Speedcore Component Library User Guide (UG065)

Speedcore eFPGA

Achronix®
Data Acceleration

Copyrights, Trademarks and Disclaimers

Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

Website: www.achronix.com
E-mail : info@achronix.com

Table of Contents

Chapter - 1: Introduction

The Achronix Speedcore component library lists the programmable fabric silicon elements which may be instantiated into a user design. These components provide access to low-level fabric primitives or, in some cases, macros which configure complex elements into advanced functions. Each entry describes the operation of the component as well as any parameters that must be initialized. Verilog and VHDL templates are also provided to aide in the implementation of user designs.

The Speedcore family includes multiple devices which not only have a different quantity of logic for each device but also different components, primarily, but not limited to, memory and arithmetic. To better understand which components a particular Speedcore device has, consult with the Speedcore Device Catalog which lists all available devices and contains tables of available resources for each core.

This guide contains the following sections:

ACX_ Prefix

All Achronix silicon components start with `ACX_` as their formal name. Therefore, when directly instantiating any component, the `ACX_xxx` name must be used. This prefix provides protection against inadvertently instantiating one of the Synplify Pro built-in primitives (primarily DFF and LUT), and distinguishes Achronix silicon components from any other library components. In addition the `ACX_xxx` wrapper exposes only the parameters and ports needed/available for a user configuration. It allows for silicon only, or test only, ports and parameters to be masked off, reducing the scope for error when directly instantiating.

When viewing Synplify Pro resource utilization reports, Synplify Pro may list multiple forms of the same component; e.g., `ACX_BRAM72K` and `BRAM72K`. The former indicates a directly instantiated component using the required `ACX_` prefix. The latter indicates an inferred component created by Synplify Pro. Both forms of the component are identical in function; the differences are only in the instantiation level. The total number of silicon components required will be the sum of these instances.

Chapter - 2: Fabric Architecture

Introduction

The Speedcore fabric architecture floorplan consists of 6-input LUTs, each with two flops, arranged as logic groups within a reconfigurable logic block (RLB6). The RLB6s are arranged in a grid, interleaved with columns of memory and arithmetic blocks. The block functions are connected by a uniform global interconnect, which enables the routing of signals between core elements. Switch boxes make the connection points between vertical and horizontal routing tracks. Inputs to and outputs from each of the functions connect to the global interconnect.

This floorplan of functional blocks and global interconnects is shown in the following figure.

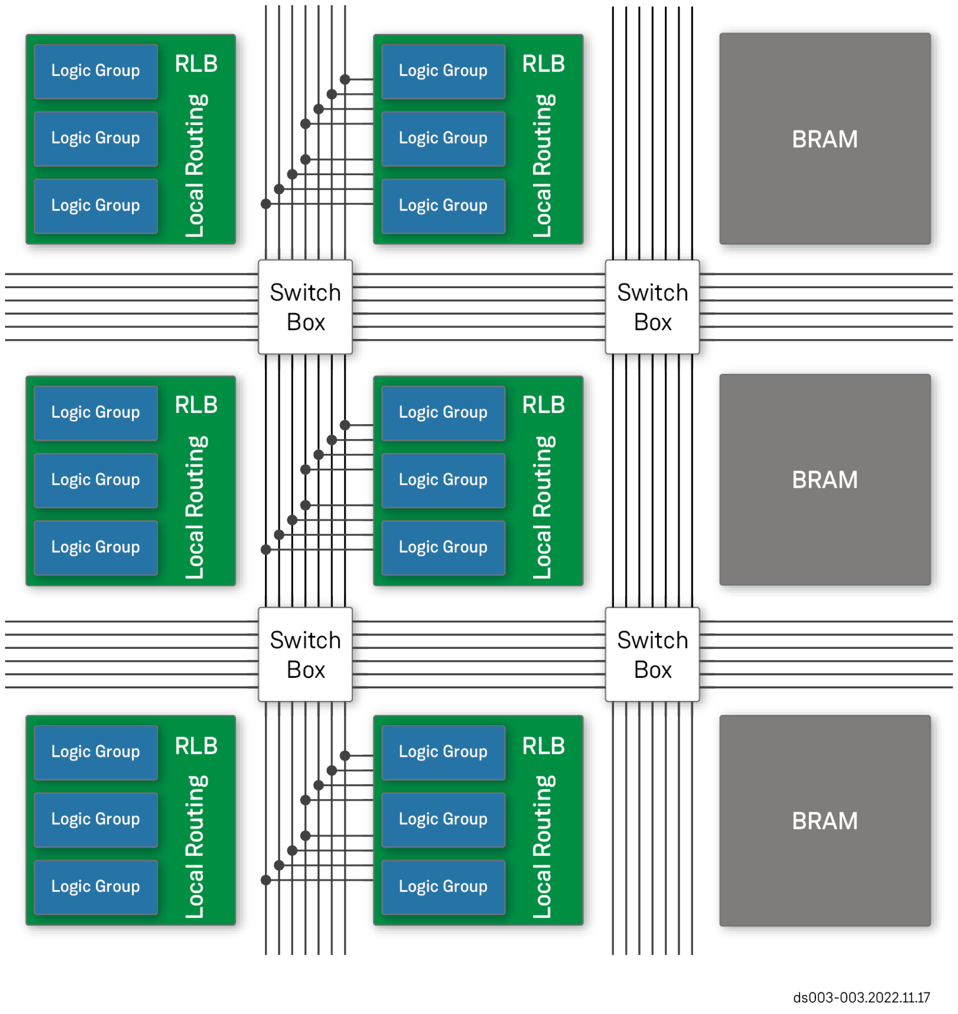


Figure 1: ***Speedcore Fabric Floorplan***

The fabric logic capabilities and functions are defined by the structure of the RLB6.

RLB6 for Gen4 Speedcore eFPGAs

The 6-input LUT-based reconfigurable logic block (RLB6) is composed of three parallel logic groups as shown in the following diagram.

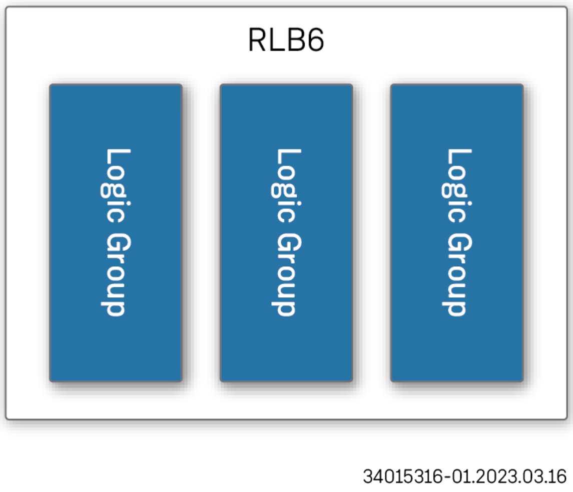


Figure 2: ***RLB6 Block Diagram***

Each logic group in a Gen4 RLB6 contains four 6-input look-up-tables (LUT6), each with two optional registers and an 8-bit fast arithmetic logic unit (ALU8) to implement logic functionality. Each logic group receives a carry-in input from the corresponding logic group in the RLB6 to the north and can propagate a carry-out output to the corresponding logic group in the RLB6 to the south.

The following table provides information on the resource counts inside an RLB6 for Gen4.

Table 1: ***RLB6 Gen4 Resource Counts***

RLB6 Resource	Count
Logic Groups	3
LUT6	12
Registers	24
8-bit ALU8	3

The following features are available using the resources in the RLB:

- 8-bit ALU for adders, counters, and comparators
- 8-to-1 MUX with single-level delay (can be inferred)
- Support for LUT chaining within the same RLB and between RLBs
- Dedicated connections for high-efficiency shift registers
- Multiplier LUT (MLUT) mode for efficient multipliers (for Speedster7t devices only)
- Ability to fan-out a clock enable or reset signal to multiple tiles without using general routing resources

- 6-input LUT configurable to function as two 5-input LUTs using shared inputs and two outputs
- Support for combining two 6-input LUTs with a dynamic select to provide 7-input LUT functionality

The following figure provides a simplified view of the circuitry inside a single logic group.

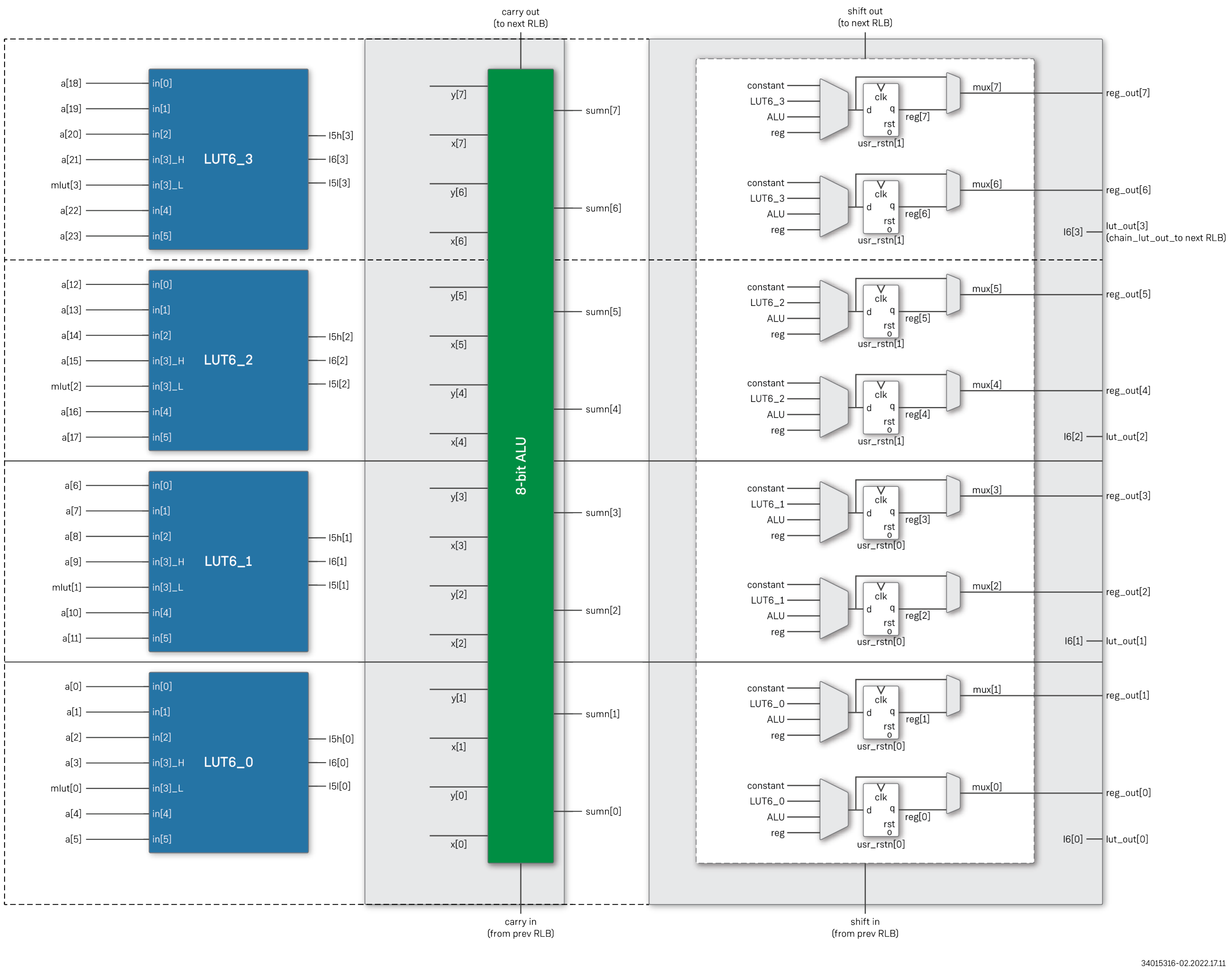


Figure 3: ***Logic Group Details***

Routing Between RLB6s

There are special considerations when routing ALU carry chains and shift registers. The Achronix Gen4 fabric has hard-wired connections on the signals `carry_in/carry_out` of each ALU. As previously mentioned, each logic group routes to the corresponding logic group in the RLB6 above or below. In other words, the ALU `carry_in/carry_out` does not route to the next ALU within the same RLB, but rather the same logic group of the next RLB6. The following figure shows the `carry_in/carry_out` routing of an ALU.

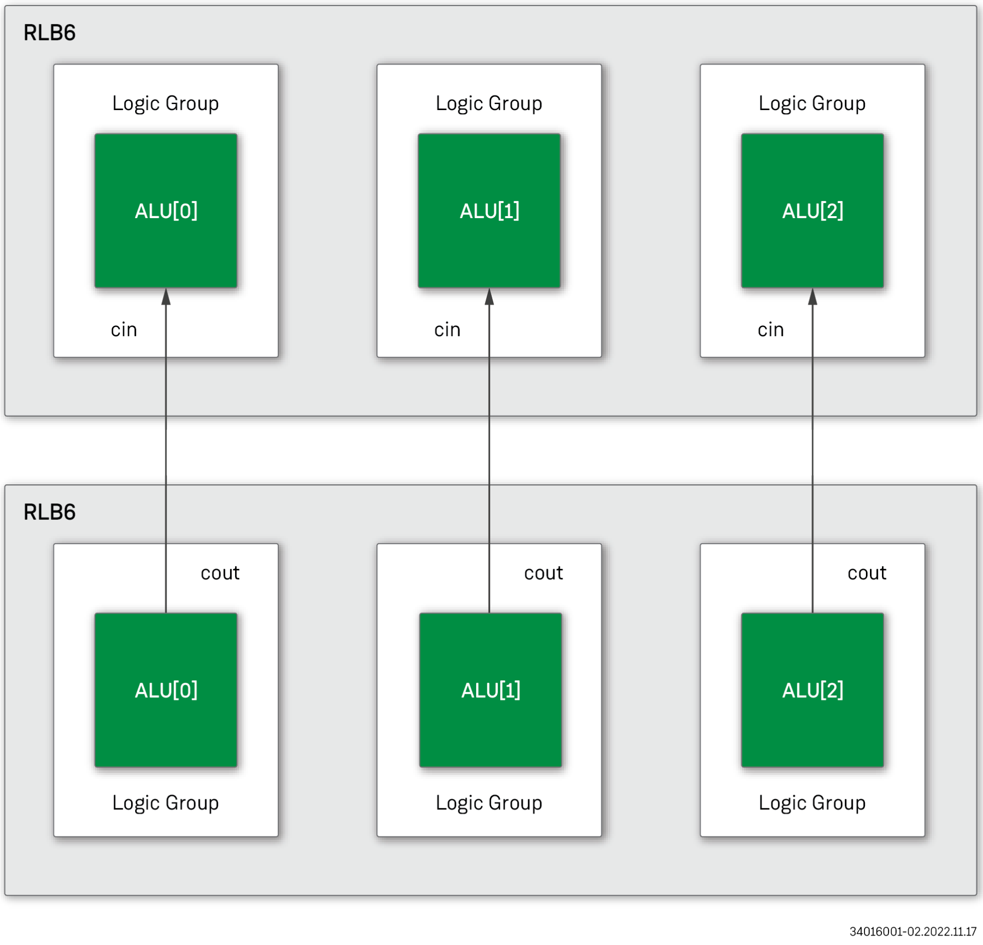


Figure 4: ***ALU Carry Chain Routing***

As true for `carry_in/carry_out`, the same is true for the signals `shift_in/shift_out` in the registers of a logic group. When creating a shift register, the registers within a logic group route to each other, but the `shift_in/shift_out` of each logic group routes to the same logic group in the next RLB6.

The following figure shows details of the routing in the Gen4 fabric.

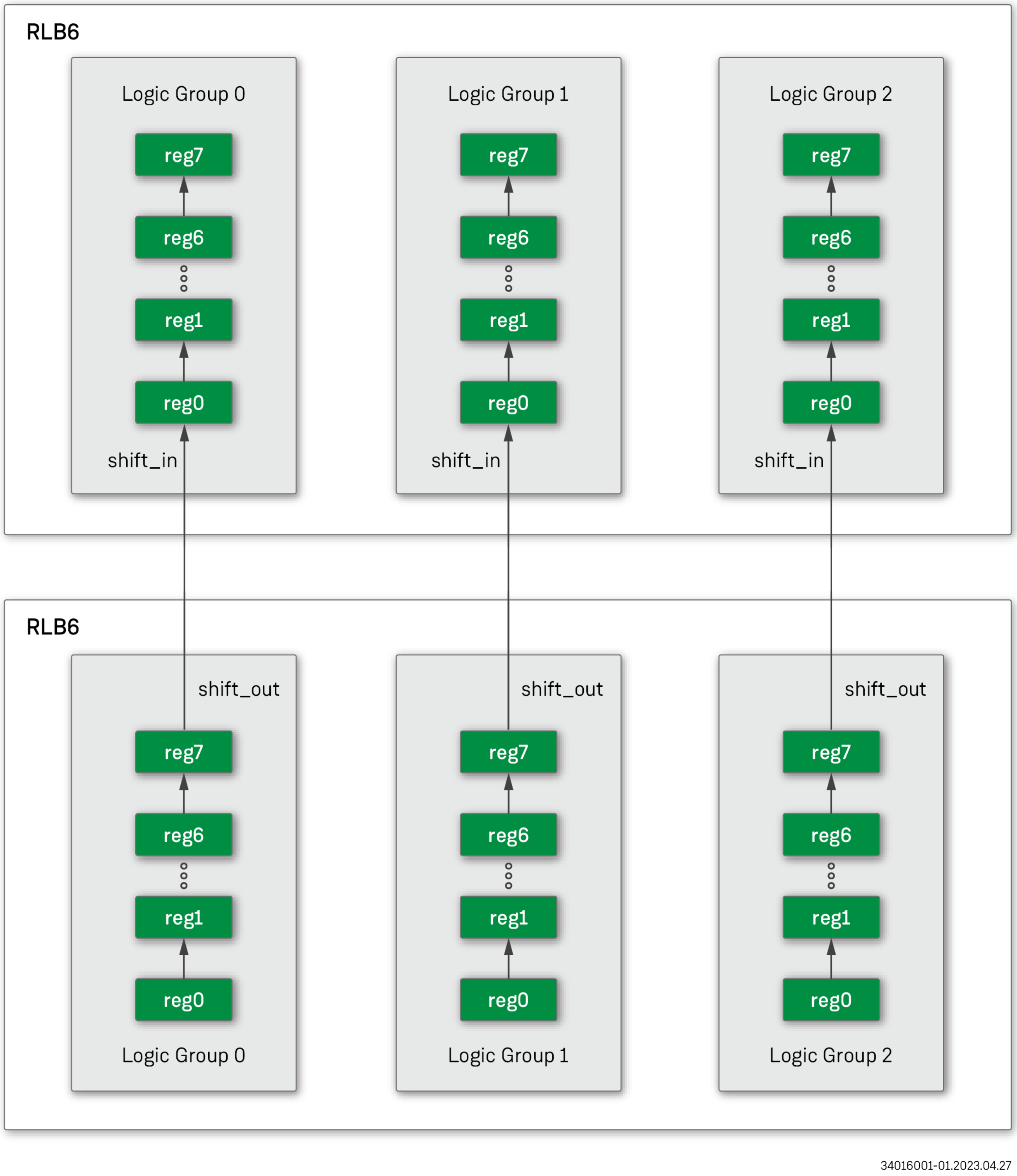


Figure 5: ***Shift Register Routing***

RLB6 Detail

Within each RLB6 are the three logic groups, each containing four 6-input LUTs (LUT6s), one ALU8, and eight registers. The logic group has ALU and flip-flop cascade paths between its associated RLB6 logic groups. The following figure shows the routing detail of one fourth of a logic group (one LUT6 and two registers).

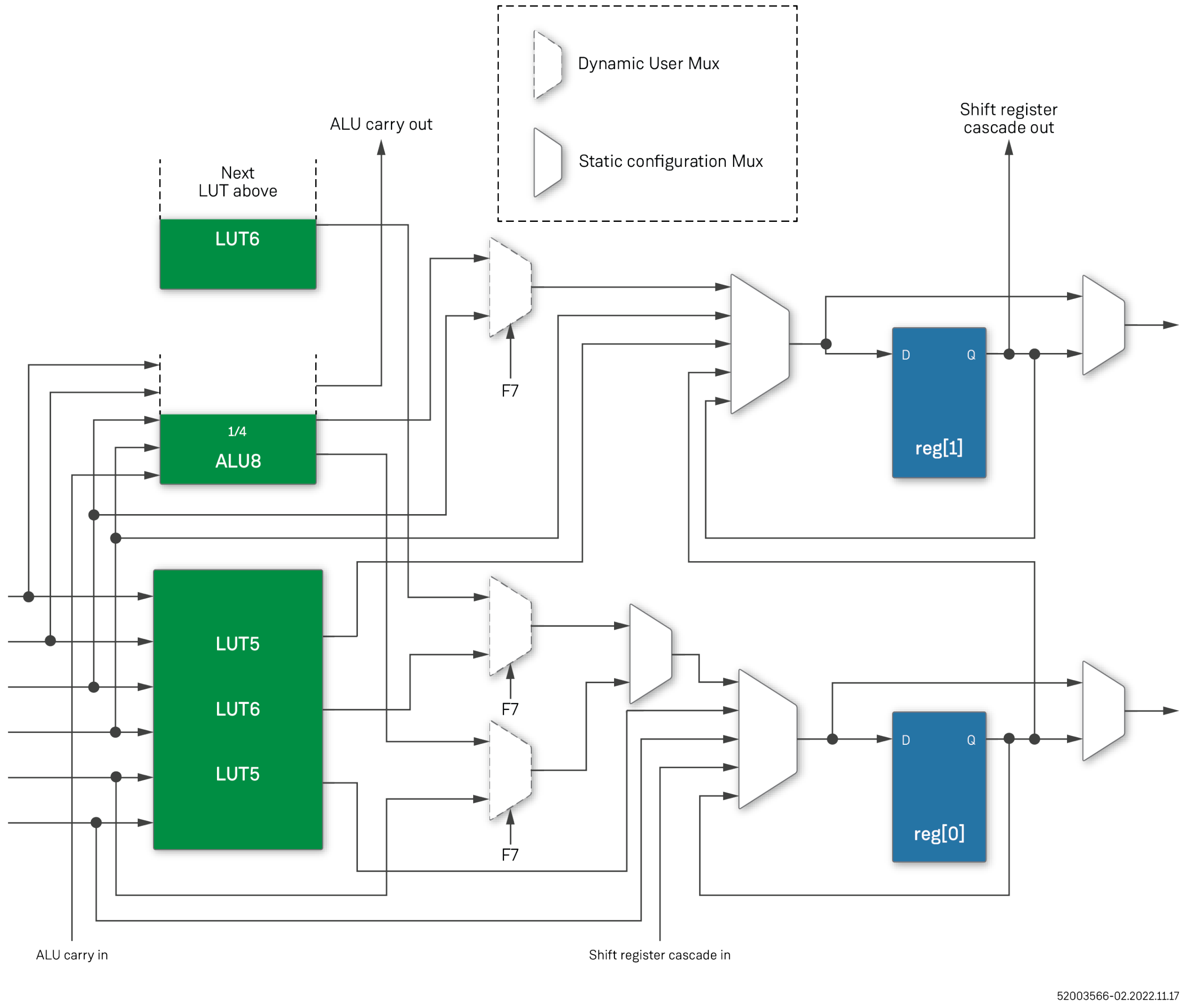


Figure 6: *One-Fourth of a Logic Group (Connection Detail)*

The diagram shows the following:

- Certain LUT6 inputs are shared with ALU8 inputs
- The LUT6 can be operated as dual 5-input LUTs (LUT5s)

- The input to each register can be selected from the following:
  - Local LUT6 output, or the LUT6 above
  - LUT5 output
  - ALU8 output (sum output)
  - LUT6 input (load input)
  - Register output (feedback path)
  - Register cascade from register below (shift register cascade)
- Some of the above inputs are statically configured by the bitstream, and other inputs can be dynamically selected. The dynamic selection is performed by the F7 signal which is an input to the logic group. The F7 allows for dynamic selection of the following:
  - Lower register – first mux: ALU sum output, or register load input (shared with LUT6 input)
  - Lower register – second mux: local LUT6 output or LUT6 above output
  - Upper register – ALU sum output, or register load input (shared with LUT6 input)

Mutually Exclusive Operations

The shared connections result in a number of mutually exclusive operations that can be achieved by a single logic group. When using all the LUT6s, the ALU8 is not available, nor is register load.

When using the ALU8:

- When ALU8 is used for A[7:0]+B[7:0]+Cin, one independent LUT6 is available.
- When ALU8 is used for A[7:0]+B[7:0], one independent LUT6 and one independent LUT2 is available.
- When ALU8 is used for A[7:0]+'Const', two independent LUT6 and one independent LUT4 are available.
- When ALU8 is used for A[3:0]+B[3:0]+Cin, two independent LUT4 are available.

When using dynamic register load, or the ALU8 sum, no LUT6s are available. When using static register load, four independent LUT4 are available.

When using F7 mux function, forming an 8:1 multiplexer (MUX8), no LUT6 or ALU8 are available.

Control Signals

Within a logic group there are eight registers, numbered reg[7:0]. These registers share control signals with each logic group having two clock, clock enable and reset inputs. The control signals are subsequently divided between the registers, with one set for registers[3:0], and the other set for registers[7:4].

Note

For designs with high utilization, ensure that as many registers as possible have common control signal sets to allow for optimum packing of the registers into logic groups.

RLB6 for Gen5 Speedcore eFPGAs

The 6-input LUT-based reconfigurable logic block (RLB6) is composed of three parallel logic groups as shown in the following diagram.

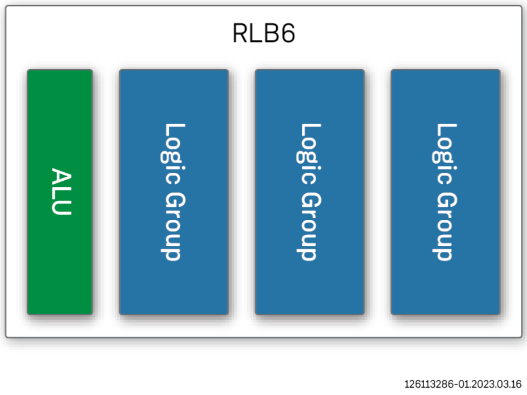


Figure 7: *RLB6 Block Diagram*

Each logic group in a Gen5 RLB6 contains four 6-input lookup tables (LUT6), each with two optional registers to implement logic functionality. Additionally, each Gen5 RLB6 includes a single 8-bit fast arithmetic logic unit (ALU8). Each ALU8 receives a carry-in input from the corresponding ALU8 in the RLB6 to the north and can propagate a carry-out output to the corresponding ALU8 in the RLB6 to the south.

The following table provides details on the resource counts inside an RLB6 for Gen5.

Table 2: *RLB6 Gen5 Resource Counts*

RLB6 Resource	Count
Logic Groups	3
LUT6	12
Registers	24
8-bit ALU8	1

The following features are available using the resources in the RLB:

- 8-bit ALU for adders, counters, and comparators
- 8-to-1 MUX with single-level delay (can be inferred)
- Support for LUT chaining within the same RLB and between RLBs
- Dedicated connections for high-efficiency shift registers
- Ability to fan-out a clock enable or reset signal to multiple tiles without using general routing resources
- 6-input LUT configurable to function as two 5-input LUTs using shared inputs and two outputs
- Support for combining two 6-input LUTs with a dynamic select to provide 7-input LUT functionality

The following figure provides a simplified view of the circuitry inside a single logic group.

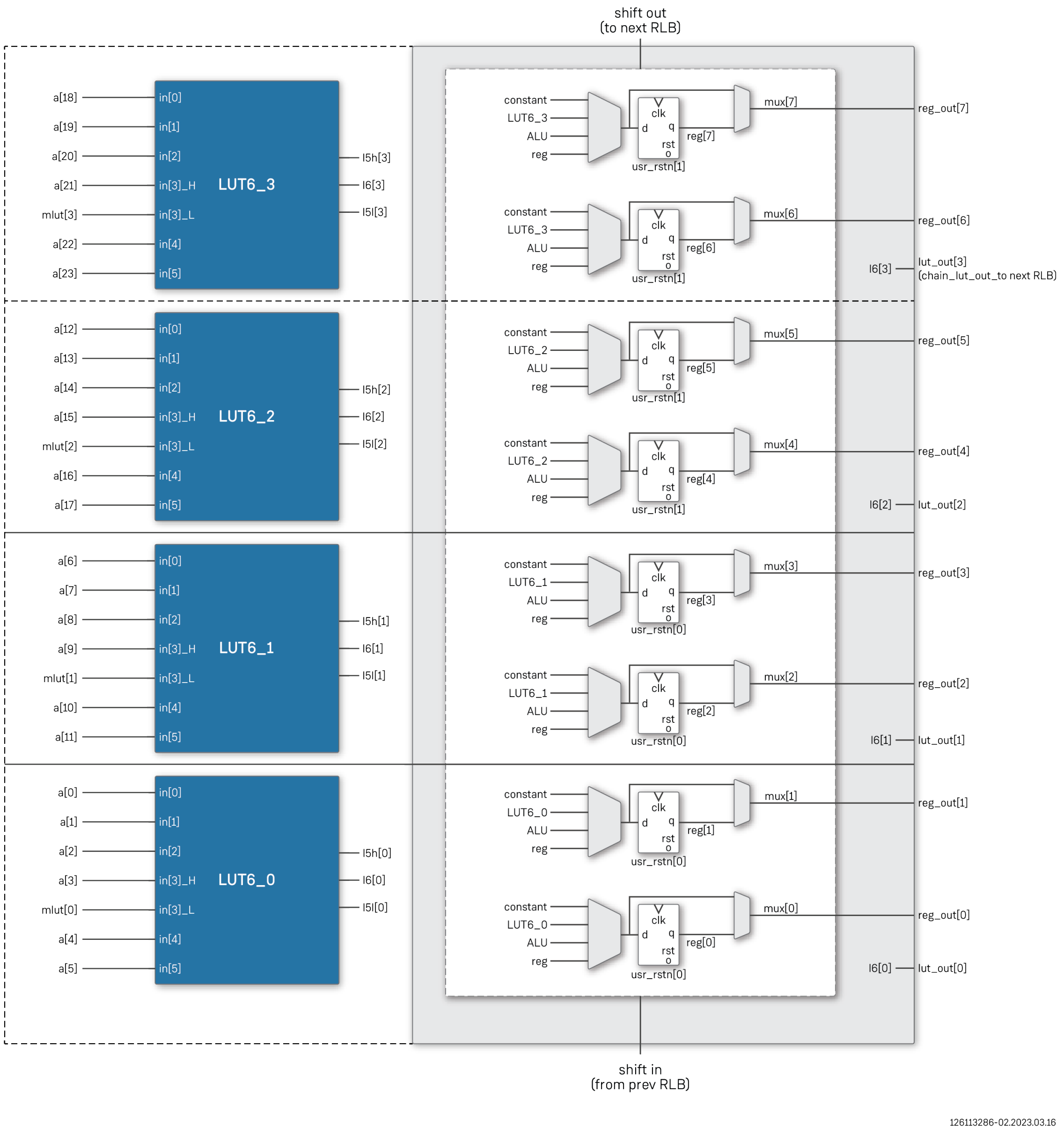


Figure 8: *Logic Group Details*

The following figure provides a simplified view of the logic groups with circuitry for the ALU inside the RLB6.

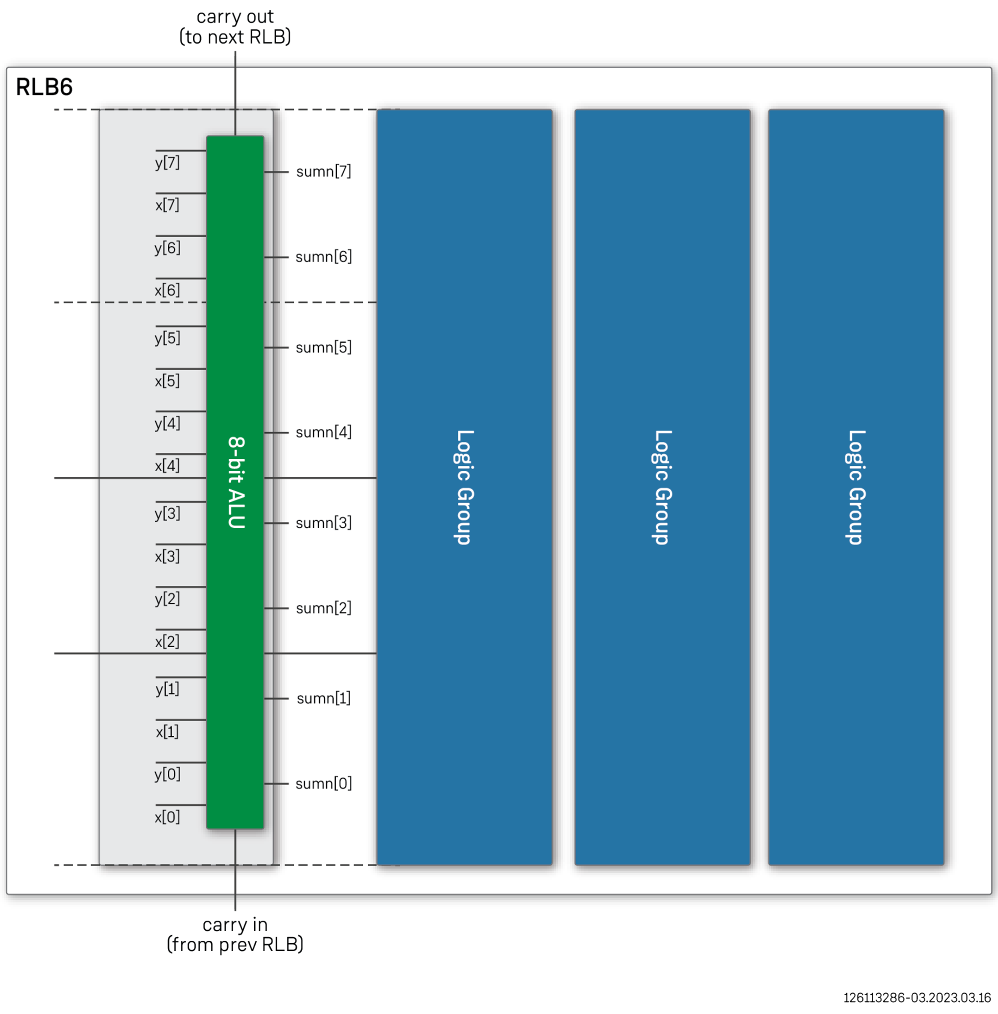


Figure 9: ***Logic Groups With ALU Details***

Routing Between RLB6s

There are special considerations when routing ALU carry chains and shift registers. The Achronix Gen5 fabric has hard-wired connections on the signals `carry_in/carry_out` of each ALU. As previously mentioned, each logic group routes to the corresponding logic group in the RLB6 above or below, and the same is true for the ALUs. In other words, the ALU `carry_in/carry_out` does not route to the next ALU to the east/west of the RLB6, but rather the ALU of the next RLB6 to the north.

The following figure shows the `carry_in/carry_out` routing of an ALU.

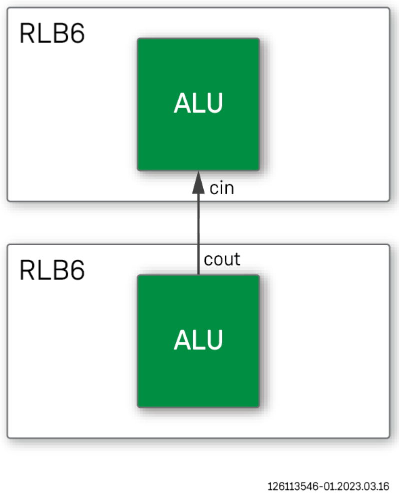
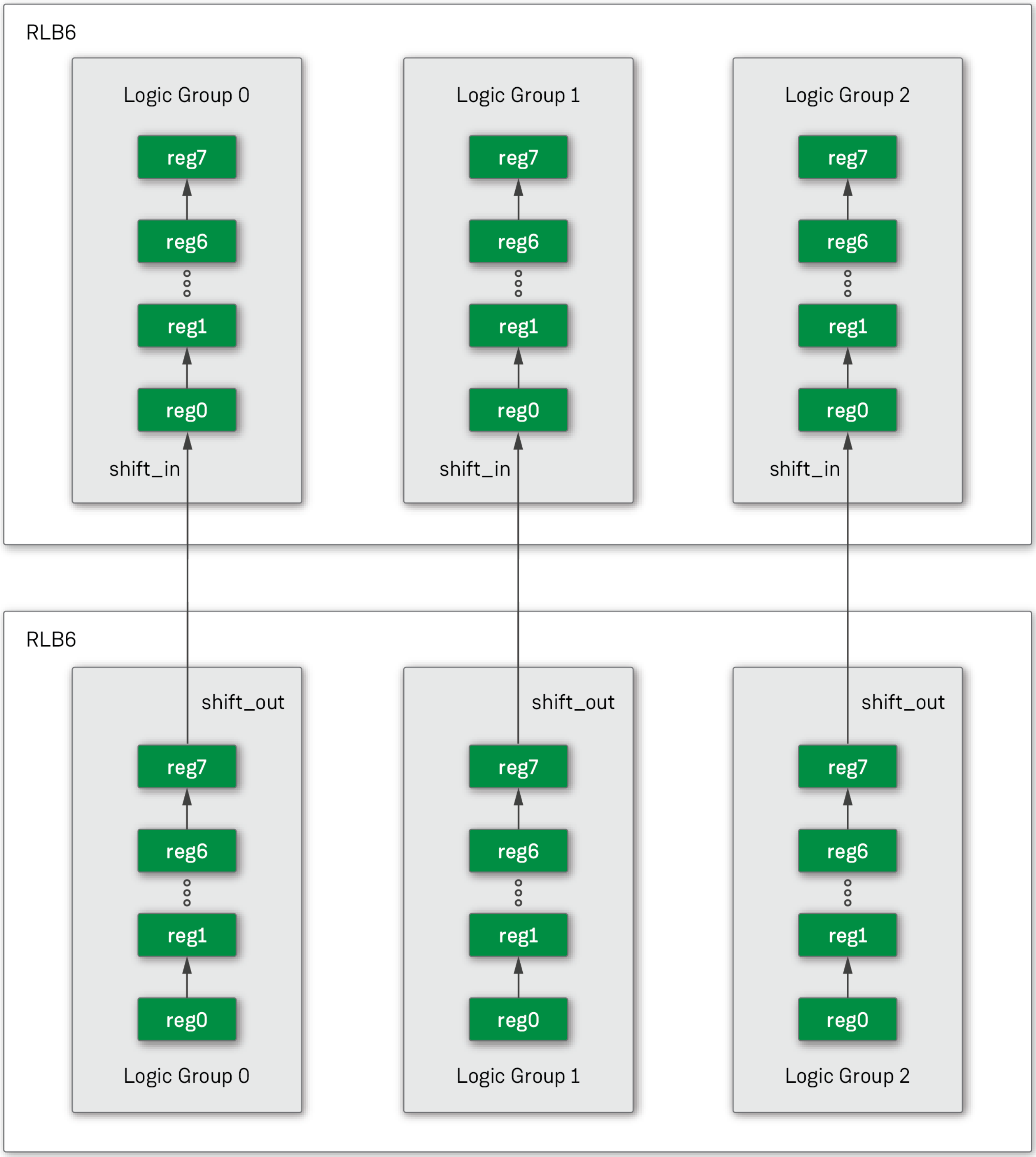


Figure 10: *ALU Carry Chain Routing*

As for `carry_in/carry_out`, the same is true for the signals `shift_in/shift_out` in the registers of a logic group. When creating a shift register, the registers within a logic group route to each other, but the `shift_in /shift_out` of each logic group routes to the same logic group in the next RLB6.

The following figure shows details of the routing in the Gen5 fabric.



126113546-02.2023.03.16

Figure 11: *Shift Register Routing*

Lookup Table (LUT) Functions

Six-Input Lookup Table (ACX_LUT6)

ACX_LUT6 implements a six-input lookup table with data inputs (`din0–din5`) and data output (`dout`), whose function is defined by the 64-bit parameter `lut_function`.

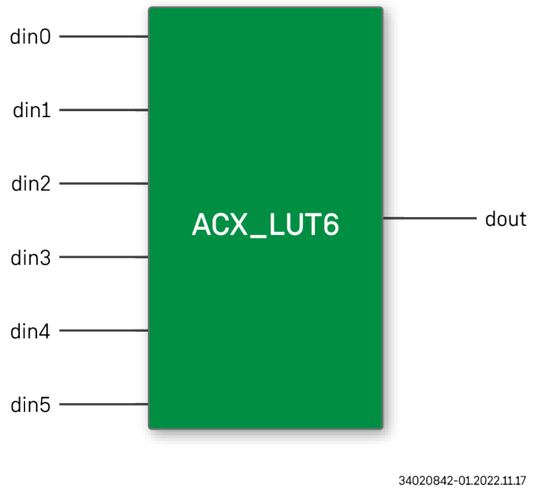


Figure 12: ***Logic Symbol***

Parameters

Table 3: ***Parameters***

Parameter	Defined Values	Default Value	Description
`lut_function`	64-bit hexadecimal value	`64'h0`	The `lut_function` parameter defines the value on the `dout` output of the LUT6 as detailed in the function table (see page 24).

Ports

Table 4: ***Pin Descriptions***

Name	Type	Description
`din0–din5`	Input	Data inputs.
`dout`	Output	Data output. The value on `dout` is the part of the `lut_function` parameter indexed by the inputs `{din5, din4, din3, din2, din1, din0}`.

Function

Table 5: *Function Table*

din5	din4	din3	din2	din1	din0	dout
0	0	0	0	0	0	`lut_function[0]`
0	0	0	0	0	1	`lut_function[1]`
0	0	0	0	1	0	`lut_function[2]`
0	0	0	0	1	1	`lut_function[3]`
0	0	0	1	0	0	`lut_function[4]`
...	...	...	...	...	...	...
1	1	1	1	0	1	`lut_function[61]`
1	1	1	1	1	0	`lut_function[62]`
1	1	1	1	1	1	`lut_function[63]`

Instantiation Templates

Verilog

```
ACX_LUT6
#(
    .lut_function     (64'h012345678abcdef)
) instance_name (
    .dout             (user_out),
    .din0             (user_in0),
    .din1             (user_in1),
    .din2             (user_in2),
    .din3             (user_in3),
    .din4             (user_in4),
    .din5             (user_in5)
);
```

VHDL

```
-- VHDL Component template for ACX_LUT6
component ACX_LUT6 is
generic (
    lut_function         : std_logic_vector( 63 downto 0) := X"0000000000000000"
);
port (
    din0                 : in  std_logic;
    din1                 : in  std_logic;
    din2                 : in  std_logic;
    din3                 : in  std_logic;
    din4                 : in  std_logic;
    din5                 : in  std_logic;
    dout                 : out std_logic
);
end component ACX_LUT6

-- VHDL Instantiation template for ACX_LUT6
instance_name : ACX_LUT6
generic map (
    lut_function         => lut_function
)
port map (
    din0                 => user_din0,
    din1                 => user_din1,
    din2                 => user_din2,
    din3                 => user_din3,
    din4                 => user_din4,
    din5                 => user_din5,
    dout                 => user_dout
);
```

Dual Five-Input Lookup Table (ACX_LUT5x2)

ACX_LUT5x2 implements dual LUT5 lookup tables with data inputs (`din0`–`din5`) and data output (`lut5ldout` and `lut5hdout`). Each of the outputs is determined by a function which is defined by the 64-bit parameter `lut_function`.

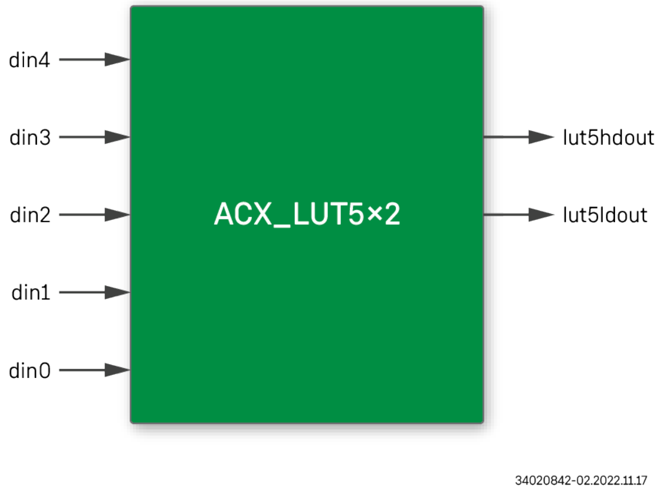


Figure 13: ***Dual LUT5 Lookup Tables***

Parameters

Table 6: ***Parameters***

Parameter	Defined Values	Default Value	Description
`lut_function`	64-bit hexadecimal value	`64'h0`	The `lut_function` parameter defines the value on both the `lut5ldout` and `lut5hdout` outputs of the LUT5x2 as detailed in function table (see page 27).

Ports

Table 7: ***Pin Descriptions***

Name	Type	Description
`din0-din4`	Input	Data inputs.
`lut5hdout`	Output	Data output. The value on `lut5hdout` is the part of the `lut_function` parameter indexed by the inputs `{1'b1, din4, din3, din2, din1, din0}`.
`lut5ldout`	Output	Data output. The value on `lut5ldout` is the part of the `lut_function` parameter indexed by the inputs `{1'b0, din4, din3, din2, din1, din0}`.

Functions

Table 8: *lut5ldout* Function Table

1'b0	din4	din3	din2	din1	din0	dout
0	0	0	0	0	0	`lut_function[0]`
0	0	0	0	0	1	`lut_function[1]`
0	0	0	0	1	0	`lut_function[2]`
0	0	0	0	1	1	`lut_function[3]`
0	0	0	1	0	0	`lut_function[4]`
...	...	...	...	...	...	...
0	1	1	1	0	1	`lut_function[29]`
0	1	1	1	1	0	`lut_function[30]`
0	1	1	1	1	1	`lut_function[31]`

Table 9: *lut5hdout* Function Table

1'b1	din4	din3	din2	din1	din0	dout
1	0	0	0	0	0	`lut_function[32]`
1	0	0	0	0	1	`lut_function[33]`
1	0	0	0	1	0	`lut_function[34]`
1	0	0	0	1	1	`lut_function[35]`
1	0	0	1	0	0	`lut_function[36]`
...	...	...	...	...	...	...
1	1	1	1	0	1	`lut_function[61]`
1	1	1	1	1	0	`lut_function[62]`
1	1	1	1	1	1	`lut_function[63]`

Instantiation Templates

Verilog

```
// Verilog template for ACX_LUT5x2
ACX_LUT5x2 #(
    .lut_function         (lut_function)
) instance_name (
    .din0                 (user_din0),
    .din1                 (user_din1),
    .din2                 (user_din2),
    .din3                 (user_din3),
    .din4                 (user_din4),
    .lut5ldout            (user_lut5ldout),
    .lut5hdout            (user_lut5hdout)
);
```

VHDL

```
-- VHDL Component template for ACX_LUT5x2
component ACX_LUT5x2 is
generic (
    lut_function         : std_logic_vector(63 downto 0) := X"0000000000000000"
);
port (
    din0                 : in  std_logic;
    din1                 : in  std_logic;
    din2                 : in  std_logic;
    din3                 : in  std_logic;
    din4                 : in  std_logic;
    lut5ldout            : out std_logic;
    lut5hdout            : out std_logic
);
end component ACX_LUT5x2

-- VHDL Instantiation template for ACX_LUT5x2
instance_name : ACX_LUT5x2
generic map (
    lut_function         => lut_function
)
port map (
    din0                 => user_din0,
    din1                 => user_din1,
    din2                 => user_din2,
    din3                 => user_din3,
    din4                 => user_din4,
    lut5ldout            => user_lut5ldout,
    lut5hdout            => user_lut5hdout
);
```

Speedcore Registers

Naming Convention

These macros are named based upon their characteristics and behavior. In each case, the name begins with DFF for D-type flip-flop. In addition to DFF, each has one or more modifiers which indicate its unique properties.

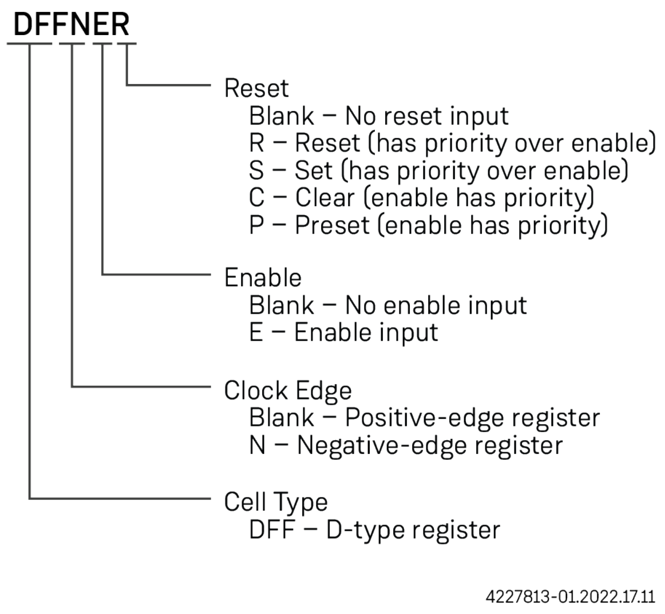

DFFNER
Reset
Blank – No reset input
R – Reset (has priority over enable)
S – Set (has priority over enable)
C – Clear (enable has priority)
P – Preset (enable has priority)
Enable
Blank – No enable input
E – Enable input
Clock Edge
Blank – Positive-edge register
N – Negative-edge register
Cell Type
DFF – D-type register
4227813-01.2022.17.11

Figure 14: ***Register Naming Convention***

Register Primitives

ACX_DFF (Positive Clock Edge D-Type Register)

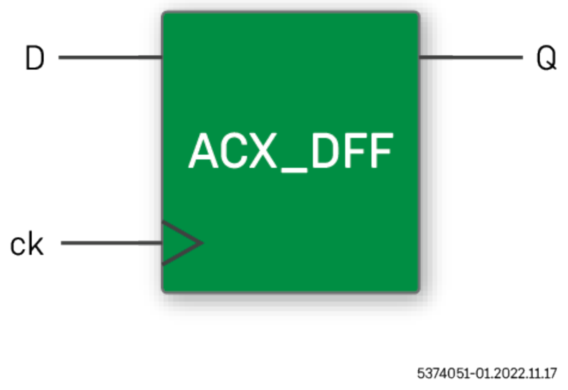


Figure 15: ***Positive Clock Edge D-Type Register***

ACX_DFF is a single D-type register with data input (d) and clock (ck) inputs and data (q) output. The data output is set to the value on the data input upon the next rising edge of the clock.

Table 10: *Parameters*

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b0`	The `init` parameter defines the initial value of the output of the DFF register. This is the value the register takes upon the initial application of power to the FPGA.

Table 11: *Pin Descriptions*

Name	Type	Description
d	Input	Data input.
ck	Input	Positive-edge clock input.
q	Output	Data output. The value present on the data input is transferred to the q output upon the rising edge of the clock.

Table 12: *Function Table*

Inputs		Output
d	**ck**	**q**
0	↑	0
1	↑	1

Instantiation Templates

Verilog

```
ACX_DFF #(
    .init    (1'b0)
) instance_name (
    .q         (user_out),
    .d         (user_din),
    .ck         (user_clock)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
----------- DONE ACHRONIX LIBRARY ---------


-- Component Instantiation
instance_name : ACX_DFF
generic map (
    init     => '0'
)
port map (
    q         => user_out,
    d         => user_din,
    ck         => user_clock
);
```

ACX_DFFE (Positive Clock Edge D-Type Register With Clock Enable)

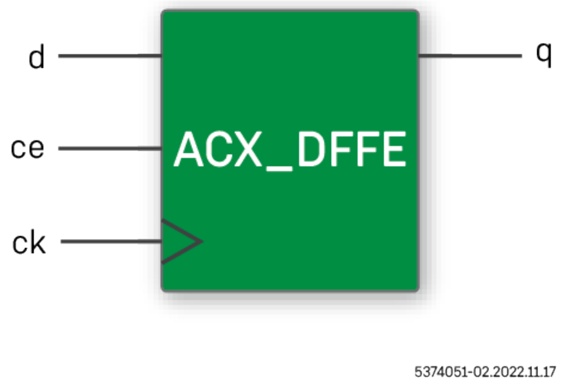


Figure 16: ***Positive Clock Edge D-Type Register With Clock Enable***

ACX_DFFE is a single D-type register with data input (`d`), clock enable (`ce`), and clock (`ck`) inputs and data (`q`) output. The data output is set to the value on the data input upon the next rising edge of the clock if the active-high clock enable input is asserted.

Table 13: ***Parameters***

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b0`	The `init` parameter defines the initial value of the output of the DFFE register. This is the value the register takes upon the initial application of power to the FPGA.

Table 14: ***Pin Descriptions***

Name	Type	Description
`d`	Input	Data input.
`ce`	Input	Active-high clock enable input.
`ck`	Input	Positive-edge clock input.
`q`	Output	Data output. The value present on the data input is transferred to the q output upon the rising edge of the clock if the clock enable input is high.

Table 15: *Function Table*

Inputs			Output
ce	d	ck	q
0	X	X	Hold
1	0	↑	0
1	1	↑	1

Instantiation Templates

Verilog

```
ACX_DFFE #(
    .init    (1'b0)
) instance_name (
    .q    (user_out),
    .d    (user_din),
    .ce    (user_clock_enable),
    .ck    (user_clock)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
----------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_DFFE
generic map (
    init     => '0'
)
port map (
    q          => user_out,
    d          => user_din,
    ce          => user_clock_enable,
    ck          => user_clock
);
```

ACX_DFFER (Positive Clock Edge D-Type Register With Clock Enable and Asynchronous/Synchronous Reset)

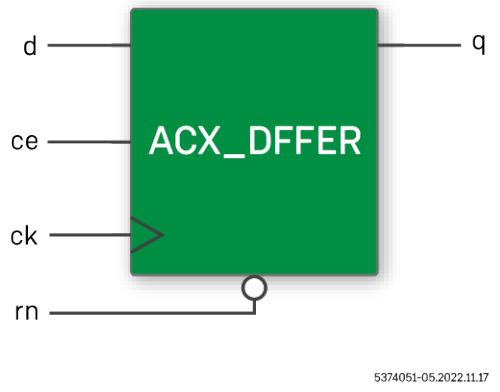


Figure 17: ***Positive Clock Edge D-Type Register With Clock Enable and Asynchronous/Synchronous Reset***

ACX_DFFER is a single D-type register with data input (`d`), clock enable (`ce`), clock (`ck`), and active-low reset (`rn`) inputs and data (`q`) output. The active-low reset input overrides all other inputs when it is asserted low and sets the data output low. The response of the q output in response to the asserted reset depends on the value of the `sr_assertion` parameter and is detailed in See ACX_DFFER Function Table with sr_assertion = "unclocked" (see page 35) and See ACX_DFFER Function Table with sr_assertion = "clocked" (see page 35). If the reset input is not asserted, the data output is set to the value on the data input upon the next rising edge of the clock if the active-high clock enable input is asserted.

Table 16: ***Parameters***

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b0`	The `init` parameter defines the initial value of the output of the DFFER register. This is the value the register takes upon the initial application of power to the FPGA.
`sr_assertion`	"unclocked", "clocked"	"unclocked"	The `sr_assertion` parameter defines the behavior of the output when the `rn` reset input is asserted. Assigning the `sr_assertion` to "unclocked" results in an asynchronous assertion of the reset signal, where the q output is set to zero upon assertion of the active-low reset signal. Assigning the `sr_assertion` to "clocked" results in a synchronous assertion of the reset signal, where the q output is set to zero at the next rising edge of the clock.

Table 17: ***Pin Descriptions***

Name	Type	Description
d	Input	Data input.
`rn`	Input	Active-low asynchronous/synchronous reset input. A low on `rn` sets the q output low independent of the other inputs if the `sr_assertion` parameter is set to "unclocked". If the `sr_assertion` parameter is set to "clocked", a low on `rn` sets the q output low at the next rising edge of the clock.
`ce`	Input	Active-high clock enable input.
`ck`	Input	Positive-edge clock input.
q	Output	Data output. The value present on the data input is transferred to the q output upon the rising edge of the clock if the clock enable input is high and the reset input is high.

Table 18: ***ACX_DFFER Function Table With sr_assertion = "unclocked"***

Inputs				Output
rn	**ce**	**d**	**ck**	**q**
0	X	X	X	0
1	0	X	X	Hold
1	1	0	↑	0
1	1	1	↑	1

Table 19: ***ACX_DFFER Function Table With sr_assertion = "clocked"***

Inputs				Output
rn	**ce**	**d**	**ck**	**q**
0	X	X	↑	0
1	0	X	X	Hold
1	1	0	↑	0
1	1	1	↑	1

Instantiation Templates

Verilog

```
ACX_DFFER #(
    .init             (1'b0),
    .sr_assertion     ("unclocked")
) instance_name (
    .q                 (user_out),
    .d                 (user_din),
    .rn                 (user_reset),
    .ce                 (user_clock_enable),
    .ck                 (user_clock)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
----------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_DFFER
generic map (
    init              => '0',
    sr_assertion      => "unclocked")
port map (
    q                  => user_out,
    d                  => user_din,
    rn                  => user_reset,
    ce                  => user_clock_enable,
    ck                  => user_clock
);
```

ACX_DFFES (Positive Clock Edge D-Type Register With Clock Enable and Asynchronous/Synchronous Set)

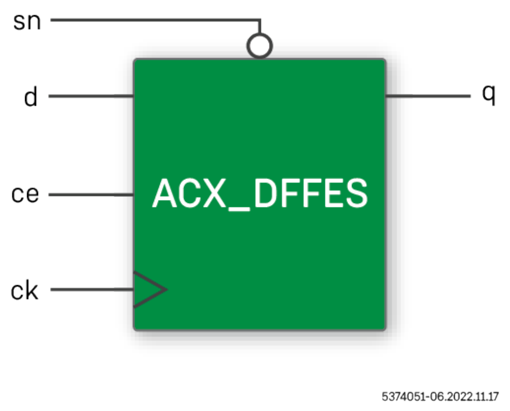


Figure 18: ***Positive Clock Edge D-Type Register With Clock Enable and Asynchronous/Synchronous Set***

ACX_DFFES is a single D-type register with data input (`d`), clock enable (`ce`), clock (`ck`), and active-low set (`sn`) inputs and data (`q`) output. The active-low set input overrides all other inputs when it is asserted low and sets the data output high. The response of the q output in response to the asserted set depends on the value of the `sr_assertion` parameter and is detailed in Table: ACX_DFFES Function Table with sr_assertion = "unclocked" (see page 35) and Table: ACX_DFFES Function Table with sr_assertion = "clocked" (see page 35). If the set input is not asserted, the data output is set to the value on the data input upon the next rising edge of the clock if the active-high clock enable input is asserted.

Table 20: ***Parameters***

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b1`	The `init` parameter defines the initial value of the output of the DFFES register. This is the value the register takes upon the initial application of power to the FPGA.
`sr_assertion`	"unclocked", "clocked"	"unclocked"	The `sr_assertion` parameter defines the behavior of the output when the `sn` set input is asserted. Assigning the `sr_assertion` to "unclocked" results in an asychronous assertion of the reset signal, where the q output is set to one upon assertion of the active-low reset signal. Assigning the `sr_assertion` to "clocked" results in a synchronous assertion of the reset signal, where the q output is set to one at the next rising edge of the clock.

Table 21: ***Pin Descriptions***

Name	Type	Description
d	Input	Data input.
sn	Input	Active-low asynchronous/synchronous set input. A low on sn sets the q output high independent of the other inputs if the `sr_assertion` parameter is set to "unclocked". If the `sr_assertion` parameter is set to "clocked", a low on `rn` sets the q output high at the next rising edge of the clock.
ce	Input	Active-high clock enable input.
ck	Input	Positive-edge clock input.
q	Output	Data output. The value present on the data input is transferred to the q output upon the rising edge of the clock if the clock enable input is high and the reset input is high.

Table 22: ***ACX_DFFES Function Table With sr_assertion = "unclocked"***

Inputs				Output
sn	**ce**	**d**	**ck**	**q**
0	X	X	X	1
1	0	X	X	Hold
1	1	0	↑	0
1	1	1	↑	1

Table 23: ***ACX_DFFES Function Table With sr_assertion = "clocked"***

Inputs				Output
sn	**ce**	**d**	**ck**	**q**
0	X	X	↑	1
1	0	X	X	Hold
1	1	0	↑	0
1	1	1	↑	1

Instantiation Templates

Verilog

```
ACX_DFFES #(
    .init            (1'b1),
    .sr_assertion    ("unclocked")
) instance_name (
    .q                (user_out),
    .d                (user_din),
    .sn                (user_set),
    .ce                (user_clock_enable),
    .ck                (user_clock)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
----------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_DFFES
generic map (
    init             => '1',
    sr_assertion     => "unclocked"
)
port map (
    q                 => user_out,
    d                 => user_din,
    sn                 => user_set,
    ce                 => user_clock_enable,
    ck                 => user_clock
);
```

ACX_DFFN (Negative Clock Edge D-Type Register)

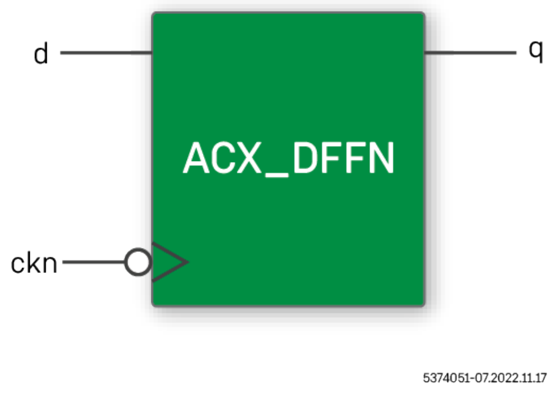


Figure 19: ***Negative Clock Edge D-Type Register***

ACX_DFFN is a single D-type register with data input (`d`) and clock (`ckn`) inputs and data (`q`) output. The data output is set to the value on the data input upon the next falling edge of the clock.

Table 24: ***Parameters***

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b0`	The init parameter defines the initial value of the output of the DFFN register. This is the value the register takes upon the initial application of power to the FPGA.

Table 25: ***Pin Descriptions***

Name	Type	Description
d	Input	Data input.
`ckn`	Input	Negative-edge clock input.
q	Output	Data output. The value present on the data input is transferred to the q output upon the falling edge of the clock.

Table 26: ***Function Table***

Inputs		Output
d	**ck**	**q**
0	↓	0
1	↓	1

Instantiation Templates

Verilog

```
ACX_DFFN #(
    .init    (1'b0)
) instance_name (
    .q        (user_out),
    .d        (user_din),
    .ckn    (user_clock)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
----------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_DFFN
generic map (
    init     => '0'
)
port map (
    q         => user_out,
    d         => user_din,
    ckn     => user_clock
);
```

ACX_DFFNER (Negative Clock Edge D-Type Register With Clock Enable and Asynchronous/Synchronous Reset)

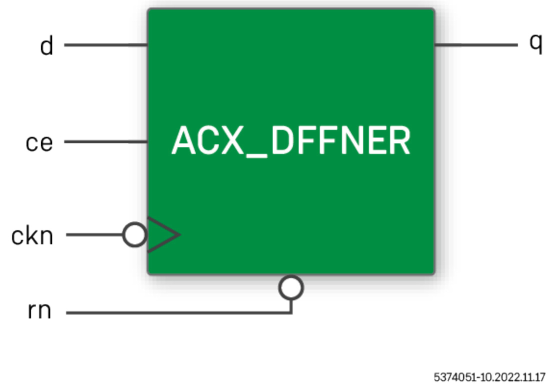


Figure 20: ***Negative Clock Edge D-Type Register With Clock Enable and Asynchronous/Synchronous Reset***

ACX_DFFNER is a single D-type register with data input (`d`), clock enable (`ce`), clock (`ckn`), and active-low reset (`rn`) inputs and data (`q`) output. The active-low reset input overrides all other inputs when it is asserted low and sets the data output low. The response of the q output in response to the asserted reset depends on the value of the `sr_assertion` parameter and is detailed in Table: ACX_DFFNER Function Table with sr_assertion = "unclocked" (see page 38) and Table: ACX_DFFNER Function Table with sr_assertion = "clocked" (see page 38). If the reset input is not asserted, the data output is set to the value on the data input upon the next falling edge of the clock if the active-high clock enable input is asserted.

Table 27: ***Parameters***

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b0`	The `init` parameter defines the initial value of the output of the DFFNER register. This is the value the register takes upon the initial application of power to the FPGA.
`sr_assertion`	"unclocked", "clocked"	"unclocked"	The `sr_assertion` parameter defines the behavior of the output when the `rn` reset input is asserted. Assigning the `sr_assertion` to "unclocked" results in an asychronous assertion of the reset signal, where the q output is set to zero upon assertion of the active-low reset signal. Assigning the `sr_assertion` to "clocked" results in a synchronous assertion of the reset signal, where the q output is set to zero at the next falling edge of the clock.

Table 28: ***Pin Descriptions***

Name	Type	Description
d	Input	Data input.
`rn`	Input	Active-low asynchronous/synchronous reset input. A low on `rn` sets the q output low independent of the other inputs if the `sr_assertion` parameter is set to "unclocked". If the `sr_assertion` parameter is set to "clocked", a low on `rn` sets the q output low at the next falling edge of the clock.
`ce`	Input	Active-high clock enable input.
`ckn`	Input	Negative-edge clock input.
q	Output	Data output. The value present on the data input is transferred to the q output upon the falling edge of the clock if the clock enable input is high and the reset input is high.

Table 29: ***ACX_DFFNER Function Table With sr_assertion = "unclocked"***

Inputs				Output
rn	**ce**	**d**	**ckn**	**q**
0	X	X	X	0
1	0	X	X	Hold
1	1	0	↓	0
1	1	1	↓	1

Table 30: ***ACX_DFFNER Function Table With sr_assertion = "clocked"***

Inputs				Output
rn	**ce**	**d**	**ckn**	**q**
0	X	X	↓	0
1	0	X	X	Hold
1	1	0	↓	0
1	1	1	↓	1

Instantiation Templates

Verilog

```
ACX_DFFNER #(
    .init            (1'b0),
    .sr_assertion    ("unclocked")
) instance_name (
    .q                (user_out),
    .d                (user_din),
    .rn                (user_reset),
    .ce                (user_clock_enable),
    .ckn            (user_clock)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
----------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_DFFNER
generic map (
    init             => '0',
    sr_assertion     => "unclocked"
)
port map (
    q                 => user_out,
    d                 => user_din,
    rn                 => user_reset,
    ce                 => user_clock_enable,
    ckn             => user_clock
);
```

ACX_DFFNES (Negative Clock Edge D-Type Register With Clock Enable and Asynchronous/Synchronous Set)

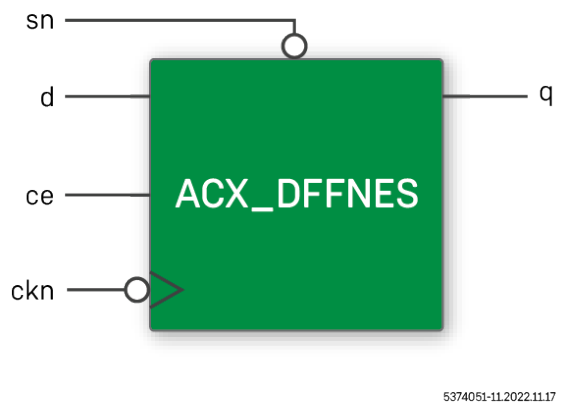


Figure 21: ***Negative Clock Edge D-Type Register With Clock Enable and Asynchronous/Synchronous Set***

ACX_DFFNES is a single D-type register with data input (`d`), clock enable (`ce`), clock (`ckn`), and active-low set (`sn`) inputs and data (`q`) output. The active-low set input overrides all other inputs when it is asserted low and sets the data output high. The response of the `q` output in response to the asserted set depends on the value of the `sr_assertion` parameter and is detailed in Table: ACX_DFFNES Function Table with sr_assertion = "unclocked" (see page 42) and Table: ACX_DFFNES Function Table with sr_assertion = "clocked" (see page 43). If the set input is not asserted, the data output is set to the value on the data input upon the next falling edge of the clock if the active-high clock enable input is asserted.

Table 31: ***Parameters***

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b1`	The `init` parameter defines the initial value of the output of the DFFNES register. This is the value the register takes upon the initial application of power to the FPGA.
`sr_assertion`	"unclocked", "clocked"	"unclocked"	The `sr_assertion` parameter defines the behavior of the output when the `sn` set input is asserted. Assigning the `sr_assertion` to "unclocked" results in an asychronous assertion of the set signal, where the `q` output is set to one upon assertion of the active-low set signal. Assigning the `sr_assertion` to "clocked" results in a synchronous assertion of the set signal, where the `q` output is set to one at the next falling edge of the clock.

Table 32: ***Pin Descriptions***

Name	Type	Description
d	Input	Data input.
sn	Input	Active-low asynchronous/synchronous set input. A low on `sn` sets the q output high independent of the other inputs if the `sr_assertion` parameter is set to "unclocked". If the `sr_assertion` parameter is set to "clocked", a low on `sn` sets the q output high at the next falling edge of the clock.
ce	Input	Active-high clock enable input.
ckn	Input	Negative-edge clock input.
q	Output	Data output. The value present on the data input is transferred to the q output upon the falling edge of the clock if the clock enable input is high and the set input is high.

Table 33: ***ACX_DFFNES Function Table With sr_assertion = "unclocked"***

Inputs				Output
sn	**ce**	**d**	**ckn**	**q**
0	X	X	X	1
1	0	X	X	Hold
1	1	0	↓	0
1	1	1	↓	1

Table 34: ***ACX_DFFNES Function Table With sr_assertion = "clocked"***

Inputs				Output
sn	**ce**	**d**	**ckn**	**q**
0	X	X	↓	1
1	0	X	X	Hold
1	1	0	↓	0
1	1	1	↓	1

Instantiation Templates

Verilog

```
ACX_DFFNES #(
    .init             (1'b1),
    .sr_assertion     ("unclocked")
) instance_name (
    .q                 (user_out),
    .d                 (user_din),
    .sn                 (user_set),
    .ce                 (user_clock_enable),
    .ckn              (user_clock)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
----------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_DFFNES
generic map (
    init             => '1',
    sr_assertion     => "unclocked"
)
port map (
    q                 => user_out,
    d                 => user_din,
    sn                 => user_set,
    ce                 => user_clock_enable,
    ckn              => user_clock
);
```

ACX_DFFNR (Negative Clock Edge D-Type Register With Asynchronous Reset)

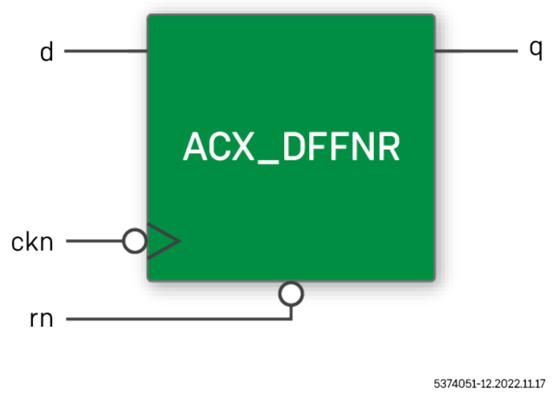


Figure 22: ***Negative Clock Edge D-Type Register With Asynchronous Reset***

ACX_DFFNR is a single D-type register with data input (`d`), clock (`ckn`), and active-low reset (`rn`) inputs and data (`q`) output. The active-low reset input overrides the other inputs when it is asserted low and sets the data output low. The response of the q output in response to the asserted reset is described under the `sr_assertion` parameter. If the reset input is not asserted, the data output is set to the value on the data input upon the next falling edge of the clock.

Table 35: ***Parameters***

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b0`	The `init` parameter defines the initial value of the output of the DFFNR register. This is the value the register takes upon the initial application of power to the FPGA.
`sr_assertion`	"unclocked", "clocked"	"unclocked"	The `sr_assertion` parameter defines the behavior of the output when the `rn` reset input is asserted. Assigning the `sr_assertion` to "unclocked" results in an asychronous assertion of the reset signal, where the q output is set low upon assertion of the active-low reset signal. Assigning the `sr_assertion` to "clocked" results in a synchronous assertion of the reset signal, where the q output is set low at the next falling edge of the clock.

Table 36: ***Pin Descriptions***

Name	Type	Description
d	Input	Data input.
`rn`	Input	Active-low asynchronous reset input. A low on `rn` sets the q output low independent of the other inputs.
`ckn`	Input	Negative-edge clock input.
q	Output	Data output. The value present on the data input is transferred to the q output upon the falling edge of the clock if the asynchronous reset input is high.

Table 37: ***Function Table With sr_assertion = "unclocked"***

Inputs			Output
rn	d	ckn	q
0	X	X	0
1	X	X	Hold
1	0	↓	0
1	1	↓	1

Table 38: ***Function Table With sr_assertion = "clocked"***

Inputs			Output
rn	d	ckn	q
0	X	↓	0
1	X	X	Hold
1	0	↓	0
1	1	↓	1

Instantiation Templates

Verilog

```
ACX_DFFNR #(
    .init    (1'b0)
) instance_name (
    .q        (user_out),
    .d        (user_din),
    .rn        (user_reset),
    .ckn    (user_clock)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
----------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_DFFNR
generic map (
    init     => '0'
)
port map (
    q         => user_out,
    d         => user_din,
    rn         => user_reset,
    ckn     => user_clock
);
```

ACX_DFFNS (Negative Clock Edge D-Type Register With Asynchronous Set)

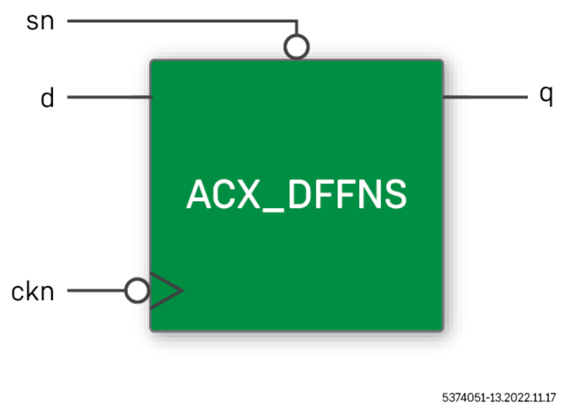


Figure 23: ***Negative Clock Edge D-Type Register With Asynchronous Set***

ACX_DFFNS is a single D-type register with data input (`d`), clock (`ckn`), and active-low set (`sn`) inputs and data (`q`) output. The active-low set input overrides the other inputs when it is asserted low and sets the data output high. The response of the `q` output in response to the asserted set is described under the `sr_assertion` parameter. If the set input is not asserted, the data output is set to the value on the data input upon the next falling edge of the clock.

Table 39: ***Parameters***

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b1`	The `init` parameter defines the initial value of the output of the DFFNS register. This is the value the register takes upon the initial application of power to the FPGA.
`sr_assertion`	"unclocked", "clocked"	"unclocked"	The `sr_assertion` parameter defines the behavior of the output when the `sn` set input is asserted. Assigning the `sr_assertion` to "unclocked" results in an asychronous assertion of the set signal, where the `q` output is set to one upon assertion of the active-low set signal. Assigning the `sr_assertion` to "clocked" results in a synchronous assertion of the set signal, where the `q` output is set to one at the next falling edge of the clock.

Table 40: ***Pin Descriptions***

Name	Type	Description
d	Input	Data input.
sn	Input	Active-low asynchronous set input. A low on sn sets the q output high independent of the other inputs.
ckn	Input	Negative-edge clock input.
q	Output	Data output. The value present on the data input is transferred to the q output upon the falling edge of the clock if the asynchronous set input is high.

Table 41: ***Function Table With sr_assertion = "unclocked"***

Inputs			Output
sn	d	ckn	q
0	X	X	1
1	X	X	Hold
1	0	↓	0
1	1	↓	1

Table 42: ***Function Table With sr_assertion = "clocked"***

Inputs			Output
sn	d	ckn	q
0	X	↓	1
1	X	X	Hold
1	0	↓	0
1	1	↓	1

Instantiation Templates

Verilog

```
ACX_DFFNS #(
    .init    (1'b1)
) instance_name (
    .q        (user_out),
    .d        (user_din),
    .sn        (user_set),
    .ckn    (user_clock)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
----------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_DFFNS
generic map (
    init     => '1'
)
port map (
    q         => user_out,
    d         => user_din,
    sn         => user_set,
    ckn     => user_clock
);
```

ACX_DFFR (Positive Clock Edge D-Type Register With Asynchronous Reset)

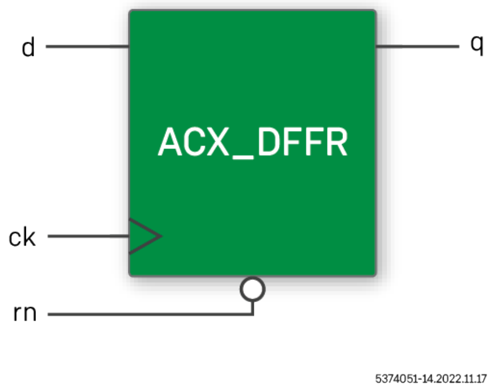


Figure 24: ***Positive Clock Edge D-Type Register With Asynchronous Reset***

ACX_DFFR is a single D-type register with data input (`d`), clock (`ck`), and active-low reset (`rn`) inputs and data (`q`) output. The active-low reset input overrides the other inputs when it is asserted low and sets the data output low. The response of the `q` output in response to the asserted reset is described under the `sr_assertion` parameter. If the reset input is not asserted, the data output is set to the value on the data input upon the next rising edge of the clock.

Note

References may be seen to DFFC in the resulting netlist. This macro is functionally equivalent to the DFFR. ACE software automatically replaces any instance of DFFC with DFFR.

Table 43: ***Parameters***

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b0`	The `init` parameter defines the initial value of the output of the DFFR register. This is the value the register takes upon the initial application of power to the FPGA.
`sr_assertion`	"unclocked", "clocked"	"unclocked"	The `sr_assertion` parameter defines the behavior of the output when the `rn` reset input is asserted. Assigning the `sr_assertion` to "unclocked" results in an asychronous assertion of the reset signal, where the q output is set low upon assertion of the active-low reset signal. Assigning the `sr_assertion` to "clocked" results in a synchronous assertion of the reset signal, where the q output is set low at the next rising edge of the clock.

Table 44: ***Pin Descriptions***

Name	Type	Description
d	Input	Data input.
`rn`	Input	Active-low asynchronous reset input. A low on `rn` sets the q output low independent of the other inputs.
`ck`	Input	Positive-edge clock input.
q	Output	Data output. The value present on the data input is transferred to the q output upon the rising edge of the clock if the asynchronous reset input is high.

Table 45: ***Function Table With sr_assertion = "unclocked"***

Inputs			Output
rn	**d**	**ck**	**q**
0	X	X	0
1	X	X	Hold
1	0	↑	0
1	1	↑	1

Table 46: ***Function Table With sr_assertion = "clocked"***

Inputs			Output
rn	**d**	**ck**	**q**
0	X	↑	0
1	X	X	Hold
1	0	↑	0
1	1	↑	1

Instantiation Templates

Verilog

```
ACX_DFFR #(
    .init    (1'b0)
) instance_name (
    .q         (user_out),
    .d         (user_din),
    .rn         (user_reset),
    .ck         (user_clock)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
----------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_DFFR
generic map (
    init     => '0'
)
port map (
    q         => user_out,
    d         => user_din,
    rn         => user_reset,
    ck         => user_clock
);
```

ACX_DFFS (Positive Clock Edge D-Type Register With Asynchronous Set)

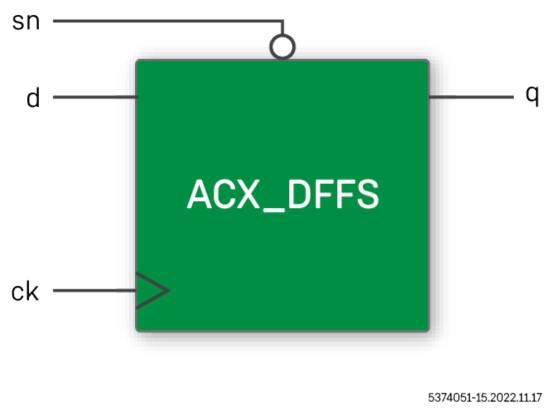


Figure 25: ***Positive Clock Edge D-Type Register With Asynchronous Set***

ACX_DFFS is a single D-type register with data input (`d`), clock (`ck`), and active-low set (`sn`) inputs and data (`q`) output. The active-low set input overrides the other inputs, when it is asserted low the data output is asserted high. The response of the q output in response to the asserted set is described under the `sr_assertion` parameter. If the set input is not asserted, the data output is set to the value on the data input upon the next rising edge of the clock.

Note

References may be seen to DFFP in the resulting netlist. This macro is functionally equivalent to the DFFS. ACE software automatically replaces any instance of DFFP with DFFS.

Table 47: ***Parameters***

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b1`	The `init` parameter defines the initial value of the output of the DFFS register. This is the value the register takes upon the initial application of power to the FPGA.
`sr_assertion`	"unclocked", "clocked"	"unclocked"	The `sr_assertion` parameter defines the behavior of the output when the `sn` set input is asserted. Assigning the `sr_assertion` to "unclocked" results in an asychronous assertion of the set signal, where the q output is set to one upon assertion of the active-low set signal. Assigning the `sr_assertion` to "clocked" results in a synchronous assertion of the set signal, where the q output is set to one at the next rising edge of the clock.

Table 48: ***Pin Descriptions***

Name	Type	Description
d	Input	Data input.
sn	Input	Active-low asynchronous set input. A low on sn sets the q output high independent of the other inputs.
ck	Input	Positive-edge clock input.
q	Output	Data output. The value present on the data input is transferred to the q output upon the rising edge of the clock if the asynchronous set input is high.

Table 49: ***Function Table With sr_assertion = "unclocked"***

Inputs			Output
sn	**d**	**ck**	**q**
0	X	↑	1
1	X	X	Hold
1	0	↑	0
1	1	↑	1

Table 50: ***Function Table With sr_assertion = "clocked"***

Inputs			Output
sn	**d**	**ck**	**q**
0	X	X	1
1	X	X	Hold
1	0	↑	0
1	1	↑	1

Instantiation Template

Verilog

```
ACX_DFFS #(
    .init    (1'b1)
) instance_name (
    .q        (user_out),
    .d        (user_din),
    .sn        (user_set),
    .ck        (user_clock)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
----------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_DFFS
generic map (
    init     => '1'
)
port map (
    q     => user_out,
    d     => user_din,
    sn     => user_set,
    ck     => user_clock
);
```

Register Macros

The following DFF modes are not natively supported by the hardware, but are transparently resolved into the appropriate primitives by ACE software.

ACX_DFFNEP (Negative Clock Edge D-Type Register With Clock Enable and Synchronous Preset)

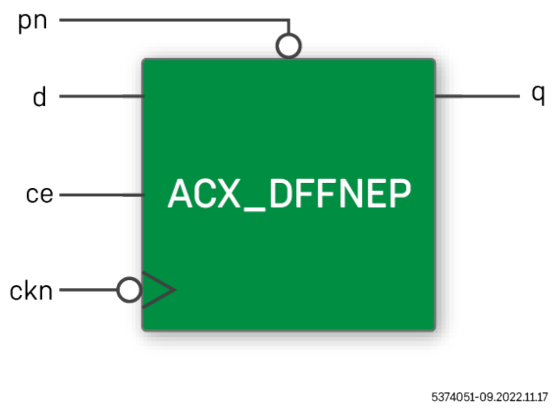


Figure 26: ***Negative Clock Edge D-Type Register With Clock Enable and Synchronous Preset***

ACX_DFFNEP is a single D-type register with data input (`d`), clock enable (`ce`), clock (`ckn`), and active-low synchronous preset (`pn`) inputs and data (`q`) output. The active-low synchronous preset input sets the data output high upon the next falling edge of the clock if it is asserted low and the clock enable signal is asserted high. If the synchronous preset input is not asserted, the data output is set to the value on the data input upon the next falling edge of the clock if the active-high clock enable input is asserted.

Table 51: ***Parameters***

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b1`	The `init` parameter defines the initial value of the output of the DFFNEP register. This is the value the register takes upon the initial application of power to the FPGA.

Table 52: ***Pin Descriptions***

Name	Type	Description
d	Input	Data input.
pn	Input	Active-low synchronous preset input. A low on pn sets the q output high upon the next falling edge of the clock if the clock enable is asserted high.
ce	Input	Active-high clock enable input.
ckn	Input	Negative-edge clock input.
q	Output	Data output. The value present on the data input is transferred to the q output upon the falling edge of the clock if the clock enable input is high and the synchronous preset input is high.

Table 53: ***Function Table***

Inputs				Output
pn	ce	d	ckn	q
X	0	X	X	Hold
0	1	X	↓	1
1	1	0	↓	0
1	1	1	↓	1

Instantiation Templates

Verilog

```
ACX_DFFNEP #(
    .init    (1'b1)
) instance_name (
    .q        (user_out),
    .d        (user_din),
    .pn        (user_preset)
    .ce        (user_clock_enable),
    .ckn    (user_clock)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
----------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_DFFNEP
generic map (
    init     => '1'
)
port map (
    q         => user_out,
    d         => user_din,
    pn         => user_preset,
    ce         => user_clock_enable,
    ckn     => user_clock
);
```

ACX_DFFEC (Positive Clock Edge D-Type Register With Clock Enable and Synchronous Clear)

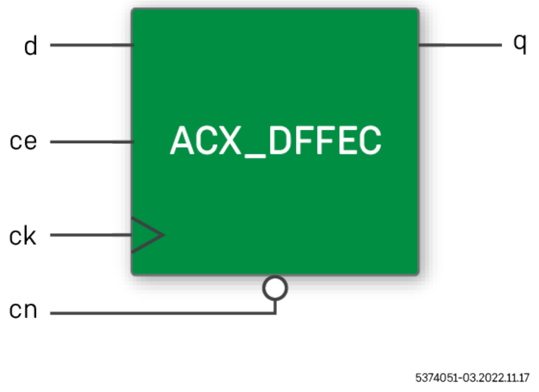


Figure 27: ***Positive Clock Edge D-Type Register With Clock Enable and Synchronous Clear***

ACX_DFFEC is a single D-type register with data input (`d`), clock enable (`ce`), clock (`ck`), and active-low synchronous clear (`cn`) inputs and data (`q`) output. The active-low synchronous clear input sets the data output low upon the next rising edge of the clock if it is asserted low and the clock enable signal is asserted high. If the synchronous clear input is not asserted, the data output is set to the value on the data input upon the next rising edge of the clock if the active-high clock enable input is asserted.

Table 54: ***Parameters***

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b0`	The `init` parameter defines the initial value of the output of the DFFEC register. This is the value the register takes upon the initial application of power to the FPGA.

Table 55: *Pin Descriptions*

Name	Type	Description
d	Input	Data input.
cn	Input	Active-low synchronous clear input. A low on cn sets the q output low upon the next rising edge of the clock if the clock enable is asserted high.
ce	Input	Active-high clock enable input.
ck	Input	Positive-edge clock input.
q	Output	Data output. The value present on the data input is transferred to the q output upon the rising edge of the clock if the clock enable input is high and the synchronous clear input is high.

Table 56: *Function Table*

Inputs				Output
cn	ce	d	ck	q
X	0	X	X	Hold
0	1	X	↑	0
1	1	0	↑	0
1	1	1	↑	1

Instantiation Templates

Verilog

```
ACX_DFFEC #(
    .init    (1'b0)
) instance_name (
    .q        (user_out),
    .d        (user_din),
    .cn        (user_clear),
    .ce        (user_clock_enable),
    .ck        (user_clock)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
----------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_DFFEC
generic map (
    init     => '0'
)
port map (
    q         => user_out,
    d         => user_din,
    cn         => user_clear,
    ce         => user_clock_enable,
    ck         => user_clock
);
```

ACX_DFFEP (Positive Clock Edge D-Type Register With Clock Enable and Synchronous Preset)

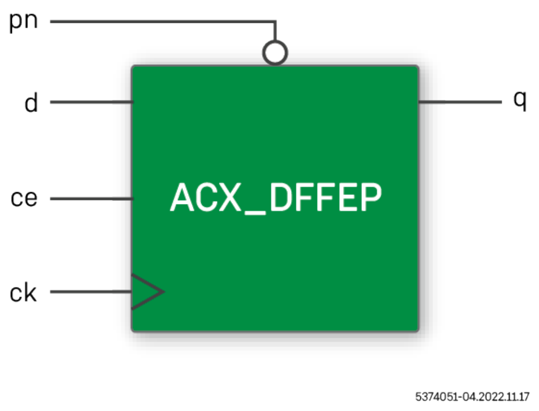


Figure 28: ***Positive Clock Edge D-Type Register With Clock Enable and Synchronous Preset***

ACX_DFFEP is a single D-type register with data input (`d`), clock enable (`ce`), clock (`ck`), and active-low synchronous preset (`pn`) inputs and data (`q`) output. The active-low synchronous preset input sets the data output high upon the next rising edge of the clock if it is asserted low and the clock enable signal is asserted high. If the synchronous preset input is not asserted, the data output is set to the value on the data input upon the next rising edge of the clock if the active-high clock enable input is asserted.

Table 57: ***Parameters***

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b1`	The `init` parameter defines the initial value of the output of the DFFEP register. This is the value the register takes upon the initial application of power to the FPGA.

Table 58: ***Pin Descriptions***

Name	Type	Description
d	Input	Data input.
pn	Input	Active-low synchronous preset input. A low on pn sets the q output high upon the next rising edge of the clock if the clock enable is asserted high.
ce	Input	Active-high clock enable input.
ck	Input	Positive-edge clock input.
q	Output	Data output. The value present on the data input is transferred to the q output upon the rising edge of the clock if the clock enable input is high and the synchronous preset input is high.

Table 59: ***Function Table***

Inputs				Output
pn	**ce**	**d**	**ck**	**q**
X	0	X	X	Hold
0	1	X	↑	1
1	1	0	↑	0
1	1	1	↑	1

Instantiation Templates

Verilog

```
ACX_DFFEP #(
    .init    (1'b1)
) instance_name (
    .q        (user_out),
    .d        (user_din),
    .pn        (user_preset),
    .ce        (user_clock_enable),
    .ck        (user_clock)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
----------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_DFFEP
generic map (
    init     => '1'
)
port map (
    q         => user_out,
    d         => user_din,
    pn         => user_preset,
    ce         => user_clock_enable,
    ck         => user_clock
);
```

ACX_DFFNEC (Negative Clock Edge D-Type Register With Clock Enable and Synchronous Clear)

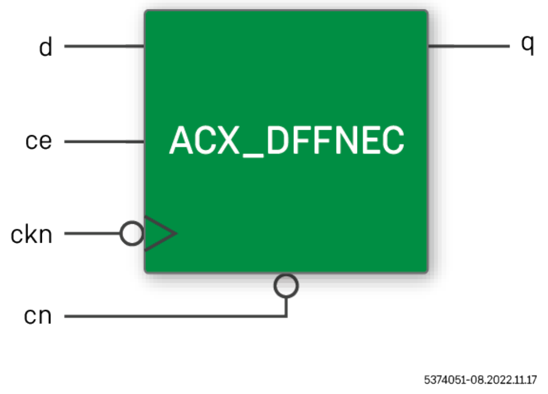


Figure 29: ***Negative Clock Edge D-Type Register With Clock Enable and Synchronous Clear***

ACX_DFFNEC is a single D-type register with data input (`d`), clock enable (`ce`), clock (`ckn`), and active-low synchronous clear (`cn`) inputs and data (`q`) output. The active-low synchronous clear input sets the data output low upon the next falling edge of the clock if it is asserted low and the clock enable signal is asserted high. If the synchronous clear input is not asserted, the data output is set to the value on the data input upon the next falling edge of the clock if the active-high clock enable input is asserted.

Table 60: ***Parameters***

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b0`	The `init` parameter defines the initial value of the output of the DFFNEC register. This is the value the register takes upon the initial application of power to the FPGA.

Table 61: ***Pin Descriptions***

Name	Type	Description
d	Input	Data input.
cn	Input	Active-low synchronous clear input. A low on cn sets the q output low upon the next falling edge of the clock if the clock enable is asserted high.
ce	Input	Active-high clock enable input.
ckn	Input	Negative-edge clock input.
q	Output	Data output. The value present on the data input is transferred to the q output upon the falling edge of the clock if the clock enable input is high and the synchronous clear input is high.

Table 62: ***Function Table***

Inputs				Output
cn	**ce**	**d**	**ckn**	**q**
X	0	X	X	Hold
0	1	X	↓	0
1	1	0	↓	0
1	1	1	↓	1

Instantiation Templates

Verilog

```
ACX_DFFNEC #(
    .init    (1'b0)
) instance_name (
    .q        (user_out),
    .d        (user_din),
    .cn        (user_clear),
    .ce        (user_clock_enable),
    .ckn    (user_clock)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
----------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_DFFNEC
generic map (
    init     => '0'
)
port map (
    q         => user_out,
    d         => user_din,
    cn         => user_clear,
    ce         => user_clock_enable,
    ckn     => user_clock
);
```

Boundary Pin Cells

In Speedcore devices, boundary pins provide the mechanism for routing signals between the core logic and surrounding ASIC logic. Boundary pins are directional buffers, with optional flip-flops. Boundary pins do not support enables nor bidirectional I/O. Boundary pins can either be handled automatically as part of the user software design flow or instantiated directly in the design.

IPIN (Input Data Pin)

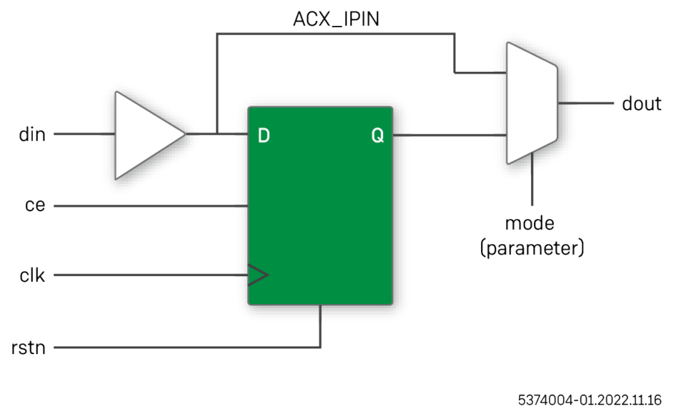


Figure 30: *ACX_IPIN Logic Diagram*

ACX_IPIN is an input boundary pin with a bypass-capable flip-flop, which supports data, reset, and enable signals. Clock signals must use ACX_CLK_IPIN. Set the `mode` parameter to `0` to use combinational mode, or to `1` to use flopped mode.

Table 63: *Ports*

Name	Type	Description
`din`	Input	Data input.
`clk`	Input	Clock input. Used only in flopped mode (`mode == 1`).
`ce`	Input	Active-high clock enable input.
`rstn`	Input	Active-low asynchronous/synchronous reset input. A low on `rstn` sets the `dout` output to the value of the `rst_value` parameter independent of the other inputs if the `sr_assertion` parameter is set to `unclocked`. If the `sr_assertion` parameter is set to `clocked`, a low on `rstn` sets the `dout` output to the value of the `rst_value` parameter at the next rising edge of the clock.
`dout`	Output	Data output.

Table 64: *Parameters*

Parameter	Defined Values	Default Value	Description
`mode`	`0, 1`	`0`	A value of `0` selects combinational mode for the IPIN, and the `clk` pin connection is ignored. A value of `1` selects flopped mode for the IPIN, and the `clk` pin must be connected to a valid clock.
`init`	`0, 1`	`0`	The initial value of the flop if mode is set to `1`.
`sr_assertion`	`unclocked, clocked`	`unclocked`	The `sr_assertion` parameter defines the behavior of the output when the `rstn` reset input is asserted. Assigning the sr_assertion to `unclocked` results in an asynchronous assertion of the reset signal, where the `dout` output is set to the value of the `rst_value` parameter upon assertion of the active-low reset signal. Assigning the `sr_assertion` to `clocked` results in a synchronous assertion of the reset signal, where the `dout` output is set to the value of the `rst_value` parameter at the next rising edge of the clock. The default value of the `sr_assertion` parameter is `unclocked`".
`rst_value`	`0, 1`	`0`	The `rst_value` parameter defines the value (`1` or `0`) that is output on the `dout` pin when reset is asserted.
`location`	`<pin_location>`	""	An optional parameter that can be used to place the instance on a site in the target device. It is recommended to use PDC constraints for placement instead of this parameter.

Table 65: *ACX_IPIN Function Table When sr_assertion = "unclocked"*

Inputs				Output
rstn	**ce**	**din**	**clk**	**dout**
0	X	X	X	`rst_value`
1	0	X	X	Hold
1	1	0	↑	0
1	1	1	↑	1

Table 66: *ACX_IPIN Function Table When sr_assertion = "clocked"*

Inputs				Output
rstn	ce	din	clk	dout
0	X	X	↑	`rst_value`
1	0	X	X	Hold
1	1	0	↑	0
1	1	1	↑	1

Instantiation Templates

Verilog – Combinational Mode

```
ACX_IPIN #(
    .mode            (0),
    .init            (0),
    .sr_assertion    ("unclocked"),
    .rst_value        (0),
    .location        ("")
) instance_name (
    .din            (user_pad),
    .clk            (),
    .ce                (),
    .rstn            (),
    .dout            (user_dout)
);
```

Verilog – Flopped Mode

```
ACX_IPIN #(
    .mode            (1),
    .init            (0),
    .sr_assertion    ("unclocked"),
    .rst_value        (0),
    .location        ("")
) instance_name (
    .din            (user_pad),
    .clk            (user_clk),
    .ce                (user_ce),
    .rstn            (user_rstn),
    .dout            (user_dout)
);
```

VHDL – Combinational Mode

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.io.all;
------------- DONE ACHRONIX LIBRARY -------

-- Component Instantiation
instance_name : ACX_IPIN
generic map (
    mode             => 0,
    init             => 0,
    sr_assertion     => "unclocked",
    rst_value         => 0,
    location         => ""
)
port map (
    din             => user_pad,
    dout             => user_dout
);
```

VHDL – Flopped Mode

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.io.all;
------------- DONE ACHRONIX LIBRARY -------

-- Component Instantiation
instance_name : ACX_IPIN
generic map (
    mode             => 1,
    init             => 0,
    sr_assertion     => "unclocked",
    rst_value         => 0,
    location         => ""
)
port map (
    din             => user_pad,
    clk              => user_clk,
    ce               => user_ce,
    rstn             => user_rstn,
    dout             => user_dout
);
```

ACX_OPIN (Output Data Pin)

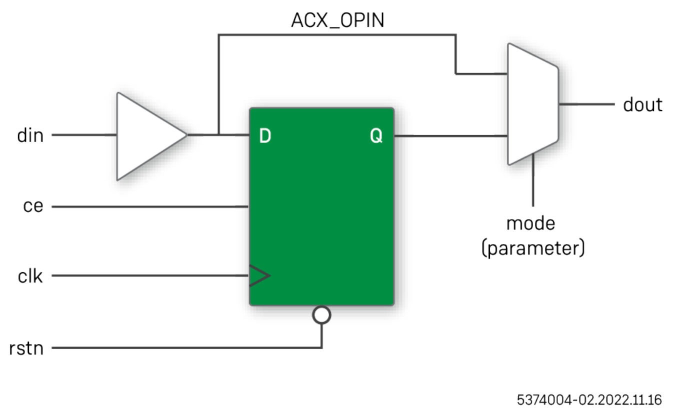


Figure 31: *ACX_OPIN Logic Diagram*

ACX_OPIN is an output boundary pin with a bypass-capable flip-flop, which supports data, reset, and enable signals. Clock signals must use ACX_CLK_OPIN. Set the `mode` parameter to `0` to use combinational mode, or to `1` to use flopped mode.

Table 67: *Ports*

Name	Type	Description
`din`	Input	Data input.
`clk`	Input	Clock input. Used only in flopped mode (`mode == 1`).
`ce`	Input	Active-high clock enable input.
`rstn`	Input	Active-low asynchronous/synchronous reset input. A low on `rstn` sets the `dout` output to the value of the `rst_value` parameter independent of the other inputs if the `sr_assertion` parameter is set to `unclocked`. If the `sr_assertion` parameter is set to `clocked`", a low on `rstn` sets the `dout` output to the value of the `rst_value` parameter at the next rising edge of the clock.
`dout`	Output	Data output.

Table 68: *Parameters*

Parameter	Defined Values	Default Value	Description
`mode`	`0, 1`	`0`	A value of `0` selects combinational mode for the OPIN, and the `clk` pin connection is ignored. A value of `1` selects flopped mode for the OPIN, and the `clk` pin must be connected to a valid clock.
`init`	`0, 1`	`0`	The initial value of the flop if `mode` is set to `1`.
`sr_assertion`	`unclocked, clocked`	`unclocked`	The `sr_assertion` parameter defines the behavior of the output when the `rstn` reset input is asserted. setting `sr_assertion` to `unclocked` results in an asynchronous assertion of the reset signal, where the `dout` output is set to the value of the `rst_value` parameter upon assertion of the active-low reset signal. Setting `sr_assertion` to `clocked` results in a synchronous assertion of the reset signal, where the `dout` output is set to the value of the `rst_value` parameter at the next rising edge of the clock. The default value of the `sr_assertion` parameter is `unclocked`.
`rst_value`	`0, 1`	`0`	The `rst_value` parameter defines the value (`1` or `0`) that is output on the `dout` pin when reset is asserted.
`location`	`<pin_location>`	""	An optional parameter that can be used to place the instance on a site in the target device. It is recommended to use PDC directives for placement instead of this parameter.

Table 69: *ACX_OPIN Function Table When sr_assertion = "unclocked"*

Inputs				Output
rstn	**ce**	**din**	**clk**	**dout**
0	X	X	X	`rst_value`
1	0	X	X	Hold
1	1	0	↑	0
1	1	1	↑	1

Table 70: *ACX_OPIN Function Table When sr_assertion = "clocked"*

Inputs				Output
rstn	**ce**	**din**	**clk**	**dout**
0	X	X	↑	`rst_value`
1	0	X	X	Hold
1	1	0	↑	0
1	1	1	↑	1

Instantiation Templates

Verilog – Combinational Mode

```
ACX_OPIN #(
    .mode            (0),
    .init            (0),
    .sr_assertion    ("unclocked"),
    .rst_value        (0),
    .location        ("")
) instance_name (
    .din            (user_din),
    .clk            (),
    .ce                (),
    .rstn            (),
    .dout            (user_pad)
);
```

Verilog – Flopped Mode

```
ACX_OPIN #(
    .mode            (1),
    .init            (0),
    .sr_assertion    ("unclocked"),
    .rst_value        (0),
    .location        ("")
) instance_name (
    .din            (user_din),
    .clk            (user_clk),
    .ce                (user_ce),
    .rstn            (user_ce),
    .dout            (user_pad)
);
```

VHDL – Combinational Mode

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.io.all;
------------- DONE ACHRONIX LIBRARY -------

-- Component Instantiation
instance_name : ACX_OPIN
generic map (
    mode             => 0,
    init             => 0,
    sr_assertion     => "unclocked",
    rst_value         => 0,
    location         => ""
)
port map (
    din             => user_din,
    dout             => user_pad
);
```

VHDL – Flopped Mode

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.io.all;
------------- DONE ACHRONIX LIBRARY -------

-- Component Instantiation
instance_name : ACX_OPIN
generic map (
    mode             => 1,
    init             => 0,
    sr_assertion     => "unclocked",
    rst_value         => 0,
    location         => ""
)
port map (
    din             => user_din,
    clk             => user_clk,
    ce               => user_ce,
    rstn             => user_rstn,
    dout             => user_pad
);
```

ACX_CLK_IPIN (Input Clock Pin)

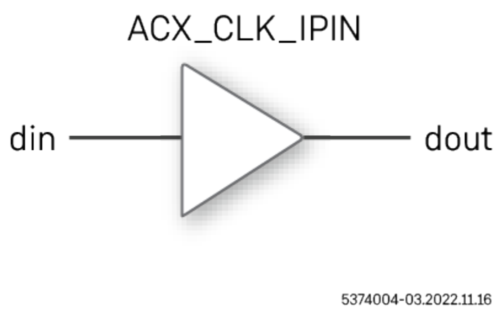


Figure 32: *ACX_CLK_IPIN Logic Symbol*

ACX_CLK_IPIN is an input boundary pin which supports only clock signals. Data and reset signals must use IPIN.

Table 71: *Ports*

Name	Type	Description
`din`	Input	Clock input.
`dout`	Output	Clock output.

Table 72: *Parameters*

Parameter	Defined Values	Default Value
`location`	`<pin_location>`	""

Table 73: *Input Function Table*

din	dout
0	0
1	1
X	X
Z	X

Instantiation Templates

Verilog

```
ACX_CLK_IPIN #(
    .location    ("")
) instance_name (
    .din        (user_pad),
    .dout        (user_clkout)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.io.all;
------------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_CLK_IPIN
generic map (
    location     => ""
)
port map (
    din         => user_pad,
    dout         => user_clkout
);
```

ACX_CLK_OPIN (Output Clock Pin)

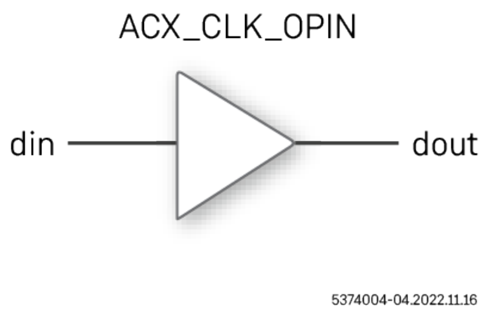


Figure 33: ***ACX_CLK_OPIN Logic Symbol***

ACX_CLK_IPIN is an output boundary pin which supports only clock signals. Data and reset signals must use OPIN.

Table 74: ***Ports***

Name	Type	Description
`din`	Input	Clock input.
`dout`	Output	Clock output.

Table 75: ***Parameters***

Parameter	Defined Values	Default Value
`location`	`<pin_location>`	""

Table 76: ***Input Function Table***

din	dout
0	0
1	1
X	X
Z	X

Instantiation Templates

Verilog

```
ACX_CLK_OPIN #(
    .location    ("")
) instance_name (
    .din        (user_clkin),
    .dout        (user_pad)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.io.all;
------------- DONE ACHRONIX LIBRARY ---------

-- Component Instantiation
instance_name : ACX_CLK_OPIN
generic map (
    location     => ""
)
port map (
    din         => user_clkin,
    dout         => user_pad
);
```

Chapter - 3: Logic Functions

ACX_SYNCHRONIZER, ACX_SYNCHRONIZER_N

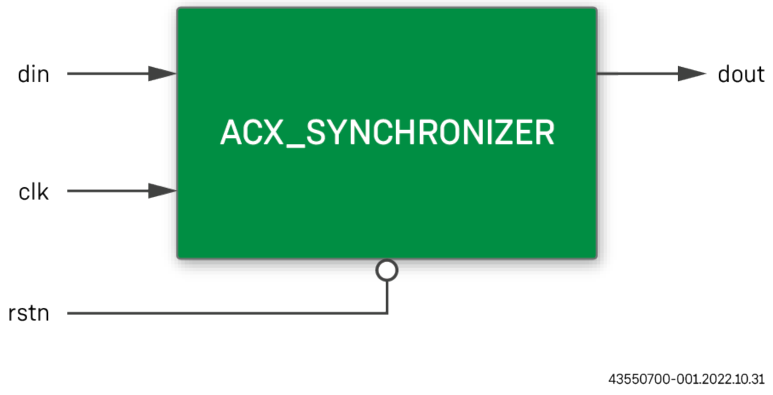


Figure 34: *ACX_SYNCHRONIZER Logic Symbol*

ACX_SYNCHRONIZER implements a data synchronizer to reduce the frequency of metastability when sampling data synchronous to one clock domain with a register clocked by another clock domain. It is strongly recommended that this macro be used for control signals that cross clock domains. Using this macro has several advantages over using a two-register synchronizer:

The ACX_SYNCHRONIZER macro uses two back-to-back registers and improves the mean time between failures (MTBF) by including ACE pragmas (and SDC) that constrain the placement of the registers relative to one another. When constructing a synchronizer from two registers (not recommended), there is a chance that the tool might separate the flip-flops within the fabric.

Embedded ACE and SDC constraints in the ACX_SYNCHRONIZER macro ensure that:

- All timing paths through the `din` input are disabled, while the `rstn` input paths are not disabled. When constructing a synchronizer from two registers (not recommended), manually add constraints to disable these paths. If such a path is timed, the tool may report false critical paths resulting in longer run-times.
- The two registers in the macro are not cloned or duplicated by the tools.

ACX_SYNCHRONIZER_N is identical to ACX_SYNCHRONIZER, except that it synchronizes to the falling edge of the reference clock instead of the rising edge.

Table 77: *Parameters*

Parameter	Defined Values	Default Value	Description
`init`	`1'b0`, `1'b1`	`1'b0`	The `init` parameter defines the initial value of the output of the synchronizer and of the intermediate register, whose results are seen after the first rising clock edge after reset. This setting is also the value that the synchronizer takes upon the initial application of power to the FPGA.

Table 78: *Pin Descriptions*

Name	Type	Clock Domain	Description
`rstn`	Input	–	Active-low reset input. Resets the value of the output register and the intermediate register to the value provided by the `init` parameter.
`din`	Input	–	Data input.
`clk`	Input		Clock reference. The `dout` signal is synchronized to the rising edge of this clock.
`dout`	Output	`clk`	Data output.

Table 79: *Function Table*

Inputs		Output
din	clk	dout
0	↑↑	0
1	↑↑	1

Using ACX_SYNCHRONIZER to Synchronize Reset

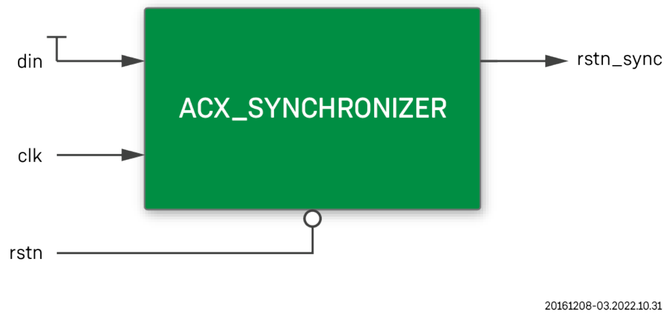


Figure 35: *ACX_SYNCHRONIZER Synchronizing Reset*

An instance of the ACX_SYNCHRONIZER module can also be used to synchronize reset signals. In this case, the active-low non-synchronous reset input is connected to the `rstn` input of the ACX_SYNCHRONIZER module, the `din` input is driven with `1'b1`, and the `init` parameter is set to `1'b0`. When the `rstn` input is asserted, the output is immediately driven to a value of `1'b0` (as determined by the `init` parameter). The `1'b1` on the data input propagates to the output after two output clock cycles; after which, when `rstn` is de-asserted, the output is set to `1'b1` on the next rising edge of `clk`.

Instantiation Templates

Verilog

```
ACX_SYNCHRONIZER #(
   .init    (1'b0)
) instance_name (
   .clk     (user_output_clock),
   .rstn    (user_reset_n),
   .din     (user_din),
   .dout    (user_out)
);
```

ACX_SHIFTREG

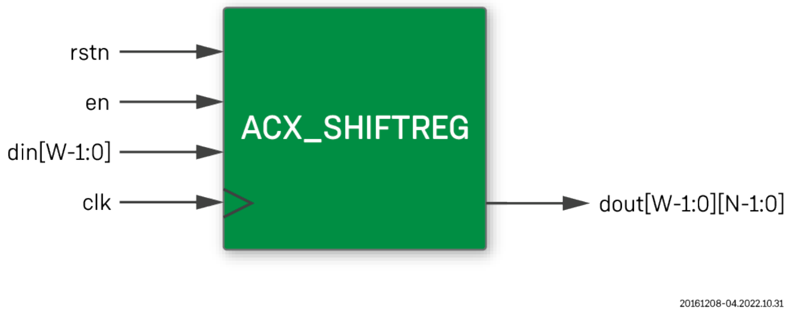


Figure 36: *ACX_SHIFTREG Logic Symbol*

ACX_SHIFTREG is a macro that provides an efficient multi-tap shift register implementation using LRAMs, with a configurable data width and delay for each tap. On each rising clock edge, the data at the `din` input pins is captured and saved by the shift register. The data is then presented on `dout[n]` after the number of cycles of delay assigned to tap *n*. De-asserting the `en` input pauses operation of the shift register, such that the data present on the input pins is not captured by the shift register, and the output does not change. For example, if the shift register is configured to have three taps, and the delays for the taps are 2, 5, and 7, then the data sampled at the input to the shift register on a given clock cycle is available at `dout[0]` after two clock cycles, at `dout[1]` after five clock cycles, and at `dout[2]` after seven clock cycles. To use this macro, include the following in the Verilog source code that instantiates the ACX_SHIFTREG macro:

```
`include "speedster7t/macros/ACX_SHIFTREG.v"
```

The shift register implementation optionally uses both edges of the clock, allowing for two taps per LRAM instance. This implementation reduces the number of LRAMs used at the expense of timing closure at higher clock frequencies.

Table 80: *Parameters*

Parameter	Defined Values	Default Value	Description
`W`	`<int>`	32	The width of the `din[]` signal, `dout[]` signals, and internal data storage.
`N`	`<int>`	1	The number of taps supported by the shift register.
`TAPS`	`[<int>]`		Array of tap latencies. The nth entry in the `TAPS` array specifies the latency of the nth tap, as seen on the `dout[n]` signals. Individual latencies are measured from `din`, each value in the array must be larger than the previous value.
`MODE`	`[0,1]`	`[0, 0, …]`	Array of modes. Setting the nth entry in the `MODE` array to `1'b1` allows that entry to be implemented using an LRAM with both rising and falling clock edges. A mode of `1'b0` uses only the rising clock edge.

Table 81: ***Pin Descriptions***

Name	Type	Description
`clk`	Input	Clock reference. All inputs and outputs are relative to the rising edge of this clock. Depending on the implementation mode, internal logic may use the falling edge of this clock.
`rstn`	Input	Active-low reset. When asserted, the value of the internal data registers are reset to `0`. Using this signal prevents the shift register data storage from being mapped to LRAMs, and the shift register is built out of core registers.
`en`	Input	Active-high clock enable. De-asserting this signals stops operation of the shift register.
`din[(W-1):0]`	Input	Data input.
`dout[(W-1):0][(N-1):0]`	Output	An array of data outputs, where `dout[(W-1):0][0]` carries the data out from the first tap, and `dout[(W-1):0][N-1]` represents the data out from the last tap.

Table 82: ***Function Table***

Inputs			Output
rstn	**en**	**clk**	**dout[n].**
0	X	X	0 (and resets all internal states).
1	0	↑	Previous `dout[n]`.
1	1	↑	`dout[n]` gets the next data element in the shift register.

Instantiation Templates

Verilog

```
ACX_SHIFTREG #(
    .W       (32),           // Data is 32 bit wide
     .N       (3),            // 3 taps
     .TAPS    ([3, 5, 7]),    // Taps at 3 cycles, 5 cycles, and 7 cycles
     .MODE    ([0,0,0])       // Rising clock edge only.
) instance_name (
     .clk     (user_clock),
    .rstn    (user_reset_n),
     .en      (user_en),
     .din     (user_din),
     .dout    (user_out_array)
);
```

Chapter - 4: Clock Functions

ACX_CLKDIV (Clock Divider)

The ACX_CLKDIV component implements a clock divider to provide an output clock at 1/2, 1/4, 1/6, or 1/8 the frequency of the input clock with a configurable offset.

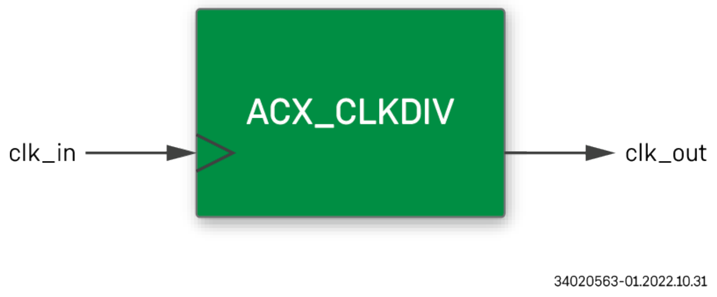


Figure 37: *ACX_CLKDIV Logic Symbol*

Table 83: *Parameters*

Parameter	Defined Values	Default Value	Description
`div_by`	`2, 4, 6, 8`	`2`	Determines the factor by which the input clock is divided.
`offset`	`0, 1, 2, 3`	`0`	Defines the number of input clock cycles by which to delay the output clock.

Table 84: *Pin Descriptions*

Name	Type	Description
`clk_in`(1)	Input	Input clock to be divided.
`clk_out`(1)	Output	Divided clock output.

Table Notes

1. Both `clk_in` and `clk_out` must connect to clock tracks within the FPGA. They cannot connect directly with data tracks.

The following timing diagram shows how the `div_by` and `offset` parameters affect the output clock.

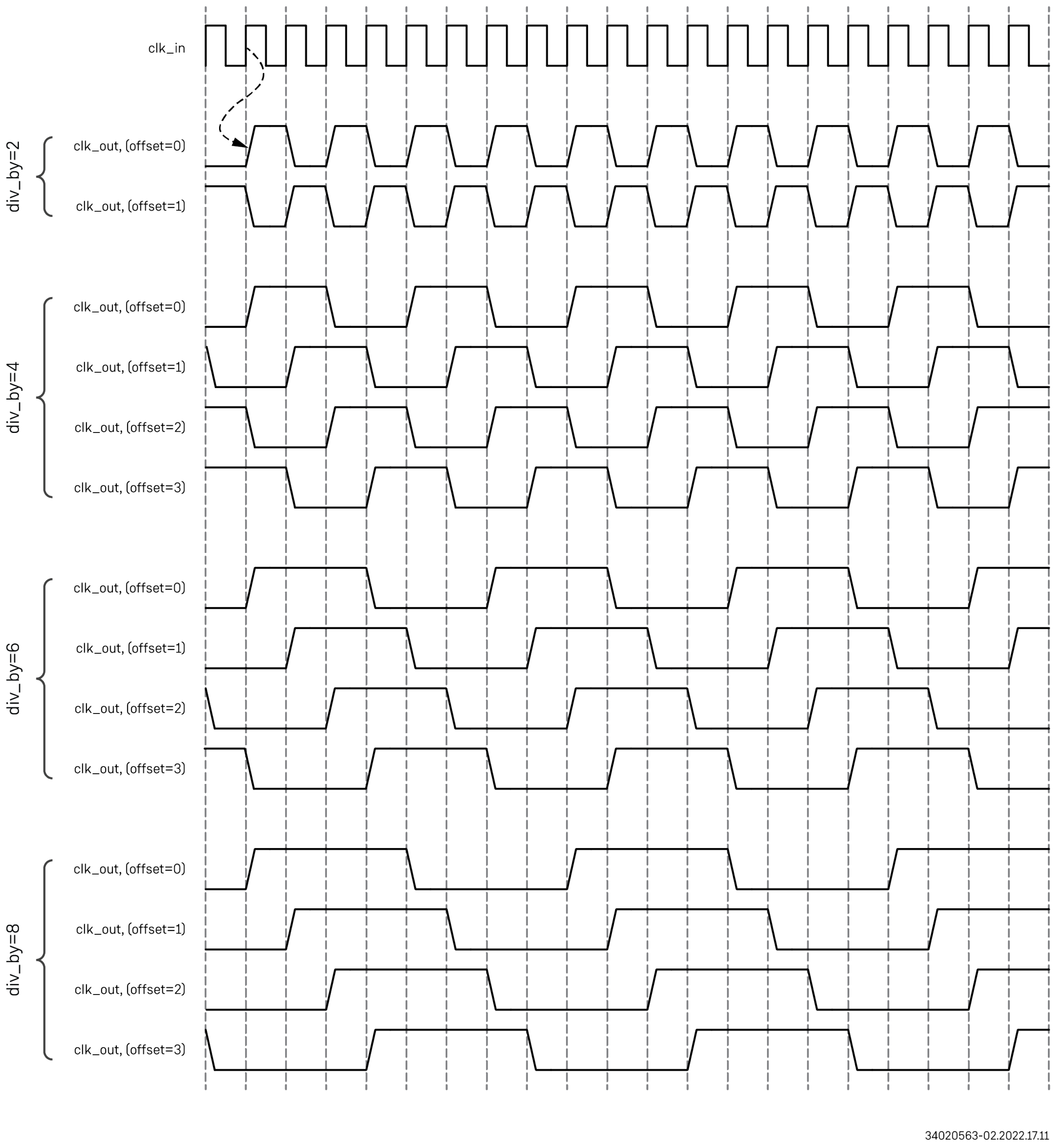


Figure 38: *Output Clock Timing Diagram*

Constraints

The ACX_CLKDIV component does *not* propagate the input clock frequency from `clk_in` to `clk_out`. Therefore, suitable constraints for `clk_out` must be specified to ensure that correct timing is applied. These constraints should be present in both the Synplify Pro and ACE constraint files.

```
# Example of constraint required with divide by 2, and offset of 0. Input clock is from the port
"i_clk_in". Output of divider connects to net named "clk_in_div_2"
create_generated_clock -name clk_div_2 -source [get_port i_clk_in] -divide_by 2 [get_nets
clk_in_div_2]
```

Instantiation Templates

Verilog

```
ACX_CLKDIV #(
   .div_by    ( 2 ),
   .offset     ( 1 )
) instance_name (
   .clk_in     (user_clk_in),
   .clk_out    (user_clk_out)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
------------- DONE ACHRONIX LIBRARY -------

-- Component Instantiation
instance_name : ACX_CLKDIV
generic map (
    div_by  => 2,
    offset  => 1
)
port map (
    clk_in  => user_clk_in,
    clk_out => user_clk_out
);
```

ACX_CLKGATE (Clock Gate)

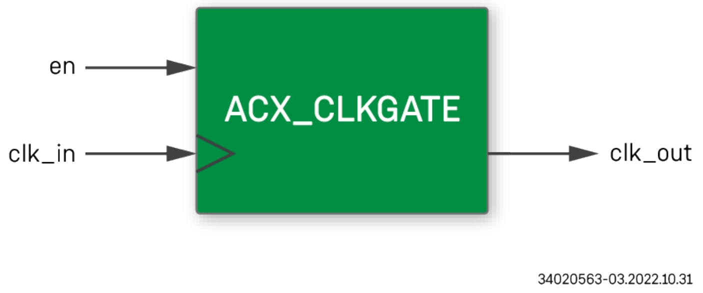


Figure 39: *ACX_CLKGATE Logic Symbol*

The ACX_CLKGATE component implements a clock gate that allows the output to toggle only when the input, `en` , is asserted high. This component disables the clock only after the clock input has transitioned low, guaranteeing that the output is glitchless. The output clock is guaranteed to never have a pulse width narrower in time than the input pulse width.

Note

When simulating the ACX_CLKGATE component, if the transition on the input signal, `en`, and the transition on the input clock arrive at the same moment, the time that it takes for the `en` transition to have an effect is dependent on how the simulator schedules events and may vary with different simulators, different designs, and different simulation models.

Table 85: *Pin Descriptions*

Name	Type	Description
`en`	Input	When asserted high, the `clk_out` output is driven by the `clk_in` input.
`clk_in`(1)	Input	Input clock to be gated.
`clk_out`(1)	Output	Gated clock output.

Table Notes

1. Both `clk_in` and `clk_out` must connect to clock tracks within the FPGA. They cannot connect directly with data tracks.

The following timing diagram illustrates the behavior of the ACX_CLKGATE component.

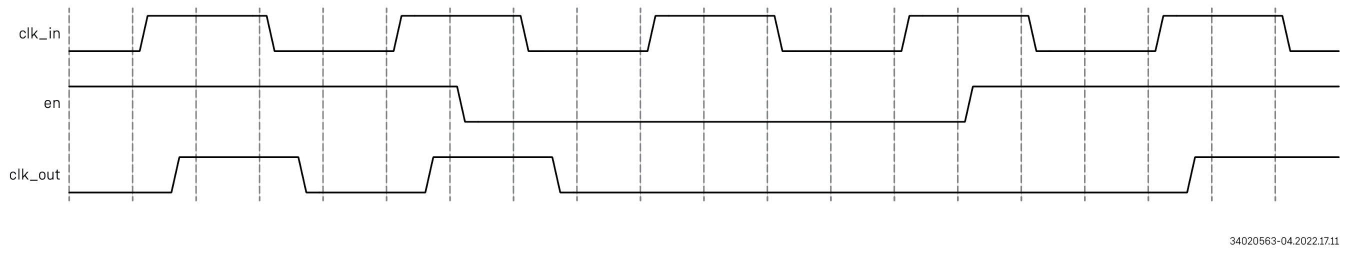


Figure 40: *ACX_CLKGATE Timing Diagram*

Constraints

The ACX_CLKGATE component does not propagate the input clock frequency from `clk_in` to `clk_out` in Synplify Pro. Therefore, it is necessary to specify additional constraints for `clk_out`. With the constraint, Synplify Pro correctly passes through the input clock domain to the output clock domain for static timing analysis purposes. ACE can propagate the input clock frequency through the ACX_CLKGATE to the output clock. The following constraint is needed by Synplify Pro.

```
# Example of defining a generated clock for ACX_CLKGATE. In this example, 'i_clkgate' is the
instance name of the ACX_CLKGATE and the input clock is 'i_clk'.
create_generated_clock -name clk_gate [ get_pins {i_clkgate/clk_out} ] -source  [get_ports
{i_clk} ] -divide_by 1
```

Instantiation Templates

Verilog

```
ACX_CLKGATE instance_name
(
   .en      (user_en),
   .clk_in  (user_clk_in),
   .clk_out (user_clk_out)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
------------- DONE ACHRONIX LIBRARY -------

-- Component Instantiation
instance_name : ACX_CLKGATE
port map (
    en      => user_en,
    clk_in  => user_clk_in,
    clk_out => user_clk_out
);
```

ACX_CLKSWITCH (Clock Switch)

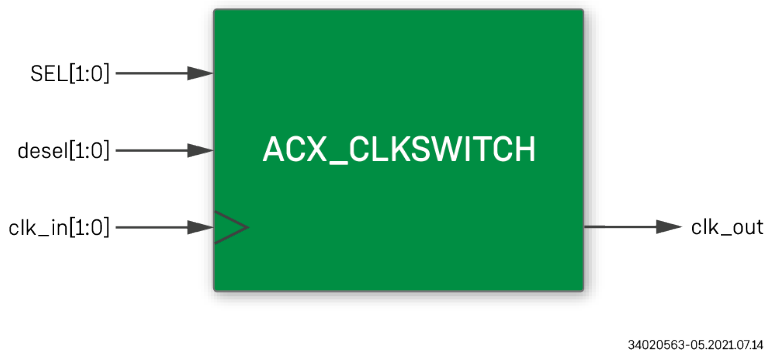


Figure 41: *ACX_CLKSWITCH Logic Symbol*

The ACX_CLKSWITCH component implements clock switching functionality allowing the output clock to be glitchlessly switched between two different clock inputs. The glitchless behavior is implemented by disabling the clock being switched *from* when that clock is at value `0`, and then enabling the clock being switched *to* when that clock has a value of `0`. In this way, the output clock never has a pulse that is narrower than the original clock or the new clock.

There are three switching behaviors depending on the value applied to the `SYNCHRONIZE_SEL` parameter:

- `0` – ensures the input, `sel[]`, for each clock is synchronized to the rising and then falling edge of the clock being selected
- `1` – synchronizes the input signal, `sel[]`, to the falling edge followed by the next falling edge of the clock being selected
- `2` – synchronizes `sel[]` to a single falling edge of the clock it is selecting (a value of `2` should only be used when the input signal, `sel[]` is synchronized to both `clk_in[0]` and `clk_in[1]`)

To ensure glitchless operation, set `SYNCHRONIZE_SEL` to the appropriate value to meet timing requirements and ensure that each bit of `sel[]` is synchronized to the clock that it is used to select.

If a clock is not toggling, then de-asserting the `sel[]` input bit for that clock does not deselect the clock. In this case, the `desel[]` input can be used to asynchronously force deselection of a clock input.

Note

When simulating the ACX_CLKSWITCH component, if the transition on the `sel[]` input signal and the transition on one of the input clocks arrive at the same moment, the time that it takes for the `sel[]` transition to have an effect is dependent on how the simulator schedules events and may vary with different simulators, different designs, and different simulation models. Using `desel[]` when the input clock is toggling can cause a glitch or partial pulse on the output.

Table 86: *Parameter Descriptions*

Parameter	Defined Values	Default Value	Description
`PRESEL`	`0, 1, 2`	`0`	Determines the operation of the CLKSWITCH at startup time to prevent the need for a clock switching event when the FPGA begins normal operation. The value should match the startup value of the input, `sel[1:0]`.
`SYNCHRONIZE_SEL`	`0, 1, 2`	`0`	Determines how many half-cycle or full cycle synchronization stages are used to synchronize the inputs, `sel[1:0]`: 0 – synchronizes the input, `sel[1:0]`, to the rising and then falling edge of the selected clock 1 – synchronizes the input, `sel[1:0]`, to two conscutive falling edges of the selected clock. 2 – synchronizes the input, `sel[1:0]`, to a single falling edge of the selected clock.

Table 87: *Pin Descriptions*

Name	Type	Description
`sel[1:0]`	Input	Assert `sel[0]` to drive the output clock from `clk_in[0]` and assert `sel[1]` to drive the output clock from `clk_in[1]`. If both bits of `sel[]` are de-asserted, the `clk_out` output stops toggling. Asserting both bits of `sel[]` at the same time results in unpredictable output.
`desel[1:0]`(1)	Input	When switching from one input clock to another clock using `sel[]`, the first clock is synchronously disabled before the second clock is enabled. If the first clock is not toggling, it can not be synchronously disabled. The `desel[]` input provides a mechanism for deselecting a clock that is not toggling. Asserting `desel[n]` asynchronously deselects `clk_in[n]`, allowing `clk_in[n]` to be deselected even when it is not toggling.
`clk_in[1:0]`(2)	Input	Input clocks.
`clk_out`(2)	Output	Output clock.

Table Notes

1. Using `desel[]` to deselect a clock while it is toggling can cause a glitch on the output clock
2. Both `clk_in[1:0]` and `clk_out` must connect to clock tracks within the FPGA. They cannot connect directly with data tracks.

The following timing diagrams show how the `SYNCHRONIZE_SEL` parameter affects the output clock.

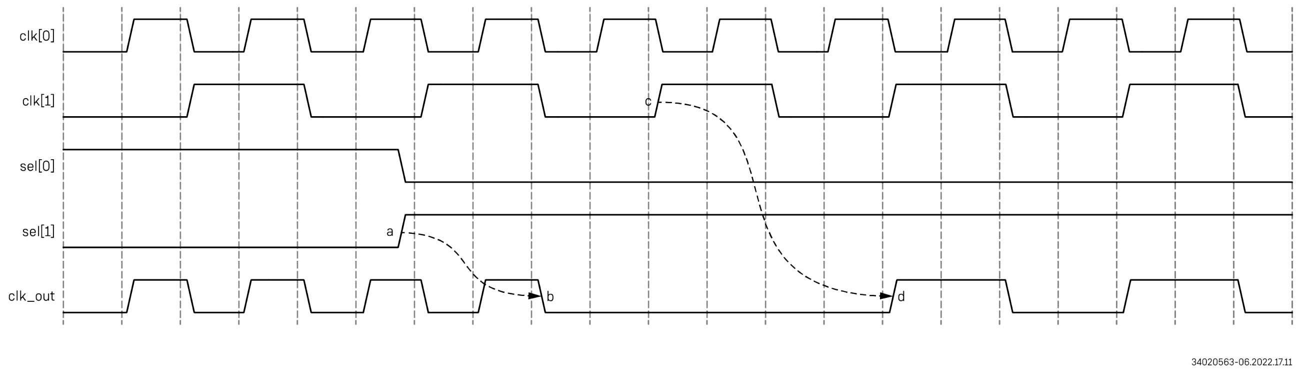


Figure 42: *SYNCHRONIZE_SEL = 0 Timing Diagram*

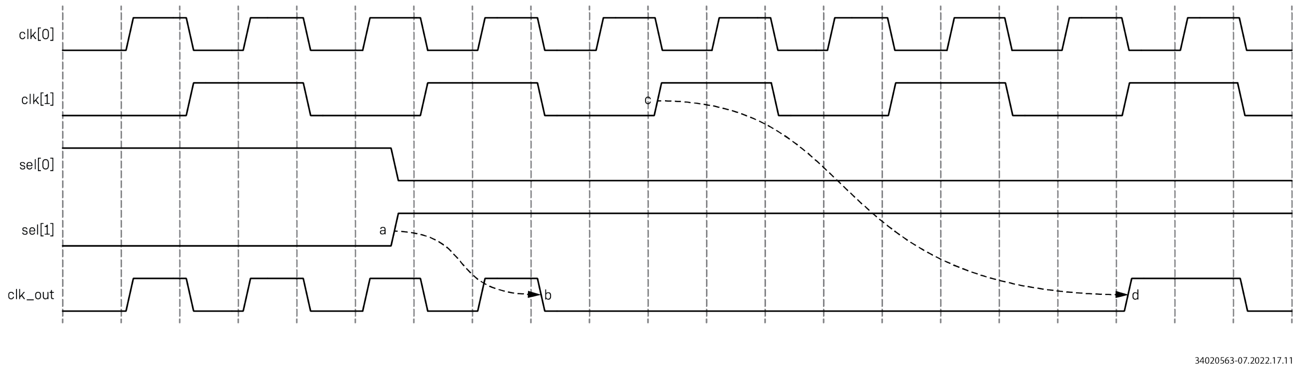


Figure 43: *SYNCHRONIZE_SEL = 1 Timing Diagram*

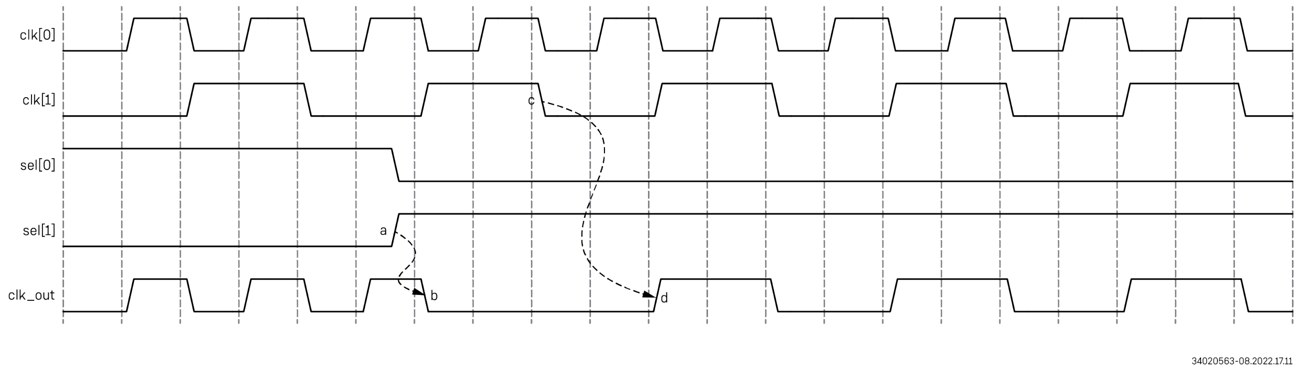


Figure 44: *SYNCHRONIZE_SEL = 2 Timing Diagram*

Constraints

The ACX_CLKSWITCH component does not propagate the input clock frequency from both `clk_in` ports to the `clk_out` port in Synplify Pro. Therefore, it is necessary to specify additional timing constraints for `clk_out`. With these added constraints, Synplify Pro correctly passes the input clock domains through to the output clock domain for static timing analysis purposes. ACE can propagate the input clock frequency through the ACX_CLKSWITCH to the output clock. The following constraints are needed by Synplify Pro.

```
# Example of defining a generated clock for ACX_CLKSWITCH. In this example, 'i_clkswitch' is the
instance name of the ACX_CLKSWITCH and the input clock is 'i_clk_0' and 'i_clk_1'.
create_generated_clock -name clk_switch0 [ get_pins {i_clkswitch/clk_out} ] -source  [get_ports
{i_clk_0} ] -divide_by 1
create_generated_clock -name clk_switch1 [ get_pins {i_clkswitch/clk_out} ] -add -master_clock
clk1 -source  [get_ports {i_clk_1} ] -divide_by 1
```

Instantiation Templates

Verilog

```
ACX_CLKSWITCH #(
   .SYNCHRONIZE_SEL ( 1 ),
   .PRESEL            ( 1 )
) instance_name (
   .sel             (user_sel),
   .desel           (user_desel),
   .clk_in          (user_clk_in),
   .clk_out         (user_clk_out)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
------------- DONE ACHRONIX LIBRARY -------

-- Component Instantiation
instance_name : ACX_CLKSWITCH
generic map (
    SYNCHRONIZE_SEL => 1 ,
    PRESEL          => 1
)
port map (
    sel             => user_sel,
    desel           => user_desel,
    clk_in          => user_clk_in,
    clk_out         => user_clk_out
);
```

Chapter - 5: Arithmetic and DSP Functions

ACX_ALU8

The ACX_ALU8 implements either an 8-bit adder or 8-bit subtractor.

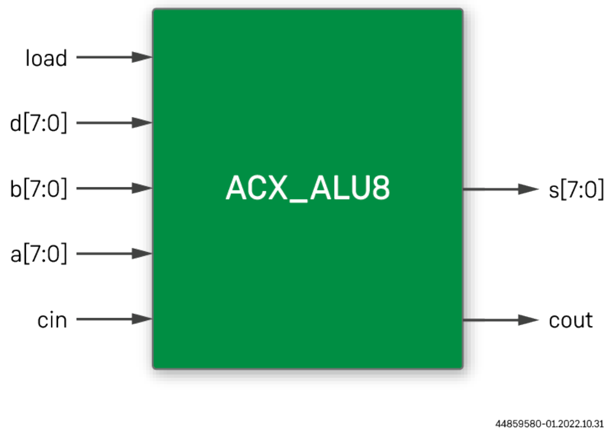


Figure 45: ***Eight-Input Adder/Subtractor With Programmable Load***

Description

The ACX_ALU8 has the following inputs:

- Adder/subtractor (`a[7:0]`, `b[7:0]`)
- Load value (`d[7:0]`)
- Load enable (`load`)
- Carry-in (`cin`)

The following outputs are generated:

- Sum/difference (`s[7:0]`)
- Carry-out (`cout`)

Asserting the load signal high assigns the `s[7:0]` output with the load value, `d[7:0]`, input.

Multiple ACX_ALU8 blocks may be combined by connecting the `cout` output of one slice to the `cin` input of the next significant eight-bit slice. Selection of whether the ACX_ALU8 is configured as an adder or subtractor is determined by the value of the `invert_b` parameter.

Note

When chaining the output, `cout`, of one ACX_ALU8 to the input, `cin`, of another ACX_ALU8, special routing details should be understood. Refer to the figure showing routing between RLBs in the section on Speedcore Fabric Architecture (see page 11).

Parameters

Table 88: *Parameters*

Parameter	Defined Values	Default Value	Description
`invert_b`	`1'b0`, `1'b1`	`1'b0`	The `invert_b` parameter determines whether the ACX_ALU8 functions as an adder or a subtractor: `1'b0` – the ACX_ALU8 performs two's complement addition of `a[7:0] + b[7:0] + cin`. `1'b1` – the ACX_ALU8 inverts the `b[7:0]` input so that two's complement subtraction of `a[7:0] – b[7:0]` can be performed. When subtraction is desired, the `cin` input must be connected to `1'b1`. With the input `b[7:0]` inverted and `cin` set to `1'b1`, in two's compliment arithmetic, this creates the value `–b`. When multiple `ACX_ALU8`s are connected to perform higher resolution subtractors, only the `cin` of the LSB of the subtractor is to be connected to `1'b1`, all other `cin` inputs must be set to `1'b0`.

Ports

Table 89: *Pin Descriptions*

Name	Type	Description
`a[7:0]`	Input	Data input a. An 8-bit two's complement signed input, where bit 7 is the most significant bit. In subtraction mode, data input a is the minuend.
`b[7:0]`	Input	Data input b. An 8-bit two's complement signed input, where bit 7 is the most significant bit. In subtraction mode, data input b is the subtrahend.
`d[7:0]`	Input	Load value input. Input `d[7:0]` is loaded onto the outputs `s[7:0]` upon the active-high assertion of the load input.
`load`	Input	Load input (active-high). Asserting the `load` input sets the `s[7:0]` output equal to the `d[7:0]` input.
`cin`	Input	Carry-In input (active-high). The `cin` is the carry-in to the ALU8. For subtraction, `cin` should be tied high.
`s[7:0]`	Output	Sum/difference output. If the `invert_b` parameter is set to `1'b0` and the `load` input is low, the `s[7:0]` output reflects the sum of the a, b, and `cin` inputs. If the `invert_b` parameter is set to `1'b1` and the `load` input is low, the `s[7:0]` output reflects the difference of the a, b, and `cin` inputs.
`cout`	Output	Carry-out output. The `cout` is set high during an add when the `s[7:0]` output overflows.

Functions

Table 90: *Function Table With invert_b = 1'b0*

load	cin	s[3:0]	Note
1	X	`d[7:0]`	Load.
0	–	`a[7:0] + b[7:0] + cin`	Add.

Table 91: *Function Table With invert_b = 1'b1*

load	cin	s[7:0]	Note
1	X	`d[7:0]`	Load.
0	1	`a[7:0] – b[7:0]`	Subtract.
0	0	`a[7:0] – b[7:0] – 1`	Subtract –1.

Instantiation Template

Verilog

```
ACX_ALU8 #(
    .invert_b    (1'b0)
) instance_name (
    .a            (user_a),
    .b            (user_b),
    .d            (user_load_value),
    .load        (user_load),
    .cin        (user_carry_in),
    .s            (user_sum),
    .cout        (user_cout)
);
```

ACX_DSP_GEN

The ACX_DSP_GEN block is optimized for fixed-point digital signal processing (DSP). Columns of ACX_DSP_GEN blocks reside within the Achronix embedded FPGA core to aid in the efficient implementation of blocks such as FIR filters and processing of wireless signals.

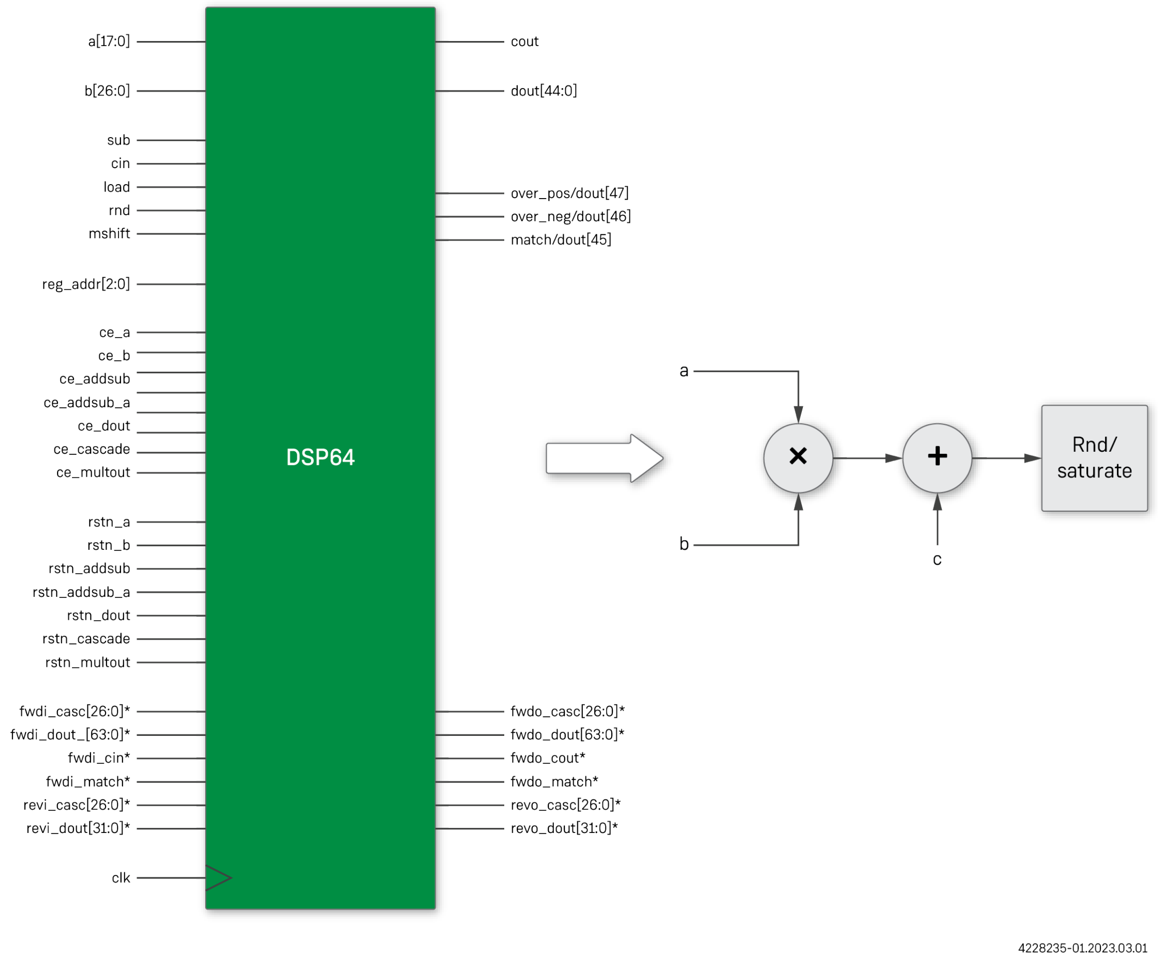


Figure 46: *ACX_DSP_GEN Tile Logic Symbol*

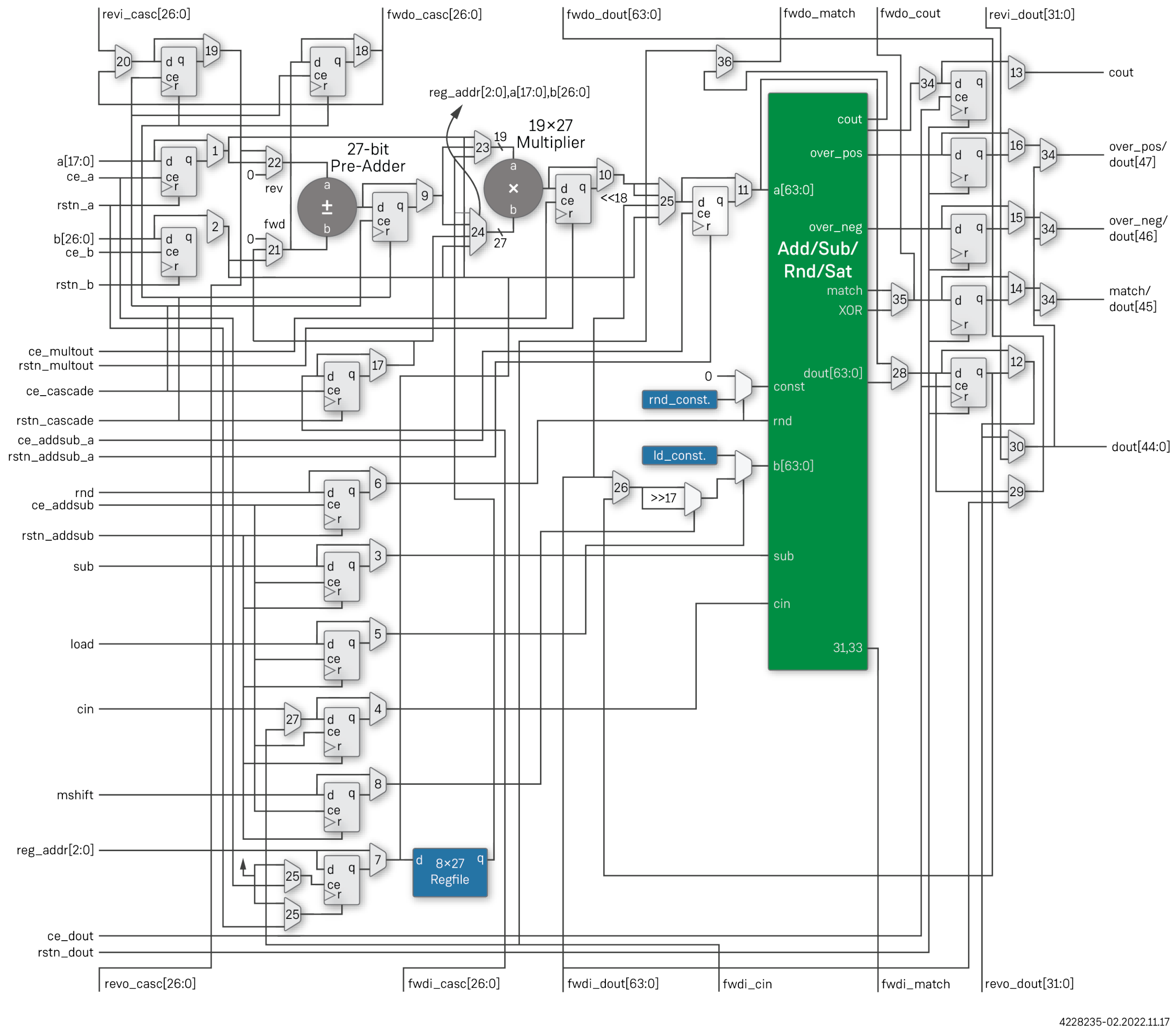


Figure 47: *ACX_DSP_GEN Block Diagram*

Note

The signals, `fwdi_*`, `fwdo_*`, `revi_*`, and `revo_*` at the top and bottom of the ACX_DSP_GEN Block Diagram denote hardwired connections to/from adjacent ACX_DSP_GEN tiles

Table 92: ***Control Parameter References Index Numbers for the ACX_DSP_GEN Block Diagram***

Index	Parameter	Index	Parameter	Index	Parameter
1	`a_del`	13	`cout_del`	25	`sel_addsub_a`
2	`b_del`	14	`match_del`	26	`sel_addsub_b`
3	`sub_del`	15	`over_neg_del`	27	`sel_cin`
4	`cin_del`	16	`over_pos_del`	28	`addsub_bypass`
5	`load_del`	17	`fwdi_casc_del`	29	`sel_fwdo_dout`
6	`rnd_del`	18	`fwdo_casc_del`	30	`sel_dout`
7	`regaddr_del`	19	`revi_casc_del`	31	`round_mode`
8	`mshift_del`	20	`sel_revi_casc`	32	`preadd_mode`
9	`preadd_del`	21	`sel_fwd_preadd`	33	`sat_mode`
10	`multout_del`	22	`sel_rev_preadd`	34	`sel_48_dout`
11	`addsub_areg_del`	23	`sel_mult_a`	35	`&match_pattern & ~(|match_mask)`
12	`dout_del`	24	`sel_mult_b`	36	`sel_fwdo_cout`

ACX_DSP_GEN Pins

Table 93: *ACX_DSP_GEN Pin Descriptions*

Name	Type	Description
`a[17:0]`	Input	Data input A, an 18-bit two's complement signed input, where bit 17 is the most significant bit.
`b[26:0]`	Input	Data input B, a 27-bit, two's complement signed input, where bit 26 is the most significant bit.
`sub`	Input	Active-high subtract input. Setting `sub` to `1` inverts the B input allowing an `a[27:0] – b[27:0]` subtraction operation to be performed. The `cin` input must be asserted during a subtract operation.
`cin`	Input	Active-high user carry input. The `cin` input to the adder/subtractor is determined by the setting of the `sel_cin` parameter.
`load`	Input	Active-high add/sub load input. Asserting `load` high results in the add/sub block summing the `a[63:0]` and the `const[63:0]` inputs. Set the `load_const` parameter to `64'h0` if a load accumulator function is desired. The `cin` input is ignored when the `load` input is asserted.
`rnd`	Input	Active-high round add/sub input. Asserting the `rnd` input high rounds the sum of `(a + b + cin)` with the rounding mode selected by the `round_mode` parameter. For rounding modes that may result in an overflow condition, enable saturation with the `sat_mode` parameter.
`mshift`	Input	Active-high multiplier shift input. Asserting `mshift` high shifts the B input of the add /sub block seventeen bits to the right with sign extension. Aids in the computation of products larger than the 19x27 bits provided natively by the multiplier unit.
`reg_addr[2:0]`	Input	Register address input. Selects one of eight 27-bit constants in the register file for the A or B inputs of the multiplier block. The output of the register is valid on the next rising edge of the clock after the address is presented on the `reg_addr` inputs. The register file contents are programmed with the 27-bit values of the `regfile_0` through `regfile_7` parameters.
`ce_a`	Input	Active-high data input A register clock enable. Set high to allow data input A to be clocked into the A-input register when the `rstn_a` signal is high.
`ce_b`	Input	Active-high data input B register clock enable. Set high to allow data input B to be clocked into the B-input register when the `rstn_b` signal is high.
`ce_addsub`	Input	Active-high add/sub input register clock enable. Set high to assert the clock enable inputs for the `rnd`, `sub`, `load`, `cin` and `mshift` input registers. Set low to allow these registers to hold their value at the next rising edge of the clock.

Name	Type	Description
ce_addsub_a	Input	Active-high add/sub A input register clock enable. Set high to allow input to the add /sub A input register to be clocked into the register when the rstn_addsub_a signal is inactive. Set low to allow the add/sub A input register to hold its value at the next rising edge of the clock.
ce_dout	Input	Active-high add/sub output register clock enable. Set high to allow the data output of the add/sub block to be clocked into the dout and cout registers when the rstn_dout signal is high.
ce_cascade	Input	Active-high cascade bus register clock enable. Set high to enable the revi_casc, fwdi_casc and fwdo_casc registers when the rstn_casc signal is high.
ce_multout	Input	Active-high multiplier output register clock enable. Set high to allow the multiplier output to be clocked into the multiplier output register when the rstn_multout signal is high.
rstn_a	Input	Active-low data input A register reset. Assert low to perform a synchronous reset of the data input A register upon the next rising edge of the clock, and set the register to the value defined by the rst_value_a parameter. The priority of rstn_a relative to the clock enable input, ce_a, is determined by the value of the regce_priority_a parameter.
rstn_b	Input	Active-low data input B register reset. Assert low to perform a synchronous reset of the data input B register upon the next rising edge of the clock, and set the register to the value defined by the rst_value_b parameter. The priority of rstn_b relative to the clock enable input, ce_b, is determined by the value of the regce_priority_b parameter.
rstn_addsub	Input	Active-low add/sub control input registers reset. Assert low to perform a synchronous reset of the rnd, sub, load, cin and mshift input registers upon the next rising edge of the clock. Upon reset, the value taken on by these registers is determined by the rst_value_<regname> parameters. The priority of rstn_addsub relative to the clock enable input, ce_addsub, is determined by the value of the regce_priority_<regname> parameters.
rstn_addsub_a	Input	Active-low add/sub A input register reset. Assert low to perform a synchronous reset of the add/sub A input register upon the next rising edge of the clock, and set the register to the value defined by the rst_value_addsub_a parameter. The priority of rstn_addsub_a relative to the clock enable input, ce_addsub_a, is determined by the value of the regce_priority_addsub_a parameter.
rstn_dout	Input	Active-low dout/cout output register reset. Assert low to performs a synchronous reset of the dout and cout output registers upon the next rising edge of the clock, and set the dout register to the value defined by the rst_value_dout parameter. Also sets the cout register to the value defined by the rst_value_cout parameter. The priority of rstn_dout relative to the clock enable input, ce_dout, is determined by the value of the regce_priority_dout parameter.
		Active-low cascade bus register reset. Assert low to perform a synchronous reset of the revi_casc, fwdi_casc and fwdo_casc registers upon the next rising edge

Name	Type	Description
`rstn_cascade`	Input	of the clock, and set the value of these registers to zero. The operation of `rstn_cascade` is independent of the value of the `ce_cascade` input.
`rstn_multout`	Input	Active-low multiplier output register reset. Assert low to perform a synchronous reset of the multiplier output register upon the next rising edge of the clock, and set the value of this register to zero. The operation of `rstn_multout` is independent of the value of the `ce_multout` input.
`clk`	Input	Clock input. Data is clocked into the input and output registers at the rising edge of this input.
`dout[44:0]`(1)	Output	Data out. The dout[44:0] output is a 45-bit signed two's complement output of the add/sub block, where bit 44 is the most significant bit. Alternatively, the dout output may be programmed to output the upper or lower 32 bits of the add/sub output as determined by the value of the del_dout parameter. This output is conditionally registered as determined by the value of the sel_dout_del parameter.
`cout`	Output	Active-high carry out. Set high if a carry was generated out of the add/sub block. This output is conditionally registered as determined by the value of the `sel_dout_del` parameter.
`over_pos`	Output	Active-high positive overflow output. Set high if an overflow of the add/sub/round block is detected for a positive number. If the saturation block is enabled, the output is limited to the maximum value determined by the rounding and saturation parameter settings (see the Saturation (see page 135) section for details). This output is conditionally registered as determined by the `match_del` parameter.
`over_neg`	Output	Active-high negative overflow output. Set high if an overflow of the add/sub/round block is detected for a negative number. If the saturation block is enabled, the output is limited to the minimum value determined by the rounding and saturation parameter settings (see the Saturation (see page 135) section for details). This output is conditionally registered as determined by the `match_del` parameter.
`match`	Output	Active-high match output. Set high if the `match_pattern` parameter (masked by the `match_mask` parameter value) matches the value of the add/sub block output (see the ACX_DSP_GEN Rounding (see page 121) chapter for details). This output is conditionally registered as determined by the `match_del` parameter.
`fwdi_casc[26:0]`	Input	Forward data cascade bus input. Aids in the development of FIR filters. The `fwdi_casc` input directly connects to the `fwdo_casc` output of the ACX_DSP_GEN block adjacent to the bottom of this block. This input must not be connected to the FPGA fabric or an error condition occurs. The forward data cbus traverses from the bottom of the ACX_DSP_GEN column to the top, connecting adjacent ACX_DSP_GEN blocks.
`fwdi_dout[63:0]`	Input	Forward accumulator cascade bus input. Daisy chains the accumulator outputs to enable fast summation of FIR filter multiplier outputs. The `fwdi_dout` input directly connects to the `fwdo_dout` output of the ACX_DSP_GEN block adjacent to the bottom of this block. This input must not be connected to the FPGA fabric or an error condition occurs. The forward accumulator cascade bus traverses from the bottom of the ACX_DSP_GEN column to the top, connecting adjacent ACX_DSP_GEN blocks.

`fwdi_cin`	Input	Forward carry cascade input. Cascades the current add/sub block with the previous ACX_DSP_GEN block to provide accumulation of widths greater than 64 bits. The `fwdi_cin` input directly connects to the `fwdo_cout` output of the ACX_DSP_GEN block adjacent to the bottom of this block. This input must not be connected to the FPGA fabric or an error condition occurs. The forward carry cascade bus traverses from the bottom of the ACX_DSP_GEN column to the top, connecting adjacent ACX_DSP_GEN blocks.
`fwdi_match`	Input	Forward match cascade input. The forward match cascade input is used to daisy-chain the current block with the previous ACX_DSP_GEN block to allow comparisons wider than the 64 bit comparison provided by a single ACX_DSP_GEN block. The `fwdi_match` input directly connects to the `fwdo_match` output of the ACX_DSP_GEN block adjacent to the bottom of this block. This input must not be connected to the FPGA fabric or an error condition occurs. The forward match cascade bus traverses from the bottom of the ACX_DSP_GEN column to the top, connecting adjacent ACX_DSP_GEN blocks.
`revi_casc[26:0]`	Input	Reverse data cascade bus input. Aids in the development of symmetric FIR filters. The `revi_casc` input directly connects to the `revo_casc` output of the ACX_DSP_GEN block adjacent to the top of this block. This input must not be connected to the FPGA fabric or an error condition occurs. The reverse data cascade bus traverses from the top of the ACX_DSP_GEN column to the bottom, connecting adjacent ACX_DSP_GEN blocks.
`revi_dout[31:0]`	Input	Reverse dout bus input. Routes the bottom 32 bits of the add/sub block to the ACX_DSP_GEN block adjacent to the top of this block. Allows this block to output the bottom 32 bits of the add/sub output while the next ACX_DSP_GEN block outputs the top 32 bits of the add/sub output, making the entire 64-bit add/sub output available to the FPGA fabric. The `revi_dout` input directly connects to the `revo_dout` output of the ACX_DSP_GEN block adjacent to the top of this block. This input must not be connected to the FPGA fabric or an error condition occurs. The reverse dout bus traverses from each ACX_DSP_GEN block to the adjacent ACX_DSP_GEN block below.
`fwdo_casc[26:0]`	Output	Forward data cascade bus output. Aids in the development of FIR filters. The `fwdo_casc` output directly connects to the `fwdi_casc` output of the ACX_DSP_GEN block adjacent to the top of this block. This output must not be connected to the FPGA fabric or an error condition occurs. The forward data cascade bus traverses from the bottom of the ACX_DSP_GEN column to the top, connecting adjacent ACX_DSP_GEN blocks.
`fwdo_dout[63:0]`	Output	Forward accumulator cascade bus output. Daisy chains the accumulator outputs to enable fast summation of FIR filter multiplier outputs. The `fwdo_dout` output directly connects to the `fwdi_dout` input of the ACX_DSP_GEN block adjacent to the top of this block. This output must not be connected to the FPGA fabric or an error condition occurs. The forward accumulator cascade bus traverses from the bottom of the ACX_DSP_GEN column to the top, connecting adjacent ACX_DSP_GEN blocks.
		Forward carry cascade output. Cascades the current add/sub block with the next ACX_DSP_GEN block to provide accumulation of widths greater than 64 bits. The `fwdo_cout` output directly connects to the `fwdi_cin` input of the ACX_DSP_GEN

Name	Type	Description
`fwdo_cout`	Output	block adjacent to the top of this block. This output must not be connected to the FPGA fabric or an error condition occurs. The forward carry cascade bus traverses from the bottom of the ACX_DSP_GEN column to the top, connecting adjacent ACX_DSP_GEN blocks.
`fwdo_match`	Output	Forward match cascade output. Daisy chains the current block with the next ACX_DSP_GEN block to allow comparisons wider than the 64 bit comparison provided by a single block. The `fwdo_match` output directly connects to the `fwdi_match` input of the ACX_DSP_GEN block adjacent to the top of this block. This output must not be connected to the FPGA fabric or an error condition occurs. The forward match cascade bus traverses from the bottom of the ACX_DSP_GEN column to the top, connecting adjacent ACX_DSP_GEN blocks.
`revo_casc[26:0]`	Output	Reverse data cascade bus output. Aids in the development of symmetric FIR filters. The `revo_casc` output directly connects to the `revi_casc` input of the ACX_DSP_GEN block adjacent to the bottom of this block. This output must not be connected to the FPGA fabric or an error condition occurs. The reverse data cascade bus traverses from the top of the ACX_DSP_GEN column to the bottom, connecting adjacent ACX_DSP_GEN blocks.
`revo_dout[31:0]`	Output	Reverse dout bus output. Routes the bottom 32 bits of the add/sub block to the ACX_DSP_GEN block (if `addsub_bypass` = `1'b0`) or the bottom 32 bits of the add /sub block A input (if `addsub_bypass` = `1'b1`) to the ACX_DSP_GEN block adjacent to the bottom of this block. This allows this block to output the top 32 bits of the add/sub output while the previous ACX_DSP_GEN block outputs the bottom 32 bits of the add/sub output, making the entire 64-bit add/sub output available to the FPGA fabric. The `revo_dout` output directly connects to the `revi_dout` input of the ACX_DSP_GEN block adjacent to the bottom of this block. This output must not be connected to the FPGA fabric or an error condition occurs. The reverse dout bus traverses from current block to the adjacent ACX_DSP_GEN block below.

Table Notes

1. The output precision of a single ACX_DSP_GEN block may be expanded to 48 bits by setting the `sel_48_dout` parameter to reallocate the `over_pos`, `over_neg` and `match` outputs as `dout[47:45]`.

Parameters

Table 94: *ACX_DSP_GEN Parameters*

Parameter	Defined Values	Default Value	Description
`init_a`	18-bit hexadecimal value	`18'h0`	Defines the power-up default value of the 18-bit data input A input register.
`init_b`	27-bit hexadecimal value	`27'h0`	Defines the power-up default value of the 27-bit data input B input register.
`init_sub`	`1'b0`, `1'b1`	`1'b0`	Defines the power-up default value of the 1-bit subtract input register.
`init_cin`	`1'b0`, `1'b1`	`1'b0`	Defines the power-up default value of the carry-in input register.
`init_load`	`1'b0`, `1'b1`	`1'b0`	Defines the power-up default value of the load input register.
`init_rnd`	`1'b0`, `1'b1`	`1'b0`	Defines the power-up default value of the round input register.
`init_mshift`	`1'b0`, `1'b1`	`1'b0`	Defines the power-up default value of the mshift input register.
`init_dout`	64-bit hexadecimal value	`64'h0`	Defines the power-up default value of the 64-bit data output register.
`init_cout`	`1'b0`, `1'b1`	`1'b0`	Defines the power-up default value of the carry-out output register.
`rst_value_a`	18-bit hexadecimal value	`18'h0`	Defines the value assigned to the 18-bit data input A input register when the `rstn_a` input is asserted concurrent with the rising edge of the clock.
`rst_value_b`	27-bit hexadecimal value	`27'h0`	Defines the value assigned to the 27-bit data input B input register when the `rstn_b` input is asserted concurrent with the rising edge of the clock.
`rst_value_sub`	`1'b0`, `1'b1`	`1'b0`	Defines the value assigned to the subtract input register when the `rstn_sub` input is asserted concurrent with the rising edge of the clock.

Parameter	Defined Values	Default Value	Description
`rst_value_cin`	`1'b0`, `1'b1`	`1'b0`	Defines the value assigned to the carry-in input register when the `rstn_cin` input is asserted concurrent with the rising edge of the clock.
`rst_value_load`	`1'b0`, `1'b1`	`1'b0`	Defines the value assigned to the load input register when the `rstn_load` input is asserted concurrent with the rising edge of the clock.
`rst_value_rnd`	`1'b0`, `1'b1`	`1'b0`	Defines the value assigned to the round input register when the `rstn_rnd` input is asserted concurrent with the rising edge of the clock.
`rst_value_mshift`	`1'b0`, `1'b1`	`1'b0`	Defines the value assigned to the mshift input register when the `rstn_mshift` input is asserted concurrent with the rising edge of the clock.
`rst_value_dout`	64-bit hexadecimal value	`64'h0`	Defines the value assigned to the 64-bit data-out output register when the `rstn_dout` input is asserted concurrent with the rising edge of the clock.
`rst_value_cout`	`1'b0`, `1'b1`	`1'b0`	Defines the value assigned to the carry-out output register when the `rstn_dout` input is asserted concurrent with the rising edge of the clock.
`rst_mode_a` (1)	`1'b0`, `1'b1`	`1'b0`	Determines whether the assertion of the reset of the data A input register is synchronous or asynchronous with respect to the `clk` input.
`rst_mode_b`(1)	`1'b0`, `1'b1`	`1'b0`	Determines whether the assertion of the reset of the data B input register is synchronous or asynchronous with respect to the `clk` input.
`rst_mode_sub`(1)	`1'b0`, `1'b1`	`1'b0`	Determines whether the assertion of the reset of the subtract input register is synchronous or asynchronous with respect to the `clk` input.
`rst_mode_cin`(1)	`1'b0`, `1'b1`	`1'b0`	Determines whether the assertion of the reset of the carry-in input register is synchronous or asynchronous with respect to the `clk` input.
`rst_mode_load`(1)	`1'b0`, `1'b1`	`1'b0`	Determines whether the assertion of the reset of the load input register is synchronous or asynchronous with respect to the `clk` input.

Parameter	Defined Values	Default Value	Description
rst_mode_rnd[1]	1'b0, 1'b1	1'b0	Determines whether the assertion of the reset of the round input register is synchronous or asynchronous with respect to the clk input.
rst_mode_mshift[1]	1'b0, 1'b1	1'b0	Determines whether the assertion of the reset of the mshift input register is synchronous or asynchronous with respect to the clk input.
rst_mode_dout[1]	1'b0, 1'b1	1'b0	Determines whether the assertion of the reset of the data out output register and the carry-out output registers are synchronous or asynchronous with respect to the clk input.
regce_priority_a	"rstreg", "regce"	"regce"	Defines the priority of the ce_a clock enable input relative to the rstn_a reset input during assertion of the rstn_a reset input on the data input A input register when parameter rst_mode_a is set high. Setting regce_priority_a to "rstreg" allows the data input A input register to be set/reset at the next rising edge of the clock without requiring the ce_a clock enable input to be active. Setting regce_priority_a to "regce" requires that the ce_a clock enable input is high for the reset operation to occur at the next rising edge of the clock.
regce_priority_b	"rstreg", "regce"	"regce"	Defines the priority of the ce_b clock enable input relative to the rstn_b reset input during an assertion of the rstn_b reset input on the data input B input register when parameter rst_mode_b is set high. Setting regce_priority_b to "rstreg" allows the data input B input register to be set/reset at the next rising edge of the clock without requiring the ce_b clock enable input to be active. Setting regce_priority_b to "regce" requires that the ce_b clock enable input is high for the reset operation to occur at the next rising edge of the clock.
regce_priority_sub[2]	"rstreg", "regce"	"regce"	Defines the priority of the ce_addsub clock enable input relative to the rstn_addsub reset input during an assertion of the rstn_addsub reset input on the sub input register when parameter rst_mode_sub is set high. Setting regce_priority_sub to "rstreg" allows the sub input register to be

Parameter	Defined Values	Default Value	Description
			set/reset at the next rising edge of the clock without requiring the ce_addsub clock enable input to be active.
regce_priority_cin[2]	"rstreg", "regce"	"regce"	Defines the priority of the ce_addsub clock enable input relative to the rstn_addsub reset input during an assertion of the rstn_addsub reset input on the cin input register when parameter rst_mode_cin is set high. Setting regce_priority_cin to "rstreg" allows the cin input register to be set/reset at the next rising edge of the clock without requiring the ce_addsub clock enable input to be active.
regce_priority_load[2]	"rstreg", "regce"	"regce"	Defines the priority of the ce_addsub clock enable input relative to the rstn_addsub reset input during an assertion of the rstn_addsub reset input on the load input register when parameter rst_mode_load is set high. Setting regce_priority_load to "rstreg" allows the load input register to be set/reset at the next rising edge of the clock without requiring the ce_addsub clock enable input to be active.
regce_priority_rnd[2]	"rstreg", "regce"	"regce"	Defines the priority of the ce_addsub clock enable input relative to the rstn_addsub reset input during an assertion of the rstn_addsub reset input on the round input register when parameter rst_mode_rnd is set high. Setting regce_priority_rnd to "rstreg" allows the round input register to be set/reset at the next rising edge of the clock without requiring the ce_addsub clock enable input to be active.
regce_priority_mshift [2]	"rstreg", "regce"	"regce"	Defines the priority of the ce_addsub clock enable input relative to the rstn_addsub reset input during an assertion of the rstn_addsub reset input on the mshift input register when parameter rst_mode_mshift is set high. Setting regce_priority_mshift to "rstreg" allows the mshift input register to be set/reset at the next rising edge of the clock without requiring the ce_addsub clock enable input to be active.

Parameter	Defined Values	Default Value	Description
`regce_priority_dout`	`"rstreg"`, `"regce"`	`"regce"`	Defines the priority of the `ce_dout` clock enable input relative to the `rstn_dout` reset input during an assertion of the `rstn_dout` reset input on the dout output register and the carry-out output register when parameter `rst_mode_dout` is set high. Setting `regce_priority_dout` to `"rstreg"` allows the Dout output register and the carry-out output register to be set/reset at the next rising edge of the clock without requiring the `ce_dout` clock enable input to be active. Setting `regce_priority_dout` to `"regce"` requires that the `ce_dout` clock enable input is high for the reset operation to occur at the next rising edge of the clock.
`a_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the data A input register is used or bypassed.
`b_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the data B input register is used or bypassed.
`sub_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the sub input register is used or bypassed.
`cin_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the cin input register is used or bypassed.
`load_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the load input register is used or bypassed.
`rnd_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the round input register is used or bypassed.
`mshift_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the mshift input register is used or bypassed.
`dout_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the dout output register is used or bypassed.
`cout_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the cout output register is used or bypassed.
`over_pos_del`	`1'b0`, `1'b1`	`1'b0`	Defines whether the `over_pos` output register is used or bypassed. Setting to `1'b0` bypasses the register while setting to `1'b1` enables the register.
			Defines whether the `over_neg` output register is used or bypassed. Setting to `1'b0`

Parameter	Defined Values	Default Value	Description
`over_neg_del`	`1'b0`, `1'b1`	`1'b0`	bypasses the register while setting to `1'b1` enables the register.
`match_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the match output register is used or bypassed.
`preadd_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the pre-adder output register is used or bypassed.
`multout_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the multiplier output register is used or bypassed.
`addsub_areg_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the add/sub A input register is used or bypassed.
`regaddr_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the register file address input register is used or bypassed.
`fwdi_casc_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the forward cascade data input register is used or bypassed.
`fwdo_casc_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the forward cascade data output register is used or bypassed.
`revi_casc_del`(3)	`1'b0`, `1'b1`	`1'b0`	Defines whether the reverse cascade data input register is used or bypassed.
`addsub_bypass`	`1'b0`, `1'b1`	`1'b1`	Defines whether the add/sub block is used or bypassed. Setting `addsub_bypass` to `1'b0` allows the add/sub block to be used, while setting `addsub_bypass` to `1'b1` bypasses the add/sub block by connecting the A input of the add/sub block to the input of the dout output register.
`sel_addsub_a`	`2'b00`–`2'b11`	`2'b00`	Defines what is routed to the input of the add /sub block A input: `2'b00` – multiplier output sign extended to 64 bits. `2'b01` – multiplier output arithmetically shifted 18 bits to the left. `sel_48_dout` must also be set to `1'b0`. `2'b10` – 64-bit sign extension of the concatenation of the `reg_addr`, A and B inputs: `sext{reg_addr[2:0],a[17:0],b[26:0]}`. Also routes the data A input register clock enable and reset signals to the register file address input registers. `2'b11` – 64-bit `fwdi_dout` input from the ACX_DSP_GEN block below the current

Parameter	Defined Values	Default Value	Description
			block. `sel_addsub_b` must also be set to `1'b0`.
`sel_addsub_b`(4) (5)	`1'b0`, `1'b1`	`1'b0`	Defines what is routed to the input of the add /sub block B input: `1'b0` – registered ACX_DSP_GEN output `dout[63:0]`. `1'b1` – 64-bit `fwdi_dout` input from the ACX_DSP_GEN block below the current block.
`sel_cin`	`1'b0`, `1'b1`	`1'b0`	Selects either the carry-in input of this ACX_DSP_GEN block or the carry-out output of the ACX_DSP_GEN block below the current block: `1'b0` – the `cin` input is routed to the add/sub block `cin` input. `1'b1` – the `fwdi_cin` input is routed to the add/sub block `cin` input.
`sel_fwdo_dout`(6)	`2'b00`–`2'b10`	`2'b00`	Defines what is routed to the forward accumulator cascade bus (`fwdo_dout`) output: `2'b00` – `addsub_bypass = 1'b0`: 64-bit unregistered add/sub block output. `2'b00` – `addsub_bypass = 1'b1`: 64-bit add/sub block A input. `2'b01` – `addsub_bypass = 1'b0`: 64-bit registered add/sub block output. `2'b01` – `addsub_bypass = 1'b1`: 64-bit registered add/sub block A input. `2'b10` – 64-bit `fwdi_dout` input. `2'b11` – Undefined.
`sel_fwdo_cout`	`1'b0`, `1'b1`	`1'b0`	Defines what is routed to the `fwdo_cout` output that is connected to the next ACX_DSP_GEN block `fwdi_cin` input: `1'b0` – the `cout` from the add/sub/rnd/sat block is routed to `fwdo_cout`. `1'b1` – the `fwdi_cout` input is routed to `fwdo_cout`.
			Defines what is routed to the 45-bit `dout` output: `2'b00` – `addsub_bypass = 1'b0`: the conditionally-registered (by `dout_del`) lower 45 bits of the add/sub block output. `2'b00` – `addsub_bypass = 1'b1`: the conditionally registered (by `dout_del`) lower

Parameter	Defined Values	Default Value	Description
`sel_dout`	`2'b00`–`2'b10`	`2'b00`	45 bits of the add/sub block A input. `2'b01` – the upper 32 bits of add/sub output: `{13'h0,Add/Sub[63:32]}`. `2'b10` – the bottom 32 bits of the add/sub block above the current ACX_DSP_GEN block: `{13'h0,revi_dout[31:0]}`. `2'b11` – undefined.
`sel_48_dout`	`1'b0`, `1'b1`	`1'b0`	Expands the dout precision from 45 to 48 bits by reallocating the `over_pos`, `over_neg` and `match` outputs as `dout[48]` through `dout[46]`: `1'b0` – the `over_pos` output is isused as positive overflow output. `1'b0` – the `over_neg` output is used as the negative overflow output. `1'b0` – the match output is used as the pattern match output. `1'b0` – the `cout` output is used as a carry for a 64-bit add/sub/rnd/sat word. `1'b1` – the `over_pos` output isused as `dout [47]`. `1'b1` – the `over_neg` output is used as `dout [46]`. `1'b1` – the match output is used as `dout [45]`. `1'b1` – the `cout` output is used as a carry for a 48-bit add/sub/rnd/sat word. This parameter also redefines the `cout` output to generate the carry signal on the 48th bit of the add/sub/rnd/sat block so that pairs of ACX_DSP_GEN blocks may be used as 48-bit slices of adders or subtractors. If `sel_48_dout` is set to `1'b1`, `sel_dout` must be set to `2'b00`.
`sel_revi_casc`	`1'b0`, `1'b1`	`1'b0`	Defines what is routed to the input of the reverse data cascade bus delay register: `1'b0` – `revi_casc` is routed to the reverse data cascade bus delay register input. `1'b1` – `fwdo_casc` of the current ACX_DSP_GEN block is routed to the reverse data cascade bus delay register input. Usually set to `1'b0` to select the `revi_casc` data input. If the current block is the middle stage of a symmetric FIR filter, where the forward data cascade bus must be looped back to the reverse data cascade bus, this parameter must be set to `1'b1`.

Parameter	Defined Values	Default Value	Description
`sel_fwd_preadd`	`2'b00`–`2'b10`	`2'b00`	Defines what is routed to the B input of the preadder: `2'b00` – `27'h0`. `2'b01` – `fwdi_casc[26:0]`. `2'b10` – data input `b[26:0]`. `2'b11` – undefined.
`sel_rev_preadd`	`2'b00`–`2'b10`	`2'b00`	Defines what is routed to the A input of the preadder: `2'b00` – `27'h0`. `2'b01` – `revi_casc[26:0]`. `2'b10` – data input `a[17:0]` sign extended to 27 bits. `2'b11` – Undefined.
`sel_mult_a`	`2'b00`–`2'b11`	`2'b00`	Defines what is routed to the A input of the multiplier: `2'b00` – data input `a[17:0]` sign extended to 19 bits: `{a[17],a[17:0]}`. `2'b01` – pre-adder output `[18:0]`. `2'b10` – register file output `[18:0]`. `2'b11` – unsigned data input `a[17:0]`: `{1'b0,a[17:0]}`.
`sel_mult_b`	`2'b00`–`2'b11`	`2'b00`	Defines what is routed to the B input of the multiplier: `2'b00` – data input `b[26:0]`. `2'b01` – `fwdi_casc[26:0]`. `2'b10` – pre-adder output `[26:0]`. `2'b11` – register file output `[26:0]`.
`preadd_mode`(7)	`2'b00`–`2'b10`	`2'b00`	Defines the functionality of the pre-adder: `2'b00` – data input A plus data input B. `2'b01` – data input A minus data input B. `2'b10` – data input B minus data input A. `2'b11` – Undefined.
`round_mode`(8)	`3'b000`–`3'b111`	`3'h0`	Defines the functionality of the rounding unit within the add/sub block: `3'b000` – no rounding. `3'b001` – round towards nearest integer. `3'b010` – round towards zero. `3'b011` – round towards infinity. `3'b100` – round towards plus infinity, round towards nearest even, round towards nearest odd, or round half down. `3'b101` – round half away from zero. `3'b110` – round half up. `3'b111` – round half towards zero.

Parameter	Defined Values	Default Value	Description
			See ACX_DSP_GEN Rounding (see page 121) for a complete description of the rounding modes.
`sat_mode`	`6'h0`–`6'h3F`	`6'h0`	Defines the saturation mode of the add/sub block. The default value of `6'h0` disables saturation and directly passes the add/sub output to the `dout` pins. A non-zero value of the `sat_mode` parameter selects the most significant bit of where saturation is detected to allow saturation of data paths less than the full 64-bit resolution of the add/sub block. See the Saturation (see page 135) section for further details. If `sel_48_dout` is set to `1'b1` , `sat_mode` must be `6'h0`.
`use_match_in`	`1'b0`, `1'b1`	`1'b0`	Allows the pattern match function to occur across multiple ACX_DSP_GEN blocks. When set to `1'b1`, requires the previous ACX_DSP_GEN block output, `fwdo_match`, to be high when evaluating the match function. Leaving the default value of `1'b0` performs the match function only within the current block. If set to `1'b1`, `round_mode` must be set to `3'h0`.
`match_pattern`	`64'h0`–`64'hFFFFFFFFFFFFFFFF`	`64'h0`	Defines the data pattern that should be used to compare against the output of the add/sub block when performing match detection.
`match_mask`	`64'h0`–`64'hFFFFFFFFFFFFFFFF`	`64'h0`	Defines which of the 64 bits at the output of the add/sub block should be used in the match detection operation. The match detection compares each bit position that contains a high bit in the `match_mask` parameter. The bit positions in the parameter that are set to zero are not used in the match function.
`round_const`	`64'h0`–`64'hFFFFFFFFFFFFFFFF`	`64'h0`	Defines the offset to be added to the output of the add/sub block if a rounding operation is to be performed. Please refer to the ACX_DSP_GEN Rounding (see page 121) section for a detailed description of the rounding function.
`load_const`	`64'h0`–`64'hFFFFFFFFFFFFFFFF`	`64'h0`	Defines what is loaded into the B side of the add/sub block if the load input is asserted. To load only the A side into the add/sub block, this parameter should be set to 64'h0.

Parameter	Defined Values	Default Value	Description
`regfile_0–regfile_7`	`27'h0–27'h7FFFFFF`	`27'h0`	The `regfile_0` through `regfile_7` parameters define the value of the output of the register file on the clock cycle after the `reg_addr[2:0]` inputs are set to `3'h0` through `3'h7`, respectively.

Table Notes

1. Setting this parameter high defines the assertion of reset to be asynchronous with respect to the clock input. Setting this parameter low defines the assertion of the reset to be synchronous with the rising edge of the `clk` input.
2. Setting this parameter to `regce` requires that the `ce_addsub` clock enable input is high for the reset operation to occur at the next rising edge of the clock.
3. Setting this parameter to `1'b0` bypasses the register while setting it to `1'b1` enables the register.
4. The 64-bit value selected by the `sel_addsub_b` input is conditionally shifted seventeen bits to the right by the mshift input before it is routed to the B input of the add/sub block.
5. If the `sel_addsub_b` parameter is set to `1'b0`, `dout_del` must be set to `1'b1`, `round_mode` must be set to `3'h0`.
6. If the 64-bit registered add/sub block output is selected, then parameter `dout_del` must be set to `1'b1`, and inputs, `rstn_dout` and `ce_dout`, must be driven appropriately.
7. The pre-adder operates on two 27-bit inputs and produces a 27-bit result. If full 27-bit resolution of the output is required, limit the dynamic range of the inputs to 26 bits and sign-extend into the 27th bit to prevent an overflow condition at the output of the pre-adder.
8. The `round_mode` parameter must be used in conjuction with the use of the `match_pattern`, `match_mask` and `round_const` parameters.

Add/Subtract/Round/Saturate Blocks

The add/sub/round/saturate block can be sub-divided into adder/subtracter, round, and saturate blocks as shown in the following figure. The A input from the multiplier to the adder/subtracter may bypass the adder/subtractor, round, and saturate blocks by enabling the `addsub_bypass` parameter.

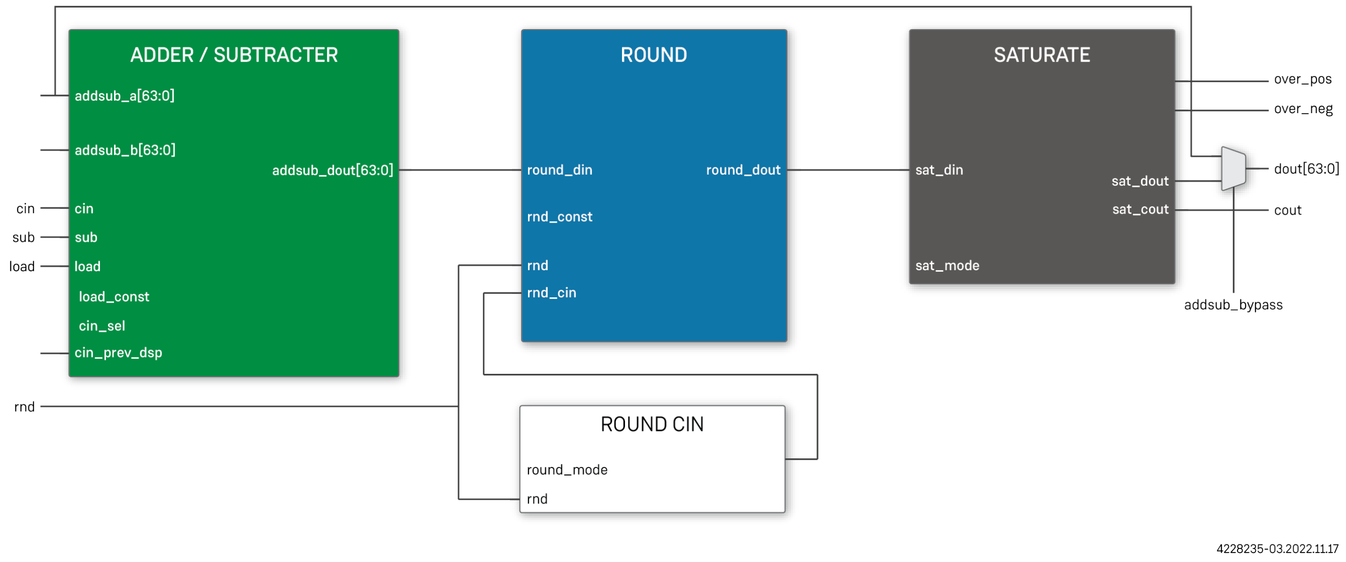


Figure 48: ***Adder/Subtracter, Round, and Saturate Blocks Example***

ACX_DSP_GEN Rounding

As mathematical operations are performed, the number of bits required to represent the number may increase. For example, in multiplication, multiplying an m-bit number and an n-bit number will result in an (m + n)-bit product. Also, adding two n-bit numbers may result in a (n+1)-bit result.

Rounding is a method used to control bit growth and approximate the result to a fixed number of bits. There are also many variations in how the rounding is performed. The ACX_DSP_GEN block supports a variety of rounding methods.

Note

A given rounding mode may have several different names all referring to the same method.

The rounding unit within the add/sub block supports the rounding modes shown in the following table. The various names for the rounding modes are also reflected in the table. The rounding operation is performed on the output of the adder/subtracter after the add or subtract operation has been performed and before the saturation unit.

Table 95: *Supported Rounding Modes*

Rounding Mode	Alternative Name
No rounding	No
Round to zero	MATLAB fix() Sign-magnitude truncation Round away from infinity
Round to infinity	–
Round to plus infinity	MATLAB ceil() Ceiling Round up
Round to minus infinity	Two's complement truncation Downward-directed rounding MATLAB floor() Round down
Round to nearest integer	MATLAB round()
Round to nearest-even	Round half to nearest-even
Round to nearest-odd	Round half to nearest-odd
Round half up	Round half towards plus infinity
Round half down	Round half towards minus infinity
Round half away from zero	Round half towards infinity
Round half towards zero	Round half away from infinity

The following sections describe each of the rounding modes. Each of the diagrams show the number X on the x axis and the corresponding rounded value of X on the y axis.

Note

When there is a dot on the end of a line segment, that end value is included on the line. For example, in the Round to Zero diagram, the line that corresponds to the range of X from one to two with the dot on the left side of the line, represents the X values greater than or equal to (>=) one and less than (<) two.

Below each rounding mode figure is a table containing the values of the `match_pattern`, `match_mask`, `round_const` and `round_mode` parameters required for the given rounding model. Additionally, for rounding to occur, the round input, `rnd`, must be asserted. At the top of the table, the number to be rounded is shown in the form of XXXX.YYYYYY, where XXXX is the integer part of the number and YYYYYY is the fractional part. For this specific example XXXX.YYYYYY, the integer part is four bits and the fractional part is six bits. Depending on where the decimal point needs to be for the specific number format, the shown values of XXXX and YYYYYY might have to be expanded or contracted about the position of the decimal point. If the Y value needs to be widened, the shown Y value should be extended to the right, copying the lsb of the shown Y value. Likewise, the XXXX value can be expanded or contracted from the leftmost position.

Note

The resultant XXXX.YYYYYY pattern must be zero-extended to the left to fill out the entire 64-bit value of the given parameter.

For the result of the rounding operation, use the XXXX bits and discard the YYYYYY bits.

For each of the rounding modes, the rounding circuitry is looking for certain conditions that determine if the number needs rounding or not. Where one rounding mode differs from the next depends upon what happens when the number is exactly half way between two integers. For example, 2.5 or -2.5, or exactly at an integer boundary (2.0 or -2.0). Since the circuitry also allows the assumed position of the decimal point to be moved, the assumed location of the decimal point must be declared.

To check for the half boundary, the match pattern is set to have a match pattern of 0.5. The programmed match_pattern value is a 64-bit number in the form of a string of zeroes on the left, a single one, and a string of zeroes to the right. The position of the one is in the 2^(-1) bit position. The `match_mask` parameter determines which of the `match_pattern` bit positions are used. To check for the integer boundary, the match pattern is set to all zeroes to the right of the decimal and the `match_mask` is set to select the bits to the right of the decimal point.

When the rounding condition is determined by the check using the `match_pattern` and `match_mask`, the `round_mode` parameter setting is used to select the conditions when the forced rounding is to occur. The rounding is usually performed by adding a one to the 2^0 bit position. The rounding circuit forces a one into the carry-in input of the rounding unit. The `round_const` parameter determines how far the carry is to ripple to the left to get to the assumed 2^0 bit position.

Round to even or odd is accomplished in a similar fashion, with the pattern detection circuitry looking for even /odd boundaries.

Round Towards Zero

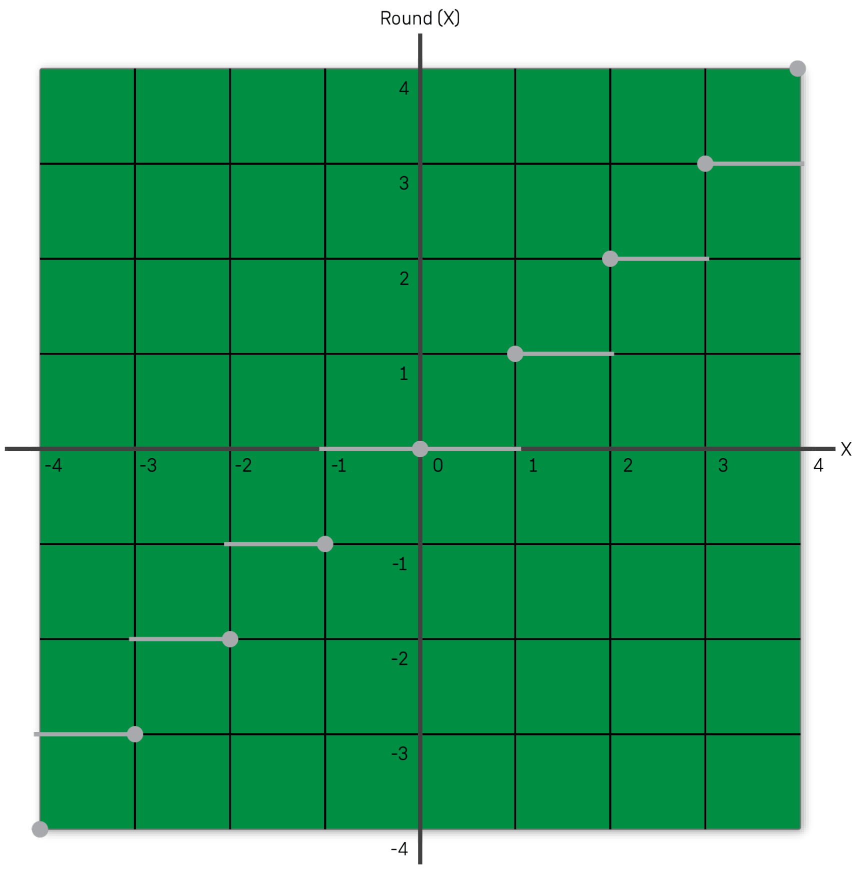


Figure 49: ***Round Towards Zero***

Table 96: ***Required Parameter Settings for Round Towards Zero Mode***

Parameter	Required Parameter Value XXXX.YYYYYY Format
`match_pattern`	0000.000000
`match_mask`	0000.111111
`round_const`	0000.111111
`round_mode`	`3'b010`

Rounding Towards Infinity

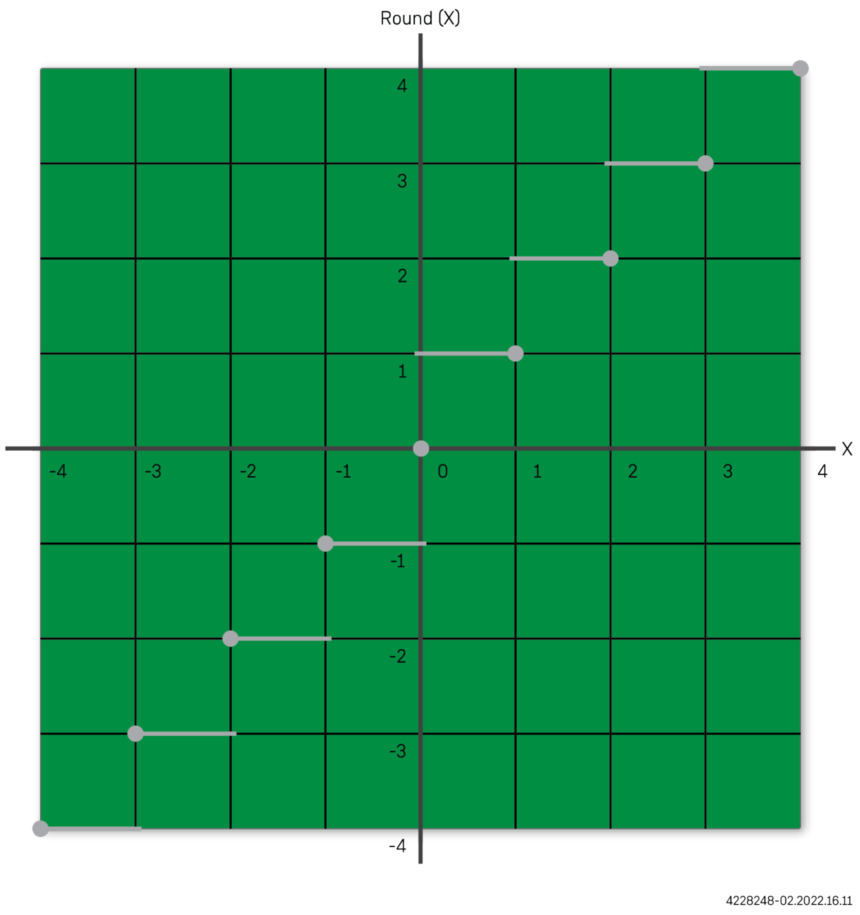


Figure 50: ***Round Towards Infinity***

Table 97: ***Required Parameter Settings for Round Towards Infinity Mode***

Parameter	Required Parameter Value XXXX.YYYYYY Format
`match_pattern`	0000.000000
`match_mask`	0000.111111
`round_const`	0000.111111
`round_mode`	`3'b011`

Round Towards Plus Infinity

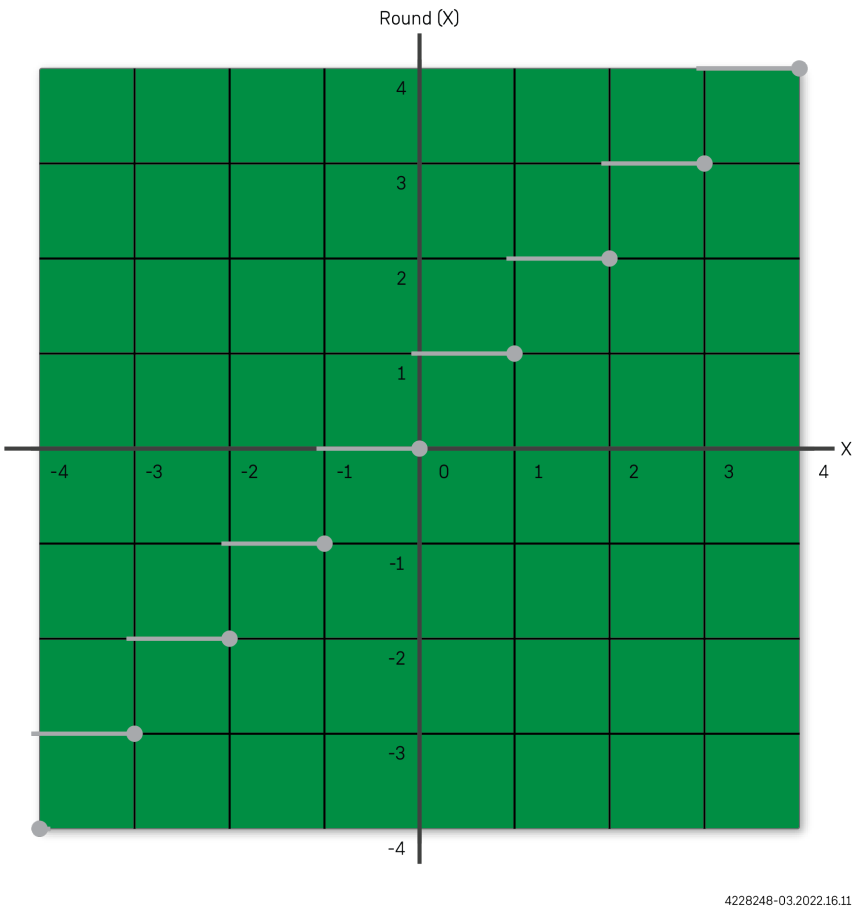


Figure 51: ***Round Towards Plus Infinity***

Table 98: ***Required Parameter Settings for Round Towards Plus Infinity Mode***

Parameter	Required Parameter Value XXXX.YYYYYY Format
`match_pattern`	0000.000000
`match_mask`	0000.111111
`round_const`	0000.111111
`round_mode`	`3'b100`

Round Towards Minus Infinity

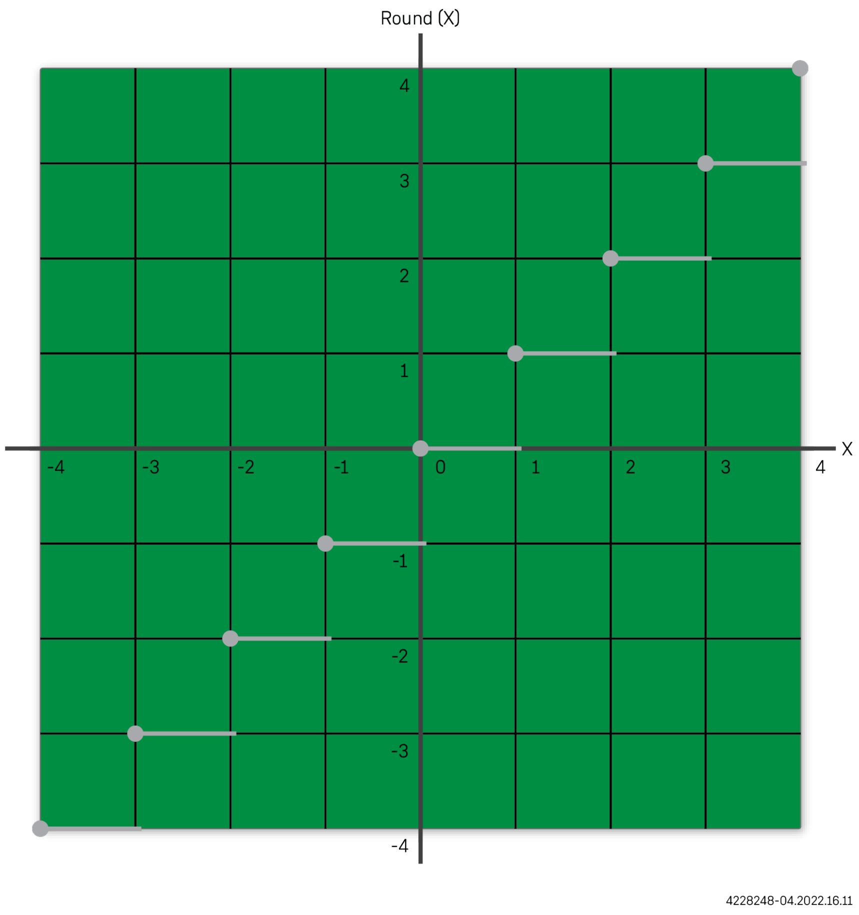


Figure 52: ***Round Towards Minus Infinity***

Table 99: ***Required Parameter Settings for Round Towards Minus Infinity Mode***

Parameter	Required Parameter Value XXXX.YYYYYY Format
`match_pattern`	xxxx.xxxxxx
`match_mask`	xxxx.xxxxxx
`round_const`	xxxx.xxxxxx
`round_mode`	`3'b000`

Round Towards Nearest Integer

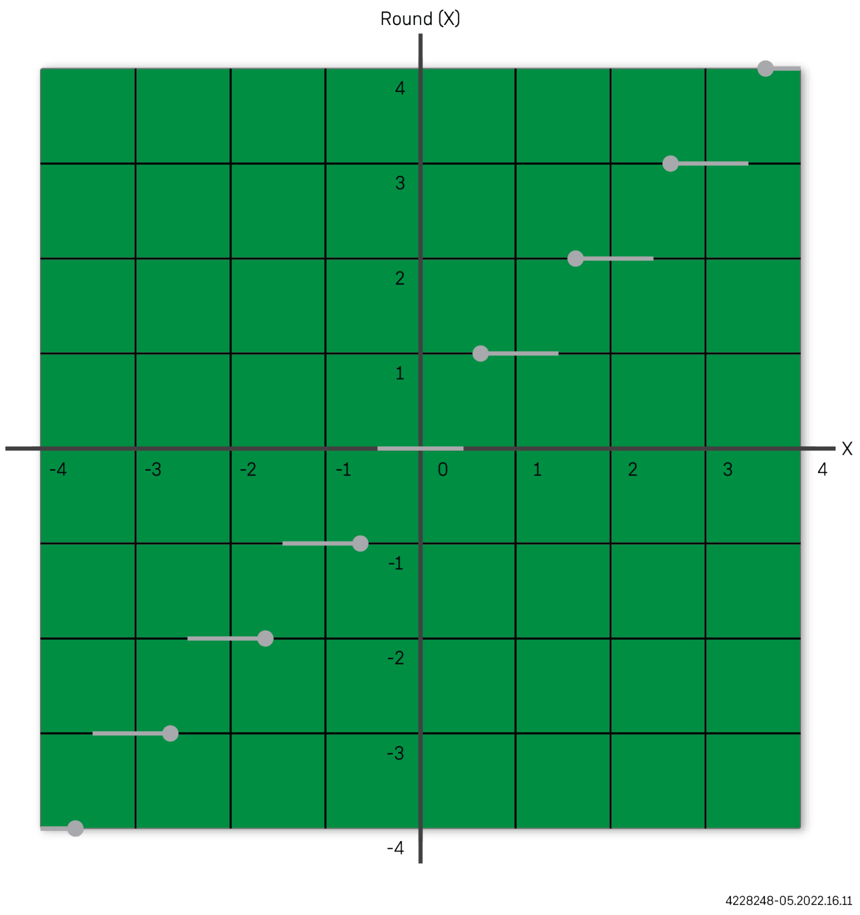


Figure 53: ***Round Towards Nearest Integer***

Table 100: ***Required Parameter Settings for Round Towards Nearest Integer Mode***

Parameter	Required Parameter Value XXXX.YYYYYY Format
`match_pattern`	0000.100000
`match_mask`	0000.111111
`round_const`	0000.111111
`round_mode`	`3'b001`

Round Towards Nearest Even

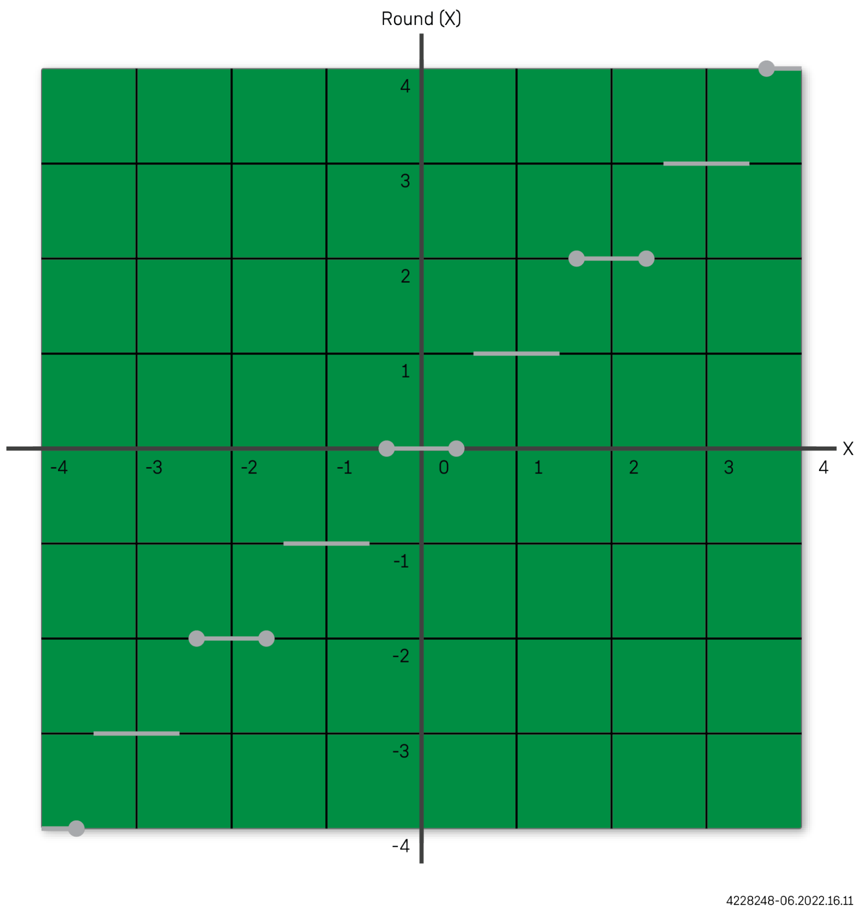


Figure 54: ***Round Towards Nearest Even***

Table 101: ***Required Parameter Settings for Round Towards Nearest Even Mode***

Parameter	Required Parameter Value XXXX.YYYYYY Format
`match_pattern`	0000.100000
`match_mask`	0001.111111
`round_const`	0000.011111
`round_mode`	`3'b100`

Round Towards Nearest Odd

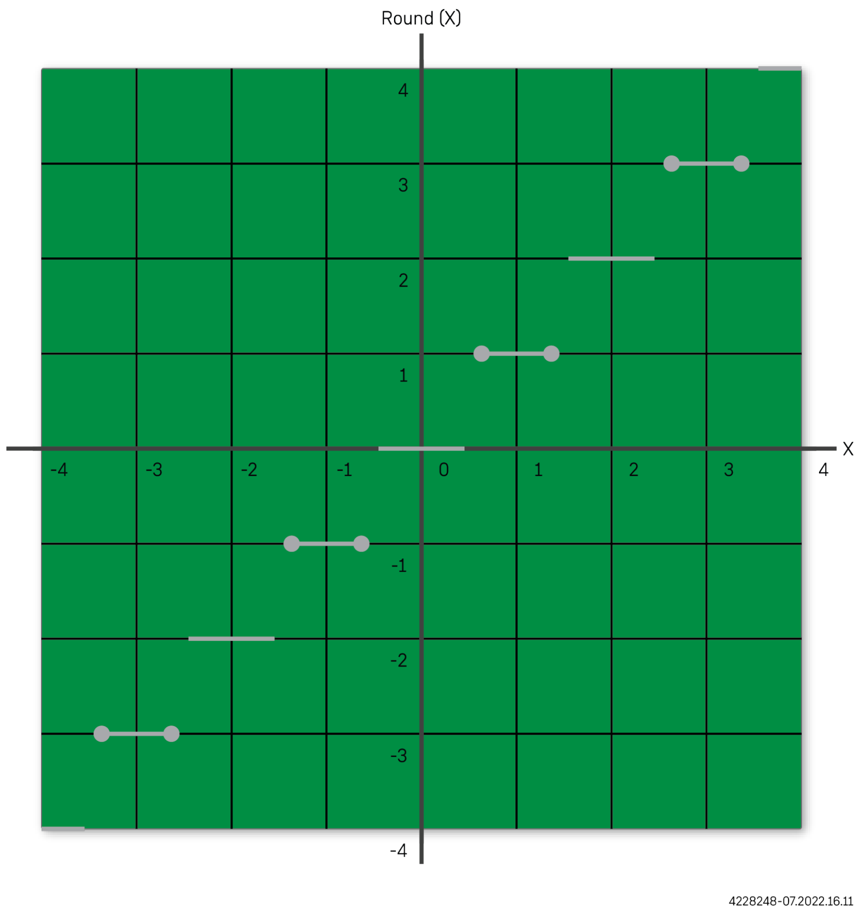


Figure 55: ***Round Towards Nearest Odd***

Table 102: ***Required Parameter Settings for Round Towards Nearest Odd Mode***

Parameter	Required Parameter Value XXXX.YYYYYY Format
`match_pattern`	0001.100000
`match_mask`	0001.111111
`round_const`	0000.011111
`round_mode`	`3'b100`

Round Half Up

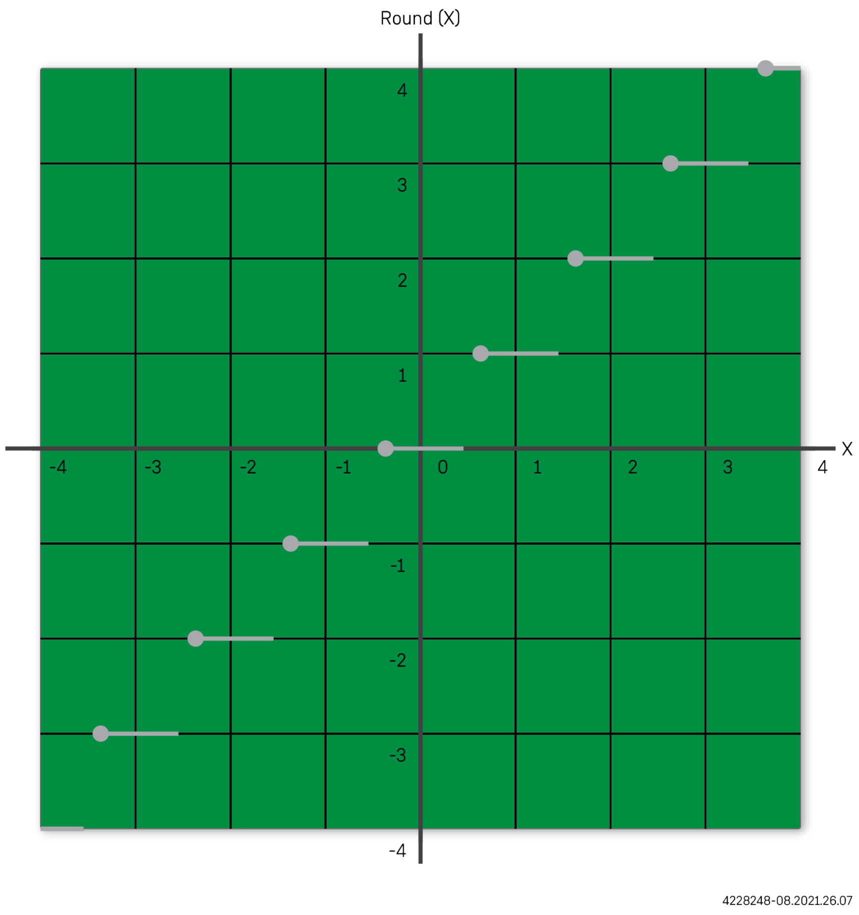


Figure 56: ***Round Half Up***

Table 103: ***Required Parameter Settings for Round Half Up Mode***

Parameter	Required Parameter Value XXXX.YYYYYY Format
`match_pattern`	0000.100000
`match_mask`	0000.100000
`round_const`	0000.111111
`round_mode`	`3'b110`

Round Half Down

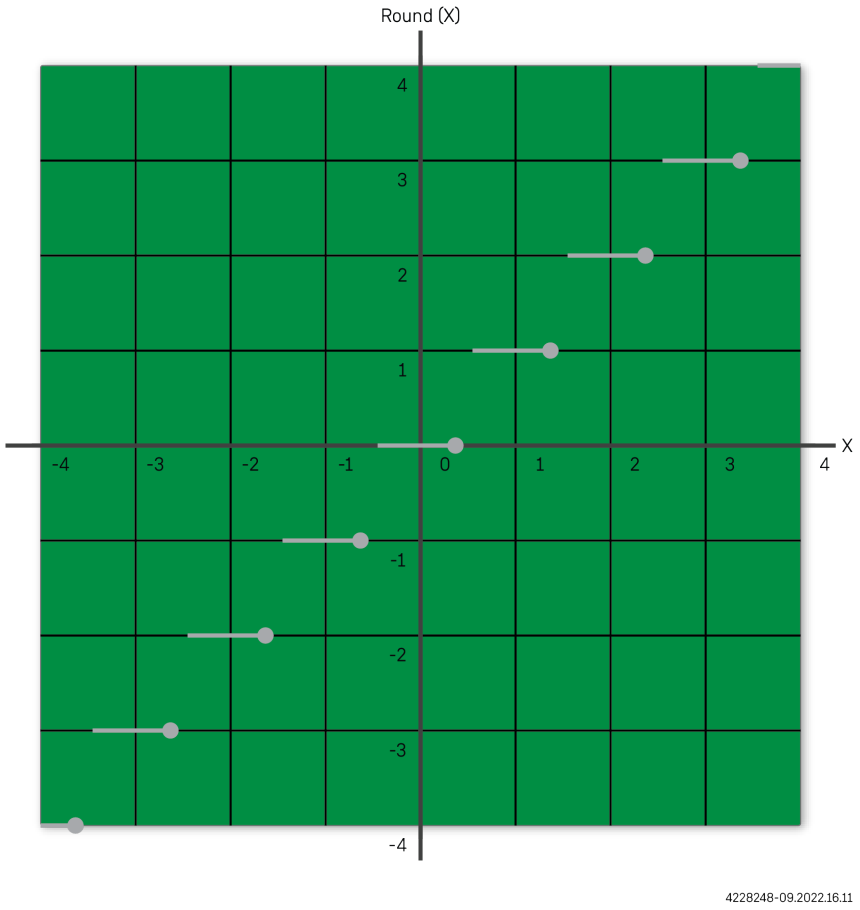


Figure 57: ***Round Half Down***

Table 104: ***Required Parameter Settings for Round Half Down Mode***

Parameter	Required Parameter Value XXXX.YYYYYY Format
`match_pattern`	0000.100000
`match_mask`	0000.111111
`round_const`	0000.011111
`round_mode`	`3'b100`

Round Half Away From Zero

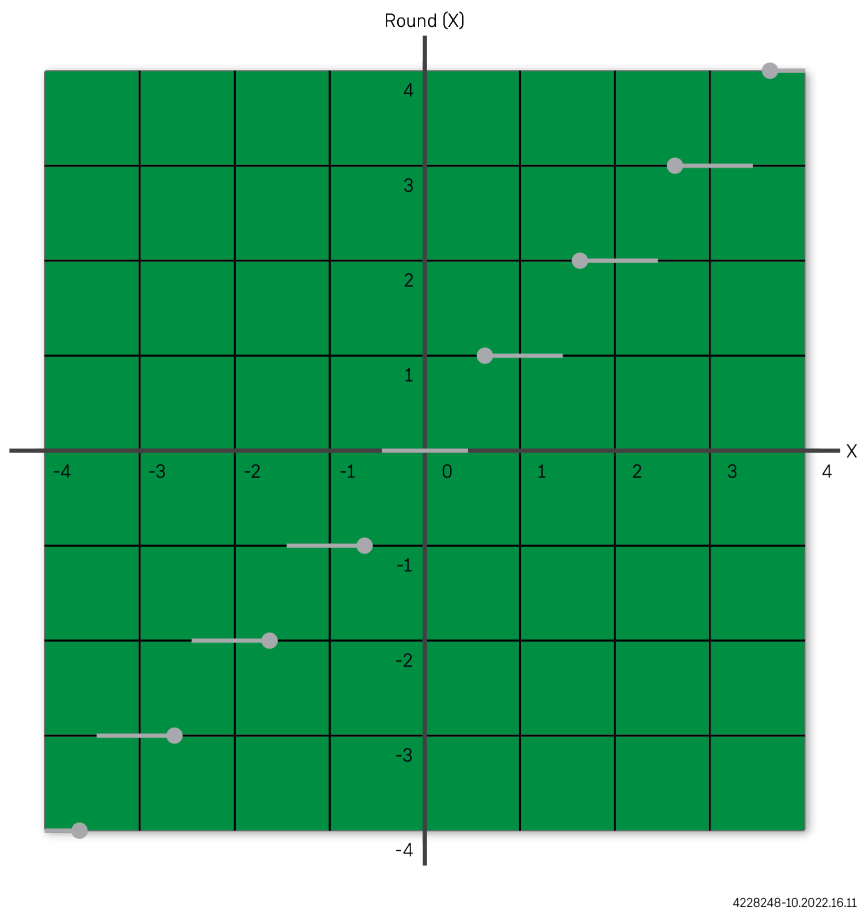


Figure 58: ***Round Half Away From Zero***

Table 105: ***Required Parameter Settings for Round Half Away from Zero Mode***

Parameter	Required Parameter Value XXXX.YYYYYY Format
`match_pattern`	0000.100000
`match_mask`	0000.111111
`round_const`	0000.011111
`round_mode`	`3'b101`

Round Half Towards Zero

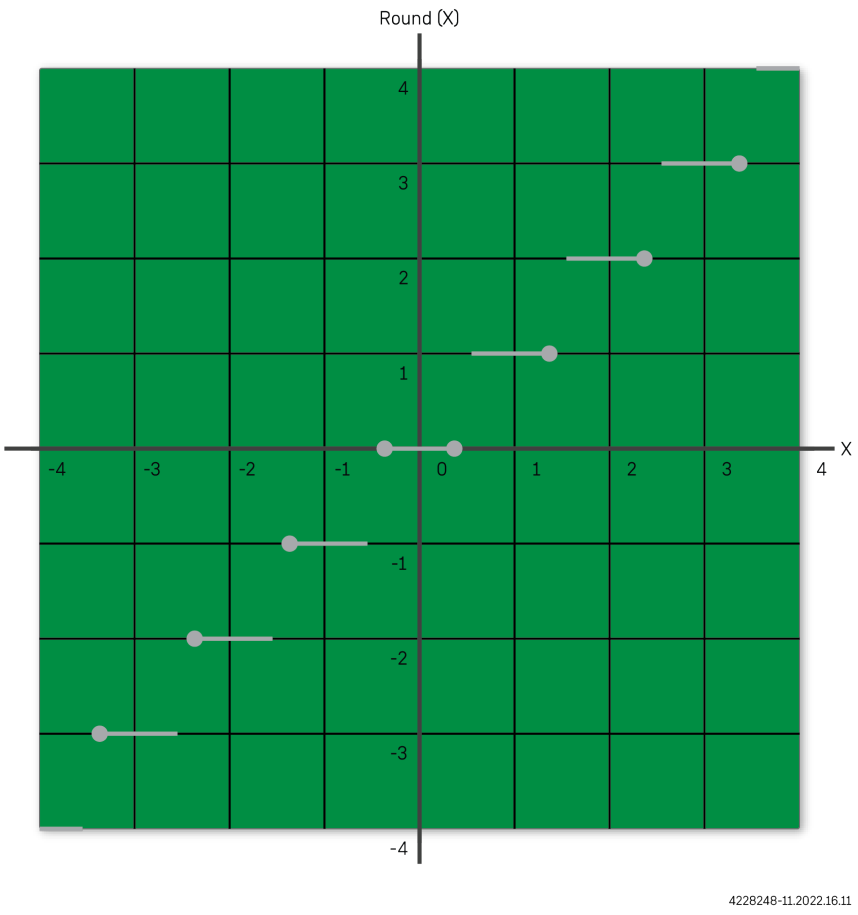


Figure 59: ***Round Half Towards Zero***

Table 106: ***Required Parameter Settings for Round Half Towards Zero Mode***

Parameter	Required Parameter Value XXXX.YYYYYY Format
`match_pattern`	0000.100000
`match_mask`	0000.111111
`round_const`	0000.011111
`round_mode`	`3'b111`

Saturation

When two numbers are added or subtracted, the result may require an additional bit to represent the result. If there is not sufficient room for bit growth in the adder/subtractor to contain the result, an overflow condition may occur.

Overflow is defined as when the maximum value allowed by the provided number of bits is exceeded. This may occur for both positive and negative numbers.

The usual remedy is to either guarantee that an overflow never occurs, or take corrective measures. One way to handle overflow is with a saturation unit. The function of the saturation unit is to correct an overflow to the maximum positive value in the case of a positive overflow, and to correct an overflow to the maximum negative value in the case of a negative overflow.

In the ACX_DSP_GEN block, an optional saturation block is available. Additionally, the bit position where the saturation is to be performed may be specified. This allows the saturation function to accommodate word widths less than the full 64 bit precision of the add/sub block.

To enable saturation, the 6-bit `sat_mode` parameter must be set to a non-zero value. The value of the `sat_mode` parameter sets the position of the msb where the rounding is to occur. For example, if the location of the msb is at the fifteenth bit (zero relative) in the add/sub block, the `sat_mode` parameter should be set to `6'h0F`.

Pre-Adder Block

The ACX_DSP_GEN block contains a 27-bit pre-adder block used to halve the number of required multipliers. The pre-adder may be configured as either an adder or subtracter so it may support either symmetric or asymmetric filter coefficients. The pre-adder may take data from either the A and B inputs to the ACX_DSP_GEN block, or the forward and reverse cascades buses. The output of the pre-adder block can be routed to the either the 19-bit or 27-bit input of the multiplier. Care should be taken to limit the input data to the pre-adder so that the resultant sum is within the resolution of the multiple input. For example, if the pre-adder output is routed to the 27-bit input of the multiplier, the input to the pre-adder should contain 26-bit data sign extended into the 27th bit so as to prevent an overflow condition at the output of the pre-adder. If using the output of the pre-adder to drive the 19-bit input of the adder, the full resolution of the 18-bit data may be used, as there is a 19-bit into the multiplier to accommodate a carry.

ACX_DSP_GEN Verilog Instantiation Template

The recommended method of including the ACX_DSP_GEN in a design is by inference (see ACX_DSP_GEN Verilog Inference Template (see page 138)). However, if the full range of ACX_DSP_GEN functions are required, or a function that cannot be fully inferred is needed, the ACX_DSP_GEN can be directly instantiated.

The Verilog instantiation template is shown in the following example.

ACX_DSP_GEN Verilog Instantiation Template

```
ACX_DSP_GEN
#(
    .init_a                    (18'h0),
    .init_b                    (27'h0),
    .init_sub                (1'b0),
    .init_cin                (1'b0),
    .init_load                (1'b0),
    .init_rnd                (1'b0),
    .init_mshift            (1'b0),
    .init_dout                (64'h0),
    .init_cout                (1'b0),
    .rst_value_a            (18'h0),
    .rst_value_b            (27'h0),
    .rst_value_sub            (1'b0),
    .rst_value_cin            (1'b0),
    .rst_value_load            (1'b0),
    .rst_value_rnd            (1'b0),
    .rst_value_mshift        (1'b0),
    .rst_value_dout            (64'h0),
    .rst_value_cout            (1'b0),
    .regce_priority_a        ("regce"),
    .regce_priority_b        ("regce"),
    .regce_priority_sub        ("regce"),
    .regce_priority_cin        ("regce"),
    .regce_priority_load    ("regce"),
    .regce_priority_rnd        ("regce"),
    .regce_priority_mshift    ("regce"),
    .regce_priority_dout    ("regce"),
    .a_del                    (1'b0),
    .b_del                    (1'b0),
    .sub_del                (1'b0),
    .cin_del                (1'b0),
    .load_del                (1'b0),
    .rnd_del                (1'b0),
    .mshift_del                (1'b0),
    .dout_del                (1'b0),
    .match_del                (1'b0),
    .preadd_del                (1'b0),
    .multout_del            (1'b0),
    .addsub_areg_del        (1'b0),
    .regaddr_del            (1'b0),
    .fwdi_casc_del            (1'b0),
    .fwdo_casc_del            (1'b0),
    .revi_casc_del            (1'b0),
    .addsub_bypass            (1'b0),
    .sel_addsub_a            (2'b00),
    .sel_addsub_b            (1'b0)
    .sel_cin                (1'b0),
    .sel_fwdo_dout             (2'b00),
    .sel_fwdo_cout             (1'b0),
    .sel_dout                 (2'b00),
    .sel_48_dout             (1'b0),
    .sel_revi_casc             (1'b0),
    .sel_fwd_preadd             (2'b00),
    .sel_rev_preadd             (2'b00),
    .sel_mult_a                 (2'b00),
    .sel_mult_b                 (2'b00),
    .preadd_mode             (2'b0),
    .round_mode                 (3'b000),
    .sat_mode                 (6'h00),
    .use_match_in             (1'b0),
    .match_pattern             (64'h0),
    .match_mask                 (64'h0),
    .round_const             (64'h0),
    .load_const                 (64'h0),
    .regfile_0                 (27'h0),
    .regfile_1                 (27'h0),
    .regfile_2                 (27'h0),
    .regfile_3                 (27'h0),
    .regfile_4                 (27'h0),
    .regfile_5                 (27'h0),
    .regfile_6                 (27'h0),
    .regfile_7                 (27'h0)
) instance_name (
    .clk                     (user_clk),
    .a                         (user_a),
    .b                         (user_b),
    .sub                     (user_sub),
    .cin                     (user_cin),
    .load                     (user_load),
    .rnd                     (user_rnd),
    .mshift                     (user_mshift),
    .reg_addr                 (user_reg_addr),
    .ce_a                     (user_ce_a),
    .ce_b                     (user_ce_b),
    .ce_addsub                 (user_ce_addsub),
    .ce_addsub_a             (user_ce_addsub_a),
    .ce_dout                 (user_ce_dout),
    .ce_cascade                 (user_ce_cascade),
    .ce_multout                 (user_ce_multout),
    .rstn_a                     (user_rstn_a),
    .rstn_b                     (user_rstn_b),
    .rstn_addsub             (user_rstn_addsub),
    .rstn_addsub_a             (user_rstn_addsub_a),
    .rstn_dout                 (user_rstn_dout),
    .rstn_cascade             (user_rstn_cascade),
    .rstn_multout             (user_rstn_multout),
    .dout                     (user_dout),
    .cout                     (user_cout),
    .over_pos                 (user_over_pos),
    .over_neg                 (user_over_neg),
    .match                     (user_match),
    .fwdi_casc                 (user_fwdi_casc),
    .fwdi_dout                 (user_fwdi_dout),
    .fwdi_cin                 (user_fwdi_cin),
    .fwdi_match                 (user_fwdi_match),
    .revi_casc                 (user_revi_casc),
    .revi_dout                (user_revi_dout),
    .fwdo_casc                (user_fwdo_casc),
    .fwdo_dout                (user_fwdo_dout),
    .fwdo_cout                (user_fwdo_cout),
    .fwdo_match                (user_fwdo_match),
    .revo_casc                (user_revo_casc),
    .revo_dout                (user_revo_dout)
);
```

ACX_DSP_GEN Verilog Inference Template

Synplify Pro supports both direct instantiation of the Speedcore ACX_DSP_GEN block as well as inferencing with specific code structures. Inferencing occurs for multiplication functions of up to 36 × 27 bits. For addition and subtraction functions, fabric logic is inferred. For direct instantiation of a DSP64, see ACX_DSP_GEN Verilog Instantiation Template (see page 136).

In addition, the ACE IP generator supports the generation of multiple DSP-based math functions which can then be directly instantiated within the user code (see the *ACE User Guide* (UG070) for details.

The following example shows how to correctly infer an 18 × 18 bit multiplier to be mapped to an ACX_DSP_GEN block.

Inferred Multiplier Example

```
`timescale 1ps/1ps
module inferred_mult_18x18_signed (ina, inb, multout, clk);
localparam a_width = 18;
localparam b_width = 18;
localparam prod_width = a_width + b_width;
input                          clk;
input  signed [a_width-1:0]    ina;
input  signed [b_width-1:0]    inb;
output signed [prod_width-1:0] multout;
reg    signed [a_width-1:0]    ina_reg = 18'h0;
reg    signed [b_width-1:0]    inb_reg = 18'h0;
reg    signed [prod_width-1:0] multout = 36'h0;
always @(posedge clk)
begin
  ina_reg <= ina;
  inb_reg <= inb;
  multout <= ina_reg * inb_reg;
end
endmodule
```

Implementing Finite Impulse Response (FIR) Filters

A finite impulse response (FIR) filter is implemented as a sum-of-products of the form:

$$y(n) = \sum_{i=0}^{i=(n-1)} x(i) \times c(i)$$

6586423_19/11/2022

Figure 60: ***Generic FIR Filter Equation***

Parallel Filter Implementation

The following figure shows the block diagram of a direct FIR filter implementation using a parallel adder tree. While this is functionally correct, it uses more resources than necessary.

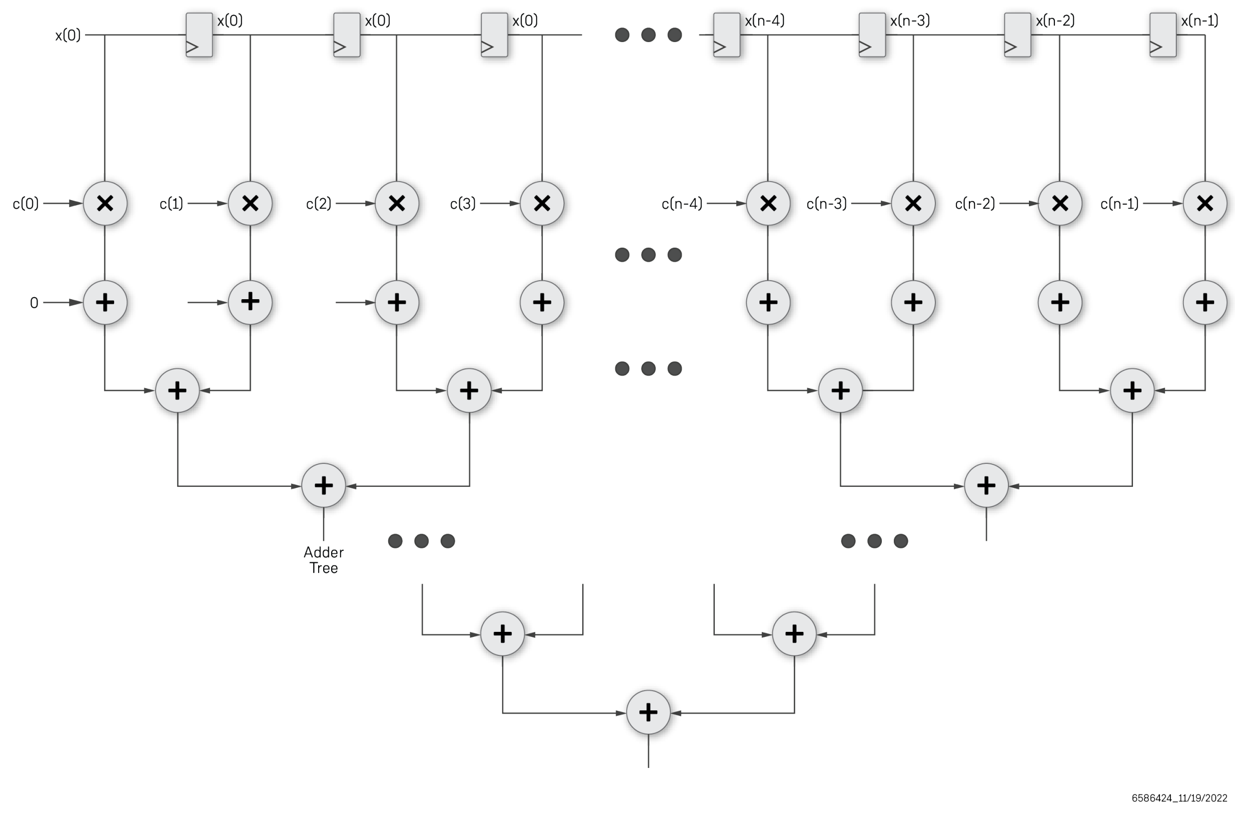


Figure 61: ***FIR Filter Block Diagram Using a Parallel Adder Tree***

The following figure shows the FIR filter implemented with a serial adder tree. While this implementation is more resource efficient, it suffers from poor performance due to the serial chain of adders.

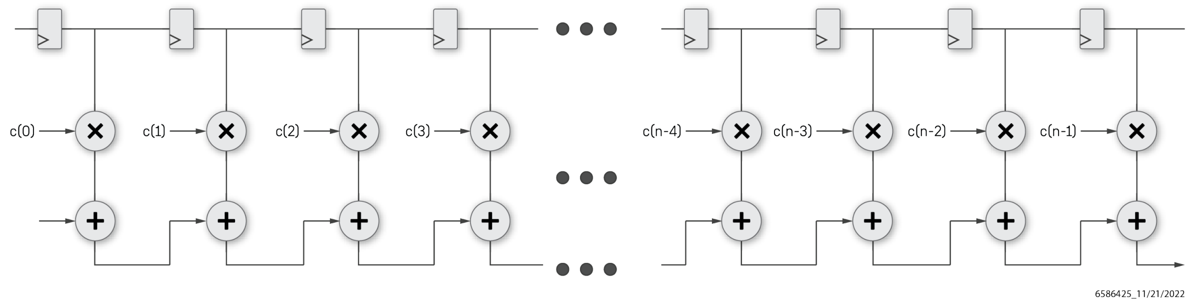


Figure 62: ***FIR Filter Block Diagram Using a Serial Adder Tree***

As shown in the following figure, A pipeline register can be added at the last filter tap without changing the functionality. An additional cycle of latency is added for this pipeline stage.

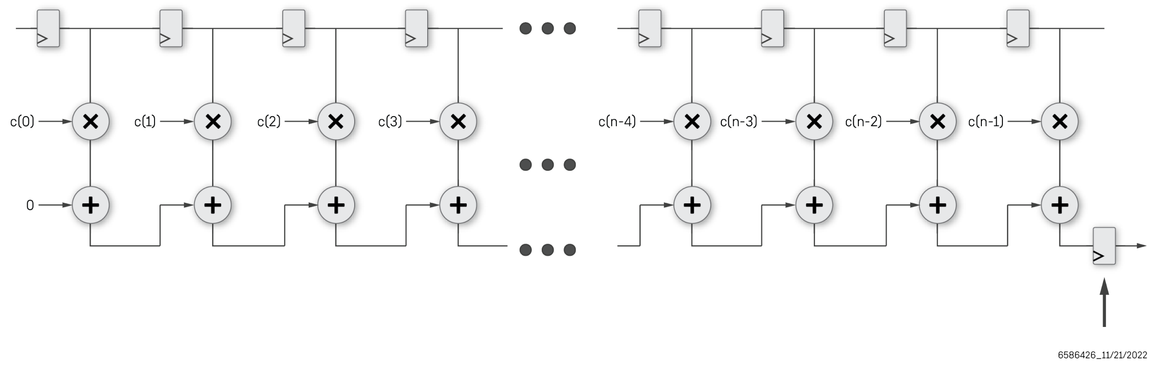


Figure 63: ***FIR Filter Block Diagram With Pipelined Output***

Performance can be improved further by adding a pair of pipeline registers at the last stage of the FIR filter as shown in the following figure. A pipeline register added at the input to the last adder must be matched by a pipeline register in front of the last multiplier to maintain proper functionality. An additional cycle of latency is added for this register pair.

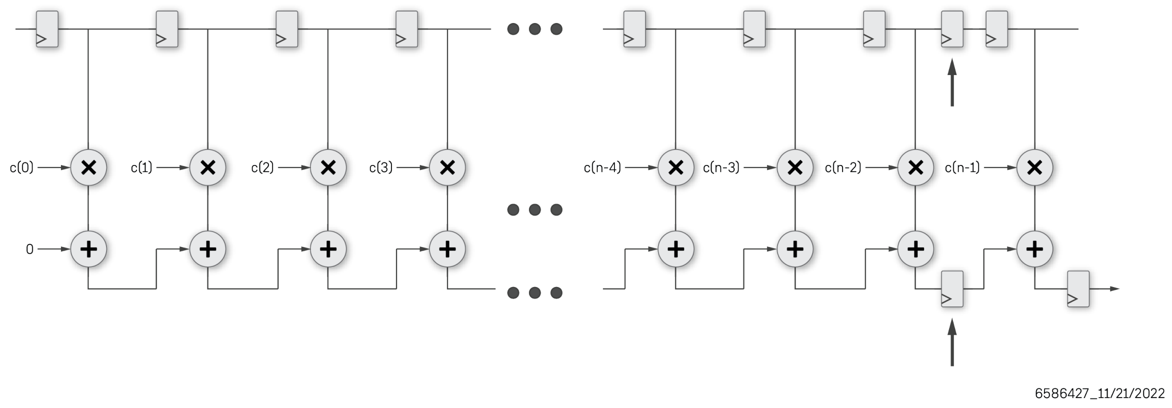


Figure 64: ***FIR Filter Block Diagram With Additional Pipeline Registers at the Last Stage***

Likewise, a pipeline register pair may be added at each stage for additional performance. The following figure shows the FIR filter with pipeline registers at the input of the adder and the multiplier at each stage. An additional cycle of latency is added for each pair of added pipeline registers. For lower latency, optionally pipeline every other stage and trade lower latency at the output versus higher performance.

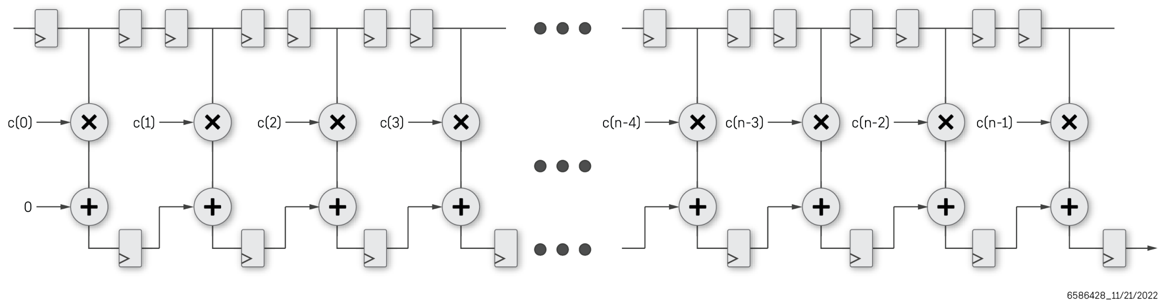


Figure 65: ***FIR Filter Block Diagram With Pipeline Registers at Each Stage***

For further timing improvements, choose to add a pipeline stage at the output of the multipliers to separate the multiplications and the additions by a cycle. If this option is selected, pipeline registers must be added to all stages of the FIR filter as shown in the following figure.

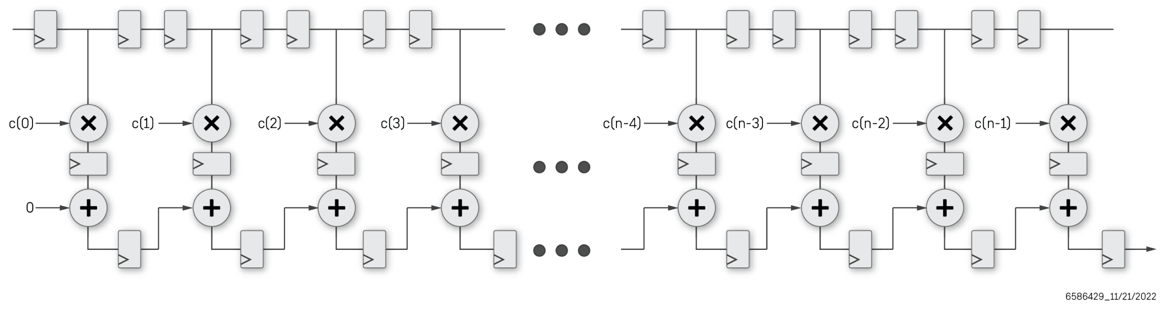


Figure 66: ***FIR Filter Block Diagram With Pipelined Multiplier Outputs***

Symmetric FIR Filter Implementation

FIR filters, in which the coefficients for the first half of the filter are symmetric to the coefficients of the second half of the filter, are said to be symmetric. There are four variants to the symmetric filter, depending on whether the filter coefficients are symmetric versus anti-symmetric, or whether the length of the filter is odd or even:

1. Odd-length symmetric impulse response filters
2. Odd-length, anti-symmetric impulse response filters
3. Even-length symmetric impulse response filters
4. Even-length, anti-symmetric impulse response filters

The following four subsections detail the four variants of symmetric FIR filters.

Odd-Length Symmetric Impulse Response FIR Filters

An odd-length symmetric FIR filter has the first half of the filters coefficients equal to the second half of the filter coefficients with the following relationship. For a filter with n (n odd) filter taps:

- coefficient(n−1) = coefficient(0)
- coefficient(n−2) = coefficient(1)
- coefficient(n−3) = coefficient(2)

⋮

- coefficient(((n−1)/2)−3) = coefficient(((n−1)/2) + 3)
- coefficient(((n−1)/2)−2) = coefficient(((n−1)/2) + 2)
- coefficient(((n−1)/2)−1) = coefficient(((n−1)/2) + 1)
- coefficient((n−1)/2) (middle tap) does not have a paired coefficient

The multipliers for the filter taps with equivalent coefficients may be shared if the corresponding data inputs for the taps are added before the multiplier.

- (coef.(0) × data(0)) + (coef.(0) × data(n−1)) = (data(0) + data(n−1)) × coef.(0)

This optimization halves the number of multipliers required to implement a FIR filter.

The figure below shows an unoptimized block diagram of a odd-length symmetric FIR filter. The following discussion and figures illustrate how to optimize the performance.

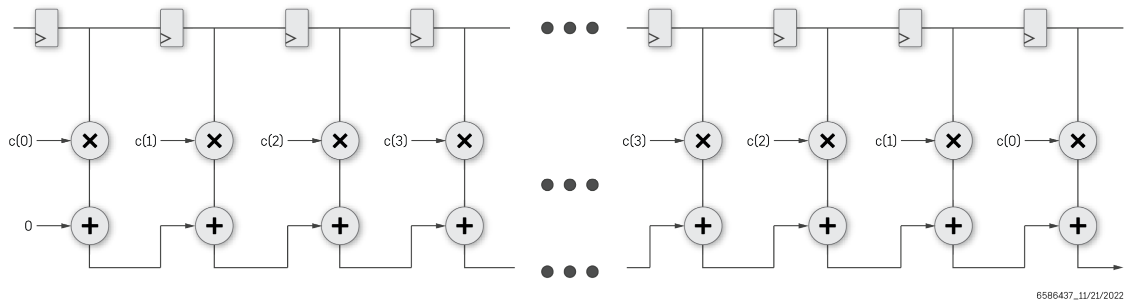


Figure 67: ***Odd-Length Symmetric FIR Filter Block Diagram***

A pre-adder is then added to the input of each multiplier with the second input of the pre-adder tied to zero. The structure shown in the following figure maintains the functionality of the FIR filter.

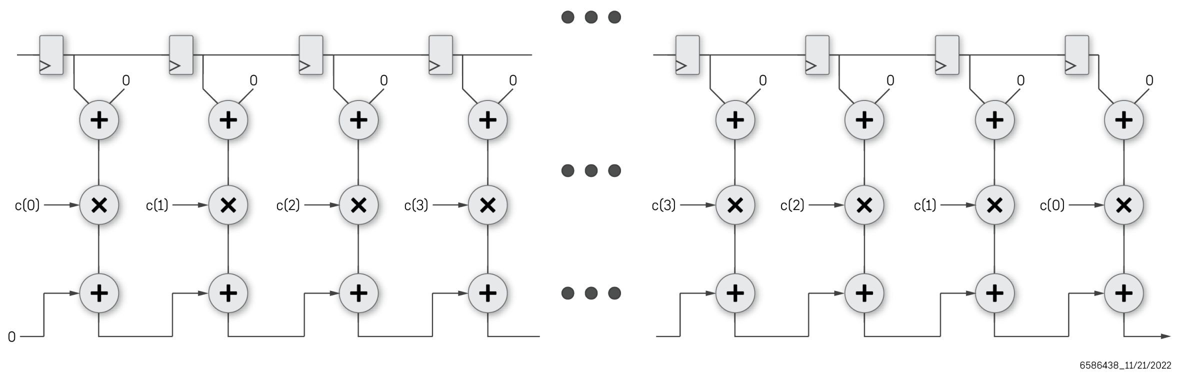


Figure 68: ***Odd-Length Symmetric FIR Filter with Pre-Adder Block Diagram***

The required number of multipliers are halved (excluding the unpaired center tap) if the shift register path is folded back and the filter taps with equivalent coefficients are connected to the second input of the pre-adder as shown in the following figure. The FIR filter maintains the functionality of the unfolded version.

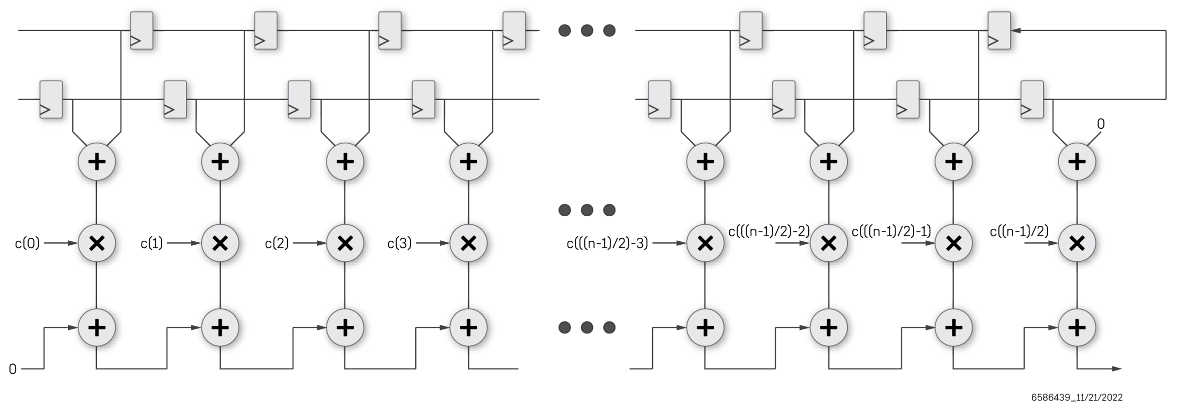


Figure 69: ***Odd-Length Symmetric FIR Filter Block With Folded Back Datapath***

The current structure of the FIR filter suffers from poor performance due to the serial chain of adders. Pipeline registers must be added to the adder chain. A pair of pipeline registers can be added in the forward path at the second-to-last filter tap to improve timing. In order to maintain functionality, one of the registers in the reverse delay line must be removed as shown in the following figure.

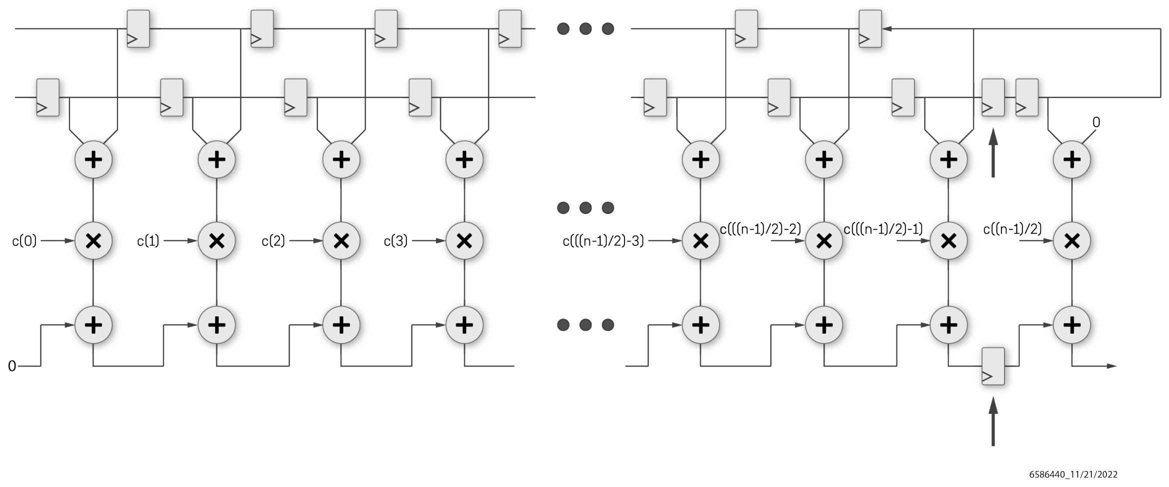


Figure 70: ***Odd-Length Symmetric FIR Filter Block With Single Pipeline Stage Added***

Likewise, an additional pipeline stage pair is added in the forward path two stages earlier. For each additional pipeline stage, a register must be removed from the backwards datapath. Pipeline stages should not be added at each stage of the folded symmetric FIR filter or the reverse datapath becomes unregistered and performance suffers. A balance must be maintained between adding pipeline stages in the forward datapath and removing pipelines stages in the reverse datapath.

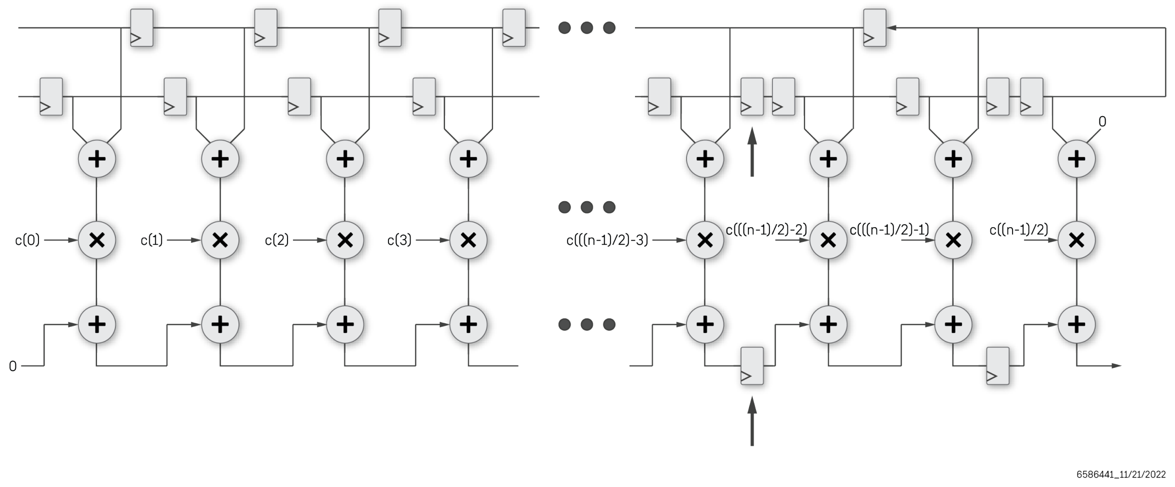


Figure 71: ***Odd-Length Symmetric FIR Filter Block With Additional Pipeline Stage Added***

To achieve a balance between adding pipeline stages in the forward datapath and removing pipeline stages in the reverse datapath, pipeline stages have been added every two stages as shown in the following figure.

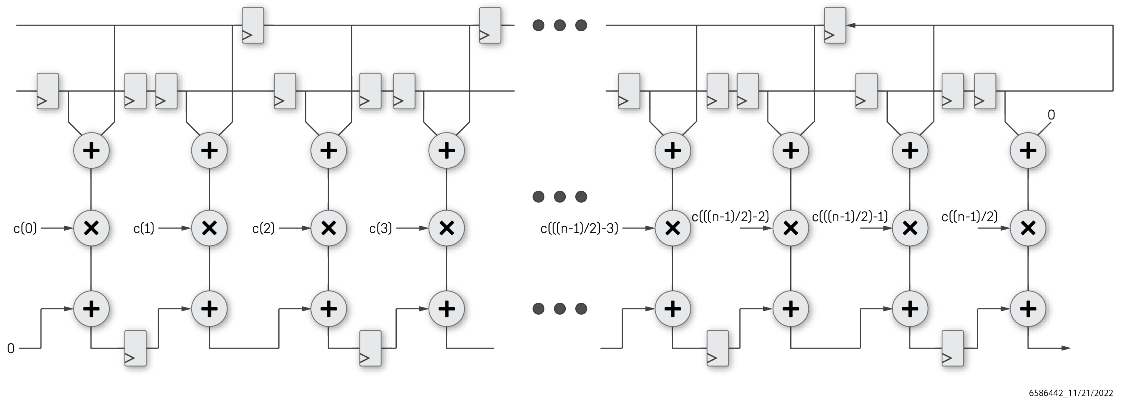


Figure 72: ***Odd-Length Symmetric FIR Filter Block with Every Other Stage Pipelined***

Further pipelining can be achieved by adding pipeline registers at the output of the multipliers and/or the output of the pre-adders. If pipelining at the output of the multipliers and/or the pre-adders is used, it must be performed at every stage of the filter to maintain proper functionality. The use of these pipeline registers is shown in the following figure.

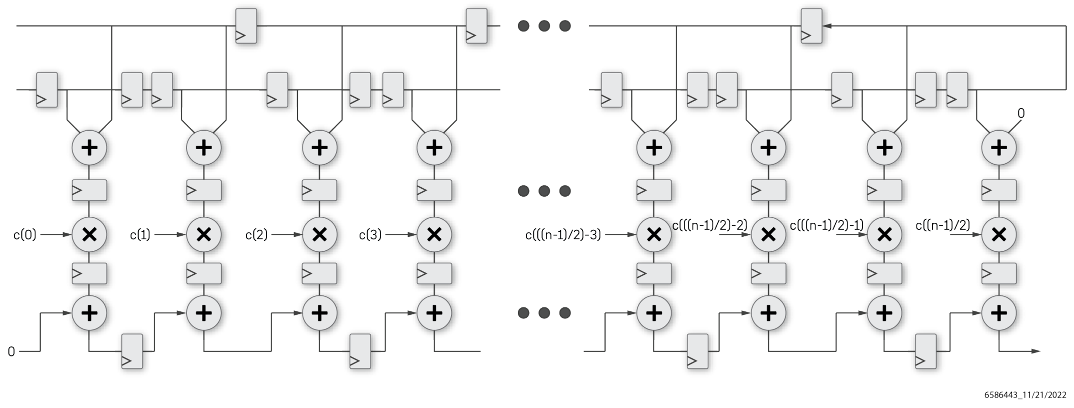


Figure 73: ***Odd-Length Symmetric FIR Filter With Pipelined Pre-Adders and Multipliers***

Odd-Length, Anti-Symmetric Impulse Response FIR Filters

An odd-length anti-symmetric FIR filter has the first half of the filter coefficients related to the second half of the filter coefficients with the following relationship. For a filter with n (n odd) filter taps:

- coefficient(n−1) = −1 × coefficient(0)
- coefficient(n−2) = −1 × coefficient(1)
- coefficient(n−3) = −1 × coefficient(2)

⋮

- coefficient(((n−1)/2)−3) = −1 × coefficient(((n−1)/2) + 3)
- coefficient(((n−1)/2)−2) = −1 × coefficient(((n−1)/2) + 2)
- coefficient(((n−1)/2)−1) = −1 × coefficient(((n−1)/2) + 1)
- coefficient((n−1)/2) (middle tap) does not have a paired coefficient

The multipliers for the filter taps with equivalent coefficients may be shared if the corresponding data inputs for the taps are added before the multiplier.

- (coef.(0) × data(0)) + (−coef.(0) × data(n−1)) = (data(0) − data(n−1)) × coef.(0)

This optimization has the same structure as an odd-length symmetric filter with the pre-adder replaced with a pre-subtracter. Pre-subtracting the data before the multiplier halves the number of multipliers required to implement a FIR filter. The final structure of the optimized even-length anti-symmetric FIR filter is shown in the following figure. Again, this structure is identical to that of the even-length symmetric FIR filter with the pre-adder replaced with a pre-subtracter.

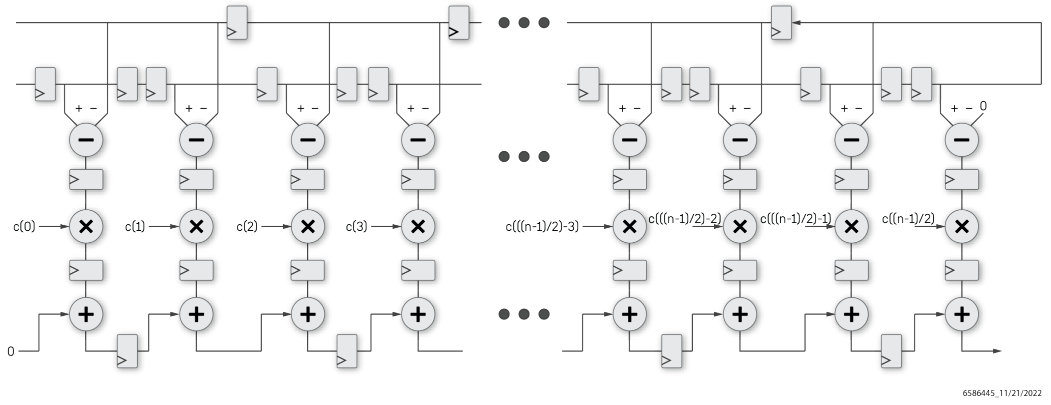


Figure 74: *Odd-Length Anti-Symmetric FIR Filter With Pipelined Pre-Adders and Multipliers*

Even-Length Symmetric Impulse Response FIR Filters

An even-length symmetric FIR filter has the first half of the filter coefficients equal to the second half of the filter coefficients with the following relationship. For a filter with n (n even) filter taps:

- coefficient(n−1) = coefficient(0)
- coefficient(n−2) = coefficient(1)
- coefficient(n−3) = coefficient(2)

⋮

- coefficient((n/2)−3) = coefficient((n/2) + 2)
- coefficient((n/2)−2) = coefficient((n/2) + 1)
- coefficient((n/2)−1) = coefficient((n/2))

The multipliers for the filter taps with equivalent coefficients may be shared if the corresponding data inputs for the taps are added before the multiplier.

- (coef.(0) × data(0)) + (coef.(0) × data(n−1)) = (data(0) + data(n−1)) × coef.(0)

This optimization halves the number of multipliers required to implement a FIR filter.

The following figure shows an unoptimized block diagram of an even-length symmetric FIR filter. The following discussion and figures illustrate how to optimize the performance.

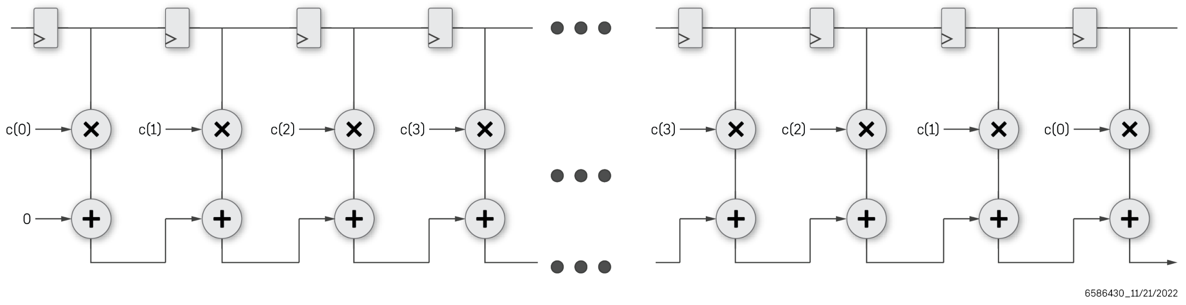


Figure 75: ***Even-Length Symmetric FIR Filter Block Diagram***

A pre-adder is then added to the input of each multiplier, with the second input of the pre-adder tied to zero. This structure, shown in the following figure, maintains the functionality of the FIR filter.

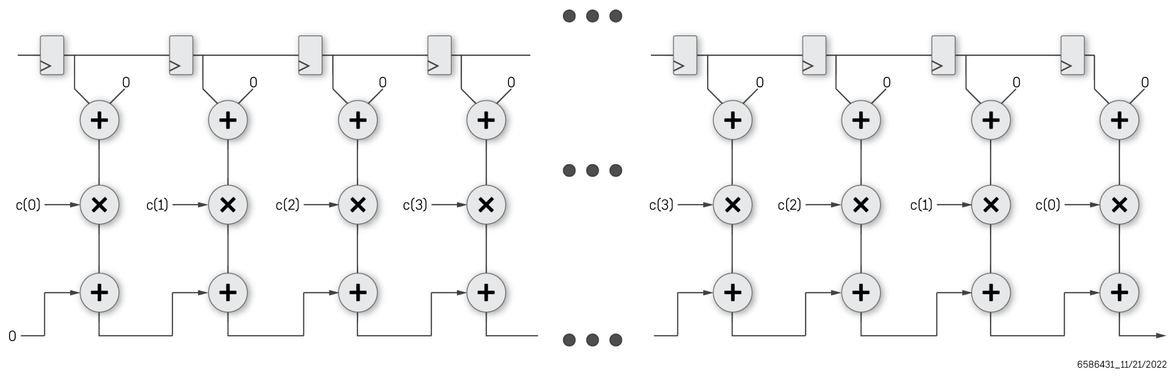


Figure 76: ***Even-Length Symmetric FIR Filter With Pre-Adder Block Diagram***

The required number of multipliers is halved if the shift register path is folded back and the filter taps with equivalent coefficients are connected to the second input of the pre-adder as shown in the following figure. The FIR filter maintains the functionality of the unfolded version.

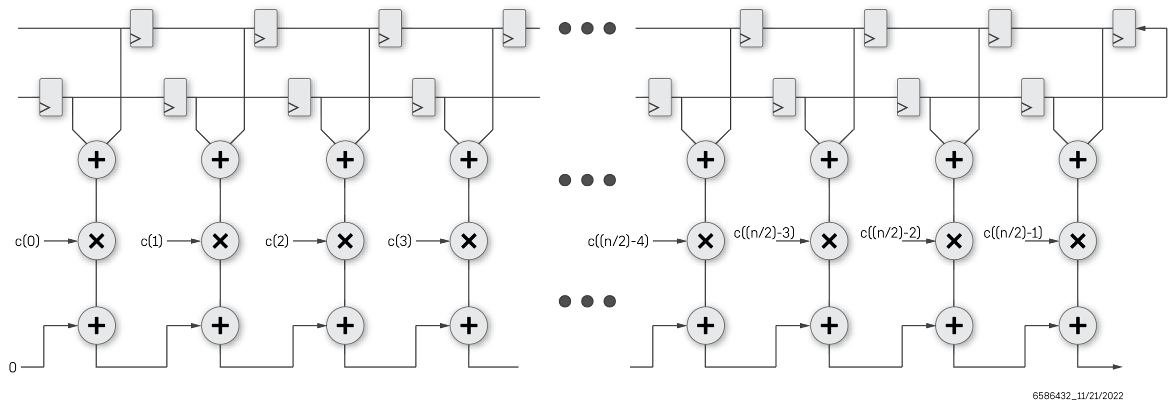


Figure 77: *Even-Length Symmetric FIR Filter Block With Folded Back Datapath*

The current structure of the FIR filter suffers from poor performance due to the serial chain of adders. Pipeline registers must be added to the adder chain. A pair of pipeline registers can be added in the forward path at the last filter tap to improve timing. In order to maintain functionality, one of the registers in the reverse delay line must be removed as shown in the following figure.

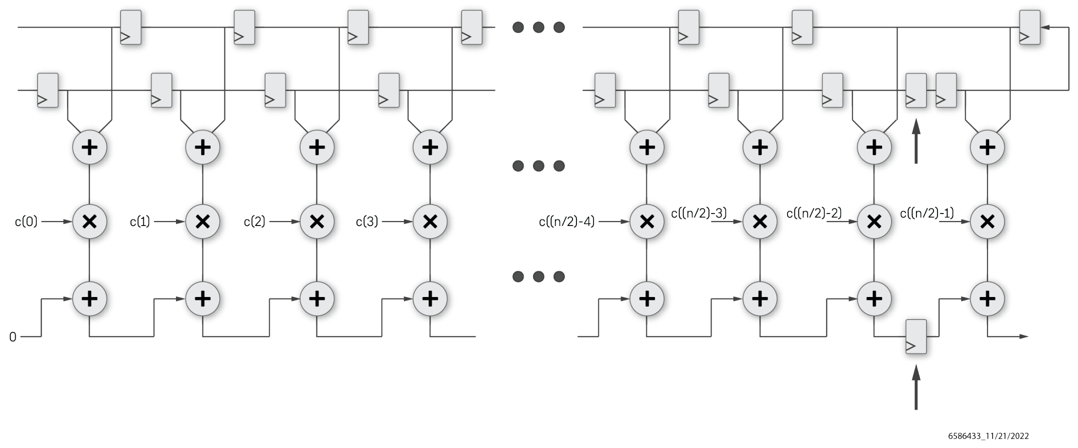


Figure 78: *Even-Length Symmetric FIR Filter Block With Single Pipeline Stage Added*

Likewise, an additional pipeline stage pair is added in the forward path, two stages earlier. For each additional pipeline stage, a register must be removed from the backwards datapath. Pipeline stages should not be added at each stage of the folded symmetric FIR filter or the reverse datapath becomes unregistered and performance suffers. A balance must be maintained between adding pipeline stages in the forward datapath and removing pipeline stages in the reverse datapath.

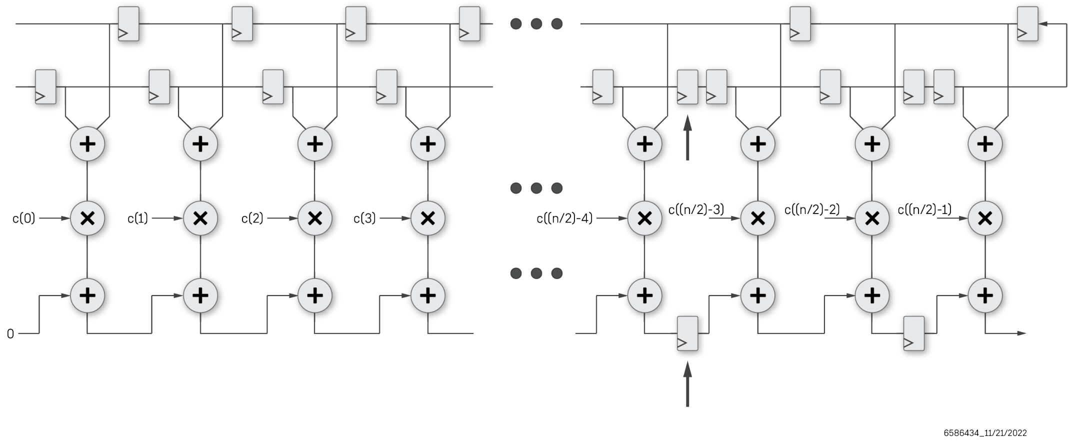


Figure 79: ***Even-Length Symmetric FIR Filter Block With Additional Pipeline Stage Added***

To achieve a balance between adding pipeline stages in the forward datapath and removing pipeline stages in the reverse datapath, pipeline stages have been added every two stages as shown below in in the following figure.

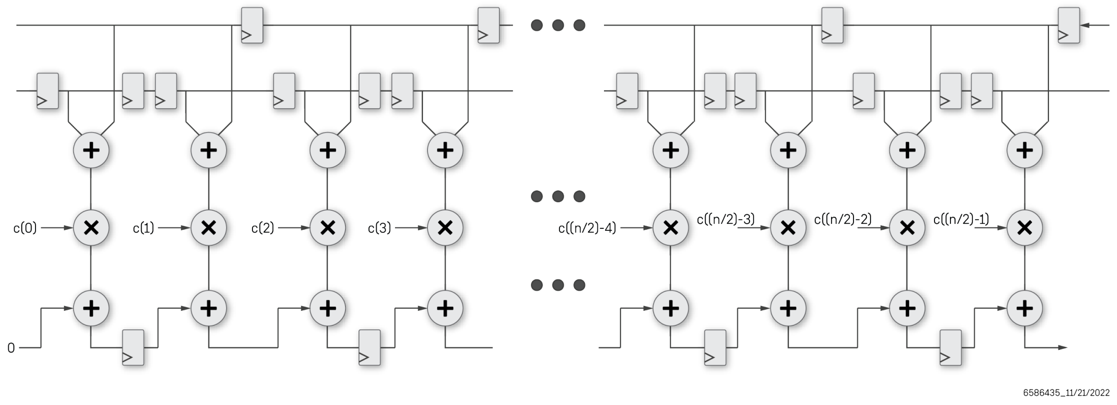


Figure 80: ***Even-Length Symmetric FIR Filter Block Mith Every Other Stage Pipelined***

Further pipelining can be achieved by adding pipeline registers at the output of the multipliers and/or the output of the pre-adders. If pipelining at the output of the multipliers and/or the pre-adders is used, it must be performed at every stage of the filter to maintain proper functionality. The use of these pipeline registers is shown in the following figure.

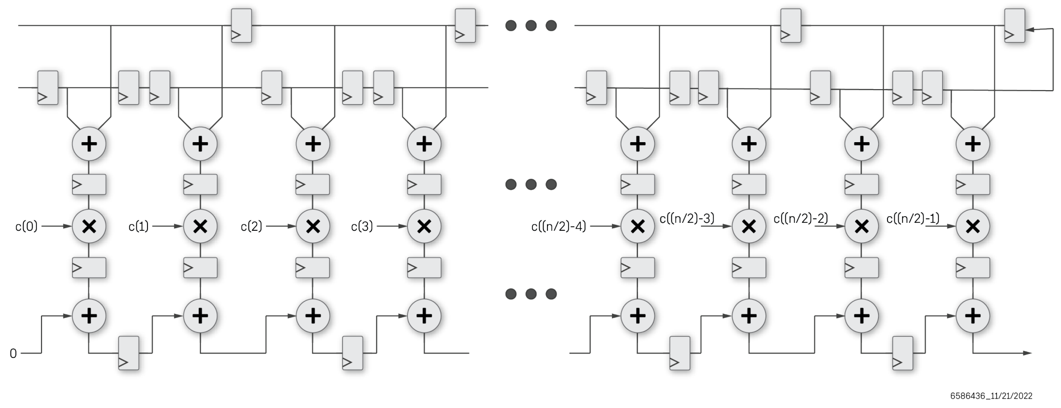


Figure 81: ***Even-Length Symmetric FIR Filter with Pipelined Pre-Adders and Multipliers***

Even-Length, Anti-Symmetric Impulse Response FIR Filters

An even-length anti-symmetric FIR filter has the first half of the filter coefficients related to the second half of the filter coefficients with the following relationship. For a filter with n (n even) filter taps:

- coefficient(n−1) = −1 × coefficient(0)
- coefficient(n−2) = −1 × coefficient(1)
- coefficient(n−3) = −1 × coefficient(2)

⋮

- coefficient((n/2)−3) = −1 × coefficient((n/2) + 2)
- coefficient((n/2)−2) = −1 × coefficient((n/2) + 1)
- coefficient((n/2)−1) = −1 × coefficient((n/2))

The multipliers for the filter taps with equivalent coefficients may be shared if the corresponding data inputs for the taps are subtracted before the multiplier.

- (coef.(0) × data(0)) + (−coef.(0) × data(n−1)) = (data(0) − data(n−1)) × coef.(0)

This optimization has the same structure as an even-length symmetric filter with the pre-adder replaced with a pre-subtracter. Presubtracting the data before the multiplier halves the number of multipliers is required to implement an FIR filter.

The final structure of the optimized even-length anti-symmetric FIR filter is shown in the following figure. This structure is identical to that of the even-length symmetric FIR filter with the pre-adder replaced with a pre-subtracter.

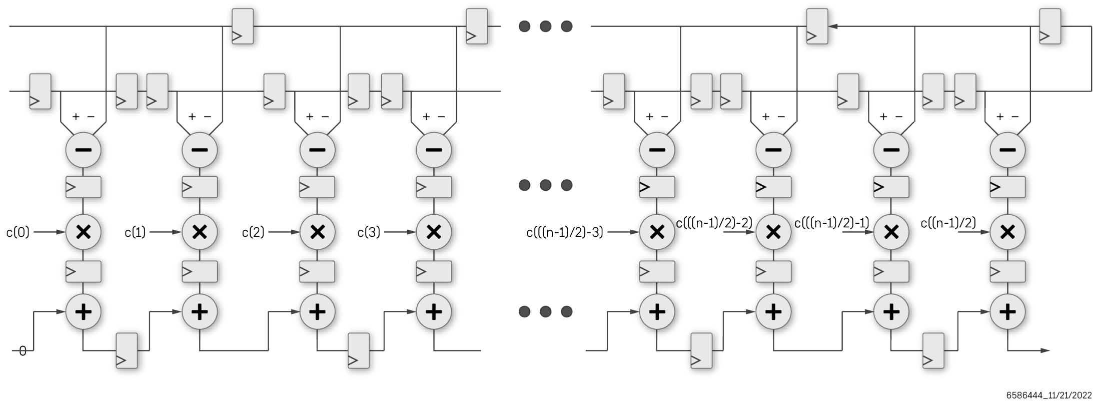


Figure 82: *Even-Length Anti-Symmetric FIR Filter With Pipelined Pre-Adders and Multipliers*

ACX_DSP_MACC_GEN

The ACX_DSP_MACC_GEN macro provides a multiply-accumulate function with an optional C input to be added to the output. The macro supports multiplication of A × B, where A and B can either be 27 × 18 or 27 × 26. The C input is 48 bits. The macro provides all multiplication and accumulate options up to and including 36 × 27 + 48-bit C with 64-bit accumulator.

Table 107: *ACX_DSP_MACC_GEN Macro Parameters*

Parameter	Values	Default	Description
`A_INPUT_WIDTH`	18 or 36	18	Width of the A input to the multiplier.
`REGISTER_INPUTS`	On/Off	Off	Optionally register the A and B inputs, adding one cycle of latency to the result.
`REGISTER_MULT`	On/Off	Off	Register the output of the multiplier, adding one cycle of latency to the result. Registering the output may be required when operating at high target frequencies.
`C_INPUT_48B`	On/Off	Off	Add a 48-bit C input, this value is added to the output.
`ACC_64_OUTPUT`	On/Off	Off	When enabled, the output accumulates, performing the function dout = dout + (A × B + C). When disabled, the function is dout = A × B + C.

The following table shows the number of DSP blocks consumed by the ACX_DSP_MACC_GEN macro based on the provided parameters.

Table 108: *ACX_DSP_MACC_GEN DSP Block Usage Based On Parameter Values*

Parameter	Value	Number of DSP	Value	Number of DSP
`A_INPUT_WIDTH`	18	1	36	2
`C_INPUT_48B`	Off	0	On	1
`ACC_64_OUTPUT`	Off	0	On	1

In the maximum configuration of 36 × 27 + 48-bit C with 64-bit accumulator, the macro uses 4 DSP blocks.

Table 109: *ACX_DSP_MACC_GEN Macro Ports*

Port Name	Direction	Description
`clk`	In	DSP clock.
`reset_n`	In	DSP reset. When `reset_n` is set to `1'b0`, `dout` = 0; When `reset_n` = `1'b1`, the DSP operates as follows.
`a_in[n:0]`	In	Signed A input, either 18 or 36 bits. This input is multiplied by the B input.
`b_in[26:0]`	In	Signed B input. This input is multiplied by the A input.
`c_in[47:0]`	In	Optional signed 48-bit C input. When present, this input is added to the multiplication of A × B, resulting in A × B + C.
`dout[63:0]`	Out	Signed sum or accumulation of A × B + C.

Timing

The output timing from ACX_DSP_MACC_GEN is dependent upon the input parameters. The parameters `REGISTER_INPUTS` and `REGISTER_MULT` each add an extra stage of latency to the result. These delays are shown in the following timing diagrams:

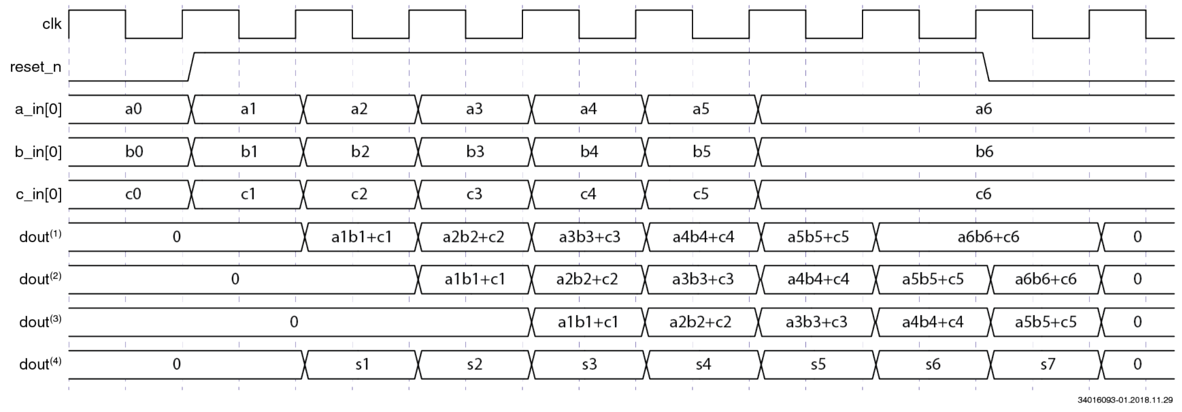


Figure 83: *DSP MACC Timing Diagram*

Note

The `dout` values shown in the timing diagram illustrate the following parameter settings:

`dout` (1) : Not registered.

`dout` (2) : `REGISTER_INPUTS` applied.

`dout` (3) : `REGISTER_INPUTS` and `REGISTER_MULT` applied.

`dout` (4) : `ACC_64_OUTPUT` applied with no registers.

ACX_DSP_ACCUMULATOR_GEN

The ACX_DSP_ACCUMULATOR_GEN macro provides for an input of up to 192 bits, which are successively accumulated. The macro uses one DSP for each 48 bits of output.

Table 110: *ACX_DSP_ACCUMULATOR_GEN Parameters*

Parameter	Values	Default	Description
`ACC_WIDTH`	32 to 192	96	Specifies the width of the data input and accumulator output.

Table 111: *ACX_DSP_ACCUMULATOR_GEN Ports*

Name	Direction	Description
`clk`	In	DSP clock.
`reset_n`	In	DSP reset. When `reset_n` is set to `1'b0`, `dout` = 0; When `reset_n` = `1'b1`, the DSP operates as follows.
`acc_enable`	In	When `acc_enable` is set to `1'b1`, the DSP accumulates `dout` with `acc_value`. When `acc_enable` is set to `1'b0`, `dout` remains constant.
`load`	In	When `load` is set to `1'b1`, the value on `acc_value[n:0]` is loaded into the DSP, and one cycle later `dout[n:0]` is set to `acc_value`. When `load` is set to `1'b0`, the DSP operates in accumulate mode. The `load` input is independent of, and has priority over `acc_enable`.
`acc_value[n:0]`	In	Unsigned load value. When `load` is set to `1'b1`, `acc_value` is used to load the DSP with an initial value.
`dout[n:0]`	Out	Unsigned result of the DSP accumulation.

Timing

The timing for the DSP accumulator is shown in the following figure.

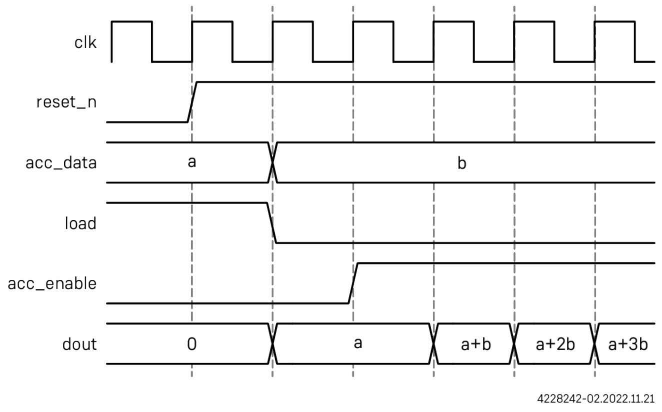


Figure 84: ***ACX_DSP_ACCUMULATOR_GEN Timing Diagram***

ACX_DSP_COUNTER_GEN

The ACX_DSP_COUNTER_GEN macro provides a counter of up to 192 bits. The macro uses one DSP for each 48 bits of output.

Table 112: ***ACX_DSP_COUNTER_GEN Macro Parameters***

Parameter	Values	Default	Description
`COUNTER_WIDTH`	32 to 192	96	Specifies the width of the data input and accumulator output.

Table 113: ***ACX_DSP_COUNTER_GEN Macro Ports***

Name	Direction	Description
`clk`	In	DSP clock.
`reset_n`	In	DSP reset. When `reset_n` is set to `1'b0`, `dout` = 0; When `reset_n` = `1'b1`, the DSP operates as follows.
`count_enable`	In	When `count_enable` is set to `1'b1`, the DSP performs a rising count on `dout`. When `count_enable` is set to `1'b0`, `dout` remains constant.
`load`	In	When `load` is set to `1'b1`, the value on `load_value[n:0]` is loaded into the DSP, and one cycle later `dout[n:0]` is set to `load_value[n:0]`. When `load` is set to `1'b0`, the DSP operates as a counter. The `load` input is independent of, and has priority over `count_enable`.
`load_value[n:0]`	In	Unsigned load value. When `load` is set to `1'b1`, `load_value` is used to load the DSP with an initial value.
`dout[n:0]`	Out	Unsigned result of the DSP count.

Timing

The following figure shows the timing diagram for the DSP counter.

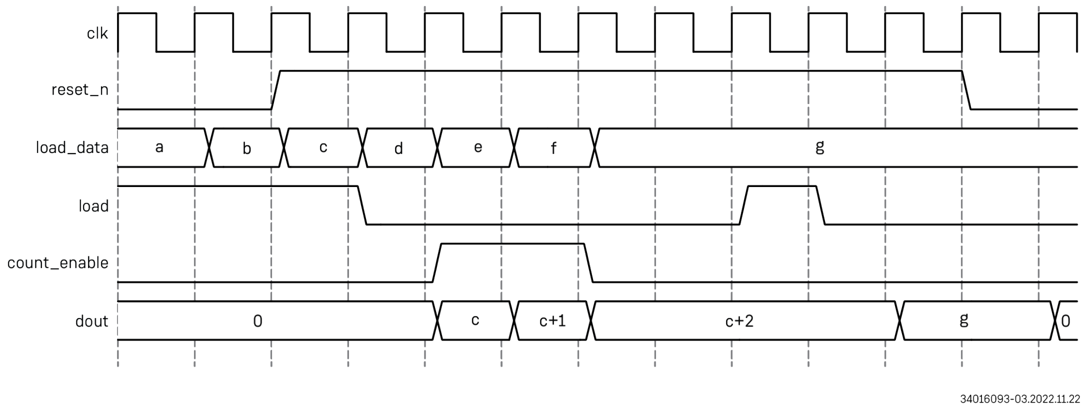


Figure 85: *DSP Counter Timing Diagram*

ACX_DSP_SUM_SQUARES_GEN

The ACX_DSP_SUM_SQUARES_GEN macro provides for N sum of squares function: $(A \pm B)^2$ where the A and B inputs can be up to 18 bits each and N can be up to 4 pairs of values. This macro consumes one DSP for each (A ±B) pair of inputs.

Table 114: ***ACX_DSP_SUM_SQUARES_GEN Macro Parameters***

Name	Values	Default	Description
`NUM_INPUT_PAIRS`	1 to 4	2	Specifies the number of A and B input pairs.
`REGISTER_AB_INPUTS`	On/Off	Off	Optionally register the A and B inputs, adding one cycle of latency to the result.
`REGISTER_MULT`	On/Off	Off	Register the output of the multiplier, adding one cycle of latency to the result. Registering may be required when operating at high target frequencies.
`ADD_SUB_N`	(A+B), (A-B), (B-A)	(A+B)	Each pair of inputs may be summed or subtracted from one another. This addition or subtraction occurs before the result is squared.

Table 115: ***ACX_DSP_SUM_SQUARES_GEN Macro Ports***

Name	Direction	Description
`clk`	In	DSP clock.
`reset_n`	In	DSP reset. When `reset_n` is set to `1'b0`, `dout` = 0; When `reset_n` = `1'b1`, the DSP operates as follows.
`a_in`	In	Array of signed A inputs. The definition of the input is [17:0] A `[NUM_INPUT_PAIRS-1:0]`. Each A input is 18 bits wide. A value must be supplied for each A input in the array.
`b_in`	In	Array of signed B inputs. The definition of the input is [17:0] B `[NUM_INPUT_PAIRS-1:0]`. Each B input is 18 bits wide. A value must be supplied for each B input in the array.
`dout[47:0]`	Out	Signed 48-bit sum of each pair of $(A \pm B)^2$.

Timing

The following figure shows the timing diagram for ACX_DSP_SUM_SQUARES_GEN.

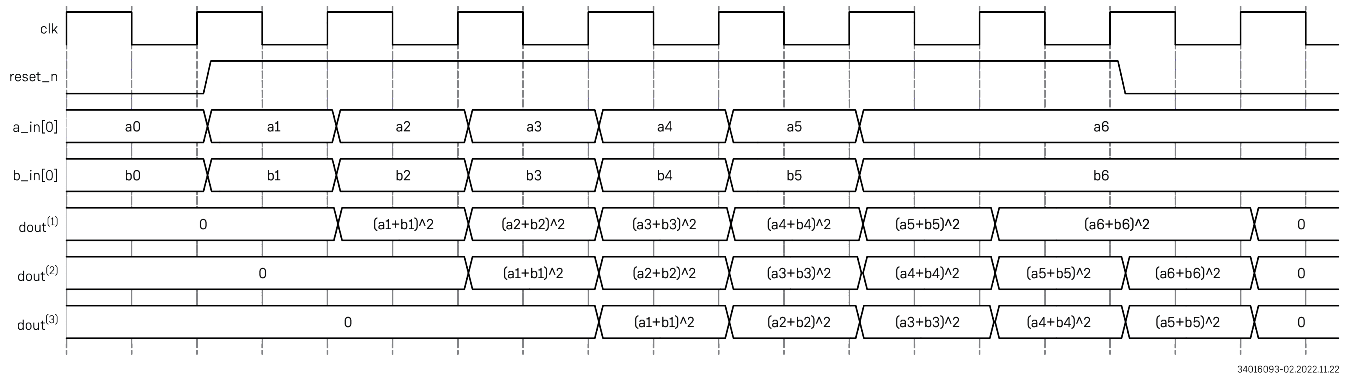


Figure 86: ***ACX_DSP_SUM_SQUARES_GEN Timing Diagram***

Note

The `dout` values shown in the timing diagram illustrate the following parameter settings:

`dout` (1) : Not registered.

`dout` (2) : `REGISTER_AB_INPUTS` applied.

`dout` (3) : `REGISTER_AB_INPUTS` and `REGISTER_MULT` applied.

Enabling each one of these parameters adds one cycle of latency to the output.

ACX_MLP72

Arithmetic within the Speedster7t architecture is primarily focused on the machine learning processing block (ACX_MLP72). This dedicated silicon block is optimized for artificial intelligence and machine learning (AI/ML) functions.

The machine learning processor block (MLP) is an array of up to 32 multipliers, followed by an adder tree, and an accumulator. The MLP is also tightly coupled with two memory blocks, a BRAM72k and LRAM2k. These memories can be used individually or in conjunction with the array of multipliers. The number of multipliers available varies with the bit width of each operand and the total width of input data. When the MLP is used in conjunction with a BRAM72k, the number of data inputs to the MLP block increases, enabling the use of additional multipliers.

The MLP offers a range of features:

- Configurable multiply precision and multiplier count. Any of the following modes are available:
  - Up to 32 multiplies for 4-bit integers or 4-bit block floating-point values in a single MLP
  - Up to 16 multiplies for 8-bit integers or 8-bit block floating-point values in a single MLP
  - Up to 4 multiplies for 16-bit integers in a single MLP
  - Up to 2 multiplies for 16-bit floating point with both 5-bit and 8-bit exponents in a single MLP
  - Up to 2 multiplies for 24-bit floating point in a single MLP
- Multiple number formats:
  - Integer
  - Floating point 16 (including B float 16)
  - Floating point 24
  - Block floating point, a method that combines the efficiency of the integer multiplier-adder tree with the range of the floating point accumulators
- Adder tree and accumulator block
- Tightly-coupled register file (LRAM) with an optional sequence controller for easily caching and feeding back results
- Tightly-coupled BRAM for reusable input data such as kernels or weights
- Cascade paths up a column of MLPs
  - Allows for broadcast of operands up a column of MLPs without using up critical routing resources
  - Allows for adder trees to extend across multiple MLPs
  - Broadcast read/write to tightly-coupled BRAMs up a column of MLPs to efficiently create large memories

Along with the numerous multiply configurations, the MLP block includes optional input and pipelining registers at various locations to support high-frequency designs. There is a deep adder tree after the multipliers with the option to bypass the adders and output the multiplier products directly. In addition, a feedback path allows for accumulation within the MLP block.

The following block diagrams show the MLP using the fixed or floating-point formats:

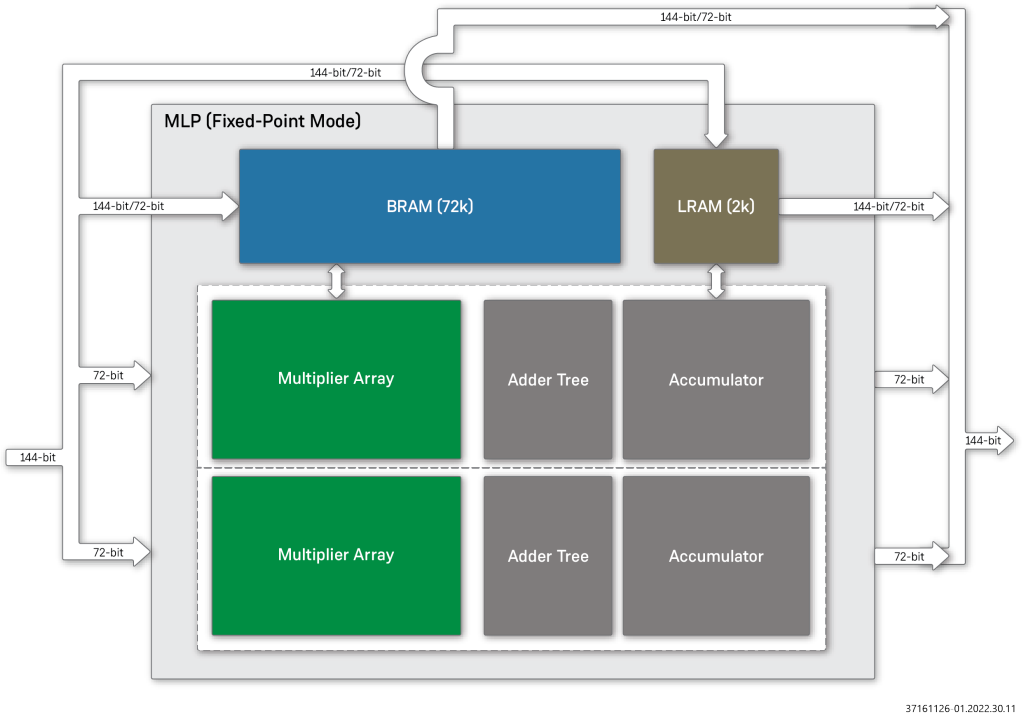


Figure 87: ***MLP Using Fixed-Point Mode***

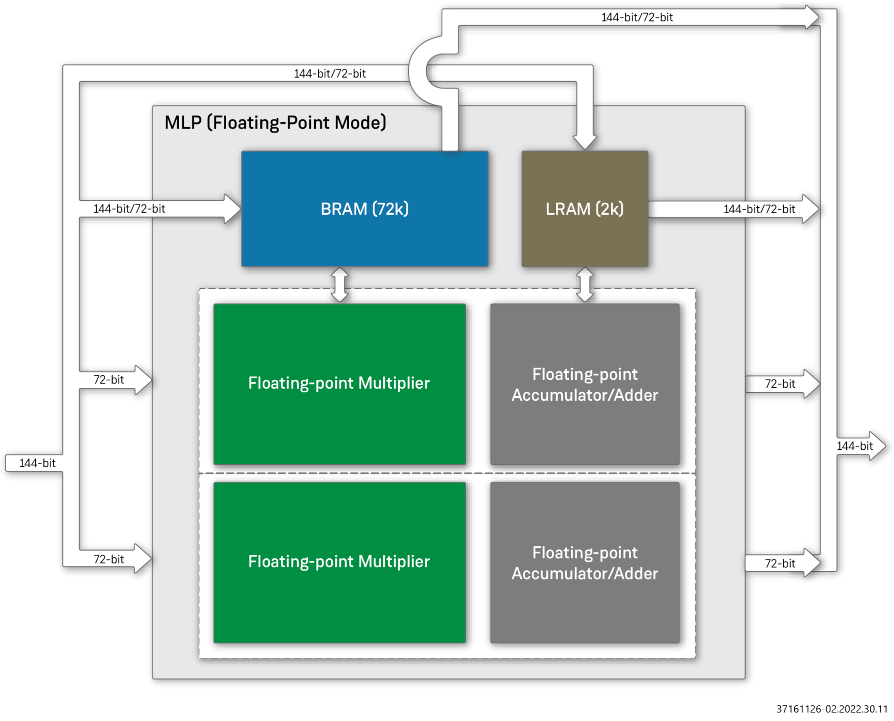

144-bit/72-bit
144-bit/72-bit
MLP (Floating-Point Mode)
144-bit/72-bit
BRAM (72k)
LRAM (2k)
144-bit/72-bit
Floating-point Multiplier
Floating-point Accumulator/Adder
72-bit
72-bit
144-bit
144-bit
Floating-point Multiplier
Floating-point Accumulator/Adder
72-bit
72-bit
37161126-02.2022.30.11

Figure 88: ***MLP Using Floating-Point Mode***

A powerful feature available in the Achronix MLP is the ability to connect several MLPs with dedicated high-speed cascade paths. The cascade paths allow for the adder tree to extend across multiple MLP blocks in a column without using extra fabric routing resources, and a data cascade/broadcast path is available to send operands across multiple MLP blocks. Cascading input or result data to multiple MLPs in parallel allows for complex, multi-element operations to be performed efficiently without the need for extra routing. The following diagram shows the cascade paths across MLPs:

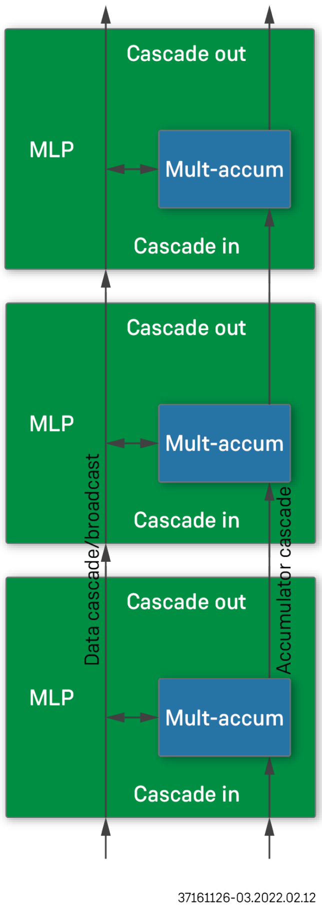


Figure 89: *MLP Cascade Path*

Note

Straight addition within the ACX_MLP72 (without a leading multiplication) is not supported.

Numerical Formats

The ACX_MLP72 can process the following numerical formats:

Table 116: *ACX_MLP72 Supported Numerical Formats*

	Formats
Integer	int3, int4, int6, int7, int8, int16
Block floating point	BFP Int3, BFP Int4, BFP Int6, BFP Int7, BFP Int8, BFP Int16
Floating point	fp3, fp4, fp6, fp8, fp16, fp16e8, fp24.

See Speedster7t MLP Number Formats for details of each of the numerical formats.

Parallel Multiplications

The following table lists the maximum number of parallel multiplies that are supported in the ACX_MLP72 as a function of the data type, and the input mode. The input modes specify from where the data input to the MLP is sourced and are described in the section Modes.

For block floating-point operations, the bit width shown is the mantissa width.

Table 117: *Parallel Multiplication Capabilities*

Data Type	x1 Mode Inputs only from FPGA Fabric	x2 Mode Inputs from FPGA Fabric and Coupled BRAM Input	x4 Mode Inputs from FPGA Fabric and Coupled BRAM Output
Integer			
Int3	12	24	32
Int4	8	16	32
Int6	6	12	16
Int7	5	10	16
Int8	4	8	16
Int16	2	4	4 (1)
Block Floating Point			
Exponents (2)	2	4/2	4
BFP Int3	10	16/20	32
BFP Int4	8	12/16	32

Data Type	x1 Mode Inputs only from FPGA Fabric	x2 Mode Inputs from FPGA Fabric and Coupled BRAM Input	x4 Mode Inputs from FPGA Fabric and Coupled BRAM Output
BFP Int6	5	8/10	16
BFP Int7	4	8/9	16
BFP Int8	4	6/8	16
BFP Int16	2	4	4 (1)
Floating Point			
fp16	1	2	2 (1)
fp16e8	1	2	2 (1)
fp24	1	2	2 (1)

Table Notes

1. The number of multiplications is limited by the available hardware multipliers, and can be achieved by using x2 input mode.
2. With x2 input mode, the number of block floating point exponents can be either 2 or 4. Using only 2 exponents allows for a greater number of mantissas to be input to the MLP, resulting in a greater number of parallel multiplications.

Memories

A key feature of the ACX_MLP72 is its tight coupling with local memories. Each ACX_MLP72 is grouped with a ACX_BRAM72K and a ACX_LRAM2K at a single silicon site. In addition to the normal fabric I/O, the ACX_MLP72, ACX_BRAM72K and the ACX_LRAM2K are also connected by dedicated, non-fabric paths. This tight coupling supports 144-bit paths between the elements, with deterministic timing, allowing full-speed operation of all multipliers operating in parallel.

This arrangement allows for efficient processing by storing input data that is reused (such as a convolution kernel or weights) and by storing results in a register file to allow for efficient burst transfers to external memory stores or other processing blocks. Using this architecture, it is possible to construct highly efficient matrix vector multiplication, 2D convolution and dot product processes that maximize the functionality of the ACX_MLP72 and its tightly-coupled memories.

Instantiation

Currently it is not possible to infer a full ACX_MLP72. In addition, due to the complexity of the full ACX_MLP72, Achronix supports, and recommends, the use of the ACX_MLP72 via libraries of macros and primitive functions derived from the full ACX_MLP72. These libraries enable implementing complex mathematical functions, all within a single block, via a simplified interface. The provided libraries include support for integer and floating-point functions.

For particular use cases not covered by the libraries of ACX_MLP72 macros and primitives, details of the full ACX_MLP72 are provided. Refer to Achronix reference designs for further examples of direct instantiation of a full ACX_MLP72.

Common Stages

Stages

Due to the complexity of the ACX_MLP72, the details that follow, including tables of parameters and ports, have been divided up into various stages. Each of these stages represents a functional stage within the ACX_MLP72, whether that be input selection or multiplier configuration. The stages are described in signal flow order, beginning with common signals and input selection, and proceeding through the multiplier stages to the output routing. Understand each stage thoroughly before configuring it via the various parameters.

The initial overview of the full ACX_MLP72 structure focuses on the integer modes. This overview details:

- Common Signals (see page 165)
- Input Selection (see page 167)
- Integer Byte Selection
- Integer Multiplier Stage
- Integer Output Stage
- LRAM

When familiar with the overall ACX_MLP72 integer structure and data flow, additional sections are provided on floating-point support:

- Block Floating Point
- Floating Point

Symmetrical Structure

In general terms, the functions of the MLP72 can be divided into two halves: upper and lower (also referred to as "ab" and "cd"). For the purposes of clarity, a number of the block diagrams which follow only show one half of the ACX_MLP72. In these cases, unless indicated otherwise, it can be assumed that the other half operates in an identical manner.

Modes

Operation of the ACX_MLP72 is commonly categorized into three operating modes, each of which reflects the number of multipliers in use, and the necessary routing of the inputs in order to supply the multipliers. The number of multipliers given in the following definitions refers to 8 bit multiplication; when 16 bit or 4 bit values are used, these values halve or double respectively.

- By-one mode (×1) – just the four multipliers in the lower half of the ACX_MLP72, mult[3:0], are in use. This requires the A and B input buses to each have 32 bits of data, or for a single input source to have 64 bits of data. Therefore any of the available input sources can be switched to these four multipliers, and it is possible to provide all the multiplier inputs from a single data source.
- By-two mode (×2) – all eight of the multipliers in the lower half of the ACX_MLP72, mult[7:0], are in use. This requires each of the A and B input buses to have 64 bits of data, so at least two of the input sources are required. In addition there are some x2 split modes whereby four of the multipliers from the lower half, and four of the multipliers from the top half of the ACX_MLP72 are used.

- By-four mode (×4) – all 16 multipliers in the ACX_MLP72 are in use. This requires two A and B input buses, each with 64 bits of data, resulting in the combined A and B input buses each having 128 bits of data. To achieve this, one of the advanced routing techniques is required. The most common method is to provide one of the 128 bit buses from the coupled ACX_BRAM72K output, and then to input the other 128 bus split between the normal MLP input and the BRAM input (each 72 bits). Methods for routing data in ×4 mode are discussed in the *Speedster7t Machine Learning Processing User Guide* (UG088).

Common Signals

There are a number of signals and parameters that are common to multiple sections of the ACX_MLP72. These common signals are primarily for controlling delay stages throughout the ACX_MLP72.

Between each functional stage there are optional registers, known as delay stages. These can be optionally enabled (using the `del_xx` parameter). If enabled, their clock enable and negative resets can be connected to any one of a common set of `ce[]` and `rstn[]` inputs. The `cesel_xx` and `rstsel_xx` parameters respectively control which of the `ce[11:0]` and `rstn[3:0]` inputs are connected to the selected delay stage. Further, for certain delay stages it is possible to control whether the reset is synchronous or asynchronous using the appropriate `rst_mode_xx` parameter.

These optional delay stages all follow the same structure as shown in the following figure.

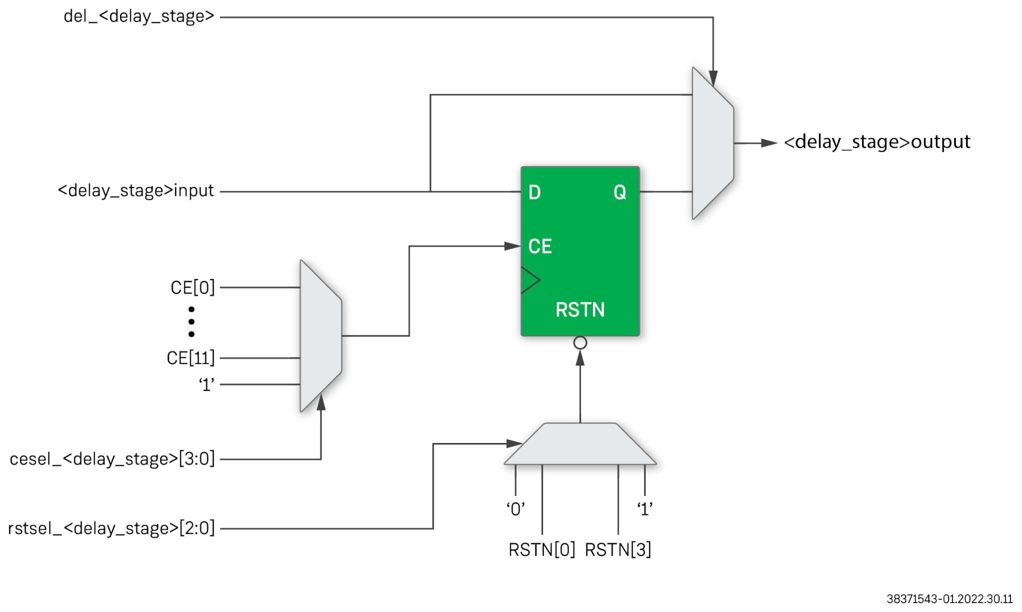


Figure 90: *Delay Stage Structure*

In the diagrams which follow, showing the various stages of the MLP72, the delay stages are shown as a register with a dotted outline indicating that they are optionally selected to be in circuit. The parameters for each delay stage are then shown in the dashed box alongside the register symbol. This representation is shown in the following figure.

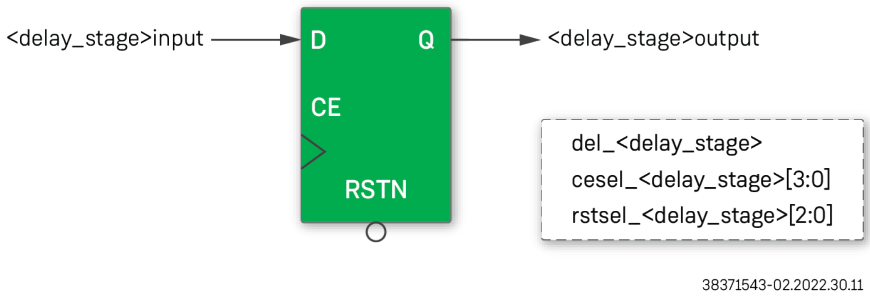


Figure 91: *Delay Stage Symbol*

Parameters

Table 118: *Common Parameters*

Parameter	Supported Values	Default Value	Description
`clk_polarity`	"rise", "fall"	"rise"	Specifies whether the registers are clocked by the rising or the falling edge of the clock.
`cesel_*[3:0]`	`4'b0000`–`4'b1101`	Must be set (0–13)	Selects the `ce` inputs for each delay stage register: `4'b0000` – `1'b0`. `4'b0001` – `ce[0]`. `4'b0010` – `ce[1]`. `4'b0011` – `ce[2]`. `4'b0100` – `ce[3]`. `4'b0101` – `ce[4]`. `4'b0110` – `ce[5]`. `4'b0111` – `ce[6]`. `4'b1000` – `ce[7]`. `4'b1001` – `ce[8]`. `4'b1010` – `ce[9]`. `4'b1011` – `ce[10]`. `4'b1100` – `ce[11]`. `4'b1101` – `1'b1`.

`rstsel_*[2:0]`	`3'b000`–`3'b101`	Must be set (0–5)	Selects the `rstn` input for each delay stage register:
			`3'b000` – `1'b0.`
			`3'b001` – `rstn[0].`
			`3'b010` – `rstn[1].`
			`3'b011` – `rstn[2].`
			`3'b100` – `rstn[3].`
			`3'b101` – `1'b1.`

`rst_mode_*`	`1'b0`–`1'b1`	`1'b0`	Selects the reset mode (clocked vs. unclocked) for each delay stage register: `1'b0` – synchronous reset mode. `1'b1` – asynchronous reset mode.

Parameter	Supported Values	Default Value	Description
`del_*`	`1'b0–1'b1`	`1'b0`	Selects if each delay stage register is enabled or bypassed: `1'b0` – delay stage register is bypassed. `1'b1` – delay stage register is enabled.

Ports

Table 119: *Common Ports*

Name	Direction	Description
`clk`	Input	Clock input. If input or output registers are enabled, they are updated on the active edge of this clock.
`ce[11:0]`	Input	Set of clock enable signals for delay stage registers. Asserting the clock enable signal for a delay stage register causes it to capture that data at it's input on the rising edge of `clk`. Has no effect when the register is disabled.
`rstn[3:0]`	Input	Set of negative reset signals for the delay stage registers. When the reset signal for a delay register stage is asserted (`1'b0`), a value of 0 is written to the output of that register on the rising edge of `clk`. Has no effect when the register is disabled.
`dft_0`	Input	Reserved for Achronix internal use. Must be left unconnected.
`dft_1`	Input	Reserved for Achronix internal use. Must be left unconnected.
`dft_2`	Input	Reserved for Achronix internal use. Must be left unconnected.

Input Selection

The ACX_MLP72 can accept inputs from a wide variety of sources. The purpose of the input selection block is to select from these sources, and generate four internal data buses. These four buses are then divided into byte lanes (byte is used as a generic term, the lanes are not necessarily 8 bits, the width is applicable to the selected number format). These byte lanes are then sent to the two banks of multipliers (high and low), with each bank consisting of 8 multipliers, and each multiplier having an A and B input.

The selected internal data buses are also output to the cascade paths so that they can be used by adjacent ACX_MLP72s in the same column.

The internal data buses, and their respective input selection are notated as `multX_Y`, where:

- X = A or B to indicate whether the bus is for the A or B input of the respective multipliers
- Y = H or L to indicate whether the bus is for the High or Low set of multipliers.

The buses are therefore named as `multa_l`, `multb_l`, `multa_h`, `multb_h`.

The high bank of multipliers has a wider selection of input data buses (8) than the low bank (4). This is shown in the following diagrams.

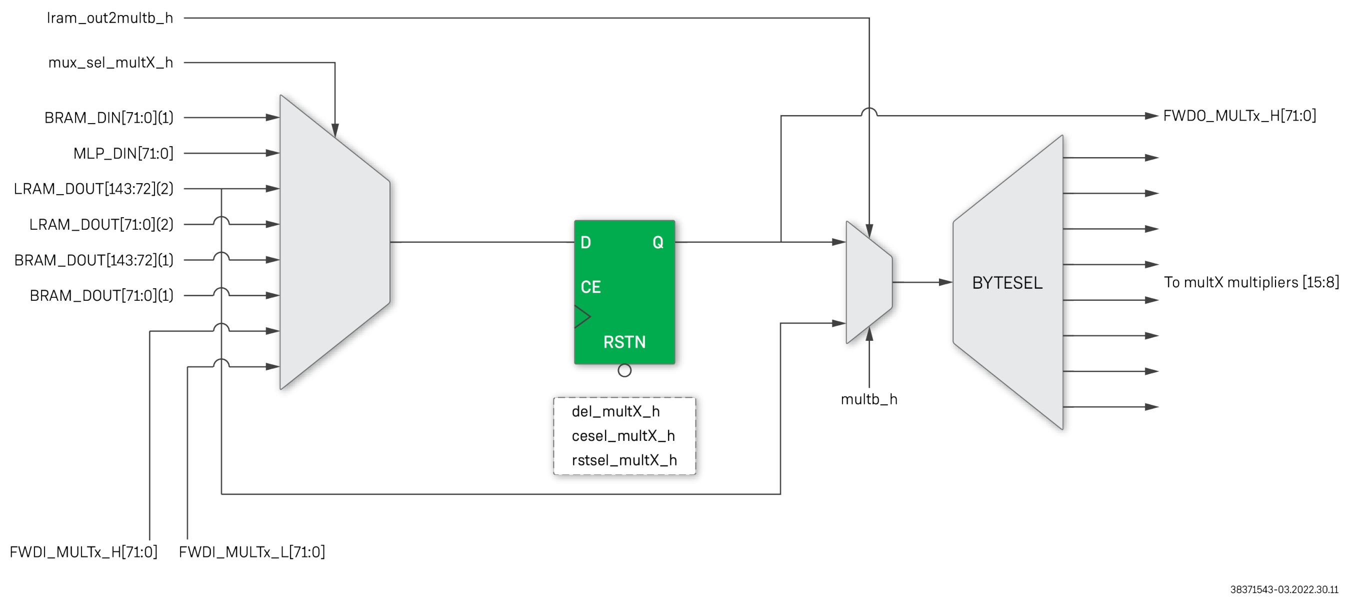


Figure 92: *High Bank Multipliers Input Selection*

Note

The noted MUX inputs in the preceding diagram have the following conditions:

1. `BRAM_DIN[71:0]` and `BRAM_DOUT[143:0]` are logical names for the respective signal paths. The physical port names vary, and are listed in the following Ports table.
2. `LRAM_DOUT[143:0]` is an internal connection only from the coupled LRAM. This is not available as an input port on the MLP.

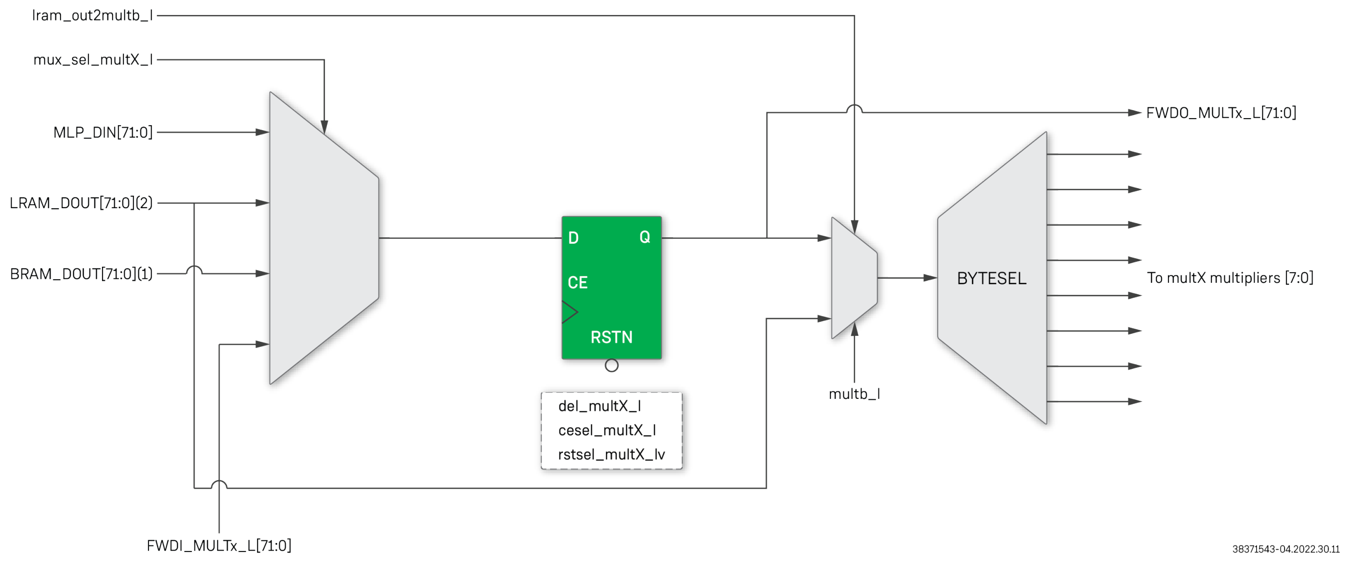


Figure 93: ***Low Bank Multipliers Input Selection***

Note

The noted MUX inputs in the preceding diagram have the following conditions:

1. BRAM_DIN[71:0] and BRAM_DOUT[143:0] are logical names for the respective signal paths. The physical port names vary, and are listed in the following Ports table.
2. LRAM_DOUT[143:0] is an internal connection only from the coupled LRAM. This is not available as an input port on the MLP.

Parameters

Table 120: ***Input Selection Parameters***

Parameter	Supported Values	Default Value	Description
`mux_sel_multa_h[2:0]`	`3'b000`–`3'b111`	`3'b000`	`3'b000` – `MLP_DIN[71:0]`. `3'b001` – `BRAM_DIN[71:0]`. `3'b010` – `LRAM_DOUT[71:0]`. (1) `3'b011` – `LRAM_DOUT[143:72]`. (1) `3'b100` – `BRAM_DOUT[71:0]`. (1) `3'b101` – `BRAM_DOUT[143:72]`. (2) `3'b110` – `FWDI_MULTA_L[71:0]`. `3'b111` – `FWDI_MULTA_H[71:0]`.
`mux_sel_multa_l[1:0]`	`2'b00`–`2'b11`	`2'b00`	`2'b00` – `MLP_DIN[71:0]`. `2'b01` – `LRAM_DOUT[71:0]`. (1) `2'b10` – `BRAM_DOUT[71:0]`. (2) `2'b11` – `FWDI_MULTA_L[71:0]`.
`mux_sel_multb_h[2:0]`	`3'b000`–`3'b111`	`3'b000`	`3'b000` – `MLP_DIN[71:0]`. `3'b001` – `BRAM_DIN[71:0]`. `3'b010` – `LRAM_DOUT[71:0]`. (1) `3'b011` – `LRAM_DOUT[143:72]`. (1) `3'b100` – `BRAM_DOUT[71:0]`. (2) `3'b101` – `BRAM_DOUT[143:72]`. (2) `3'b110` – `FWDI_MULTB_L[71:0]`. `3'b111` – `FWDI_MULTB_H[71:0]`.
`mux_sel_multb_l[1:0]`	`2'b00`–`2'b11`	`2'b00`	`2'b00` – `MLP_DIN[71:0]`. `2'b01` – `LRAM_DOUT[71:0]`. (1) `2'b10` – `BRAM_DOUT[71:0]`. (2) `2'b11` – `FWDI_MULTB_L[71:0]`.
`lram_out2multb_l`	`1'b0`–`1'b1`	`1'b0`	Routes `LRAM_DOUT[71:0]` direct to the `multb_l` bus, bypassing `mux_sel_multb_l`: `1'b0` – `'b'` input to the multipliers is the bus selected by `mux_sel_multb_l`. `1'b1` – `'b'` input to the low bank of multipliers is `LRAM_DOUT[71:0]`. (1)
`lram_out2multb_h`	`1'b0`–`1'b1`	`1'b0`	Routes `LRAM_DOUT[143:72]` direct to the `multb_h` bus, bypassing `mux_sel_multb_h`: `1'b0` – `'b'` input to the multipliers is the bus selected by `mux_sel_multb_h`. `1'b1` – `'b'` input to the high bank of multipliers is `LRAM_DOUT[143:72]`. (1)
`cesel_multX_Y[3:0]`	`4'b0000`–`4'b1101`	Must be set (0–13)	Selects the `ce` inputs for each register group: `4'b0000` – `1'b0`. `4'b0001` – `ce[0]`. `4'b0010` – `ce[1]`. `4'b0011` – `ce[2]`. `4'b0100` – `ce[3]`. `4'b0101` – `ce[4]`. `4'b0110` – `ce[5]`. `4'b0111` – `ce[6]`. `4'b1000` – `ce[7]`. `4'b1001` – `ce[8]`. `4'b1010` – `ce[9]`. `4'b1011` – `ce[10]`. `4'b1100` – `ce[11]`. `4'b1101` – `1'b1`.
			Selects the `rstn` input for each register group:

Parameter	Supported Values	Default Value	Description
rstsel_multX_Y[2:0]	3'b000–3'b101	Must be set (0–5)	3'b000 – 1'b0. 3'b001 – rstn[0]. 3'b010 – rstn[1]. 3'b011 – rstn[2]. 3'b100 – rstn[3]. 3'b101 – 1'b1.
rst_mode_multX_Y	1'b0–1'b1	1'b0	Selects the reset mode (clocked vs. unclocked) for each register group: 1'b0 – synchronous reset mode. 1'b1 – asynchronous reset mode.
del_multX_Y	1'b0–1'b1	1'b0	Controls if each register group is enabled: 1'b0 – pipeline register is disabled. 1'b1 – pipeline register is enabled.

Table Notes

1. LRAM_DOUT[143:0] is an internal connection only from the coupled LRAM. This is not available as an input port on the MLP.
2. BRAM_DIN[71:0] and BRAM_DOUT[143:0] are logical names for the respective signal paths. The physical port names vary, and are listed in the following Ports table.

Ports

Table 121: ***Input Selection Ports***

Name	Direction	Description
din[71:0]	Input	MLP_DIN[71:0] data inputs.
mlpram_bramdin2mlpdin[71:0](1)	Input	Dedicated path from co-sited ACX_BRAM72K. Connects BRAM_DIN[71:0] port to MLP.
mlpram_bramdout2mlp[143:0](1)	Input	Dedicated path from co-sited ACX_BRAM72K. Connects BRAM_DOUT[143:0] to MLP.
fwdi_multa_h[71:0]	Input	Forward cascade path inputs for multiplier A inputs, higher multiplier block.
fwdi_multb_h[71:0]	Input	Forward cascade path inputs for multiplier B inputs, higher multiplier block.
fwdi_multa_l[71:0]	Input	Forward cascade path inputs for multiplier A inputs, lower multiplier block.
fwdi_multb_l[71:0]	Input	Forward cascade path inputs for multiplier B inputs, lower multiplier block.
fwdo_multa_h[71:0]	Output	Forward cascade path output for multiplier A inputs, higher multiplier block.

Name	Direction	Description
`fwdo_multb_h[71:0]`	Output	Forward cascade path output for multiplier B inputs, higher multiplier block. This bus is the selection from `mult_sel_multb_h` and is not affected by the value of `lram_out2multb_h`.
`fwdo_multa_l[71:0]`	Output	Forward cascade path output for multiplier A inputs, lower multiplier block.
`fwdo_multb_l[71:0]`	Output	Forward cascade path output for multiplier B inputs, lower multiplier block. This bus is the selection from `mult_sel_multb_l` and is not affected by the value of `lram_out2multb_l`.

Table Notes

1. This port can only be connected to the equivalent, same-named output on a ACX_BRAM72K. This port cannot be driven directly by fabric logic. A BRAM must be instantiated to use this connection.

Integer Modes

The most straightforward operation of the ACX_MLP72 is in integer mode, when up to 32 parallel multiplications can be performed, and combined with various adder and accumulation stages.

Byte Selection

When the four input buses have been selected; the buses are divided up into "byte" lanes. These lanes are then sent to each multiplier. Normally byte implies an 8-bit signal, however in this instance the signal width varies and is dependent upon the selected number format. Throughout this description, byte is used as nomenclature for the selected group of bits sent to each multiplier. The byte selection is controlled by the two parameters, `bytesel_00_07` to select the words from `multa_l` and `multb_l` into multipliers [7:0], and `bytesel_08_15` to select the words from `multa_h` and `multb_h` for multipliers[15:8].

In most applications, `bytesel_00_07` and `bytesel_08_15` are assigned the same value. However, it is possible to assign different values, particularly when treating the MLP72 as two independent halves. In addition, for the expanded modes (×2 and ×4) `bytesel_00_07` value may retain the same value as for the ×1 mode configuration, with just `bytesel_08_15` changing to map the bytes to the upper multipliers.

The sources for `multa_l`, `multb_l`, `multa_h`, and `multb_h` are selected independently. With particular `bytesel` mappings, the same input source could be used for the a and b multiplier inputs. For instance, if selecting Int8 in 1x mode (only 4 multipliers used), then both `multa_l` and `multb_l` can be set to select the `MLP_DIN[71:0]` input. If this input is packed as `MLP_DIN[71:0] = {8'h00, b3, b2, b1, b0, a3, a2, a1, a0}`, then using the correct `bytesel`, the a and b inputs to the 4 multipliers can be selected from just this one single input. (As reference, in this example, `bytesel_00_07` and `bytesel_08_15` should both be set to 'h0).

The following tables show the integer byte selection from each input bus, based on the values of `bytesel`. The tables are grouped by the required number format. Greyed out cells are not used, and should be set to 1'b0.

Int8

A total of up to 16 multipliers can be used, in either ×1, ×2, ×4 or a split mode.

Table 122: ***Four Multipliers (×1 Mode – bytesel_00_07 = 'h00; bytesel_08_15 = 'h00)***

Input Bus	[71:64]	[63:56]	[55:48]	[47:40]	[39:32]	[31:24]	[23:16]	[15:8]	[7:0]
multa_l						a3	a2	a1	a0
multb_l		b3	b2	b1	b0				
multa_h	Unused								
multb_h	Unused								

Table 123: ***Eight Multipliers (×2 Mode – bytesel_00_07 = 'h01; bytesel_08_15 = 'h01)***

Input Bus	[71:64]	[63:56]	[55:48]	[47:40]	[39:32]	[31:24]	[23:16]	[15:8]	[7:0]
multa_l		a7	a6	a5	a4	a3	a2	a1	a0
multb_l		b7	b6	b5	b4	b3	b2	b1	b0
multa_h	Unused								
multb_h	Unused								

Table 124: ***Sixteen Multipliers (×4 Mode – bytesel_00_07 = 'h01; bytesel_08_15 = 'h21)***

Input Bus	[71:64]	[63:56]	[55:48]	[47:40]	[39:32]	[31:24]	[23:16]	[15:8]	[7:0]
multa_l		a7	a6	a5	a4	a3	a2	a1	a0
multb_l		b7	b6	b5	b4	b3	b2	b1	b0
multa_h		a15	a14	a13	a12	a11	a10	a9	a8
multb_h		b15	b14	b13	b12	b11	b10	b9	b8

The following mode uses 4 multipliers from the lower half, and 4 multipliers from the top half of the MLP72.

Table 125: ***Eight Multipliers (×2 Split Mode – bytesel_00_07 = 'h00; bytesel_08_15 = 'h20)***

Input Bus	[71:64]	[63:56]	[55:48]	[47:40]	[39:32]	[31:24]	[23:16]	[15:8]	[7:0]
multa_l						a3	a2	a1	a0
multb_l		b3	b2	b1	b0				
multa_h						a11	a10	a9	a8
multb_h		b11	b10	b9	b8				

Int7

A total of up to 16 multipliers can be used, in either ×1, ×2, ×4 or a split mode.

Table 126: ***Five Multipliers (×1 Mode – bytesel_00_07 = 'h07; bytesel_08_15 = 'h07)***

Input Bus	[71:70]	[69:63]	[62:56]	[55:49]	[48:42]	[41:35]	[34:28]	[27:21]	[20:14]	[13:7]	[6:0]
multa_l							a4	a3	a2	a1	a0
multb_l		b4	b3	b2	b1	b0					
multa_h	Unused										
multb_h	Unused										

Table 127: ***Ten Multipliers (×2 Mode – bytesel_00_07 = 'h08; bytesel_08_15 = 'h08)***

Input Bus	[71:70]	[69:63]	[62:56]	[55:49]	[48:42]	[41:35]	[34:28]	[27:21]	[20:14]	[13:7]	[6:0]
multa_l				a7	a6	a5	a4	a3	a2	a1	a0
multb_l				b7	b6	b5	b4	b3	b2	b1	b0
multa_h		a9	a8								
multb_h		b9	b8								

Table 128: ***Sixteen Multipliers (×4 Mode – bytesel_00_07 = 'h08; bytesel_08_15 = 'h28)***

Input Bus	[71:70]	[69:63]	[62:56]	[55:49]	[48:42]	[41:35]	[34:28]	[27:21]	[20:14]	[13:7]	[6:0]
multa_l				a7	a6	a5	a4	a3	a2	a1	a0
multb_l				b7	b6	b5	b4	b3	b2	b1	b0
multa_h				a15	a14	a13	a12	a11	a10	a9	a8
multb_h				b15	b14	b13	b12	b11	b10	b9	b8

The following mode uses 5 multipliers from the lower half, and 5 multipliers from the top half of the MLP72.

Table 129: ***Ten Multipliers (×2 Split Mode – bytesel_00_07 = 'h07; bytesel_08_15 = 'h27)***

Input Bus	[71:70]	[69:63]	[62:56]	[55:49]	[48:42]	[41:35]	[34:28]	[27:21]	[20:14]	[13:7]	[6:0]
multa_l							a4	a3	a2	a1	a0
multb_l		b4	b3	b2	b1	b0					
multa_h							a12	a11	a10	a9	a8
multb_h		b12	b11	b10	b9	b8					

Int6

A total of up to 16 multipliers can be used, in either ×1, ×2, ×4 or a split mode.

Table 130: ***Six Multipliers (×1 Mode – bytesel_00_07 = 'h0a. bytesel_08_15 = 'h0a)***

Input Bus	[71:66]	[65:60]	[59:54]	[53:48]	[47:42]	[41:36]	[35:30]	[29:24]	[23:18]	[17:12]	[11:6]	[5:0]
multa_l							a5	a4	a3	a2	a1	a0
multb_l	b5	b4	b3	b2	b1	b0						
multa_h	Unused											
multb_h	Unused											

Table 131: ***Twelve Multipliers (×2 Mode – bytesel_00_07 = 'h0b; bytesel_08_15 = 'h0b)***

Input Bus	[71:66]	[65:60]	[59:54]	[53:48]	[47:42]	[41:36]	[35:30]	[29:24]	[23:18]	[17:12]	[11:6]	[5:0]
multa_l					a7	a6	a5	a4	a3	a2	a1	a0
multb_l					b7	b6	b5	b4	b3	b2	b1	b0
multa_h	a11	a10	a9	a8								
multb_h	b11	b10	b9	b8								

Table 132: ***Sixteen Multipliers (×4 Mode – bytesel_00_07 = 'h0b; bytesel_08_15 = 'h2b)***

Input Bus	[71:66]	[65:60]	[59:54]	[53:48]	[47:42]	[41:36]	[35:30]	[29:24]	[23:18]	[17:12]	[11:6]	[5:0]
multa_l					a7	a6	a5	a4	a3	a2	a1	a0
multb_l					b7	b6	b5	b4	b3	b2	b1	b0
multa_h					a15	a14	a13	a12	a11	a10	a9	a8
multb_h					b15	b14	b13	b12	b11	b10	b9	b8

The following mode uses 6 multipliers from the lower half, and 6 multipliers from the top half of the MLP72.

Table 133: ***Twelve Multipliers (×2 Split Mode – bytesel_00_07 = 'h0a; bytesel_08_15 = 'h2a)***

Input Bus	[71:66]	[65:60]	[59:54]	[53:48]	[47:42]	[41:36]	[35:30]	[29:24]	[23:18]	[17:12]	[11:6]	[5:0]
multa_l							a5	a4	a3	a2	a1	a0
multb_l	b5	b4	b3	b2	b1	b0						
multa_h							a13	a12	a11	a10	a9	a8
multb_h	b13	b12	b11	b10	b9	b8						

Int4

MLP72 supports up to 32 int4 multipliers. This is achieved by internally dividing each of the native int8 multipliers into two. There are no separate `bytesel` modes for int4. Instead, use the int8 `bytesel` modes, packing two int4 arguments per int8 value. The number of mapped int4 multiplications is double the number of int8 multiplications for the same mode.

Int3

MLP72 supports up to 32 int3 multipliers. This is achieved by internally dividing each of the native int8 multipliers into two. There are no separate `bytesel` modes for int3. Instead, use the int6 `bytesel` modes, packing two int3 arguments per int6 value. The number of mapped int3 multiplications is double the number of int6 multiplications for the same mode.

Int16

A total of up to 4 multiplications can be performed in parallel, in either ×1, ×2, split or compact mode.

Table 134: ***Two Multiplications (×1 Mode – bytesel_00_07 = 'h11. bytesel_08_15 = 'h11)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
multa_l				a1	a0
multb_l		b1	b0		
multa_h	Unused				
multb_h	Unused				

Table 135: ***Four Multiplications (×2 Mode – bytesel_00_07 = 'h12. bytesel_08_15 = 'h12)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
multa_l				a1	a0
multb_l				b1	b0
multa_h		a3	a2		
multb_h		b3	b2		

The following mode achieves 2 multiplications in the lower half, and 2 multiplications in the top half of the MLP72.

Table 136: ***Four Multiplications (×2 Split Mode – bytesel_00_07 = 'h11. bytesel_08_15 = 'h31)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
multa_l				a1	a0
multb_l		b1	b0		
multa_h				a3	a2
multb_h		b3	b2		

The following mode achieves 2 multiplications in the lower half, and 2 multiplications in the top half of the MLP72.

Table 137: ***Four Multiplications (×2 Compact Mode – bytesel_00_07 = 'h12. bytesel_08_15 = 'h32)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
multa_l				a1	a0
multb_l				b1	b0
multa_h				a3	a2
multb_h				b3	b2

Parameters

Table 138: ***Integer Byte Selection Parameters***

Parameter	Supported Values	Default Value	Description
`bytesel_00_07[4:0]`	`5'h00`–`5'h12`	`5'h00`	`5'h00` – Int8 ×1 and ×2 split mode. `5'h01` – Int8 ×2 and ×4 mode. `5'h07` – Int7 ×1 and ×2 mode. `5'h08` – Int7 ×2 and ×4 mode. `5'h0A` – Int6 ×1 and ×2 split mode. `5'h0B` – Int6 ×2 and ×4 mode. `5'h11` – Int16 ×1 mode. `5'h12` – Int16 ×2 mode.
`bytesel_08_15[5:0]`	`6'h00`–`6'h2B`	`6'h00`	`6'h00` – Int8 ×1 mode. `6'h01` – Int8 ×2 mode. `6'h07` – Int7 ×1 mode. `6'h08` – Int7 ×2 mode. `6'h0A` – Int6 ×1 mode. `6'h0B` – Int6 ×2 mode. `6'h11` – Int16 ×1 mode. `6'h12` – Int16 ×2 mode. `6'h20` – Int8 ×2 split mode. `6'h21` – Int8 ×4 mode. `6'h27` – Int7 ×2 split mode. `6'h28` – Int7 ×4 mode. `6'h2A` – Int6 ×2 split mode. `6'h2B` – Int6 ×4 mode. `6'h31` – Int16 ×2 split mode. `6'h32` – Int16 ×2 compact mode.

Multiplier Stage

The ACX_MLP72 contains 16 integer multipliers, each of which can multiply two 8 bit values. These multipliers can then either be combined to support multiplication of larger integer values such as 16 bit, or else subdivided to support double the multiplication capacity for 4 and 3 bit integers. The multipliers are divided into two banks, high and low, and each bank is fed from the corresponding input stage.

Within each bank, there are 8 multipliers which are summed as two groups of 4. These intermediate sums are then optionally summed, or subtracted from each other. Finally the sum of each bank is added together to give an overall result, representing the sum of all 16 input multipliers.

The input to each integer multiplier supports an optional delay stage. For multipliers[3:0] each individual input has it's own delay stage control, including control of the reset mode. For multipliers[15:4], the delay stages are controlled in banks of 4, corresponding to the group of 4 multipliers which are initially summed together.

The structure of integer multiplication, summing and delay stages is shown in the following figure. The parameters which control signal selection, delay stage selection, add or subtract are shown as text only alongside the component they apply to.

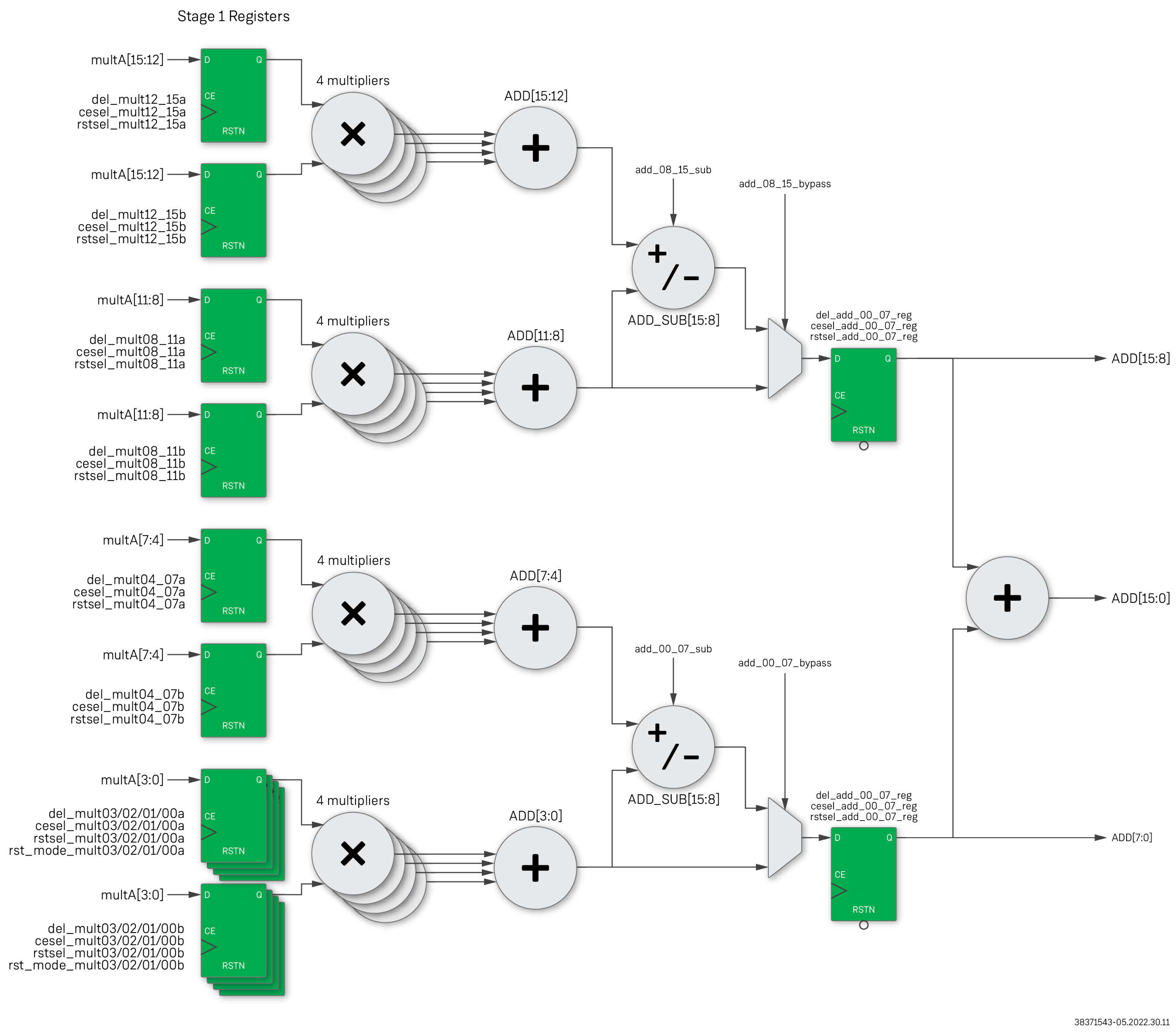


Figure 94: *Multiplication Stage Structure (Integer)*

Parallel Multiplications

The ACX_MLP72 combines multipliers in appropriate structures based on the selected number formats. Each multiplier natively supports an Int8 × Int8 multiplication with a 16 bit result. These multipliers can then also be split to perform two parallel Int4 × Int4, or Int3 × Int3 multiplications. In these split modes, the output of the multiplier can either be the two individual results (8 bits each, configured by the `multmode_xx_xx` parameter `SNOADD` mode), or the sum of the two results. 16 parallel multiplications can then be achieved for number formats of 8 bits and 6 bits. Finally, for number formats greater than 8 bits, such as Int16, a lower number of parallel integer multiplications is achieved as the multipliers are combined to compute the larger result. The maximum number of parallel multiplications for each number format is shown in the following table.

Table 139: ***Maximum Possible Integer Multiplications***

Number Format	Maximum Parallel Multiplications
Int3	32
Int4	32
Int6	16
Int8	16
Int16	4

Number Formats

For details of the number formats used within the ACX_MLP72, refer to Number formats . In addition to the actual number formats listed, there is a further processed format, `Sign - No Add`, that is specific to the ACX_MLP72.

Sign - No ADD (SNOADD)

SNOADD is an output only number format from a multiplier. When the multiplier is set to Int4 or Int3 format, the multiplier is split into two separate multipliers, each performing either a Int4 × Int4, or Int3 × Int3 multiplication. The multiplier can then be set to either add the two results together, to give the multiply-accumulate sum of the two input pairs, or alternatively the multiplier can be set to output the two results in parallel, each using 8 bits of the 16 bit multiplier output. It is not intended that there would be any further processing of this value within the MLP72, instead this split value can be sent directly to the ACX_MLP72 output stage.

Format Consistency

Between the Input Selection and the Integer Multiplier Stage (see page 179) the ACX_MLP72 selects the appropriate slice of the input bus to route to each multiplier. This slice selection is dependent upon the input number format, and is controlled by the `bytesel_xx_xx` parameters, and detailed in Byte Selection (see page 173). Equally the multiplier modes are controlled by the `multmode_xx_xx` parameters, which are dependent upon the selected number format. The `bytesel` and `multmode` parameters must be consistent in terms of number format and sizes in order to achieve correct multiplication results.

Parameters

Parameters that are specific to the integer multiplication stage are detailed in the following table. For the purposes of clarity, the delay stage parameters are not shown in this table, instead they are shown in the previous Figure (see page 180).

Table 140: ***Integer Multiplication Parameters***

Parameter	Supported Values	Default Value	Description
`multmode_00_07[4:0]`	`5'h00`–`5'h11`	`5'h00`	`5'h00` – SIGNED 8×8. `5'h01` – UNSIGNED 8×8. `5'h02` – SMAG 8×8 (SignMAGnitude). `5'h03` – SIGNED 7×7. `5'h04` – SMAG 7×7 (SignMAGnitude). `5'h05` – SIGNED 6×6. `5'h06` – SMAG 6×6 (SignMAGnitude). `5'h07` – SIGNED 4×4. `5'h08` – SMAG 4×4 (SignMAGnitude). `5'h09` – SNOADD 4×4 (Sign-NOADDer). `5'h0A` – SIGNED 3×3. `5'h0B` – SMAG 3×3 (SignMAGnitude). `5'h0C` – SNOADD 3×3 (Sign-NOADDer). `5'h0D` – SIGNED 16×16. `5'h0E` – SA_UB 16×16 (SignedA_UnsignedB). `5'h0F` – UA_SB 16×16 (UnsignedA_SignedB). `5'h10` – UNSIGNED 16×16. `5'h11` – NO OP (NO OPeration). `5'h12` – A SIGNED, B UNSIGNED 8×8. `5'h13` – A UNSIGNED, B SIGNED 8×8.
`multmode_08_15[4:0]`	`5'h00`–`5'h11`	`5'h00`	`5'h00` – SIGNED 8×8. `5'h01` – UNSIGNED 8×8. `5'h02` – SMAG 8×8 (SignMAGnitude). `5'h03` – SIGNED 7×7. `5'h04` – SMAG 7×7 (SignMAGnitude). `5'h05` – SIGNED 6×6. `5'h06` – SMAG 6×6 (SignMAGnitude). `5'h07` – SIGNED 4×4. `5'h08` – SMAG 4×4 (SignMAGnitude). `5'h09` – SNOADD 4×4 (Sign-NOADDer). `5'h0A` – SIGNED 3×3. `5'h0B` – SMAG 3×3 (SignMAGnitude). `5'h0C` – SNOADD 3×3 (Sign-NOADDer). `5'h0D` – SIGNED 16×16. `5'h0E` – SA_UB 16×16 (SignedA_UnsignedB). `5'h0F` – UA_SB 16×16 (UnsignedA_SignedB). `5'h10` – UNSIGNED 16×16. `5'h11` – NO OP (NO OPeration). `5'h12` – A SIGNED, B UNSIGNED 8×8. `5'h13` – A UNSIGNED, B SIGNED 8×8.
`add_00_07_bypass`	`1'b0`–`1'b1`	`1'b0`	Controls if ADD07 is bypassed: `1'b0` – `ADD0_7_REG` input selects ADD07 output. `1'b1` – `ADD0_7_REG` input selects ADD03 output.
`add_00_07_sub`	`1'b0`–`1'b1`	`1'b0`	Controls if ADD07 is in subtract mode: `1'b0` – ADD07 performs A + B. `1'b1` – ADD07 performs A – B.
`add_08_15_bypass`	`1'b0`–`1'b1`	`1'b0`	Controls if ADD815 is bypassed: `1'b0` – ADD8_15_REG input selects ADD815 output. `1'b1` – ADD8_15_REG input selects ADD811 output.
`add_08_15_sub`	`1'b0`–`1'b1`	`1'b0`	Controls if ADD815 is in subtract mode: `1'b0` – ADD815 performs A + B. `1'b1` – ADD815 performs A – B.

Output Stage

The ACX_MLP72 output stage supports addition, subtraction or accumulation of the output from the multiplier stage. Other signals from the BRAM and LRAM may also be combined or routed through for specific configurations.

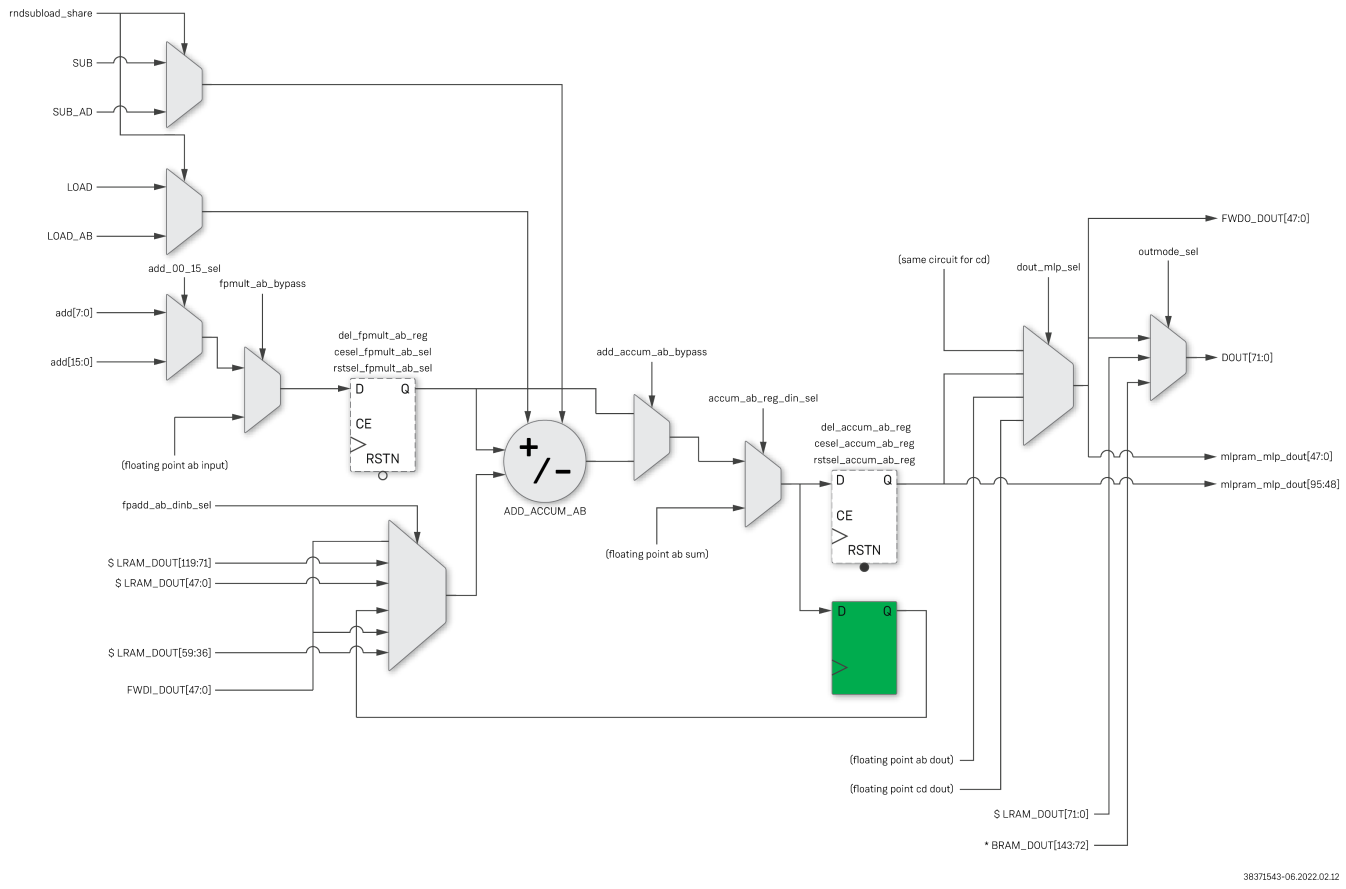


Figure 95: *Output Stage*

Parameters

Table 141: ***Output Stage Parameters***

Parameter	Supported Values	Default Value	Description
`add_00_15_sel`	`1'b0`–`1'b1`	`1'b0`	Selects if the output of ADD015 is used: `1'b0` – ADD0_7_REG output is routed toward FPMULT_AB_REG. `1'b1` – ADD015 output is routed toward FPMULT_AB_REG.
`fpmult_ab_bypass`	`1'b0`–`1'b1`	`1'b0`	Select to bypass (A*B) Floating-Point Multiplier: `1'b0` – floating-Point Multiplier is enabled. `1'b1` – floating-Point Multiplier is bypassed; integer multiplier is selected.
`fpmult_cd_bypass`	`1'b0`–`1'b1`	`1'b0`	Select to bypass (C*D) Floating-Point Multiplier: `1'b0` – floating-Point Multiplier is enabled. `1'b1` – floating-Point Multiplier is bypassed; integer multiplier is selected.
`fpadd_cd_dina_sel`	`1'b0`–`1'b1`	`1'b0`	Select the value between (C*D) Floating-Point multiplier and (A*B) Accumulator: `1'b0` – selection the value from (C*D) Floating-Point-Multiplier. `1'b1` – selection the value from (A*B) Accumulator. This selector is not shown on the diagram above.
`fpadd_cd_dinb_sel[2:0]`	`3'b000`–`3'b100`	`3'b000`	Select the addend, or subtrahend for the CD Accumulator: `3'b000` – 48-bit `ACCUM_CD_REG` input (registered). `3'b001` – 48-bit MLP Forward Cascaded input `FWDI_DOUT[47:0]`. `3'b010` – 48-bit `LRAM_DOUT[47:0]`. `3'b011` – reserved. `3'b100` – 48-bit AB Accumulator data output.
`fpadd_ab_dinb_sel[2:0]`	`3'b000`–`3'b101`	`3'b000`	Select the addend, or subtrahend for the AB Accumulator: `3'b000` – 48-bit ACCUM_AB_REG input (always registered). `3'b001` – 48-bit MLP Forward Cascaded input `FWDI_DOUT[47:0]`. `3'b010` – 48-bit `LRAM_DOUT[47:0]`.(1) `3'b011` – 24-bit `LRAM_DOUT[59:36]` (top 24 bits tied to zero). `3'b100` – 24-bit MLP Forward Cascade input `FWDI_DOUT[47:24]` (top 24 bits tied to zero). `3'b101` – 48-bit `LRAM_DOUT[119:72]`.
`add_accum_ab_bypass`	`1'b0`–`1'b1`	`1'b0`	Select to bypass the AB accumulator output: `1'b0` – integer AB accumulator value is used. `1'b1` – bypass integer AB accumulator.
`add_accum_cd_bypass`	`1'b0`–`1'b1`	`1'b0`	Select to bypass the CD accumulator output: `1'b0` – integer CD accumulator value is used. `1'b1` – bypass integer CD accumulator.
`out_reg_din_sel[2:0]`	`3'b000-3'b110`	`2'b00`	Select `out_reg` input: `3'b000` – value is from Mult8×4. `3'b010` – output of floating point FP_ADD_CD accumulator. `3'b011` – output or bypass of integer CD accumulator, as set by `add_accum_cd_bypass`. `3'b100` – 8-bit wide A ± B output. `3'b110` – value is Mult16×2. This selector is not shown on the diagram above.
`accum_ab_reg_din_sel`	`1'b0`–`1'b1`	`1'b0`	Select between integer and floating point AB result: `1'b0` – value from integer AB accumulator block. `1'b1` – value from floating point FP_ADD_AB accumulator block.
			Select values for the forward DOUT cascade path: `2'b00` – value from optionally registered output `OUT_REG[63:0]` (Not shown on diagram). `2'b01` – concatenated outputs of upper and lower MLP outputs `{24'h0,ACCUM_AB_REG`

Parameter	Supported Values	Default Value	Description
`dout_mlp_sel[1:0]`	`2'b00–2'b11`	`2'b00`	`[23:0],OUT_REG[23:0]}`, used to pass floating point values via `fwdo_dout`. `2'b10` – value from optionally registered output `ACCUM_AB_REG[47:0]`. `2'b11` – concatenated lower 36 bits from upper and lower MLP outputs `{ACCUM_AB_REG [35:0],OUT_REG[35:0]}`.
`outmode_sel[1:0]`	`2'b00–2'b11`	`2'b00`	Select final `DOUT` value: `2'b00` – 72-bit output of value selected by parameter `dout_mlp_sel[1:0]`. `2'b01` – `LRAM_DOUT[71:0]`.(1) `2'b10` – `BRAM_DOUT[143:72]`. `2'b11` – optionally registered concatenated outputs of floating point format conversion registers with status `{20'h0,fp_ab_status, fp_cd_status, accum_ab_reg, out_reg}`.
`rndsubload_share`	`1'b0–1'b1`	`1'b0`	Select to share Round, Sub, and Load input from the upper (cd sum) half with the lower (ab sum) half.

Table Notes

1. LRAM_DOUT is not a physical port on the ACX_MLP72. It is an internal only connection from the associated tightly-coupled ACX_LRAM.

Ports

Table 142: *Output Stage Ports*

Name	Direction	Description
`load`	Input	`rndsubshare = 1'b0` – when the upper half cd_add_accum accumulator is enabled, load the accumulator with the `add[15:8]` sum. `rndsubshare = 1'b1` – load both ab_add_accum and cd_add_accum with their respective sum inputs.
`load_ab`	Input	`rndsubshare = 1'b0` – when the lower half ab_add_accum accumulator is enabled, load the accumulator with the output of the add_00_15_sel multiplexer. `rndsubshare = 1'b1` – unused.
`sub`	Input	`rndsubshare = 1'b0` – configure upper half cd_add_accum adder to subtraction mode. `rndsubshare = 1'b1` – configure both add_accum adders to subtraction mode.
`sub_ab`	Input	`rndsubshare = 1'b0` – configure lower half ab_add_accum adder to subtraction mode. `rndsubshare = 1'b1` – unused.
`dout[71:0]`	Output	The result of the multiply-accumulate operation.
`fwdi_dout[47:0]`	Input	MLP72 internally calculated result, cascaded from the ACX_MLP72 block below.
`fwdo_dout[47:0]`	Output	MLP72 internally calculated results, cascaded up to the ACX_MLP72 block above.
`mlpram_mlp_dout[95:0]`	Output	Bits[47:0] ACX_MLP72 internally calculated result truncated to 48 bits. Bits[95:48] result of the ab sum path. The intended operation of `mlpram_mlp_dout` is when `dout_mlp_sel` selects the result of the cd sum path. Then `mlpram_mlp_dout` is a concatenation of the cd and ab sums, each truncated to 48 bits.

Integrated LRAM

The ACX_MLP72 has an integrated Logic 2-kb RAM (LRAM) tightly bonded to both its external inputs and internal signals. This LRAM enables local storage and reuse of both input values, and output results. The LRAM is often referred to as a register file, particularly when it is configured to store and replay ACX_MLP72 results. The LRAM can be configured as 36 bits × 64, 72 bits × 32, or 144 bits × 16, dependent upon the application.

Standalone LRAM

If an LRAM independent of the MLP72 is required, use the dedicated ACX_LRAM2K_SDP or ACX_LRAM2K_FIFO (see page 449) primitive, appropriate to the application. These primitives have only the required ACX_LRAM2K ports and parameters, simplifying instantiation.

Note

When an ACX_LRAM2K is instantiated directly, the associated ACX_MLP72 is not available due to the use of shared pins.

LRAM Operational Modes

When the LRAM is used as an integrated part of the ACX_MLP72, it can be operated in three modes (the mode values correspond to the values set for the `lram_input_control_mode` and `lram_output_control_mode` parameters):

- Mode 0 (default) – LRAM is slaved to co-sited ACX_BRAM72K. Using the `wrmsel` and `rdmsel` address enables on the co-sited ACX_BRAM72K, the LRAM operates as an extension to the ACX_BRAM72K, supporting additional address space. The data, read and write signals are connected from the ACX_BRAM72K to the LRAM using the dedicated signal paths. This mode is intended for initializing the LRAM via the NoC during power-up.
- Mode 1 – LRAM operates as either a RAM or FIFO (dependent upon `lram_fifo_enable`). Re-purposing several dual-use inputs (CE, RSTN, EXPB), the LRAM can store the results of the ACX_MLP72 calculation, and its output can be routed back into the ACX_MLP72 Input Selection stage. For details of how the ACX_MLP72 inputs can be re-purposed to the LRAM, see LRAM Virtual Ports. (see page 186)
- Mode 2 – the LRAM must be set to operate as a FIFO in Mode 1 (`lram_fifo_enable` = 1'b1). Mode 2 then adds additional signals that allow the reset of the FIFO address generators (see FIFO Address Generators (see page 189)). This additional flexibility allows the LRAM to store groups of results or coefficients that do not necessarily match the length of the FIFO, i.e., their length is not a power of 2^n.

Note

Although `lram_input_control_mode` and `lram_output_control_mode` are separate parameters, it is anticipated that in normal operation they would both be set to the same value. If the user application requires these parameters to be set to differing values, it is recommended to discuss the requirements with Achronix Support.

LRAM Virtual Ports

When the LRAM is configured within the ACX_MLP72, several of the ACX_MLP72 ports are re-purposed to the LRAM. These configurations are also dependent upon the operating mode. These re-purposed ports have logical internal signal names and can be considered virtual ports to the LRAM. The mapping of these virtual ports is detailed in the following table.

Table 143: ***LRAM Virtual Port Mapping***

		External Pin		
Virtual Port Name	**Description**	**Mode 0**	**Mode 1**	**Mode 2**
`lram_wraddr[5:0]`	Write address.	`mlpram_wraddr`	`expb[7:2]`	`6'h0`
`lram_wren`	Write enable.	`mlpram_wren`	`ce[7]`	`ce[7]`
`lram_rdaddr[5:0]`	Read address.	`mlpram_rdaddr`	`{expb[1:0],ce[11:8]}`	`6'h0`
`lram_rden`	Read enable.	`mlpram_rden`	`ce[6]`	`ce[6]`
`lram_rstregn`	Output register reset, (optionally block memory reset).	`1'b1`	`rstn[0]`	`rstn[0]`
`lram_fsm_wrrst`	Reset FIFO write address pointer.	`1'b0`	`1'b0`	`ce[9]`
`lram_fsm_rdrst`	Reset FIFO read address pointer.	`1'b0`	`1'b0`	`ce[8]`

Interconnection Diagram

The block diagram and interconnection of the LRAM is shown in the following figure.

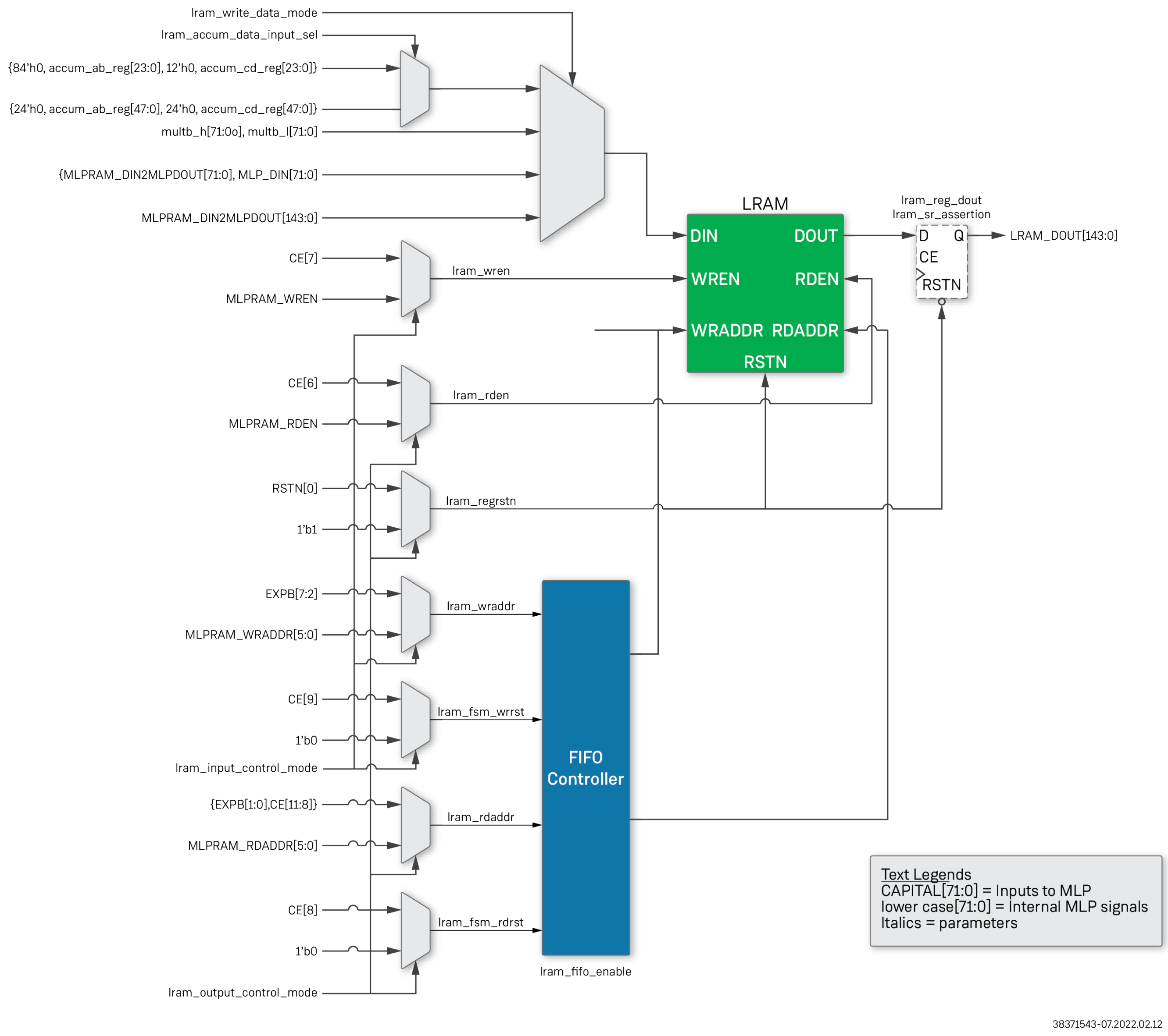


Figure 96: *LRAM connectivity*

Note

The `LRAM_DOUT[143:0]` port is an internal connection only to the coupled LRAM and is not available as an output port from the MLP. Inputs prefixed with `MLPRAM_` are dedicated paths and can only be connected to equivalent, same-named outputs on a co-sited ACX_BRAM72K and cannot be driven directly by fabric logic.

FIFO Address Generators

The LRAM is designed with particular flexibility around its FIFO address generators. allowing them to be used as built-in generic address counters. This mode of operation is particularly useful when partial sums produced by the MLP are written to the LRAM, to be read back some cycles later as input to the MLP for further additions. Using built-in address pointers rather than external address counters reduces user logic, and allows the virtual `lram_wraddr` and `lram_rdaddr` ports defined above to be used as `ce` and `expb` inputs.

Three separate features can be enabled to transform the FIFO address pointers into regular address counters. When these features are used, the FIFO counters no longer satisfy normal FIFO operation, they allow over and underflow, and for entries to be read multiple times. Although the FIFO status flags are still computed, user logic should ignore them as the pointers no longer maintain the FIFO property; this applies to the `full`, `empty`, `almost_full`, `almost_empty`, `write_error`, and `read_error flags`.

Length Adjustment

The ACX_MLP72 supports programmable end locations for both the write and read address generators. These thresholds are set using the `lram_fifo_wrptr_maxval` and `lram_fifo_rdptr_maxval` parameters. When an address pointer is equal to the specified `maxval` threshold, the next increment assigns the address counter back to 0.

Mode 2 Pointer Reset

In Mode 2 (requirement that `lram_fifo_enable` = 1'b1) two external pins are re-purposed as internal FIFO address generator resets:

- Asserting `lram_fsm_wrrst` resets the FIFO write pointer to 0 on the next active edge of `lram_wrclk`.
- Asserting `lram_fsm_rdrst` resets the FIFO read pointer to 0 on the next active edge of `lram_rdclk`.

These additional signals allow the read or write pointers to be dynamically reset.

Ignore Flags

Normally, in FIFO mode, a write in the full state has no effect: no memory location is changed, and the write pointer is not incremented. Likewise, a read from an empty FIFO does not change the output, and the read pointer is not incremented. However, when the `lram_fifo_ignore_flags` parameter is set, these rules are not followed: A write always writes the current memory location and increments the write pointer, and a read always returns the value stored at the current location and increments the read pointer. (The increments wrap around as specified by their `maxval` thresholds).

Parameters

Table 144: *LRAM Parameters*

Parameter	Supported Values	Default Value	Description
`lram_wrclk_polarity`	"rise", "fall"	"rise"	Specifies whether registers are clocked by the rising or falling edge of the clock.
`lram_rdclk_polarity`	"rise", "fall"	"rise"	Specifies whether registers are clocked by the rising or falling edge of the clock.
`lram_sync_mode`	`1'b0`–`1'b1`	`1'b0`	Set LRAM synchronous mode: `1'b0` – write clock and read clock are asynchronous. `1'b1` – write clock and read clock are the same clock (synchronous).
`lram_reg_dout`	`1'b0`–`1'b1`	`1'b0`	Enable optional `LRAM_DOUT[143:0]` register: `1'b0` – LRAM read data is asynchronous read, no register. `1'b1` – LRAM read data is synchronous read, register enabled.
`lram_sr_assertion`	`1'b0`–`1'b1`	`1'b0`	Set reset mode for the output register: `1'b0` – synchronous reset mode. `1'b1` – asynchronous reset mode. If `lram_reg_dout` = 1'b0, then this parameter has no effect.
`lram_fifo_enable`	`1'b0`–`1'b1`	`1'b0`	Enable LRAM FIFO mode: `1'b0` – LRAM is not in FIFO mode. `1'b1` – LRAM is in FIFO mode.
`lram_clear_enable`(1)	`1'b0`–`1'b1`	`1'b0`	Enable LRAM block memory clear: `1'b0` – LRAM block memory clear is disabled. `1'b1` – when the virtual port `lram_regrstn` is asserted (1'b0), the contents of the LRAM memory are reset to 0.
`lram_write_width[1:0]`	`2'b00`–`2'b10`	`2'b00`	Select LRAM write data width and depth value: `2'b00` – data is 72-bit wide and 32 deep. `2'b01` – data is 36-bit wide and 64 deep. `2'b10` – data is 144-bit wide and 16 deep.
`lram_read_width[1:0]`	`2'b00`–`2'b10`	`2'b00`	Select LRAM read data width and depth value: `2'b00` – data is 72-bit wide and 32 deep. `2'b01` – data is 36-bit wide and 64 deep. `2'b10` – data is 144-bit wide and 16 deep.
`lram_input_control_mode[1:0]`	`2'b00`–`2'b11`	`2'b00`	Select LRAM Input control mode: `2'b00` – BRAM controls LRAM write control. `2'b01` – LRAM uses MLP inputs. `2'b10` – LRAM uses MLP inputs with additional FIFO controller FSM inputs. `2'b11` – LRAM is off/disabled. This controls the source of wraddr and wren.
`lram_output_control_mode[1:0]`	`2'b00`–`2'b11`	`2'b00`	Select LRAM output control mode: `2'b00` – BRAM controls LRAM read control. `2'b01` – LRAM uses MLP inputs. `2'b10` – LRAM uses MLP inputs with additional FIFO controller FSM inputs. `2'b11` – LRAM is off/disabled. This controls the source of `rdaddr`, `rden` and `regrstn`:
			`LRAM_DIN[143:0]` source: `2'b00` – `mlpram_din2mlpdout[143:0]`. BRAM internal ×144-bit write data. `2'b01` – aggregation of {`mlpram_din2mlpdout[71:0]`, `MLP_DIN[71:0]`}. BRAM internal ×72-bit input and MLP ×72-bit data in. `2'b10` – input selected by `lram_accum_data_input_sel`.

Parameter	Supported Values	Default Value	Description
`lram_write_data_mode[1:0]`	`2'b00–2'b11`	`2'b00`	`2'b11` – aggregation of mutliplier "b" input buses, {`multb_h[71:0]`, `multb_l [71:0]`}.
`lram_accum_data_input_sel`	`1'b0–1'b1`	`1'b0`	Select Accumulated data for `LRAM_DIN[143:0]`: `1'b0` – aggregation of {`24'h0`, `ADD_ACCUM_AB[47:0]`, `24'h0`, `ADD_ACCUM_CD [47:0]`}. ×144-bit mode. `1'b1` – aggregation of {`72'h0`, `12'h0`, `ADD_ACCUM_AB[23:0]`, `12'h0`, `ADD_ACCUM_CD[23:0]`}. ×72-bit mode.
`lram_fifo_wrptr_maxval[6:0]`	`7'h00–7'h7F`	`7'h7F`	LRAM FIFO write pointer maximum value (must be `'h7F` for normal FIFO operation)
`lram_fifo_rdptr_maxval[6:0]`	`7'h00–7'h7F`	`7'h7F`	LRAM FIFO read pointer maximum value (must be `'h7F` for normal FIFO operation)
`lram_fifo_sync_mode`	`1'b0–1'b1`	`1'b0`	Enable LRAM FIFO synchronous mode: `1'b0` – LRAM FIFO is in asynchronous mode. `1'b1` – LRAM FIFO is in synchronous mode.
`lram_fifo_afull_threshold[6:0]`	`7'h00–7'h3F`	`7'h3F`	Set LRAM FIFO almost full threshold. User-defined configuration bit. Recommended values are less than `7'h3F`.
`lram_fifo_aempty_threshold`	`7'h00–7'h0F`	`7'h00`	Set LRAM FIFO almost empty threshold. User-defined configuration bit. Recommended values are not less than `7'h01`.
`lram_fifo_ignore_flags`	`1'b0–1'b1`	`1'b0`	Enable LRAM FIFO address pointers to ignore empty/full status `1'b0` – LRAM FIFO does not write when the FIFO is full (asserting `write_error`) and does not read when the FIFO is empty (asserting `read_error`). This is normal FIFO behavior. `1'b1` – a write always writes to memory and increments the write pointer, regardless of `full` status. A read always reads from memory and increments the read pointer, regardless of `empty` status. In this mode, the read and write pointers act as regular address counters without operating as a FIFO. Ignore the `full`, `empty`, `almost_full`, `almost_empty`, `write_error`, and `read_error` flags.
`lram_fifo_fwft_mode`	`1'b0–1'b1`	`1'b0`	Enable LRAM FIFO in first-word-fall-through (FWFT) mode: `1'b1` – FWFT support is enabled. `1'b0` – FWFT is not enabled.

Table Notes

1. The LRAM output register is always reset when `lram_regrstn` is asserted low, independent of the state of `lram_clear_enable`.

Ports

Table 145: *LRAM Ports*

Name	Direction	Description
`lram_wrclk`	Input	Write side clock input for LRAM.
`lram_rdclk`	Input	Read side clock input for LRAM.
`mlpram_din2mlpdout[143:0]`(1)	Input	Connects BRAM data input, either BRAM_DIN or BRAM internal din, to LRAM_DIN.
`mlpram_rdaddr[5:0]`(1)	Input	Allows BRAM to control LRAM read address.
`mlpram_wraddr[5:0]`(1)	Input	Allows BRAM to control LRAM write address.

Name	Direction	Description
`mlpram_rden`(1)	Input	Allows BRAM to control LRAM read enable.
`mlpram_wren`(1)	Input	Allows BRAM to control LRAM write enable.
`mlpram_sbit_error`(1)	Input	Allows BRAM to pass through single bit error indication.
`mlpram_dbit_error`(1)	Input	Allows BRAM to pass through double bit error indication.
`sbit_error`	Output	Co-sited BRAM72K dedicated pass through of `mlpram_sbit_error`.
`dbit_error`	Output	Co-sited BRAM72K dedicated pass through of `mlpram_dbit_error`.
`empty`	Output	LRAM FIFO empty flag.
`full`	Output	LRAM FIFO full flag.
`almost_empty`	Output	LRAM FIFO almost empty flag.
`almost_full`	Output	LRAM FIFO almost full flag.
`write_error`	Output	Asserted when LRAM in FIFO mode, and write enable is asserted when LRAM FIFO is full.
`read_error`	Output	Asserted when LRAM in FIFO mode, and read enable is asserted when LRAM FIFO is empty.

Table Notes

1. All inputs prefixed with `mlpram_` are a dedicated path from the co-sited ACX_BRAM72K and are for when the BRAM and LRAM operate as a co-joined pair. The inputs can only be connected to equivalent, same-named outputs on the ACX_BRAM72K and cannot be driven directly by fabric logic. Instantiate a ACX_BRAM72K to use these connections. If used, same site placement constraints must be used for the paired ACX_BRAM72K and ACX_MLP72.

Block Floating-Point Modes

The ACX_MLP72 can be operated in either Integer, block floating-point or floating-point modes. The block floating-point structure follows the integer structure with some differences around the use of the multipliers.

Input Selection

The selection of the input source to multiplier bus is the same as for integer. Refer to Input Selection (see page 193) for details

Multiplication Operation

Block floating point combines the integer multiplier-adder tree with the floating-point multipliers. The input consists of integer mantissas (in signed magnitude format) and a shared exponent. The mantissa arguments follow the same convention as integer mode: a0 refers to the 'a' input of mult0, etc.

The exponents are named ea and eb for the 'ab' floating point result, and ec and ed for the 'cd' floating point result. In all block floating-point modes, there is space for an 8-bit exponent, but a separate parameter may be set to indicate that only a 5-bit exponent should be used.

In some modes, there is not sufficient data width in the input bus for all exponents. In these instances, the separate `expb[7:0]` input of the MLP is used to pass eb (and in some cases ed). Since there is only one `expb[]` input, if both eb and ed are mapped to expb, they must be equal. The `expb[]` input has dual purpose; it is also used to input LRAM addresses. As a result, a number of the block floating-point modes are incompatible with some LRAM modes.

The block floating point operation computes:

- `mult_ab = (a0*b0 + ... + a7*b7) * 2ea * 2eb`
- `mult_cd = (a8*b8 + ... + a15*b15) * 2ec * 2ed`

Byte Selection

Note

The following byte selection tables are listed by the mantissa size, which have the same conventions and names as their integer equivalents.

BFP Int8

Table 146: *Int8 3 Multiplications (×1 Mode – bytesel_00_07 = 'h03; bytesel_08_15 = 'h03)*

Input Bus	[71:64]	[63:56]	[55:48]	[47:40]	[39:32]	[31:24]	[23:16]	[15:8]	[7:0]
`multa_l`						`ea`	`a2`	`a1`	`a0`
`multb_l`		`eb`	`b2`	`b1`	`b0`				
`multa_h`	Unused								
`multb_h`	Unused								

Table 147: ***Int8 4 Multiplications (×1 Mode – bytesel_00_07 = 'h04. bytesel_08_15 = 'h04)***

Input Bus	[71:64]	[63:56]	[55:48]	[47:40]	[39:32]	[31:24]	[23:16]	[15:8]	[7:0]
`multa_l`	`ea`					`a3`	`a2`	`a1`	`a0`
`multb_l`(1)		`b3`	`b2`	`b1`	`b0`				
`multa_h`	Unused								
`multb_h`	Unused								

Table Notes

1. The `eb` input is driven directly from the `expb[7:0]` pins.

Table 148: ***Int8 6 Multiplications (×2 Mode Split – bytesel_00_07 = 'h03; bytesel_08_15 = 'h23)***

Input Bus (1)	[71:64]	[63:56]	[55:48]	[47:40]	[39:32]	[31:24]	[23:16]	[15:8]	[7:0]
`multa_l`						`ea`	`a2`	`a1`	`a0`
`multb_l`		`eb`	`b2`	`b1`	`b0`				
`multa_h`						`ec`	`a10`	`a9`	`a8`
`multb_h`		`ed`	`b10`	`b9`	`b8`				

Table Notes

1. A and B input data fields are numbered to reflect the multiplier to which they are applied.

Table 149: ***Int8 8 Multiplications (×2 Mode Exponent Split – ; bytesel_00_07 = 'h04; bytesel_08_15 = 'h24)***

Input Bus	[71:64]	[63:56]	[55:48]	[47:40]	[39:32]	[31:24]	[23:16]	[15:8]	[7:0]
`multa_l`	`ea`					`a3`	`a2`	`a1`	`a0`
`multb_l`(1)		`b3`	`b2`	`b1`	`b0`				
`multa_h`	`ec`					`a11`	`a10`	`a9`	`a8`
`multb_h`(1)		`b11`	`b10`	`b9`	`b8`				

Table Notes

1. The `eb` and `ed` exponents are the same, and are both taken from the `expb[7:0]` pins.

Table 150: ***Int8 8 Multiplications (×2 Mode – bytesel_00_07 = 'h05; bytesel_08_15 = 'h05)***

Input Bus	[71:64]	[63:56]	[55:48]	[47:40]	[39:32]	[31:24]	[23:16]	[15:8]	[7:0]
`multa_l`	`ea`	`a7`	`a6`	`a5`	`a4`	`a3`	`a2`	`a1`	`a0`
`multb_l`	`eb`	`b7`	`b6`	`b5`	`b4`	`b3`	`b2`	`b1`	`b0`
`multa_h`									
`multb_h`									

Table 151: ***Int8 16 Multiplications (×4 Mode – bytesel_00_07 = 'h05; bytesel_08_15 = 'h25)***

Input Bus	[71:64]	[63:56]	[55:48]	[47:40]	[39:32]	[31:24]	[23:16]	[15:8]	[7:0]
`multa_l`	`ea`	`a7`	`a6`	`a5`	`a4`	`a3`	`a2`	`a1`	`a0`
`multb_l`	`eb`	`b7`	`b6`	`b5`	`b4`	`b3`	`b2`	`b1`	`b0`
`multa_h`	`ec`	`a15`	`a14`	`a13`	`a12`	`a11`	`a10`	`a9`	`a8`
`multb_h`	`ed`	`b15`	`b14`	`b13`	`b12`	`b11`	`b10`	`b9`	`b8`

BFP Int7

Table 152: ***Int7 4 Multiplications (×1 Mode – bytesel_00_07 = 'h09; bytesel_08_15 = 'h09)***

Input Bus	[71:64]	[63:56]	[55:49]	[48:42]	[41:35]	[34:28]	[27:21]	[20:14]	[13:7]	[6:0]
`multa_l`		`ea`					`a3`	`a2`	`a1`	`a0`
`multb_l`	`eb`		`b3`	`b2`	`b1`	`b0`				
`multa_h`	Unused									
`multb_h`	Unused									

Table 153: ***Int7 8 Multiplications (×2 Mode Split – bytesel_00_07 = 'h09; bytesel_08_15 = 'h29)***

Input Bus	[71:64]	[63:56]	[55:49]	[48:42]	[41:35]	[34:28]	[27:21]	[20:14]	[13:7]	[6:0]
`multa_l`		`ea`					`a3`	`a2`	`a1`	`a0`
`multb_l`	`eb`		`b3`	`b2`	`b1`	`b0`				
`multa_h`		`ec`					`a11`	`a10`	`a9`	`a8`
`multb_h`	`ed`		`b11`	`b10`	`b9`	`b8`				

Table 154: *Int7 9 Multiplications (×2 Mode – bytesel_00_07 = 'h1b; bytesel_08_15 = 'h1b)*

Input Bus	[71:64]	63	[62:56]	[55:49]	[48:42]	[41:35]	[34:28]	[27:21]	[20:14]	[13:7]	[6:0]
`multa_l`	`ea`			`a7`	`a6`	`a5`	`a4`	`a3`	`a2`	`a1`	`a0`
`multb_l`	`eb`			`b7`	`b6`	`b5`	`b4`	`b3`	`b2`	`b1`	`b0`
`multa_h`	`ec`		`a8`								
`multb_h`	`ed`		`b8`								

Table 155: *Int7 16 Multiplications (×4 Mode – bytesel_00_07 = 'h1C; bytesel_08_15 = 'h1C)*

Input Bus	[71:64]	[63:56]	[55:49]	[48:42]	[41:35]	[34:28]	[27:21]	[20:14]	[13:7]	[6:0]
`multa_l`		`ea`	`a7`	`a6`	`a5`	`a4`	`a3`	`a2`	`a1`	`a0`
`multb_l`		`eb`	`b7`	`b6`	`b5`	`b4`	`b3`	`b2`	`b1`	`b0`
`multa_h`		`ec`	`a15`	`a14`	`a13`	`a12`	`a11`	`a10`	`a9`	`a8`
`multb_h`		`ed`	`b15`	`b14`	`b13`	`b12`	`b11`	`b10`	`b9`	`b8`

BFP Int6

Table 156: *Int6 4 Multiplications (×1 Mode – bytesel_00_07 = 'h0D; bytesel_08_15 = 'h0D)*

Input Bus	[71:64]	[63:56]	[55:50]	[49:44]	[43:38]	[37:32]	[31:24]	[23:18]	[17:12]	[11:6]	[5:0]
`multa_l`							`ea`	`a3`	`a2`	`a1`	`a0`
`multb_l`		`eb`	`b3`	`b2`	`b1`	`b0`					
`multa_h`	Unused										
`multb_h`	Unused										

Table 157: *Int6 5 Multiplications (×1 Mode – bytesel_00_07 = 'h0E; bytesel_08_15 = 'h0E)*

Input Bus	[71:64]	[63:60]	[59:54]	[53:48]	[47:42]	[41:36]	[35:30]	[29:24]	[23:18]	[17:12]	[11:6]	[5:0]
`multa_l`	`ea`							`a4`	`a3`	`a2`	`a1`	`a0`
`multb_l`(1)			`b4`	`b3`	`b2`	`b1`	`b0`					
`multa_h`	Unused											
`multb_h`	Unused											

Table Notes

1. The eb input is driven directly from the `expb[7:0]` pins.

Table 158: ***Int6 8 Multiplications (×2 Mode – bytesel_00_07 = 'h0D; bytesel_08_15 = 'h2D)***

Input Bus	[71:64]	[63:56]	[55:50]	[49:44]	[43:38]	[37:32]	[31:24]	[23:18]	[17:12]	[11:6]	[5:0]
`multa_l`							`ea`	`a3`	`a2`	`a1`	`a0`
`multb_l`		`eb`	`b3`	`b2`	`b1`	`b0`					
`multa_h`							`ec`	`a11`	`a10`	`a9`	`a8`
`multb_h`		`ed`	`b11`	`b10`	`b9`	`b8`					

Table 159: ***Int6 10 Multiplications (×2 split Mode – bytesel_00_07 = 'h0E; bytesel_08_15 = 'h2E)***

Input Bus	[71:64]	[63:60]	[59:54]	[53:48]	[47:42]	[41:36]	[35:30]	[29:24]	[23:18]	[17:12]	[11:6]	[5:0]
`multa_l`	`ea`							`a4`	`a3`	`a2`	`a1`	`a0`
`multb_l`[(1)]			`b4`	`b3`	`b2`	`b1`	`b0`					
`multa_h`	`ec`							`a12`	`a11`	`a10`	`a9`	`a8`
`multb_h`[(1)]			`b12`	`b11`	`b10`	`b9`	`b8`					

Table Notes

1. The `eb` and `ed` exponents are the same, and are both taken from the `expb[7:0]` pins.

Table 160: ***Int6 10 Multiplications (×2 split Mode – bytesel_00_07 = 'h0F; bytesel_08_15 = 'h0F)***

Input Bus	[71:64]	[63:60]	[59:54]	[53:48]	[47:42]	[41:36]	[35:30]	[29:24]	[23:18]	[17:12]	[11:6]	[5:0]
`multa_l`	`ea`				`a7`	`a6`	`a5`	`a4`	`a3`	`a2`	`a1`	`a0`
`multb_l`	`eb`				`b7`	`b6`	`b5`	`b4`	`b3`	`b2`	`b1`	`b0`
`multa_h`	`ec`		`a9`	`a8`								
`multb_h`	`ed`		`b9`	`b8`								

Table 161: ***Int6 16 Multiplications (×4 Mode – bytesel_00_07 = 'h10; bytesel_08_15 = 'h10)***

Input Bus	[71:64]	[63:56]	[55:48]	[47:42]	[41:36]	[35:30]	[29:24]	[23:18]	[17:12]	[11:6]	[5:0]
`multa_l`		`ea`		`a7`	`a6`	`a5`	`a4`	`a3`	`a2`	`a1`	`a0`
`multb_l`		`eb`		`b7`	`b6`	`b5`	`b4`	`b3`	`b2`	`b1`	`b0`
`multa_h`		`ec`		`a15`	`a14`	`a13`	`a12`	`a11`	`a10`	`a9`	`a8`
`multb_h`		`ed`		`b15`	`b14`	`b13`	`b12`	`b11`	`b10`	`b9`	`b8`

BFP Int4 and Int3

There are 32 multipliers of these types. There are no separate bytesel modes for block floating point int4 and block floating point int3. Instead, use the BFP int8 bytesel modes for BFP int4, packing two int4 arguments per int8 value; the number of mapped int4 multiplications is double the number of int8 multiplications for the same mode. Likewise, use the BFP int6 bytesel modes for BFP int3, packing two int3 arguments per int6 value.

BFP Int16

Unlike the other block floating-point modes, the BFP int16 input must be in two's complement format (there is no 16-bit signed magnitude support). A single BFP Int16 multiplication uses four multipliers, mult0, …, mult3, in the same way that four multipliers are required for integer Int16 multiplication.

Table 162: ***Int16 2 Multiplications (×1 Mode – bytesel_00_07 = 'h11; bytesel_08_15 = 'h11)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
`multa_l`	`ea`			`a1`	`a0`
`multb_l`(1)		`b1`	`b0`		
`multa_h`	Unused				
`multb_h`	Unused				

Table Notes

1. The eb input is driven directly from the `expb[7:0]` pins.

Table 163: ***Int16 4 Multiplications (×2 split Mode – bytesel_00_07 = 'h11; bytesel_08_15 = 'h31)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
`multa_l`	`ea`			`a1`	`a0`
`multb_l`(1)		`b1`	`b0`		
`multa_h`	`ec`			`a3`	`a2`
`multb_h`(1)		`b3`	`b2`		

Table Notes

1. The eb and ed exponents are the same, and are both taken from the `expb[7:0]` pins

Table 164: ***Int16 4 Multiplications (×2 Mode – bytesel_00_07 = 'h12; bytesel_08_15 = 'h12)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
`multa_l`	`ea`			`a1`	`a0`
`multb_l`	`eb`			`b1`	`b0`

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
`multa_h`	`ec`	`a3`	`a2`		
`multb_h`	`ed`	`b3`	`b2`		

Ports

Table 165: ***Block Floating-Point Inputs***

Name	Direction	Description
`expb[7:0]`	Input	Separate exponent input.

Parameters

Table 166: ***Block Floating-Point Byte Selection Parameters***

Parameter	Supported Values	Default Value	Description
`bytesel_00_07[4:0]`	`5'h00–5'h1C`	`5'h00`	`5'h03` – block floating point (BFP) Int8. 3 or 6 multiplications. `5'h04` – BFP Int8 separate expb. 4 or 8 multiplications. `5'h05` – BFP Int8 ×2/×4 mode. 8 or 16 multiplications. `5'h09` – BFP Int7 x1/×2 mode. 4 or 8 multiplications. `5'h0D` – BFP Int6. `5'h0E` – BFP Int6 separate expb. `5'h0F` – BFP Int6 ×2 mode. `5'h10` – BFP Int6 ×4 mode. `5'h1B` – BFP Int7 ×2 mode. 9 multiplications. `5'h1C` – BFP Int7 ×4 mode. 16 multiplications.
`bytesel_08_15[5:0]`	`6'h00–6'h3A`	`6'h00`	`6'h03` – BFP Int8 3 multiplications. `6'h04` – BFP Int8 separate expb. 4 multiplications. `6'h05` – BFP Int8 ×2 mode. 8 multiplications. `6'h09` – BFP Int7 ×1 mode. 4 multiplications. `6'h0D` – BFP Int6. `6'h0E` – BFP Int6 separate expb. `6'h0F` – BFP Int6 ×2 mode. `6'h10` – BFP Int6 ×4 mode. `6'h1B` – BFP Int7 ×2 mode. 9 multiplications. `6'h1C` – BFP Int7 ×4 mode. 16 multiplications. `6'h23` – BFP Int8 6 multiplications. `6'h24` – BFP Int8 separate expb. 8 multiplications. `6'h25` – BFP Int8 ×4 mode. 16 multiplications. `6'h29` – BFP Int7 ×2 mode. 8 multiplications.
`fpmult_ab_blockfp`	`1'b0–1'b1`	`1'b0`	Select (A×B) regular floating point or block floating point: `1'b0` – regular floating point (input – floating point numbers). `1'b1` – block floating point (input – integer mantissas and shared exponent).
`fpmult_ab_blockfp_mode[2:0]`	`3'b000–3'b100`	`3'b000`	Select size of integer multipliers for (A×B) block floating point: `3'b000` – 8×8. `3'b001` – 16×16. `3'b011` – 3×3. `3'b100` – 4×4. `3'b110` – 6×6. `3'b111` – 7×7.
			Select (C×D) regular floating point or block floating point:

Parameter	Supported Values	Default Value	Description
`fpmult_cd_blockfp`	`1'b0`–`1'b1`	`1'b0`	`1'b0` – regular floating point (input – floating point numbers). `1'b1` – block floating point (input – integer mantissas and shared exponent).
`fpmult_cd_blockfp_mode[2:0]`	`3'b000`–`3'b100`	`3'b000`	Select size of integer multipliers for (C×D) block floating point: `3'b000` – 8×8. `3'b001` – 16×16. `3'b011` – 3×3. `3'b100` – 4×4. `3'b110` – 6×6. `3'b111` – 7×7.

Floating-Point Modes

For single and twin floating-point multiplications or addition, use the existing ACX_MLP72 Floating-Point Library. This library consists of macros which instantiate the ACX_MLP72 suitably configured for different floating-point operations. However, if the library does not contain macros suitably configured for the user's needs, then the following details enable configuring the base ACX_MLP72 to perform a large number of differing floating-point operations.

There are two floating-point multipliers, mult_ab with inputs 'a' and b, and mult_cd with inputs c and d. In some byte selection modes there is only space for a, b, and c. In those cases, d = 1.0. This configuration can be used to compute *Result* = *a* × *b* + *c*.

Before configuring the ACX_MLP72 for floating-point operation, understand how the differing types of floating point numbers are represented and manipulated within the ACX_MLP72 as detailed in Number Formats.

Byte Selection

The following byte selection values are available for floating-point inputs. In the configurations with three inputs, resulting in a × b + c, the d input is automatically set to a value of 1.0 internal to the ACX_MLP72.

Note

BFLOAT16 refers to the Tensor flow nomenclature "Brain Float 16 bits". This term should not be confused with block floating point which is referred to as BFP.

BFLOAT16

Table 167: ***Bfloat16. a × b + c. 8-bit Exponent. d=1.0 (×1 Mode – bytesel_00_07 = 'h13; bytesel_08_15 = 'h13)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
`multa_l`					a
`multb_l`				b	
`multa_h`			c		
`multb_h`	Unused				

Table 168: ***Bfloat16. Two Multipliers. 8-bit exponent (×2 Split Mode – bytesel_00_07 = 'h13; bytesel_08_15 = 'h33)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
`multa_l`					a
`multb_l`				b	
`multa_h`					c
`multb_h`				d	

Table 169: ***Bfloat16. Two Multipliers. 8-bit exponent (×2 Mode – bytesel_00_07 = 'h14; bytesel_08_15 = 'h14)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
`multa_l`					a
`multb_l`				b	
`multa_h`			c		
`multb_h`		d			

Table 170: ***Bfloat16. Two Multipliers. 8-bit exponent (×2 Alternate Mode – bytesel_00_07 = 'h15; bytesel_08_15 = 'h15)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
`multa_l`					a
`multb_l`					b
`multa_h`				c	
`multb_h`				d	

Table 171: ***Bfloat16. Two Multipliers. 8-bit exponent (×2 Compact Mode – bytesel_00_07 = 'h15; bytesel_08_15 = 'h35)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
`multa_l`					a
`multb_l`					b
`multa_h`					c
`multb_h`					d

FP16

Table 172: ***Floating Point 16. a × b + c; 5-bit Exponent; d = 1.0 (×1 Mode – bytesel_00_07 = 'h16; bytesel_08_15 = 'h16)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
`multa_l`					a
`multb_l`				b	
`multa_h`			c		
`multb_h`	Unused				

Table 173: ***Floating Point 16. Two Multipliers ; 5-bit Exponent (×2 Split Mode – bytesel_00_07 = 'h16; bytesel_08_15 = 'h36)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
`multa_l`					a
`multb_l`				b	
`multa_h`					c
`multb_h`				d	

Table 174: ***Floating Point 16. Two Multipliers; 5-bit Exponent. (×2 Mode – bytesel_00_07 = 'h17; bytesel_08_15 = 'h17)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
`multa_l`					a
`multb_l`				b	
`multa_h`			c		
`multb_h`		d			

Table 175: ***Floating Point 16. Two Multipliers; 5-bit Exponent. (×2 Alternate Mode – bytesel_00_07 = 'h18; bytesel_08_15 = 'h18)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
`multa_l`					a
`multb_l`					b
`multa_h`				c	
`multb_h`				d	

Table 176: ***Floating Point 16. Two Multipliers; 5-bit Exponent. (×2 Compact Mode – bytesel_00_07 = 'h18; bytesel_08_15 = 'h38)***

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
multa_l					a
multb_l					b
multa_h					c
multb_h					d

FP24

Table 177: ***Floating Point 24. a × b + c. 8-bit Exponent. d = 1.0 (×1 Mode – bytesel_00_07 = 'h19; bytesel_08_15 = 'h19)***

Input Bus	[71:48]	[47:24]	[23:0]
multa_l			a
multb_l		b	
multa_h	c		
multb_h	Unused		

Table 178: ***Floating Point 24. Two Multipliers; 8-bit Exponent (×2 Split Mode – bytesel_00_07 = 'h19; bytesel_08_15 = 'h39)***

Input Bus	[71:48]	[47:24]	[23:0]
multa_l			a
multb_l		b	
multa_h			c
multb_h		d	

Table 179: ***Floating Point 24. Two Multipliers; 8-bit Exponent. (×2 Mode – bytesel_00_07 = 'h1A; bytesel_08_15 = 'h1A)***

Input Bus	[71:48]	[47:24]	[23:0]
multa_l			a
multb_l			b
multa_h		c	
multb_h		d	

Table 180: ***Floating Point 24. Two Multipliers; 8-bit Exponent (×2 Compact Mode – bytesel_00_07 = 'h1A; bytesel_08_15 = 'h3A)***

Input Bus	[71:48]	[47:24]	[23:0]
`multa_l`			a
`multb_l`			b
`multa_h`			c
`multb_h`			d

Parameters

Table 181: ***Floating-Point Byte Selection Parameters***

Parameter	Supported Values	Default Value	Description
`bytesel_00_07[4:0]`	`5'h00–5'h1C`	`5'h00`	`5'h13` – BFLOAT16. 1 or 2 multiplications. `5'h14` – BFLOAT16. 2 multiplications. `5'h15` – BFLOAT16. 2 multiplications. `5'h16` – FP16. 1 or 2 multiplications. `5'h17` – FP16. 2 multiplications. `5'h18` – FP16. 2 multiplications. `5'h19` – FP24. 1 or 2 multiplications. `5'h1A` – FP24. 2 multiplications.
`bytesel_08_15[5:0]`	`6'h00–6'h3A`	`6'h00`	`6'h13` – BFLOAT16. ×1 mode. 1 multiplication. `6'h14` – BFLOAT16. ×2 mode. 2 multiplications. `6'h15` – BFLOAT16. ×2 alternate mode. 2 multiplications. `6'h16` – FP16. ×1 mode. 1 multiplications. `6'h17` – FP16. ×2 mode. 2 multiplications. `6'h18` – FP16. ×2 alternate mode. 2 multiplications. `6'h19` – FP24. ×1 mode. 1 multiplication. `6'h1A` – FP24. ×2 mode. 2 multiplications. `6'h33` – BFLOAT16. ×2 split mode. 2 multiplications. `6'h35` – BFLOAT16. ×2 compact mode. 2 multiplications. `6'h36` – FP16. ×2 split mode. 2 multiplications. `6'h38` – FP16. ×2 compact mode. 2 multiplications. `6'h39` – FP24. ×2 split mode. 2 multiplications. `6'h3A` – FP24. ×2 split mode. 2 multiplications.

Multiplication Stage

The ACX_MLP72 floating-point multiplication stage consists of two 24-bit full floating-point multipliers, and a 24-bit full floating-point adder. The two multipliers perform parallel calculations of A×B and C×D. The adder sums the two results to provide A×B + C×D.

There are two outputs from the multiplication stage. The lower half output can be selected between A×B, or (A×B + C×D). The upper half output is always C×D.

The numerical formats used by the multipliers and adder are determined by the format set by the byte selection parameters, and in addition, the `fpmult_ab_exp_size` and `fpmult_cd_exp_size` parameters.

Warning!

The `fpmult_ab_exp_size` and `fpmult_cd_exp_size` parameters must be consistent with the byte selection (`bytesel_xx_xx`) parameters in terms of the selected number format. If they are inconsistent, the final output result will be incorrect.

The following diagram shows the floating-point multiplication stage. The sign and exponent inputs are sourced from the input selection and byte selection multiplexers. There are optional multi-stage delay registers for the sign and exponent paths, and single delay registers for the multiplier outputs.

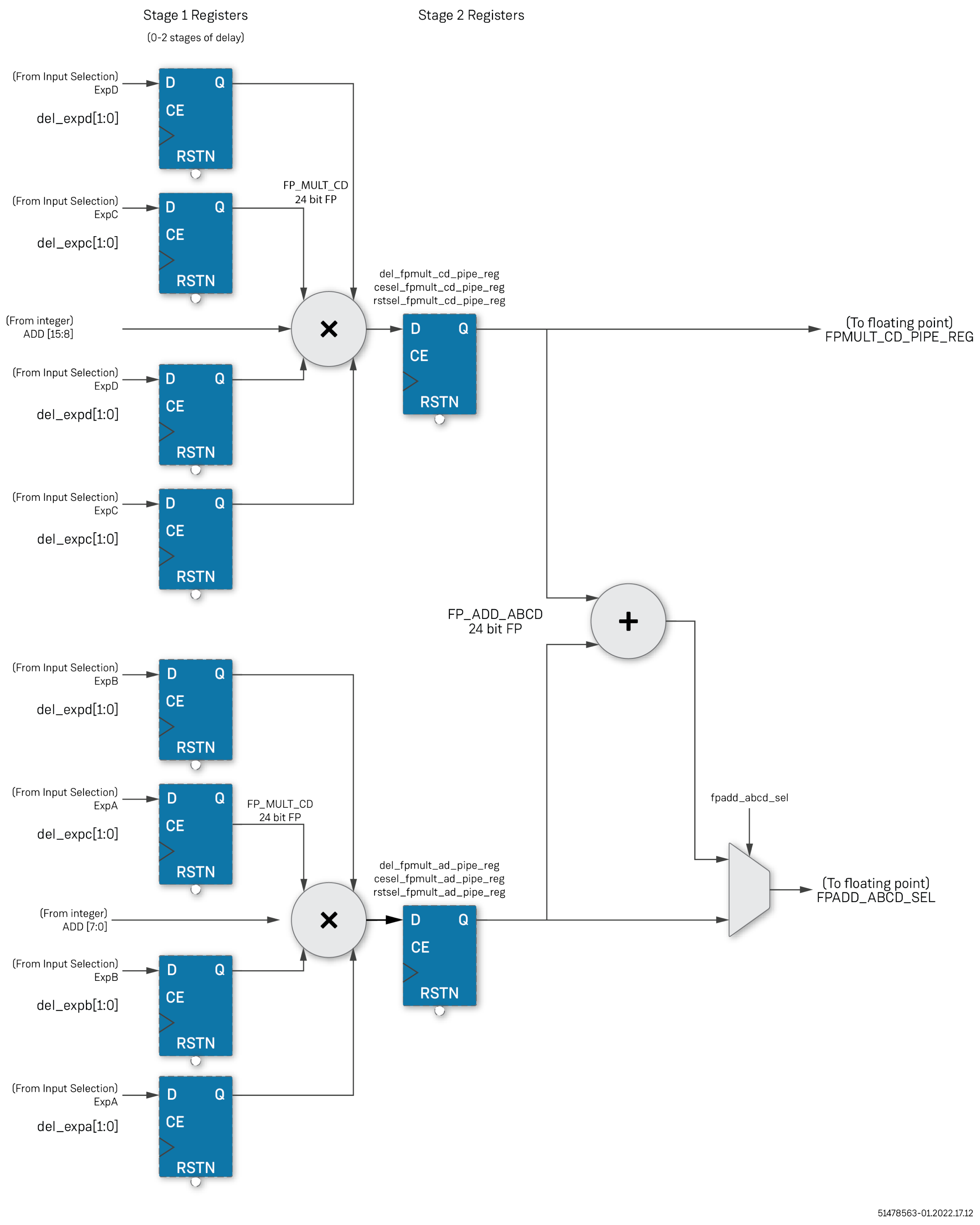


Figure 97: ***Floating-Point Multiplier Stage***

Parameters

Table 182: ***Floating-Point Multiplication Stage Parameters***

Parameter	Supported Values	Default Value	Description
`del_expa_reg[1:0]`	`2'b00–2'b11`	`2'b00`	Number of delay stages applied to floating point A input sign and exponent from byte selection to FP_MULT_AB.
`del_expb_reg[1:0]`	`2'b00–2'b11`	`2'b00`	Number of delay stages applied to floating point B input sign and exponent from byte selection to FP_MULT_AB.
`del_expc_reg[1:0]`	`2'b00–2'b11`	`2'b00`	Number of delay stages applied to floating point C input sign and exponent from byte selection to FP_MULT_CD.
`del_expd_reg[1:0]`	`2'b00–2'b11`	`2'b00`	Number of delay stages applied to floating point D input sign and exponent from byte selection to FP_MULT_CD.
`fpadd_abcd_sel`	`1'b0–1'b1`	`1'b0`	FPADD_ABCD select: `1'b0` – FPMULT_AB output routed to FPMULT_AB_REG. `1'b1` – Sum of FPMULT_AB + FPMULT_CD output routed to FPMULT_AB_REG.
`fpmult_ab_blockfp`	`1'b0–1'b1`	`1'b0`	Select (A×B) regular floating point or block floating point: `1'b0` – Regular floating point (input – floating-point numbers). `1'b1` – Block floating point (input – integer mantissas and shared exponent).
`fpmult_ab_exp_size`	`1'b0–1'b1`	`1'b0`	Exponents ea and eb are represented by biased unsigned integers ea and eb: `1'b0` – Bits ea/eb are 8 bits. `1'b1` – Bits ea/eb are 5 bits.
`fpmult_cd_blockfp`	`1'b0–1'b1`	`1'b0`	Select (C×D) regular floating point or block floating point: `1'b0` – Regular floating point (input – floating point numbers). `1'b1` – Block floating point (input – integer mantissas and shared exponent).
`fpmult_cd_exp_size`	`1'b0–1'b1`	`1'b0`	Exponents ec and ed are represented by biased unsigned integers ec and ed: `1'b0` – Bits ec/ed are 8 bits. `1'b1` – Bits ec/ed are 5 bits.

Output Stage

The floating-point output stage has a common path and structure to the integer output stage. The ACX_MLP72 can be configured to select either the integer or the equivalent floating-point inputs at particular stages. The output supports two 24-bit full floating-point adders which can be configured for either addition or accumulation. Further the adders can be loaded (to start an accumulation), can be set for subtraction, and support optional rounding modes.

The final output stage supports formatting the floating-point output to any one of the three floating-point formats supported within the ACX_MLP72. This ability allows the ACX_MLP72 to externally support consistently sized floating-point inputs and outputs (such as fp16 or bfloat16), while internally performing all calculations at fp24.

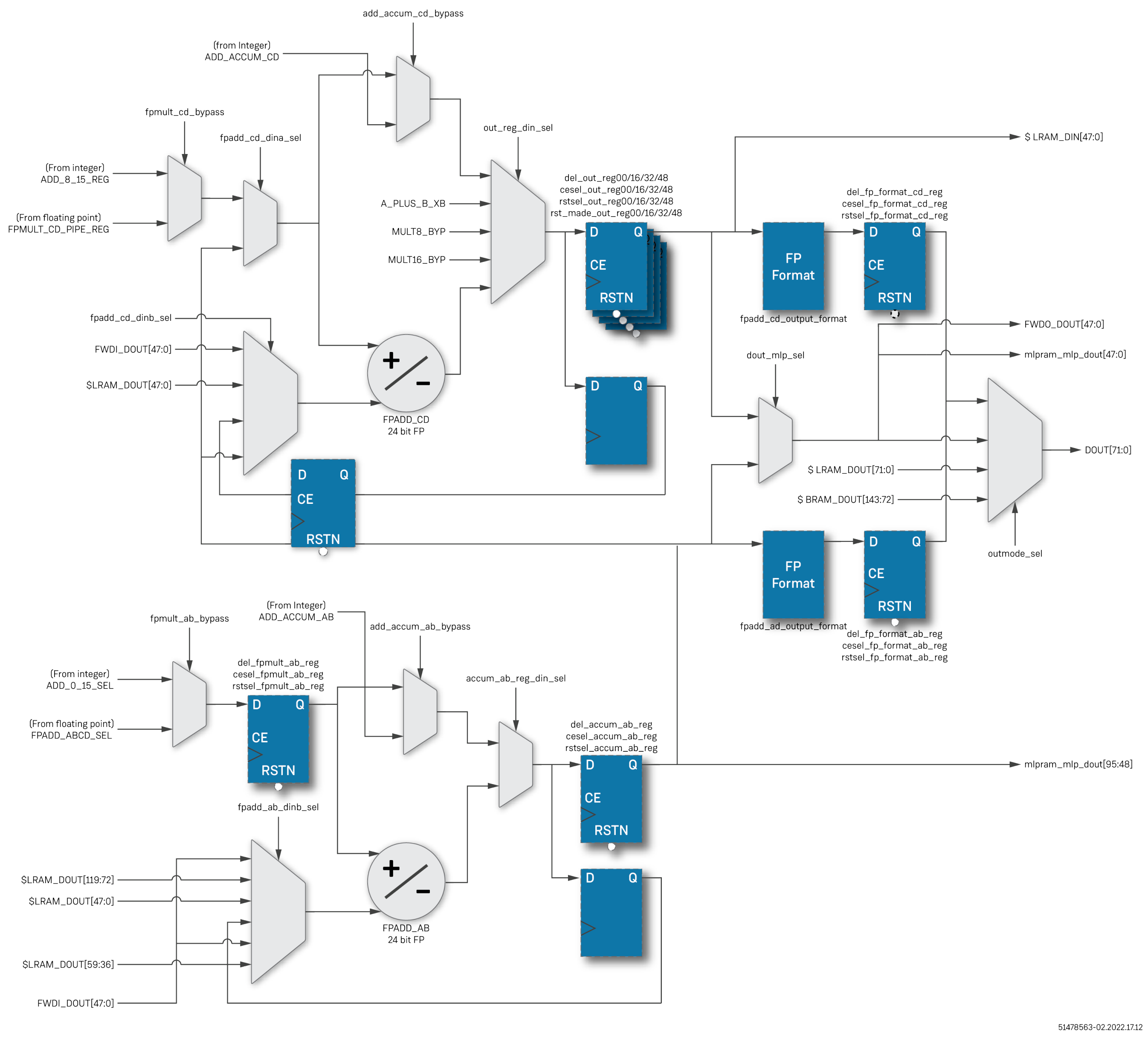


Figure 98: *Floating-Point Output Stage*

OUT_REG

The optional delay register outputting the top-half (CD) calculation is titled OUT_REG. This register bank is 64 bits and can optionally be enabled and reset in four banks of 16 bits each. This feature enables for power saving if the required output is less than 64 bits. Only the required banks need be enabled; the other banks can be left out of circuit or held in reset.

Parameters

Table 183: ***Floating-Point Output Stage Parameters***

Parameter	Supported Values	Default Value	Description
`accum_ab_reg_din_sel`	`1'b0–1'b1`	`1'b0`	Select between integer and floating-point AB result: `1'b0` – Value from integer AB accumulator block. `1'b1` – Value from floating-point AB accumulator block.
`add_accum_ab_bypass`	`1'b0–1'b1`	`1'b0`	Select to bypass the AB accumulator output: `1'b0` – AB accumulator value is used. `1'b1` – Bypass AB accumulator.
`add_accum_cd_bypass`	`1'b0–1'b1`	`1'b0`	Select to bypass the CD accumulator output: `1'b0` – CD accumulator value is used. `1'b1` – Bypass CD accumulator.
`dout_mlp_sel[1:0]`	`2'b00–2'b10`	`2'b00`	Select individual or concatenated results from OUT_REG and ACCUM_AB_REG: `2'b00` – Value from optionally registered output {8'h0, OUT_REG[63:0]}. `2'b01` – Concatenated outputs of upper and lower MLP outputs {24'h0, ACCUM_AB_REG[23:0], OUT_REG[23:0]}. `2'b10` – Value from optionally registered output {24'h0, ACCUM_AB_REG [47:0]}. `2'b11` – Concatenated lower 36 bits from upper and lower MLP outputs {ACCUM_AB_REG[35:0], OUT_REG[35:0]}.
`fpadd_ab_nornd`	`1'b0–1'b1`	`1'b0`	Disable FPADD_AB adder/accumulator rounding: `1'b0` – FPADD_AB round to even mode. `1'b1` – FPADD_AB rounding disabled (truncation).
`fpadd_ab_dinb_sel[2:0]`	`3'b000–3'b101`	`3'b000`	Select the addend, or subtrahend for the FPADD_AB adder/accumulator: `3'b000` – 48-bit ACCUM_AB_REG input (always registered). `3'b001` – 48-bit MLP forward cascaded input FWDI_DOUT[47:0]. `3'b010` – 48-bit LRAM_DOUT[47:0]. `3'b011` – 24-bit LRAM_DOUT[59:36] (top 24 bits tied to zero). `3'b100` – 24-bit MLP forward cascade input FWDI_DOUT[47:24] (top 24 bits tied to zero). `3'b101` – 48-bit LRAM_DOUT[119:72].
`fpadd_ab_output_format[1:0]`	`2'b00–2'b10`	`2'b00`	Selection of floating-point output format of FPADD_AB floating-point adder /accumulator: `2'b00` – Output format is FP24. `2'b01` – Output format is BFLOAT16. `2'b10` – Output format is FP16.
`fpadd_cd_dina_sel`	`1'b0–1'b1`	`1'b0`	Select the value between (C×D) floating-point multiplier and (A×B) accumulator: `1'b0` – Select the output from the (C×D) floating-point multiplier. `1'b1` – Select the output from the (A×B) accumulator.
`fpadd_cd_dinb_sel[2:0]`	`3'b000–3'b100`	`3'b000`	Select the addend, or subtrahend for the CD accumulator: `3'b000` –48-bit ACCUM_CD_REG input (registered). `3'b001` – 48-bit MLP forward cascaded input FWDI_DOUT[47:0]. `3'b010` – 48-bit LRAM_DOUT[47:0]. `3'b011` – Reserved. `3'b100` – 48-bit AB Accumulator data output.
`fpadd_cd_nornd`	`1'b0–1'b1`	`1'b0`	Disable FPADD_CD rounding: `1'b0` – FPADD_CD round to even mode. `1'b1` – FPADD_CD rounding disabled (truncation).

Parameter	Supported Values	Default Value	Description
`fpadd_cd_output_format[1:0]`	`2'b00`–`2'b10`	`2'b00`	Selection of floating-point output format from FPADD_CD floating-point adder /accumulator: `2'b00` – Output format is FP24. `2'b01` – Output format is BFLOAT16. `2'b10` – Output format is FP16.
`fpmult_ab_bypass`	`1'b0`–`1'b1`	`1'b0`	Select to bypass (A×B) floating-point multiplier: `1'b0` – Floating-point Multiplier output is selected. `1'b1` – Integer multiplier output is selected.
`fpmult_cd_bypass`	`1'b0`–`1'b1`	`1'b0`	Select to bypass (C×D) floating-point multiplier: `1'b0` – Floating-point multiplier output is selected. `1'b1` – Integer multiplier output is selected.
`out_reg_din_sel[2:0]`	`3'b000`–`3'b100`	`2'b00`	Select to bypass floating-point value and accumulator value: `3'b000` – Value is from Mult8×4. `3'b010` – FP_ADD_CD floating-point value. `3'b011` – Bypass FP_ADD_CD accumulator value. `3'b100` – 8-wide A +/– B output. `3'b110` – Value is Mult16×2.
`outmode_sel[1:0]`	`2'b00`–`2'b10`	`2'b00`	Select source of MLP DOUT: `2'b00` – 72-bit output of value selected by parameter dout_mlp_sel[1:0]. `2'b01` – LRAM_DOUT[71:0]. `2'b10` – BRAM_DOUT[143:72]. `2'b11` – Optionally registered concatenated outputs of floating-point format conversion registers with status {20'h0, w_fp_ab_status_reg, w_fp_cd_status_reg, w_accum_ab_reg_output_format_reg, w_out_reg_output_format_reg}.

Instantiation Template

Verilog

```
ACX_MLP72 #(
    .mux_sel_multa_l                (mux_sel_multa_l),
    .mux_sel_multa_h                (mux_sel_multa_h),
    .mux_sel_multb_l                (mux_sel_multb_l),
    .mux_sel_multb_h                (mux_sel_multb_h),
    .del_multa_l                    (del_multa_l),
    .del_multa_h                    (del_multa_h,),
    .del_multb_l                    (del_multb_l),
    .del_multb_h                    (del_multb_h),
    .cesel_multa_l                  (cesel_multa_l),
    .cesel_multa_h                  (cesel_multa_h),
    .cesel_multb_l                  (cesel_multb_l),
    .cesel_multb_h                  (cesel_multb_h),
    .rstsel_multa_l                 (rstsel_multa_l),
    .rstsel_multa_h                 (rstsel_multa_h),
    .rstsel_multb_l                 (rstsel_multb_l),
    .rstsel_multb_h                 (rstsel_multb_h),
    .del_mult00a                    (del_mult00a),
    .del_mult01a                    (del_mult01a),
    .del_mult02a                    (del_mult02a),
    .del_mult03a                    (del_mult03a),
    .del_mult04_07a                 (del_mult04_07a),
    .del_mult08_11a                 (del_mult08_11a),
    .del_mult12_15a                 (del_mult12_15a),
    .del_mult00a                    (del_mult00a),
    .del_mult01a                    (del_mult01a),
    .del_mult02a                    (del_mult02a),
    .del_mult03a                    (del_mult03a),
    .del_mult04_07a                 (del_mult04_07a),
    .del_mult08_11a                 (del_mult08_11a),
    .del_mult12_15a                 (del_mult12_15a),
    .cesel_mult00a                  (cesel_mult00a),
    .cesel_mult01a                  (cesel_mult01a),
    .cesel_mult02a                  (cesel_mult02a),
    .cesel_mult03a                  (cesel_mult03a),
    .cesel_mult04_07a               (cesel_mult04_07a),
    .cesel_mult08_11a               (cesel_mult08_11a),
    .cesel_mult12_15a               (cesel_mult12_15a),
    .rstsel_mult00a                 (rstsel_mult00a),
    .rstsel_mult01a                 (rstsel_mult01a),
    .rstsel_mult02a                 (rstsel_mult02a),
    .rstsel_mult03a                 (rstsel_mult03a),
    .rstsel_mult04_07a              (rstsel_mult04_07a),
    .rstsel_mult08_11a              (rstsel_mult08_11a),
    .rstsel_mult12_15a              (rstsel_mult12_15a),
    .bytesel_00_07                  (bytesel_00_07),
    .bytesel_08_15                  (bytesel_08_15),
    .multmode_00_07                 (multmode_00_07),
    .multmode_08_15                 (multmode_08_15),
    .add_00_07_bypass               (add_00_07_bypass),
    .add_08_15_bypass               (add_08_15_bypass),
    .del_add_00_07_reg              (del_add_00_07_reg),
    .del_add_08_15_reg              (del_add_08_15_reg),
    .cesel_add_00_07_reg                (cesel_add_00_07_reg),
    .cesel_add_08_15_reg                (cesel_add_08_15_reg),
    .rstsel_add_00_07_reg               (rstsel_add_00_07_reg),
    .rstsel_add_08_15_reg               (rstsel_add_08_15_reg),
    .add_00_15_sel                      (add_00_15_sel),
    .fpmult_ab_bypass                   (fpmult_ab_bypass),
    .fpmult_cd_bypass                   (fpmult_cd_bypass),
    .fpadd_ab_dinb_sel                  (fpadd_ab_dinb_sel),
    .add_accum_ab_bypass                (add_accum_ab_bypass),
    .accum_ab_reg_din_sel               (accum_ab_reg_din_sel),
    .del_accum_ab_reg                   (del_accum_ab_reg),
    .cesel_accum_ab_reg                 (cesel_accum_ab_reg),
    .rstsel_accum_ab_reg                (rstsel_accum_ab_reg),
    .rndsubload_share                   (rndsubload_share),
    .del_rndsubload_reg                 (del_rndsubload_reg),
    .cesel_rndsubload_reg               (cesel_rndsubload_reg),
    .rstsel_rndsubload_reg              (rstsel_rndsubload_reg),
    .dout_mlp_sel                       (dout_mlp_sel),
    .outmode_sel                        (outmode_sel),
) i_mlp72 (
    .clk                                (clk),
    .din                                (din),
    .mlpram_bramdout2mlp                (mlpram_bramdout2mlp),
    .mlpram_bramdin2mlpdin              (mlpram_bramdin2mlpdin),
    .mlpram_mlp_dout                    (mlpram_mlp_dout),
    .sub                                (sub),
    .load                               (load),
    .sub_ab                             (sub_ab),
    .load_ab                            (load_ab),
    .ce                                 (ce),
    .rstn                               (rstn),
    .expb                               (expb),
    .dout                               (dout),
    .sbit_error                         (sbit_error),
    .dbit_error                         (dbit_error),
    .full                               (full),
    .almost_full                        (almost_full),
    .empty                              (empty),
    .almost_empty                       (almost_empty),
    .write_error                        (write_error),
    .read_error                         (read_error),
    .fwdo_multa_h                       (fwdo_multa_h),
    .fwdo_multb_h                       (fwdo_multb_h),
    .fwdo_multa_l                       (fwdo_multa_l),
    .fwdo_multb_l                       (fwdo_multb_l),
    .fwdo_dout                          (fwdo_dout),
    .mlpram_din                         (mlpram_din),
    .mlpram_dout                        (mlpram_dout),
    .mlpram_we                          (mlpram_we),
    .fwdi_multa_h                       (fwdi_multa_h),
    .fwdi_multb_h                       (fwdi_multb_h),
    .fwdi_multa_l                       (fwdi_multa_l),
    .fwdi_multb_l                       (fwdi_multb_l),
    .fwdi_dout                          (fwdi_dout),
    .mlpram_din2mlpdout                 (mlpram_din2mlpdout),
    .mlpram_rdaddr                      (mlpram_rdaddr),
    .mlpram_wraddr                      (mlpram_wraddr),
    .mlpram_dbit_error                  (mlpram_dbit_error),
    .mlpram_rden                        (mlpram_rden),
    .mlpram_sbit_error              (mlpram_sbit_error),
    .mlpram_wren                    (mlpram_wren),
    .lram_wrclk                     (lram_wrclk),
    .lram_rdclk                     (lram_rdclk)
);
```

MLP72_INT

The ACX_MLP72_INT supports up to 12 integer multiply operations, followed by an adder tree and an optional accumulate. The number of arithmetic operations that can be supported depends on the operand width, where more arithmetic operations can be supported per clock cycle with narrower operands. Inputs can be encoded as unsigned integers, signed two's-complement integers, or signed-magnitude integers. Outputs are always 48-bit signed integers.

The supported arithmetic equations are as follows. The first equation represents the functionality of the block when the accumulator is disabled, and the second represents the functionality of the block when the accumulator is enabled, and `dout'` is the previous value of the accumulator block. The number of operations as a function of operand width are as shown.

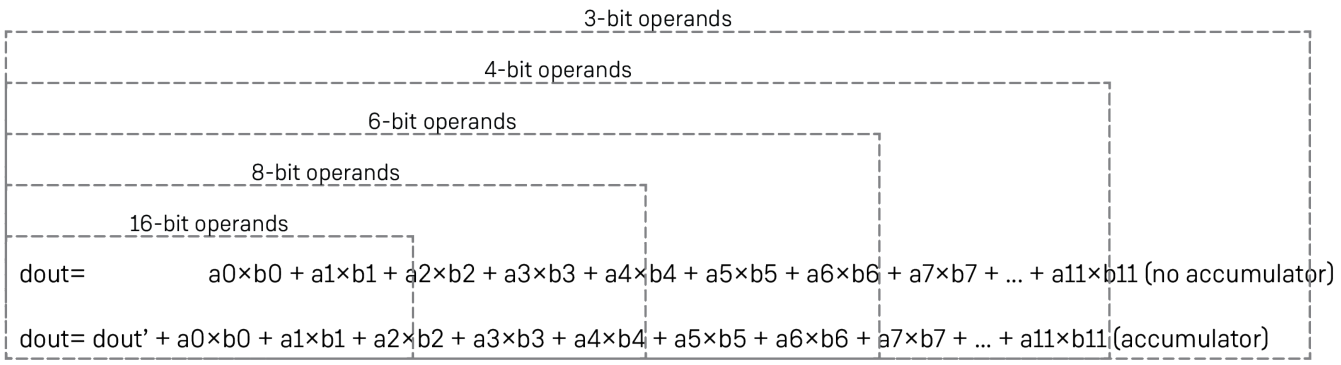

3-bit operands
4-bit operands
6-bit operands
8-bit operands
16-bit operands
dout= a0×b0 + a1×b1 + a2×b2 + a3×b3 + a4×b4 + a5×b5 + a6×b6 + a7×b7 + ... + a11×b11 (no accumulator)
dout= dout' + a0×b0 + a1×b1 + a2×b2 + a3×b3 + a4×b4 + a5×b5 + a6×b6 + a7×b7 + ... + a11×b11 (accumulator)
37160452-01.2022.16.11

Figure 99: ***ACX_MLP72_INT Arithmetic Expressions***

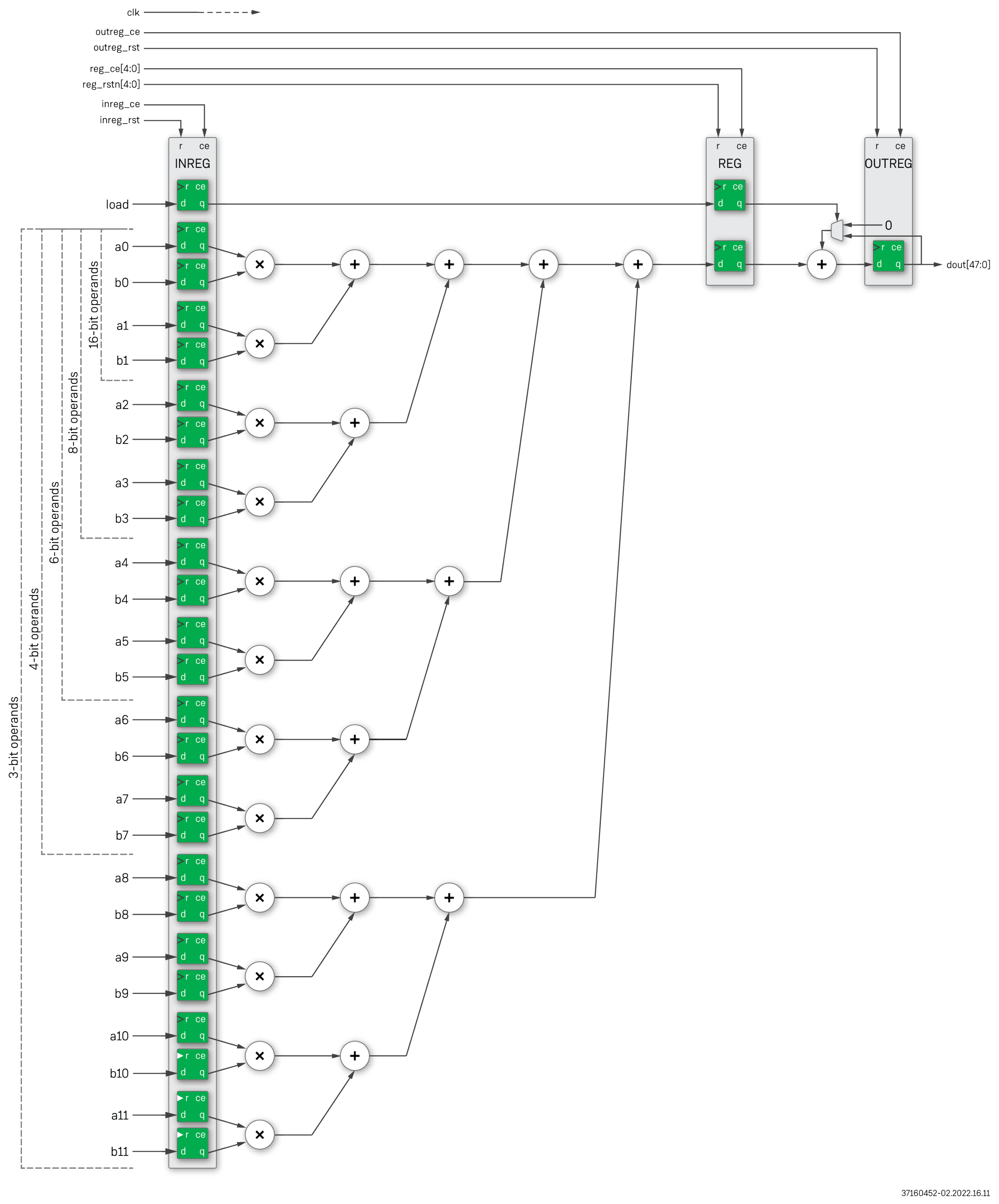


Figure 100: *ACX_MLP72_INT Block Diagram*

Parameters

Table 184: *ACX_MLP72_INT Parameters*

Parameter	Supported Values	Default Value	Description
`clk_polarity`	"rise", "fall"	"rise"	Determines which edge of the input clock to use: "rise" – rising edge of clock. "fall" – falling edge of clock.
`operand_width`	3, 4, 6, 7, 8, 16	8	Determines the width of the a and b input operands.
`number_format`	0, 1, 2, 3, 4	1	Determines the format of the input operands and the output result: 0 – unsigned (only supported for operand_width of 8 and 16). 1 – signed two's complement. 2 – signed-magnitude (only supported for operand_width of 8 or less). 3 – unsigned "A" input with signed "B" input (only supported for operand_width of 16). 4 – signed "A" input with unsigned "B" input (only supported for operand_width of 16).
`accumulator_enable`	0, 1	1	Controls whether or not the optional accumulator is enabled: 0 – accumulator is not enabled. 1 – accumulator is enabled.

`inreg_enable`
`reg_enable`
`outreg_enable`
	0, 1	0	Controls whether or not the input register, intermediate registers and output register is enabled:
			0 – disable the register.
			1 – enable the register. Results in extra latency.

`inreg_sr_assertion`	"clocked", "unclocked"	"clocked"	Controls whether the assertion of the reset of the input registers is synchronous or asynchronous with respect to the `clk` input: "clocked" – synchronous reset; the register is reset upon the next rising edge of the clock when the associated `rstn` signal is asserted low. This mode is supported for all `operand_widths`. "unclocked" – asynchronous reset. The register is reset immediately when the associated `rstn` signal is asserted low. See the section, Asynchronous Reset Rules, (see page 220) for more details.
`outreg_sr_assertion`	"clocked", "unclocked"	"clocked"	Controls whether the assertion of the reset of the output registers is synchronous or asynchronous with respect to the `clk` input: "clocked" – synchronous reset. The register is reset upon the next rising edge of the clock when the associated `rstn` signal is asserted low. "unclocked" – asynchronous reset. The register is reset immediately when the associated `rstn` signal is asserted low.

Ports

Table 185: *ACX_MLP72_INT Pin Descriptions*

Name	Direction	Description
`clk`	Input	Clock input. If input or output registers are enabled, they are updated on the active edge of this clock.
`load`	Input	When the accumulator is enabled, this signal controls when to accumulate versus load the accumulator with the newly calculated sums (without accumulating). The `load` signal is also registered if `inreg_enable` is enabled.
`din[71:0]`	Input	Data inputs.

`inreg_rstn`
`reg_rstn`
`outreg_rstn`
	Input	Register reset signal for each register stage. When the register reset signal for each register stage is asserted, a value of 0 is written to all of the registers in that register stage on the rising edge of `clk`. This signal has no effect when the register is disabled.

`inreg_ce`
`reg_ce`
`outreg_ce`
	Input	Register clock enable signal for each register stage. Asserting the register clock enable signal for a register stage causes it to capture that data at its input on the rising edge of `clk`. This signal has no effect when the register is disabled.

`dout[47:0]`	Output	The result of the multiply-accumulate operation.

Input Data Mapping

The assignment of the 72-bit input data to the 'a' and 'b' operands is as shown in the following table. The data input is easily assigned as a single concatenation, such as (for 8-bit mode):

```
din = {a0, a1, a2, a3, b0, b1, b2, b2};
```

Table 186: ***A Operand Input Data Mapping***

A Operands	Input Widths					
	3-bit	4-bit	6-bit	7-bit	8-bit	16-bit
a0	din[2:0]	din[3:0]	din[5:0]	din[6:0]	din[7:0]	din[15:0]
a1	din[5:3]	din[7:4]	din[11:6]	din[13:7]	din[15:8]	din[31:16]
a2	din[8:6]	din[11:8]	din[17:12]	din[20:14]	din[23:16]	
a3	din[11:9]	din[15:12]	din[23:18]	din[27:21]	din[31:24]	
a4	din[14:12]	din[19:16]	din[29:24]	din[34:28]		
a5	din[18:15]	din[23:20]	din[35:30]			
a6	din[20:18]	din[27:24]				
a7	din[23:21]	din[31:28]				
a8	din[26:24]					
a9	din[29:27]					
a10	din[32:30]					
a11	din[35:33]					

Table 187: ***B Operand Input Data Mapping***

B Operands	Input Widths					
	3-bit	**4-bit**	**6-bit**	**7-bit**	**8-bit**	**16-bit**
b0	din[38:36]	din[35:32]	din[41:36]	din[41:35]	din[39:32]	din[47:32]
b1	din[41:39]	din[39:36]	din[47:42]	din[48:42]	din[47:40]	din[63:48]
b2	din[44:42]	din[43:40]	din[53:48]	din[55:49]	din[55:48]	
b3	din[47:45]	din[47:44]	din[59:54]	din[62:56]	din[63:56]	
b4	din[50:48]	din[51:48]	din[65:60]	din[69:63]		
b5	din[49:51]	din[55:52]	din[71:66]			
b6	din[56:54]	din[59:56]				
b7	din[59:57]	din[63:60]				
b8	din[62:60]					
b9	din[65:63]					
b10	din[68:66]					
b11	din[71:69]					

Output Formatting and Error Conditions

The number format of the data output is the same as the format of the data input, as controlled by the number_format parameter. The output register is always 48 bits wide, regardless of the number format or input data width.

Asynchronous Reset Rules

Asynchronous reset mode on input registers, `inreg_sr_assertion="unclocked"`, is only supported in the lower four internal multiply units. The upper multiply units only support `inreg_sr_assertion="clocked"`. Therefore, to use `inreg_sr_assertion="unclocked"`, do one of the following:

- Tie off the upper multipliers and do not use them
- Set `inreg_enable=0` and replace the input registers of the MLP with fabric DFFs

Note

For optimal MLP performance on upper multipliers, use synchronous ("clocked") resets in a design.

When `accumulator_enable` is set to 1, then set `inreg_sr_assertion="clocked"`; `inreg_sr_assertion="unclocked"` is not supported when using the accumulator feature. The following table describes valid scenarios when `inreg_sr_assertion` can be set to `"unclocked"` when the input register is enabled.

Table 188: *ACX_MLP72_INT Asynchronous Reset Rules*

operand_width	accumulator_enable	Multipliers For Use With inreg_sr_assertion of "unclocked"	Multipliers to be Tied Off and Not Used with inreg_sr_assertion of "unclocked"
3	0	Lower 8 multipliers (sets of A/B inputs).	Upper 4 multipliers (sets of A/B inputs).
4	0	All 8 multipliers (sets of A/B inputs).	
6	0	Lower 4 multipliers (sets of A/B inputs).	Upper 2 multipliers (sets of A/B inputs).
7	0	Lower 4 multipliers (sets of A/B inputs).	Upper 1 multiplier (set of A/B inputs).
8	0	All 4 multipliers (sets of A/B inputs).	
16	0	Lower 1 multiplier (set of A/B inputs).	Upper 1 multiplier (set of A/B inputs).

Inference

The ACX_MLP72_INT is inferrable using RTL constructs commonly used to infer multiplication and addition operations, such as those shown.

Data widths which fall between the supported values infer the next largest input size and, if appropriate, sign extend the input when it is defined as a signed value. For example, 9-bit signed signals would be extended to be 16-bit signed inputs of the ACX_MLP72_INT.

Examples

inreg_enable=0, outreg_enable=0, 4 inputs

```
x = a0 * b0 + a1 * b1 + a2 * b2 + a3 * b3;
```

inreg_enable=0, outreg_enable=1

```
always @(posedge clk) begin
  x <= a0 * b0 + a1 * b1 + a2 * b2 + a3 * b3;
end
```

inreg_enable=0, outreg_enable=1, Asynchronous Reset

```
always @(posedge clk, negedge rstn) begin
  if (rstn == 1'b0)
    x <= 'h0;
  else if (en == 1'b1)
    x <= a0 * b0 + a1 * b1 + a2 * b2 + a3 * b3;
end
```

Instantiation Template

Verilog

```
   ACX_MLP72_INT #(
       .clk_polarity        (clk_polarity        ),
       .operand_width       (operand_width       ),
       .number_format       (number_format       ),
       .accumulator_enable  (accumulator_enable  ),
       .inreg_enable        (inreg_enable        ),
       .reg_enable          (reg_enable          ),
       .outreg_enable       (outreg_enable       ),
       .inreg_sr_assertion  (inreg_sr_assertion  ),
       .outreg_sr_assertion (outreg_sr_assertion )
   ) instance_name (
      .clk                     (clk         ),
      .load                    (load        ),
      .din                     (din         ),
      .inreg_rstn              (inreg_rstn  ),
      .inreg_ce                (inreg_ce    ),
      .reg_rstn             (reg_rstn    ),
      .reg_ce               (reg_ce      ),
      .outreg_rstn             (outreg_rstn ),
      .outreg_ce               (outreg_ce   ),
      .dout                    (dout        )
   );
```

MLP72_INT8_MULT_4X

The ACX_MLP72_INT8_MULT_4X primitive is a simple multiplier block with support for up to four parallel multipliers using 8-bit two's-complement signed, signed magnitude, or unsigned integers. For higher performance operation, additional input and/or output registers can be enabled. Enabling each register causes an additional cycle of latency.

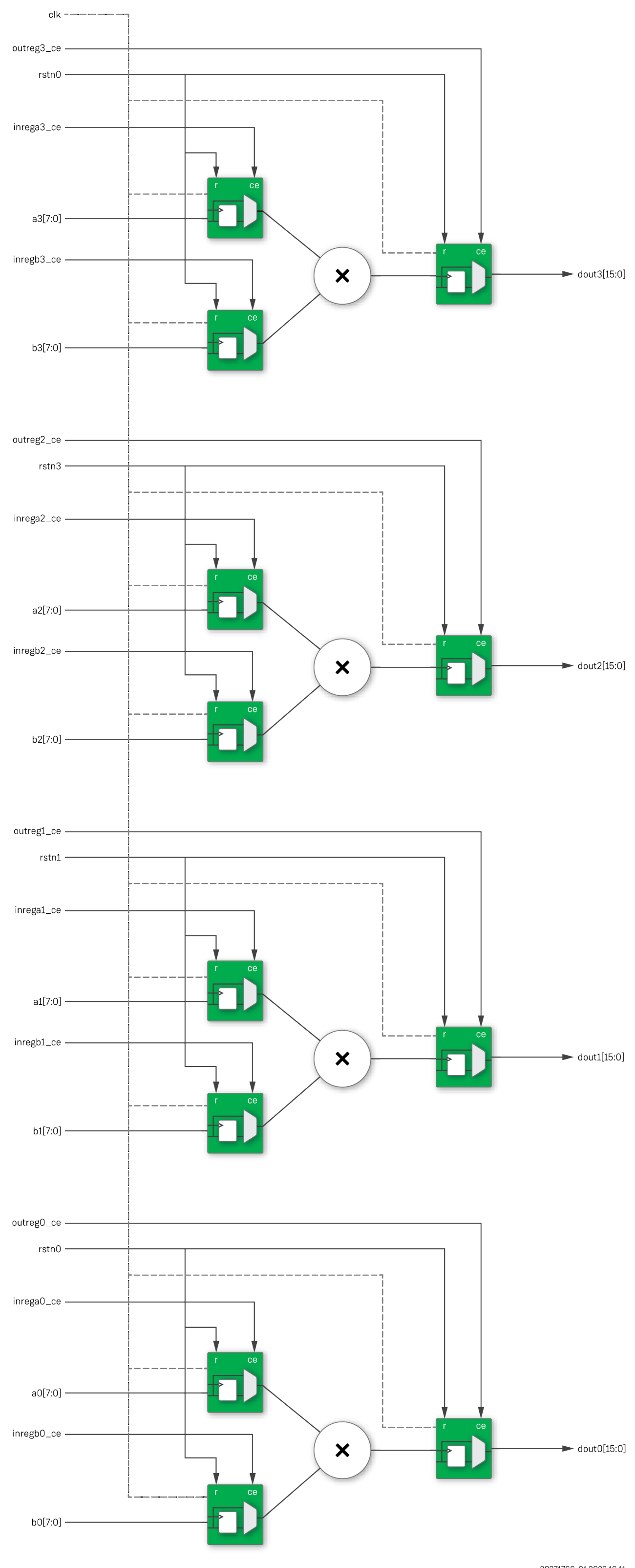


Figure 101: ***ACX_MLP72_INT8_MULT_4X Block Diagram***

Parameters

Table 189: ***ACX_MLP72_INT8_MULT_4X Parameters***

Parameter	Supported Values	Default Value	Description
`clk_polarity`	"rise", "fall"	"rise"	Controls which edge of the input clock to use: "rise" – rising edge of clock. "fall" – falling edge of clock.
`number_format`	0, 1, 2	0	Controls the number format to use for all data inputs for each of the four multipliers: 0 – unsigned. 1 – signed two's complement. 2 – signed-magnitude.

`inrega3_enable`
`inregb3_enable`
`inrega2_enable`
`inregb2_enable`
`inrega1_enable`
`inregb1_enable`
`inrega0_enable`
`inregb0_enable`
`outreg3_enable`
`outreg2_enable`
`outreg1_enable`
`outreg0_enable`
	0, 1	0	Controls whether or not the input and output registers are enabled:
			0 – disable the register.
			1 – enable the register. Results in extra latency.

`inrega3_sr_assertion`
`inregb3_sr_assertion`
`inrega2_sr_assertion`
`inregb2_sr_assertion`
`inrega1_sr_assertion`
`inregb1_sr_assertion`
`inrega0_sr_assertion`
`inregb0_sr_assertion`
`outreg3_sr_assertion`
`outreg2_sr_assertion`
`outreg1_sr_assertion`
`outreg0_sr_assertion`
	"clocked"," unclocked"	"clocked"	Controls whether the assertion of the reset of the input and output registers is synchronous or asynchronous with respect to the `clk` input:
			"clocked" – synchronous reset. The register is reset upon the next rising edge of the clock when the associated `rstn` signal is asserted low.
			"unclocked" – asynchronous reset. The register is reset immediately when the associated `rstn` signal is asserted low.

Ports

Table 190: ***ACX_MLP72_INT8_MULT_4X Pin Descriptions***

Name	Direction	Description
`clk`	Input	Clock input. If input or output registers are enabled, they are updated on the active edge of this clock.

`a0[7:0]`
`a1[7:0]`
`a2[7:0]`
`a3[7:0]`
	Input	Operand A input, in the specified number_format.

`b0[7:0]`
`b1[7:0]`
`b2[7:0]`
`b3[7:0]`
	Input	Operand B input, in the specified number_format.

`rstn0`
`rstn1`
`rstn2`
`rstn3`
	Input	Register resets. When a given `reg_rstn` is asserted (active low), a value of 0 is written to the input register upon the next active edge of `clk`. Synchronous or asynchronous reset assertion is determined by the `outreg/inreg_sr_assertion` parameter.

`inrega3_ce`
`inregb3_ce`
`inrega2_ce`
`inregb2_ce`
`inrega1_ce`
`inregb1_ce`
`inrega0_ce`
`inregb0_ce`
	Input	Input register clock enable (active high). When the `inreg_enable` parameter is 1, de-asserting the `inreg_ce` signal causes the MLP72_INT8_MULT to keep the contents of the input register unchanged.

`outreg3_ce`
`outreg2_ce`
`outreg1_ce`
`outreg0_ce`
	Input	Output register clock enable (active high). When the `outreg_enable` parameter is 1, de-asserting the `outreg_ce` signal causes the MLP72_INT8_MULT to keep the contents of the output register unchanged.

`dout0[15:0]`
`dout1[15:0]`
`dout2[15:0]`
`dout3[15:0]`
	Output	The result of the multiply operation.

Timing Diagrams

The following timing diagram shows typical use of ACX_MLP72_INT8_MULT_4X, where both `inreg_enable` and `outreg_enable` are true, and all control inputs are active high.

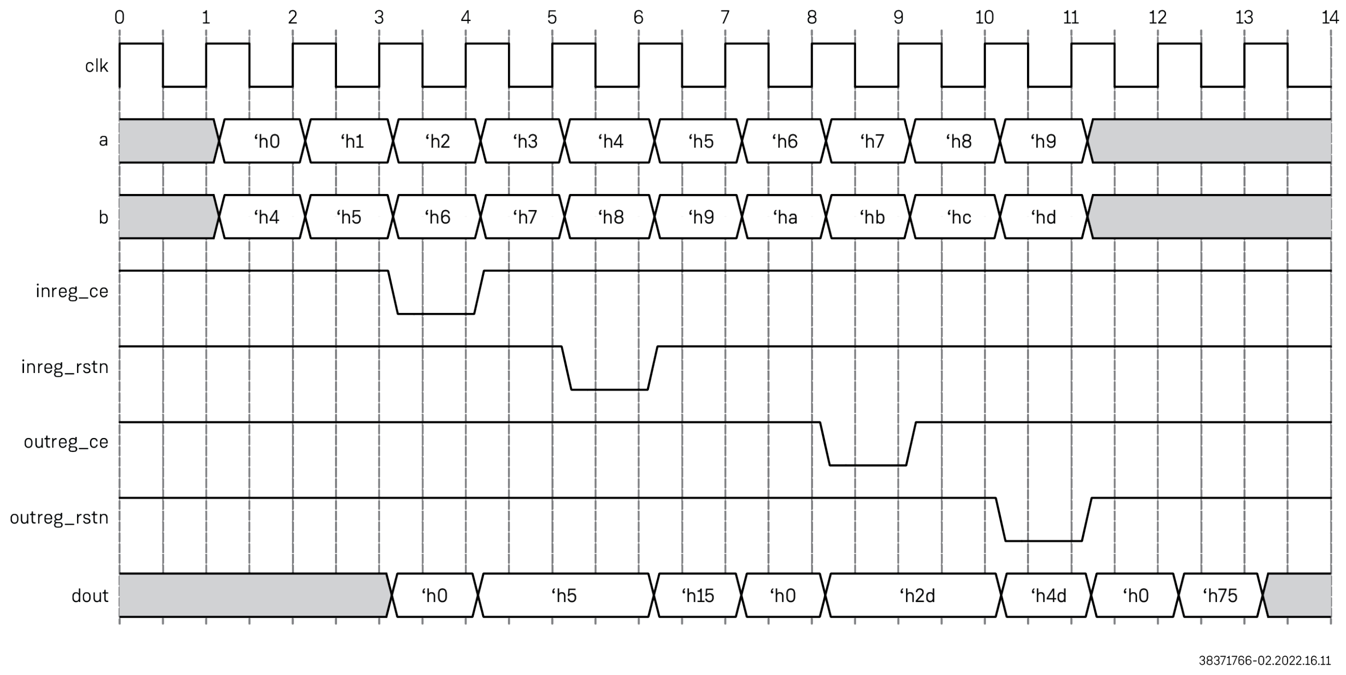


Figure 102: ***Timing Diagram for Single Multiplier Channel***

Inference

The ACX_MLP72_INT8_MULT_4X is inferrable using RTL constructs commonly used to infer multiplication operations, such as those shown in the following examples.

Note

- This component is appropriate for integer data widths of 8 bits and less.
- For widths larger than 8 bits, use ACX_MLP72_INT16_MULT_2X.
- As an inference target, it is only necessary to use a single pair of inputs and a single output. If there are other compatible instances in a design, they are merged during the build flow.

Examples

inreg_enable = 0, outreg_enable=0

```
x1 = a1 * b2;
```

inreg_enable = 0, outreg_enable=1

```
always @(posedge clk) begin
  x2 <= a2 * b2;
end
```

inreg_enable = 0, outreg_enable=1, with reset

```
always @(posedge clk) begin
  if (rstn == 1'b0)
    x3 <= 'h0;
  else if (en)
    x3 <= a3 * b3;
end
```

inreg_enable=1, outreg_enable=1, with input clock enable and output clock enable

```
always @(posedge clk) begin
  if (rstn == 1'b0) begin
    a4_d <= 'h0;
    b4_d <= 'h0;
  end else if (inreg_ce == 1'b1) begin
    a4_d <= a4;
    b4_d <= a4;
  end
end

always @(posedge clk) begin
  if (rstn == 1'b0)
    x4 <= 'h0;
  else if (outreg_ce == 1'b1)
    x4 <= a4_d * b4_d;
end
```

Instantiation Template

Verilog

```
    ACX_MLP72_INT8_MULT_4X
      #(
        .clk_polarity         (clk_polarity          ),
        .number_format        (number_format         ),
        .inrega3_enable       (inrega3_enable        ),
        .inregb3_enable       (inregb3_enable        ),
        .inrega2_enable       (inrega2_enable        ),
        .inregb2_enable       (inregb2_enable        ),
        .inrega1_enable       (inrega1_enable        ),
        .inregb1_enable       (inregb1_enable        ),
        .inrega0_enable       (inrega0_enable        ),
        .inregb0_enable       (inregb0_enable        ),
        .outreg3_enable       (outreg3_enable        ),
        .outreg2_enable       (outreg2_enable        ),
        .outreg1_enable       (outreg1_enable        ),
        .outreg0_enable       (outreg0_enable        ),
        .inrega3_sr_assertion (inrega3_sr_assertion  ),
        .inregb3_sr_assertion (inregb3_sr_assertion  ),
        .inrega2_sr_assertion (inrega2_sr_assertion  ),
        .inregb2_sr_assertion (inregb2_sr_assertion  ),
        .inrega1_sr_assertion (inrega1_sr_assertion  ),
        .inregb1_sr_assertion (inregb1_sr_assertion  ),
        .inrega0_sr_assertion (inrega0_sr_assertion  ),
        .inregb0_sr_assertion (inregb0_sr_assertion  ),
        .outreg3_sr_assertion (outreg3_sr_assertion  ),
        .outreg2_sr_assertion (outreg2_sr_assertion  ),
        .outreg1_sr_assertion (outreg1_sr_assertion  ),
        .outreg0_sr_assertion (outreg0_sr_assertion  )
    ) instance_name (
       .clk                       (clk        ),
       .a0                       (a0         ),
       .a1                       (a1         ),
       .a2                       (a2         ),
       .a3                       (a3         ),
       .b0                       (b0         ),
       .b1                       (b1         ),
       .b2                       (b2         ),
       .b3                       (b3         ),
       .rstn0                    (rstn0      ),
       .rstn1                    (rstn1      ),
       .rstn2                    (rstn2      ),
       .rstn3                    (rstn3      ),
       .inrega3_ce               (inrega3_ce ),
       .inregb3_ce               (inregb3_ce ),
       .inrega2_ce               (inrega2_ce ),
       .inregb2_ce               (inregb2_ce ),
       .inrega1_ce               (inrega1_ce ),
       .inregb1_ce               (inregb1_ce ),
       .inrega0_ce               (inrega0_ce ),
       .inregb0_ce               (inregb0_ce ),
       .outreg3_ce               (outreg3_ce ),
       .outreg2_ce               (outreg2_ce ),
       .outreg1_ce               (outreg1_ce ),
       .outreg0_ce              (outreg0_ce ),
       .dout0                   (dout0      ),
       .dout1                   (dout1      ),
       .dout2                   (dout2      ),
       .dout3                   (dout3      )
    );
```

MLP72_INT16_MULT_2X

The ACX_MLP72_INT16_MULT_2X primitive is a simple multiplier block with support for up to two parallel 16-bit multipliers using 16-bit two's-complement signed, signed magnitude, or unsigned integers. For higher performance operation, additional input and/or output registers can be enabled. Enabling each register causes an additional cycle of latency.

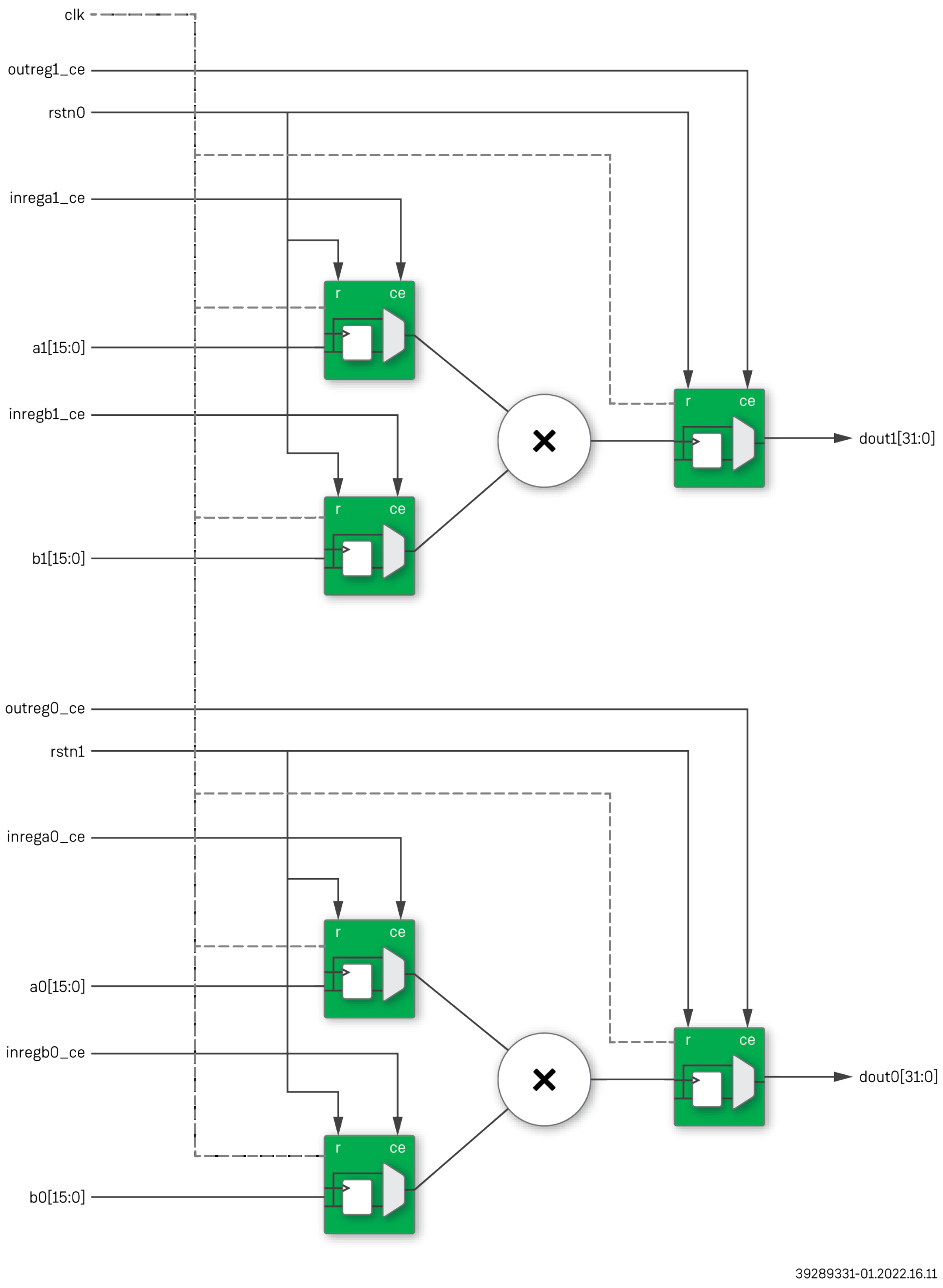


Figure 103: ***ACX_MLP72_INT16_MULT_2X Block Diagram***

Parameters

Table 191: ***ACX_MLP72_INT16_MULT_2X Parameters***

Parameter	Supported Values	Default Value	Description
`clk_polarity`	"rise", "fall"	"rise"	Controls which edge of the input clock to use: "rise" – rising edge of clock. "fall" – falling edge of clock.
`number_format`	0, 1, 2, 3	0	Controls the number format to use for all data inputs for both of the multipliers: 0 – unsigned. 1 – signed two's complement. 2 – signed "A" input with unsigned "B" input. 3 – unsigned "A" input with signed "B" input.

`inrega0_enable`
`inrega1_enable`
`inregb0_enable`
`inregb1_enable`
`outreg0_enable`
`outreg1_enable`
	0, 1	0	Controls whether or not the input and output registers are enabled:
			0 – disable the register.
			1 – enable the register. Results in extra latency.

`inrega0_sr_assertion`
`inregb0_sr_assertion`
`outreg0_sr_assertion`
`outreg1_sr_assertion`
	"clocked"," unclocked"	"clocked"	Controls whether the assertion of reset for the input and output registers is synchronous or asynchronous with respect to the `clk` input:
			"clocked" – synchronous reset. The register is reset upon the next rising edge of the clock when the associated `rstn` signal is asserted low.
			"unclocked" – asynchronous reset. The register is reset immediately when the associated `rstn` signal is asserted low.

`inrega1_sr_assertion` `inregb1_sr_assertion`	"clocked" (1)	"clocked"	The hardware only supports synchronous reset with respect to the `clk` input for the upper multiplier input registers. If a circuit uses asynchronous reset, then `inrega1_enable` and `inregb1_enable` should be set to 0, and the upper multiplier input register must be instantiated outside the MLP72_INT16_MULT_2X as a DFF. "clocked" – synchronous reset. The register is reset upon the next rising edge of the clock when the associated `rstn` signal is asserted low.

Table Notes

1. For optimal MLP performance on upper multipliers, use synchronous ("clocked") resets.

Ports

Table 192: ***ACX_MLP72_INT16_MULT_2X Pin Descriptions***

Name	Direction	Description
`clk`	Input	Clock input. If input or output registers are enabled, they are updated on the active edge of this clock.

`a0[15:0]`
`a1[15:0]`
	Input	Operand A input, in the specified number_format.

`b0[15:0]`
`b1[15:0]`
	Input	Operand B input, as specified by the number_format.

`inrega0_ce`
`inrega1_ce`
`inregb0_ce`
`inregb1_ce`
`outreg0_ce`
`outreg1_ce`
	Input	Input register clock enable (active high). When the `inreg_enable` parameter is 1, de-asserting the `inreg_ce` signal causes the MLP72_INT16_MULT2X to keep the contents of the input register unchanged.

`rstn0` `rstn1`	Input	Register resets. When a given `reg_rstn` is asserted (active low), a value of 0 is written to the input register upon the next active edge of `clk`. Synchronous or asynchronous reset assertion is determined by the `<outreg/inreg>_sr_assertion` parameter.

`dout1[31:0]`
`dout0[31:0]`
	Output	The result of the multiply operation.

Timing Diagrams

The following timing diagram shows typical use of the ACX_MLP72_INT16_MULT2X, where both `inreg_enable` and `outreg_enable` are true, and all control inputs are active high.

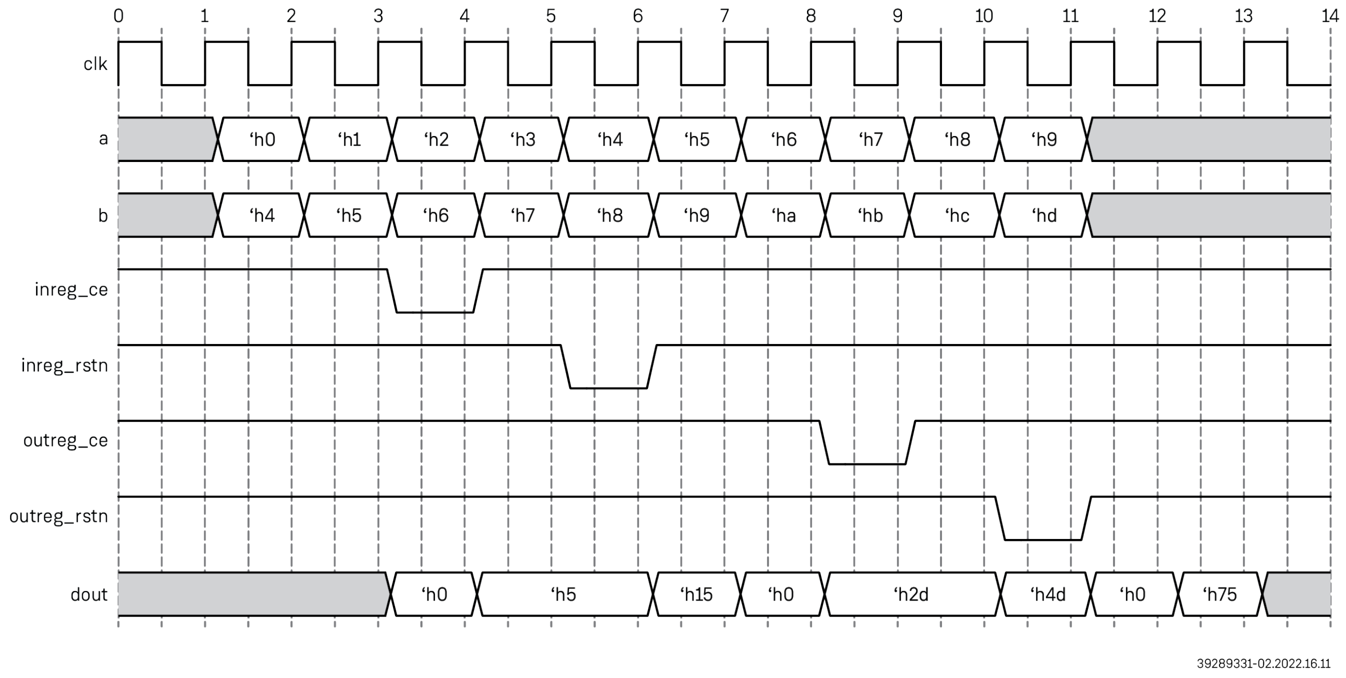


Figure 104: ***Timing Diagram for a Single Multiplier Channel***

Inference

The ACX_MLP72_INT16_MULT_2X is inferrable using RTL constructs commonly used to infer multiplication operations, such as those shown in the following examples.

Note

This component is appropriate for integer data widths between 9 and 16 bits, inclusive:

- For widths between 9 and 15 inclusive, sign extend the inputs and truncate the output as appropriate.
- For widths narrower than 9 bits, use ACX_MLP72_INT8_MULT_4X.

As an inference target, it is only necessary to use a single pair of inputs and a single output. If there are other compatible primitives in the design, they are merged during the build flow.

Examples

inreg_enable=0, outreg0_enable=0

```
x = a * b;
```

inreg_enable=0, outreg_enable=1

```
always @(posedge clk) begin
  x <= a * b;
end
```

inreg_enable=0, outreg0_enable=1, synchronous reset

```
always @(posedge clk) begin
  if (rstn == 1'b0)
    x <= 'h0;
  else if (en == 1'b1)
    x <= a * b;
end
```

inreg_enable=1, outreg_enable=1, asynchronous resets

```
always @(posedge clk, negedge rstn) begin
  if (rstn == 1'b0) begin
    a_d <= 'h0;
  end else if (inrega_ce == 1'b1) begin
    a_d <= a;
  end
end

always @(posedge clk, negedge rstn) begin
  if (rstn == 1'b0) begin
    b_d <= 'h0;
  end else if (inregb_ce == 1'b1) begin
    b_d <= b;
  end
end

always @(posedge clk, negedge rstn) begin
  if (rstn == 1'b0)
    x <= 'h0;
  else if (outreg_ce == 1'b1)
    x <= a_d * b_d;
end
```

Instantiation Template

Verilog

```
ACX_MLP72_INT16_MULT_2X
  #(
    .clk_polarity         (clk_polarity         ),
    .number_format        (number_format        ),
    .inrega0_enable       (inrega0_enable       ),
    .inregb0_enable       (inregb0_enable       ),
    .inrega1_enable       (inrega1_enable       ),
    .inregb1_enable       (inregb1_enable       ),
    .outreg0_enable       (outreg0_enable       ),
    .outreg1_enable       (outreg1_enable       ),
    .inrega0_sr_assertion (inrega0_sr_assertion ),
    .inregb0_sr_assertion (inregb0_sr_assertion ),
    .inrega1_sr_assertion (inrega1_sr_assertion ),
    .inregb1_sr_assertion (inregb1_sr_assertion ),
    .outreg0_sr_assertion (outreg0_sr_assertion ),
    .outreg1_sr_assertion (outreg1_sr_assertion )
) instance_name (
   .clk                   (clk                  ),
   .a0                    (a0                   ),
   .b0                    (b0                   ),
   .a1                    (a1                   ),
   .b1                    (b1                   ),
   .rstn0                 (rstn0                ),
   .rstn1                 (rstn1                ),
   .inrega0_ce            (inrega0_ce           ),
   .inregb0_ce            (inregb0_ce           ),
   .inrega1_ce            (inrega1_ce           ),
   .inregb1_ce            (inregb1_ce           ),
   .outreg0_ce            (outreg0_ce           ),
   .outreg1_ce            (outreg1_ce           ),
   .dout0                 (dout0                ),
   .dout1                 (dout1                )
);
```

Integer Library

The Achronix integer library provides macros that use the ACX_MLP72 to perform common integer operations. In addition, the library enables the use of the MLUT logic cell to efficiently implement integer multiplication with programmable logic. To use the library, include the following in the Verilog source code that instantiates any of the integer library macros:

```
`include "speedster7t/common/acx_integer.sv"
```

MLP Registers

The ACX_MLP72 has a number of internal registers that can be enabled to pipeline operations. Pipelining allows for higher clock frequencies, but operations take more clock cycles. Generally, for operation at the maximum fabric speed, all registers need to be enabled, but for lower frequencies some may be omitted.

For the integer library, modules support input registers, and one or more pipeline registers. The latter are simply identified by the number of desired pipeline stages. All registers are disabled by default.

Clock Enable and Reset

The input registers typically have separate clock enables for the 'a' and 'b' inputs and a shared reset. The pipeline registers have a shared clock enable and a shared reset, separate from the input registers. Many designs do not need clock enables and resets, in which case these inputs can simply be tied to `1'b1` (in particular, the accumulator is normally started with a load signal rather than a reset).

Accumulation

Most operations have an option to accumulate results. When accumulation is enabled, a new accumulation is started by asserting the `load` signal. When `load` is high, the previous value of the internal accumulation register is ignored, and the new value is stored. The output is then set to this value. When `load` is low, the old and new values are added, and the sum is stored. The output is this sum.

The `load` signal is internally pipelined to have the same latency as the input. If a set of inputs start a new accumulation, then `load` must be high when those inputs are presented. If accumulation is not enabled, then the `load` signal is ignored.

The accumulator uses an internal register, independent of the pipelining. In particular, accumulation may be used with `pipeline_regs = 0`, though this setting results in a lower frequency.

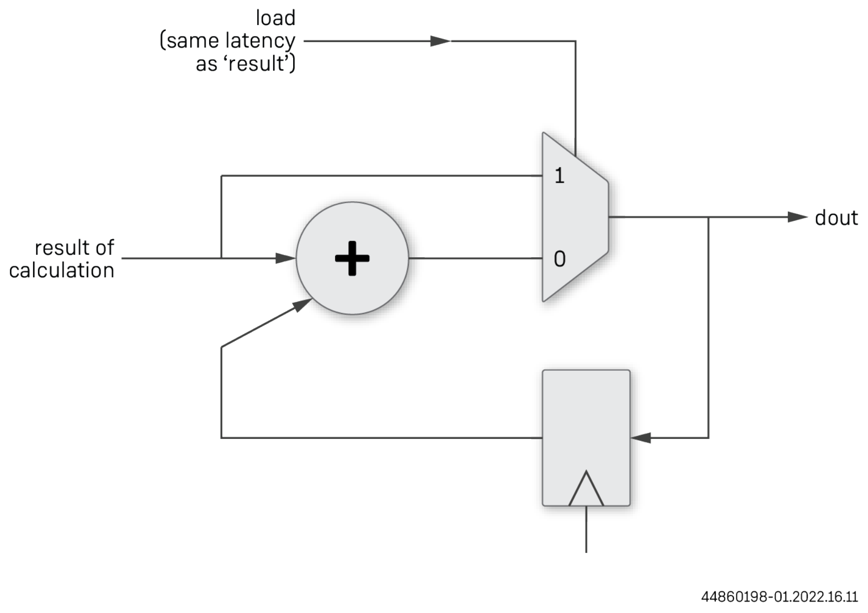


Figure 105: ***Accumulator With Load Signal***

ACX_INT_MULT

The ACX_INT_MULT module implements integer multiplication with fabric logic or with the ACX_MLP72, and delivers the following features:

- N × N multiplication, for N = 3–8, 16, 32
- Either input can be signed or unsigned
- Optional accumulator
- Optional registers to enable higher frequency operation

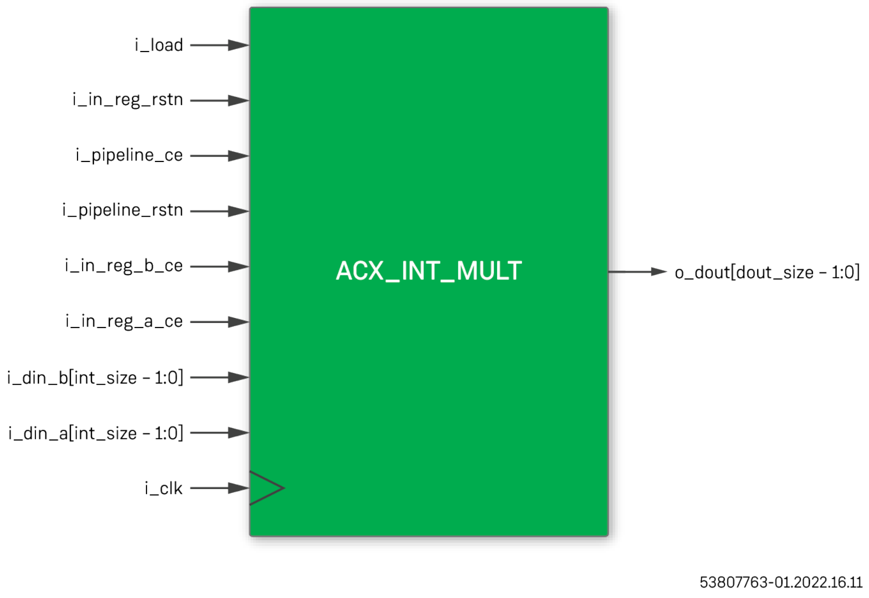


Figure 106: ***Integer Multiplier With Optional Accumulate***

Parameters

Table 193: *ACX_INT_MULT Parameters*

Parameter	Supported Values	Default	Description
`int_size`	3, 4, 5, 6, 7, 8, 16, 32	8	Number of bits of each integer input.
`int_unsigned_a`	0, 1	0	0 – `i_din_a` is signed (two's complement). 1 – `i_din_a` is unsigned.
`int_unsigned_b`	0, 1	0	0 – `i_din_b` is signed (two's complement). 1 – `i_din_b` is unsigned.
`accumulate`	0, 1	0	0 – no accumulation: `dout = i_din_a * i_din_b` 1 – accumulation: `dout` is the accumulated value. The start of accumulation is signaled by asserting `i_load=1`.
`in_reg_enable`	0, 1	0	0 – no input registers. 1 – `i_din_a` and `i_din_b` are registered. The input registers are controlled by the `i_in_reg_a_ce`, `i_in_reg_b_ce`, and `i_in_reg_rstn` inputs. Enabling the input register adds one cycle of latency.
`pipeline_regs`	0, 1, 2 (3)	0	The number of pipeline registers, not counting the input register. The total latency is `pipeline_regs + in_reg_enable`. A value of 3 is only allowed if `int_size=32` and `accumulate=1`. For all other cases, the valid values are 0, 1, and 2.
`dout_size`	Output (see page 244)		Width of the `o_dout` output. The default and range are determined by several other parameters, as explained in the Output (see page 244) section. Signed results are sign-extended as necessary. Values that do not fit are truncated at the high-order bits.
`architecture`	"auto", "rlb", "mlp"	auto	This string-valued parameter determines the implementation method. Refer to Architecture (see page 244) for more information. "rlb" – implementation is with reconfigurable logic, including MLUT, ALU8, and DFF's. "mlp" – implementation uses a single MLP72. "auto" – equivalent to "rlb" if `int_size <= 8`; equivalent to "mlp" if `int_size > 8`.

Ports

Table 194: *ACX_INT_MULT Pin Descriptions*

Name	Direction	Description
`i_clk`	Input	Clock input, used for the (optional) registers and accumulator.
`i_din_a[(int_size-1):0]`	Input	A data input to multiplier.
`i_din_b[(int_size-1):0]`	Input	B data input to multiplier.

`i_in_reg_a_ce`	Input	if `in_reg_enable=0` – ignored.
		if `in_reg_enable=1` – clock enable for `i_din_a`.

`i_in_reg_b_ce`	Input	if `in_reg_enable=0` – ignored.
		if `in_reg_enable=1` – clock enable for `i_din_b`.

`i_in_reg_rstn`	Input	if `in_reg_enable=0` – ignored. if `in_reg_enable=1` – synchronous active-low reset for input registers.
`i_pipeline_ce`	Input	if `pipeline_regs=0` – ignored. if `pipeline_regs>0` – clock enable for pipeline and accumulator registers.
`i_pipeline_rstn`	Input	if `pipeline_regs=0` – ignored. if `pipeline_regs>0` – synchronous active-low reset for pipeline and accumulator registers.
`i_load`	Input	if `accumulate=0` – ignored. if `accumulate=1` – resets the accumulator to `i_din_a*i_din_b`, ignoring the previous value. This signal is internally pipelined to have the same latency as `i_din_a` and `i_din_b`.
`o_dout[dout_size-1:0]`	Output	Result of multiplication and accumulation.

Usage and Inference

ACX_INT_MULT is intended for situations where direct control over the implementation of multiplication is required, in particular, when a fabric logic based implementation is desired or when manual control over the registers is needed. Alternatively, integer multiplication written as `a*b` in RTL is recognized and inferred, using an MLP-based implementation similar to the one provided by this module.

In addition to direct instantiation in Verilog or VHDL, an instance of ACX_INT_MULT can also be created in the ACE IP Configuration Perspective. See *Speedster7t Soft IP User Guide* (UG103) for details.

Architecture

For small integer sizes (`int_size` ≤ 8), by default, the multiplier is constructed using reconfigurable logic which uses the efficient Achronix MLUT feature to reduce LUT count compared to other FPGAs.

For `int_size` ≤ 8, the `architecture` parameter can be used to select an implementation with an ACX_MLP72 (this includes all registers and the accumulator). However, while this setting can result in a faster design, using an entire ACX_MLP72 for a single multiplication is not an efficient use of resources. Better efficiency can be achieved by using the ACX_INT_MULT_N module, which allows combining several multiplications in a single ACX_MLP72. Alternatively, one can write `a*b` and let Synplify and ACE handle the implementation, which also maps to an ACX_MLP72, and may pack several multiplications into a single ACX_MLP72 (packing decisions are based on the netlist connectivity).

For `int_size` = 16, the implementation always uses a single ACX_MLP72 (this includes all registers and the accumulator). As before, resource usage can be improved by using ACX_INT_MULT_N to combine two 16×16 multiplications in a single ACX_MLP72. Alternatively, writing `a*b` also uses an ACX_MLP72 implementation, and may pack two multiplications depending on netlist connectivity.

For `int_size` = 32, the implementation always uses a single ACX_MLP72. The ACX_MLP72 includes most of the registers, but not the accumulator. If accumulation is enabled, the accumulator and associated register are implemented with fabric logic.

Output

For multiplication, the default output size is two times the input size, but a smaller `dout_size` can be specified if the result is known to fit. When accumulation is enabled, typically a larger output size is required. For fabric logic based implementations (`int_size` ≤ 8), and for `int_size` = 32, the accumulator is built with fabric logic and with `dout_size` bits as specified by the user. For ACX_MLP72 based implementations with `int_size` ≤ 16, the accumulator is a maximum of 48 bits wide. The default and limits are summarized in the following table. The output format is unsigned if both inputs are unsigned, otherwise the output is signed (two's complement).

Table 195: *`dout_size`* Default and Limits

		accumulate=0		accumulate=1	
int_size	**architecture**	**Default**	**Max**	**Default**	**Max**
3–8	auto, rlb	2 × int_size	48	2 × int_size	48
3–8	mlp	2 × int_size	48	48	48
16	auto, mlp	32	48	48	48
32	auto, mlp	64	64	64	any

Instantiation Templates

Verilog

```
// Verilog template for ACX_INT_MULT
ACX_INT_MULT #(
    .int_size       (int_size       ),
    .int_unsigned_a (int_unsigned_a ),
    .int_unsigned_b (int_unsigned_b ),
    .accumulate     (accumulate     ),
    .in_reg_enable  (in_reg_enable  ),
    .pipeline_regs  (pipeline_regs  ),
    .dout_size      (dout_size      ),
    .architecture   (architecture   )
) instance_name (
    .i_clk           (user_i_clk                  ),
    .i_din_a         (user_i_din_a[int_size-1 : 0] ),
    .i_din_b         (user_i_din_b[int_size-1 : 0] ),
    .i_in_reg_a_ce   (user_i_in_reg_a_ce          ),
    .i_in_reg_b_ce   (user_i_in_reg_b_ce          ),
    .i_in_reg_rstn   (user_i_in_reg_rstn          ),
    .i_pipeline_ce   (user_i_pipeline_ce          ),
    .i_pipeline_rstn (user_i_pipeline_rstn        ),
    .i_load          (user_i_load                 ),
    .o_dout          (user_o_dout[dout_size-1 : 0] )
);
```

VHDL

```
-- VHDL Component template for ACX_INT_MULT
component ACX_INT_MULT is
generic (
    int_size             : integer := 8;
    int_unsigned_a       : integer := 0;
    int_unsigned_b       : integer := 0;
    accumulate           : integer := 0;
    in_reg_enable        : integer := 0;
    pipeline_regs        : integer := 0;
    dout_size            : integer := 48;
    architecture         : string  := "auto"
 );
port (
    i_clk                : in  std_logic;
    i_din_a              : in  std_logic_vector( int_size-1 downto 0 );
    i_din_b              : in  std_logic_vector( int_size-1 downto 0 );
    i_in_reg_a_ce        : in  std_logic;
    i_in_reg_b_ce        : in  std_logic;
    i_in_reg_rstn        : in  std_logic;
    i_pipeline_ce        : in  std_logic;
    i_pipeline_rstn      : in  std_logic;
    i_load               : in  std_logic;
    o_dout               : out std_logic_vector( dout_size-1 downto 0 )
);
end component ACX_INT_MULT

-- VHDL Instantiation template for ACX_INT_MULT
instance_name : ACX_INT_MULT
generic map (
    int_size             => int_size,
    int_unsigned_a       => int_unsigned_a,
    int_unsigned_b       => int_unsigned_b,
    accumulate           => accumulate,
    in_reg_enable        => in_reg_enable,
    pipeline_regs        => pipeline_regs,
    dout_size            => dout_size,
    architecture         => architecture
 )
port map (
    i_clk                => user_i_clk,
    i_din_a              => user_i_din_a,
    i_din_b              => user_i_din_b,
    i_in_reg_a_ce        => user_i_in_reg_a_ce,
    i_in_reg_b_ce        => user_i_in_reg_b_ce,
    i_in_reg_rstn        => user_i_in_reg_rstn,
    i_pipeline_ce        => user_i_pipeline_ce,
    i_pipeline_rstn      => user_i_pipeline_rstn,
    i_load               => user_i_load,
    o_dout               => user_o_dout
);
```

ACX_INT_MULT_N

The ACX_INT_MULT_N module computes N parallel multiplications with all numbers using the same format. There is no accumulation option. The macro has the following features:

- K × K multiplication, with K = 3–8, or 16
- Either input (a, b, or both) can be signed or unsigned
- N parallel multiplications (all the same format)
- Optional registers to enable higher clock frequency

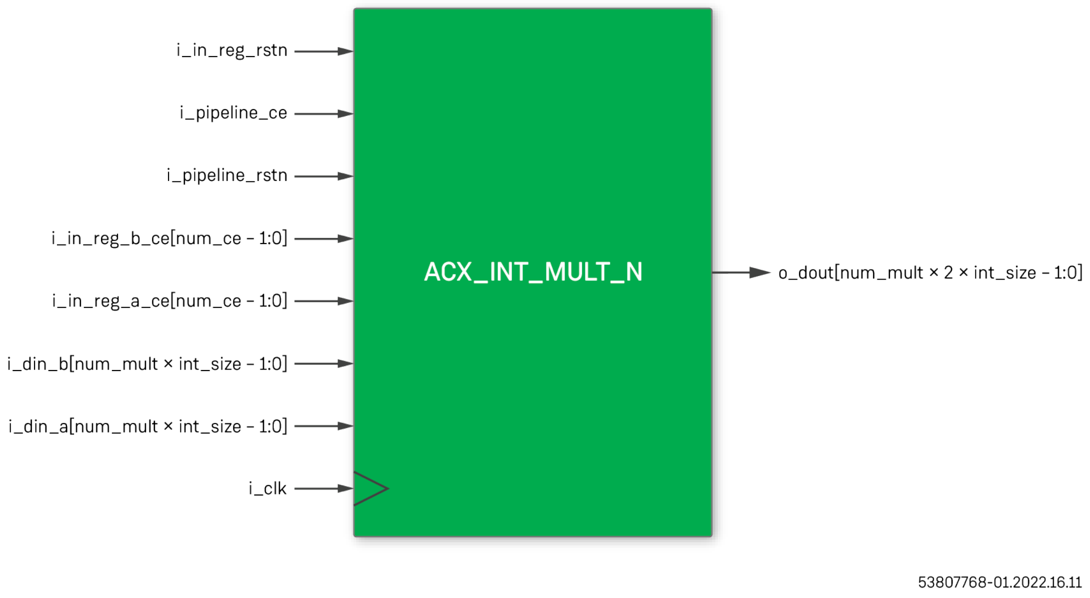

i_in_reg_rstn
i_pipeline_ce
i_pipeline_rstn
i_in_reg_b_ce[num_ce - 1:0]
i_in_reg_a_ce[num_ce - 1:0]
i_din_b[num_mult × int_size - 1:0]
i_din_a[num_mult × int_size - 1:0]
i_clk
ACX_INT_MULT_N
o_dout[num_mult × 2 × int_size - 1:0]
53807768-01.2022.16.11

Figure 107: ***N Integer Parallel Multiplications***

Parameters

Table 196: *ACX_INT_MULT_N Parameters*

Parameter	Supported Values	Default	Description
`int_size`	3, 4, 5, 6, 7, 8, 16	8	Number of bits of each integer input.
`num_mult`	1–8	1	Number of parallel multiplications. Refer to Maximum Parallel Multiplications (see page 250) for the limit per number format.
`int_unsigned_a`	0, 1	0	0 – `i_din_a(i)` is signed (two's complement). 1 – `i_din_a(i)` is unsigned.
`int_unsigned_b`	0, 1	0	0 – `i_din_b(i)` is signed (two's complement). 1 – `i_din_b(i)` is unsigned.
`in_reg_enable`	0, 1	0	0 – No input registers. 1 – `i_din_a` and `i_din_b` are registered. The input registers are controlled by the `i_in_reg_a_ce`, `i_in_reg_b_ce`, and `i_in_reg_rstn` inputs. Enabling the input register adds one cycle of latency.
`pipeline_regs`	0, 1	0	The number of pipeline registers, not counting the input register. The total latency is `pipeline_regs + in_reg_enable`.

An internal parameter, `num_ce`, is generated from the above parameters. This parameter determines the number of clock enables supported. The calculation of `num_ce` is shown in the following example.

```
localparam integer num_ce = (int_size <= 4)? (num_mult + 1)/2 : num_mult
```

Ports

Table 197: *ACX_INT_MULT_N Pin Descriptions*

Name	Direction	Description
`i_clk`	Input	Clock input, used for the (optional) registers.
`i_din_a[(num_mult*int_size-1):0]`	Input	Packed (see page 249) vector of A data input to multipliers.
`i_din_b[(num_mult*int_size-1):0]`	Input	Packed (see page 249) vector of B data input to multipliers.

`i_in_reg_a_ce[(num_ce-1):0]`	Input	if `in_reg_enable` = 0 – ignored.
		if `in_reg_enable` = 1 – clock enable for `i_din_a`. Refer to Clock Enables (see page 249).

`i_in_reg_b_ce[(num_ce-1):0]`	Input	if `in_reg_enable` = 0 – ignored.
		if `in_reg_enable` = 1 – clock enable for `i_din_b`. Refer to Clock Enables (see page 249).

`i_in_reg_rstn`	Input	if `in_reg_enable` = 0 – ignored. if `in_reg_enable` = 1 – synchronous active-low reset for input registers.
`i_pipeline_ce`	Input	if `pipeline_regs` = 0 – ignored. if `pipeline_regs` > 0 – clock enable for pipeline registers.
`i_pipeline_rstn`	Input	if `pipeline_regs` = 0 – ignored. if `pipeline_regs` > 0 – synchronous active-low reset for pipeline registers.
`o_dout[(num_mult*2*int_size-1):0]`	Output	Packed (see page 249) vector of multiplication results. The results are unsigned if both inputs are unsigned. Otherwise, the results are signed (two's complement).

Data Packing

Inputs and outputs are packed in single input and output vectors.

```
a(i)    = i_din_a[i*int_size +: int_size];
b(i)    = i_din_b[i*int_size +: int_size];
dout(i) = o_dout[i*2*int_size +: 2*int_size];
```

Clock Enables

If the input register is enabled, each input has its own clock enable if `int_size` >= 5. For `int_size` = 3 or 4, two adjacent inputs share the same clock enable. For example, `a(0)` and `a(1)` share `i_in_reg_a_ce[0]`, etc.

Maximum Parallel Multiplications

Parameter `num_mult` specifies the number of parallel multiplications. The maximum is determined by the input format (if either input is unsigned, the "Unsigned" column applies).

Table 198: ***Maximum Parallel Multiplications***

`int_size`	Max Signed Multiplications	Max Unsigned Multiplications
3	8	8
4	8	4
5,6,7,8	4	4
16	2	2

Usage and Inference

ACX_INT_MULT_N maps to a single ACX_MLP72. This macro is intended for situations where direct control over the implementation of multiplications is required, including the use of registers and the choice of which multiplications to combine in a single ACX_MLP72. Alternatively, integer multiplication written as `a*b` in RTL is recognized and inferred using an ACX_MLP72-based implementation, and combines multiplications based on netlist connectivity.

In addition to direct instantiation in Verilog or VHDL, an instance of ACX_INT_MULT_N can also be created in the ACE IP Configuration Perspective. See *Speedster7t Soft IP User Guide* (UG103) for details.

Instantiation Templates

Verilog

```
// Verilog template for ACX_INT_MULT_N
ACX_INT_MULT_N #(
    .int_size       (int_size       ),
    .num_mult       (num_mult       ),
    .int_unsigned_a (int_unsigned_a ),
    .int_unsigned_b (int_unsigned_b ),
    .in_reg_enable  (in_reg_enable  ),
    .pipeline_regs  (pipeline_regs  )
) instance_name (
    .i_clk           (user_i_clk                              ),
    .i_din_a         (user_i_din_a[num_mult*int_size-1 : 0]   ),
    .i_din_b         (user_i_din_b[num_mult*int_size-1 : 0]   ),
    .i_in_reg_a_ce   (user_i_in_reg_a_ce[num_ce-1 : 0]        ),
    .i_in_reg_b_ce   (user_i_in_reg_b_ce[num_ce-1 : 0]        ),
    .i_in_reg_rstn   (user_i_in_reg_rstn                      ),
    .i_pipeline_ce   (user_i_pipeline_ce                      ),
    .i_pipeline_rstn (user_i_pipeline_rstn                    ),
    .o_dout          (user_o_dout[num_mult*2*int_size-1 : 0] )
);
```

VHDL

```
-- VHDL Component template for ACX_INT_MULT_N
component ACX_INT_MULT_N is
generic (
    int_size             : integer := 8;
    num_mult             : integer := 1;
    int_unsigned_a       : integer := 0;
    int_unsigned_b       : integer := 0;
    in_reg_enable        : integer := 0;
    pipeline_regs        : integer := 0
);
port (
    i_clk                : in  std_logic;
    i_din_a              : in  std_logic_vector( num_mult*int_size-1 downto 0 );
    i_din_b              : in  std_logic_vector( num_mult*int_size-1 downto 0 );
    i_in_reg_a_ce        : in  std_logic_vector( num_ce-1 downto 0 );
    i_in_reg_b_ce        : in  std_logic_vector( num_ce-1 downto 0 );
    i_in_reg_rstn        : in  std_logic;
    i_pipeline_ce        : in  std_logic;
    i_pipeline_rstn      : in  std_logic;
    o_dout               : out std_logic_vector( num_mult*2*int_size-1 downto 0 )
);
end component ACX_INT_MULT_N

-- VHDL Instantiation template for ACX_INT_MULT_N
instance_name : ACX_INT_MULT_N
generic map (
    int_size             => int_size,
    num_mult             => num_mult,
    int_unsigned_a       => int_unsigned_a,
    int_unsigned_b       => int_unsigned_b,
    in_reg_enable        => in_reg_enable,
    pipeline_regs        => pipeline_regs,
    out_reg_enable       => out_reg_enable
)
port map (
    i_clk                => user_i_clk,
    i_din_a              => user_i_din_a,
    i_din_b              => user_i_din_b,
    i_in_reg_a_ce        => user_i_in_reg_a_ce,
    i_in_reg_b_ce        => user_i_in_reg_b_ce,
    i_in_reg_rstn        => user_i_in_reg_rstn,
    i_pipeline_ce        => user_i_pipeline_ce,
    i_pipeline_rstn      => user_i_pipeline_rstn,
    o_dout               => user_o_dout
);
```

ACX_INT_MULT_ADD

The ACX_INT_MULT_ADD module computes a parallel sum of products, SUM a(i) × b(i), with optional accumulation. This macro features:

- K × K multiplication, for K = 3 – 8, or 16
- Inputs can be signed or unsigned
- Sum of N parallel multiplications
- Optional accumulator
- Optional registers to enable higher frequency

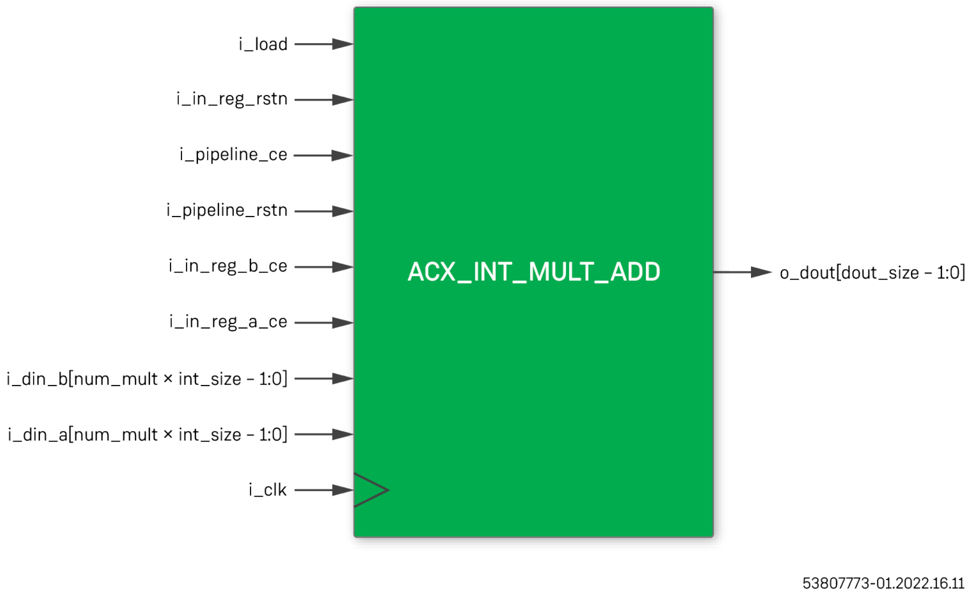


Figure 108: ***N Integer Sum of Products With Optional Accumulation***

Parameters

Table 199: *ACX_INT_MULT_ADD Parameters*

Parameter	Supported Values	Default	Description
`int_size`	3, 4, 5, 6, 7, 8, 16	8	Number of bits of each integer input.
`num_mult`	1–24	1	Number of parallel multiplications. Refer to Maximum Parallel Multiplications (see page 256) for the limits per number format.
`int_unsigned_a`	0, 1	0	0 – `i_din_a` is signed (two's complement). 1 – `i_din_a` is unsigned.
`int_unsigned_b`	0, 1	0	0 – `i_din_b` is signed (two's complement). 1 – `i_din_b` is unsigned.
`accumulate`	0, 1	0	0 – No accumulation: `dout = SUM(i_din_a(i)*i_din_b(i))`. 1 – Accumulation: `dout` is the accumulated value. The start of accumulation is signaled by asserting `i_load=1`.
`in_reg_enable`	0, 1	0	0 – No input registers. 1 – `i_din_a` and `i_din_b` are registered. The input registers are controlled by the `i_in_reg_a_ce`, `i_in_reg_b_ce`, and `i_in_reg_rstn` inputs. Enabling the input register adds one cycle of latency.
`pipeline_regs`	0, 1, 2	0	The number of pipeline registers, not counting the input register. The total latency is `pipeline_regs + in_reg_enable`.
`dout_size`	≤ 48	48	Width of the `o_dout` output. Values that do not fit are truncated at the high-order bits.

Ports

Table 200: *ACX_INT_MULT_ADD Pin Descriptions*

Name	Direction	Description
`i_clk`	Input	Clock input, used for the (optional) registers and accumulator.
`i_din_a[num_mult*int_size-1 : 0]`	Input	Packed (see page 255) vector of A data input to multipliers.
`i_din_b[num_mult*int_size-1 : 0]`	Input	Packed (see page 255) vector of B data input to multipliers

`i_in_reg_a_ce`	Input	if `in_reg_enable=0` – ignored.
		if `in_reg_enable=1` – clock enable for `i_din_a`.

`i_in_reg_b_ce`	Input	if `in_reg_enable=0` – ignored.
		if `in_reg_enable=1` – clock enable for `i_din_b`.

`i_in_reg_rstn`	Input	if `in_reg_enable=0` – ignored. if `in_reg_enable=1` – synchronous active-low reset for input registers.
`i_pipeline_ce`	Input	if `pipeline_regs=0` – ignored. if `pipeline_regs>0` – clock enable for pipeline and accumulator registers.
`i_pipeline_rstn`	Input	if `pipeline_regs=0` – ignored. if `pipeline_regs>0` – synchronous active-low reset for pipeline and accumulator registers.
`i_load`	Input	if `accumulate=0` – ignored. if `accumulate=1` – resets the accumulator to `SUM(i_din_a*i_din_b)`, ignoring the previous value. This signal is internally pipelined to have the same latency as `i_din_a` and `i_din_b`.
`o_dout[(dout_size-1):0]`	Output	Sum of products, or result of accumulation.

Input Packing

Inputs are packed in single input vectors:

```
a(i) = i_din_a[i*int_size +: int_size];
b(i) = i_din_b[i*int_size +: int_size];
```

Maximum Parallel Multiplications

Parameter `num_mult` specifies the number of parallel multiplications. The ACX_MLP72 used by the module has two input modes, normal and wide. Wide mode enables more parallel multiplications per ACX_MLP72. However, in this mode, the adjacent ACX_BRAM72K site is used as route-through, meaning it is no longer available for BRAM placement. The selection between normal and wide mode is automatically made based on the number of requested multiplications and the size of the inputs.

The following table lists the maximum number of parallel multiplications for each of the two modes. If either input is unsigned, the **Unsigned** columns apply. Wide mode is only selected if `num_mult` is larger than the maximum for normal mode. For example, for `int_size` = 8, if `num_mult` <= 4, ACX_INT_MULT_ADD requires one ACX_MLP72, but if `num_mult` > 4, ACX_INT_MULT_ADD requires one ACX_MLP72 and one ACX_BRAM72K.

Table 201: ***Maximum Number of Parallel Multiplications***

	Normal Mode		Wide Mode	
int_size	**Max Signed Multiplications**	**Max Unsigned Multiplications**	**Max Signed Multiplications**	**Max Unsigned Multiplications**
3	12	8	24	16
4	8	6	16	12
5	6	6	12	12
6	6	5	12	10
7	5	4	10	8
8	4	4	8	8
16	2	2	4	4

Usage and Inference

The ACX_INT_MULT_ADD module gives direct control over the multiply-add functionality of the ACX_MLP72. In particular, it enables the use of wide mode to increase the number of parallel multiplications. Alternatively, a sum of products written in RTL, such as `x=a0*b0 + a1*b1` is recognized and inferred. However, an inferred multiply-add does not use wide mode and is currently limited to int8 and int16.

In addition to direct instantiation in Verilog or VHDL, an instance of ACX_INT_MULT_ADD can also be created in the ACE IP Configuration Perspective. See *Speedster7t Soft IP User Guide* (UG103) for details.

Instantiation Templates

Verilog

```
// Verilog template for ACX_INT_MULT_ADD
ACX_INT_MULT_ADD #(
    .int_size       (int_size       ),
    .num_mult       (num_mult       ),
    .int_unsigned_a (int_unsigned_a ),
    .int_unsigned_b (int_unsigned_b ),
    .accumulate     (accumulate     ),
    .in_reg_enable  (in_reg_enable  ),
    .pipeline_regs  (pipeline_regs  ),
    .dout_size      (dout_size      )
) instance_name (
    .i_clk           (user_i_clk                          ),
    .i_din_a         (user_i_din_a[num_mult*int_size-1 : 0] ),
    .i_din_b         (user_i_din_b[num_mult*int_size-1 : 0] ),
    .i_in_reg_a_ce   (user_i_in_reg_a_ce                  ),
    .i_in_reg_b_ce   (user_i_in_reg_b_ce                  ),
    .i_in_reg_rstn   (user_i_in_reg_rstn                  ),
    .i_pipeline_ce   (user_i_pipeline_ce                  ),
    .i_pipeline_rstn (user_i_pipeline_rstn                ),
    .i_load          (user_i_load                         ),
    .o_dout          (user_o_dout[dout_size-1 : 0]        )
);
```

VHDL

```
-- VHDL Component template for ACX_INT_MULT_ADD
component ACX_INT_MULT_ADD is
generic (
    int_size            : integer := 8;
    num_mult            : integer := 1;
    int_unsigned_a      : integer := 0;
    int_unsigned_b      : integer := 0;
    accumulate          : integer := 0;
    in_reg_enable       : integer := 0;
    pipeline_regs       : integer := 0;
    dout_size           : integer := 48
);
port (
    i_clk               : in  std_logic;
    i_din_a             : in  std_logic_vector( num_mult*int_size-1 downto 0 );
    i_din_b             : in  std_logic_vector( num_mult*int_size-1 downto 0 );
    i_in_reg_a_ce       : in  std_logic;
    i_in_reg_b_ce       : in  std_logic;
    i_in_reg_rstn       : in  std_logic;
    i_pipeline_ce       : in  std_logic;
    i_pipeline_rstn     : in  std_logic;
    i_load              : in  std_logic;
    o_dout              : out std_logic_vector( dout_size-1 downto 0 )
);
end component ACX_INT_MULT_ADD

-- VHDL Instantiation template for ACX_INT_MULT_ADD
instance_name : ACX_INT_MULT_ADD
generic map (
    int_size            => int_size,
    num_mult            => num_mult,
    int_unsigned_a      => int_unsigned_a,
    int_unsigned_b      => int_unsigned_b,
    accumulate          => accumulate,
    in_reg_enable       => in_reg_enable,
    pipeline_regs       => pipeline_regs,
    dout_size           => dout_size
)
port map (
    i_clk               => user_i_clk,
    i_din_a             => user_i_din_a,
    i_din_b             => user_i_din_b,
    i_in_reg_a_ce       => user_i_in_reg_a_ce,
    i_in_reg_b_ce       => user_i_in_reg_b_ce,
    i_in_reg_rstn       => user_i_in_reg_rstn,
    i_pipeline_ce       => user_i_pipeline_ce,
    i_pipeline_rstn     => user_i_pipeline_rstn,
    i_load              => user_i_load,
    o_dout              => user_o_dout
);
```

Floating-Point Library

Introduction

The Achronix floating-point library provides macros that instantiate the ACX_MLP72 to perform common floating-point operations. To use the library, include the following in the Verilog source code:

```
`include "speedster7t/common/acx_floating_point.sv"
```

MLP Registers

The MLP has a number of internal registers that can be enabled to pipeline operations. Pipelining allows for higher clock frequencies, but operations take more clock cycles. Generally, for operation at the maximum fabric speed, all registers need to be enabled, but for lower frequencies some may be omitted.

For the floating-point library, modules support input registers and one or more pipeline registers. The latter are simply identified by the number of desired pipeline stages. All registers are by default disabled (bypassed).

Clock Enable and Reset

The input registers typically have separate clock enables for the 'a' and 'b' inputs, and a shared reset. The pipeline registers have a shared clock enable and a shared reset, separate from the input registers. Many designs do not need clock enables and resets, in which case these inputs can simply be tied to `1'b1` (in particular, the accumulator is normally started with a load signal rather than a reset).

Accumulation

Most operations have an option to accumulate results. When accumulation is enabled, a new accumulation is started by asserting the `load` signal. When `load` is high, the previous value of the internal accumulation register is ignored, and the new value is stored. The output is then set to this value. When `load` is low, the old and new values are added, and the sum is stored. The output is this sum.

The `load` signal is internally pipelined to have the same latency as the input. If a set of inputs start a new accumulation, then `load` must be high when those inputs are presented. If accumulation is not enabled, then the `load` signal is ignored.

The accumulator uses an internal register, independent of the pipelining. In particular, accumulation may be used with `pipeline_regs = 0`, though this setting results in a lower frequency.

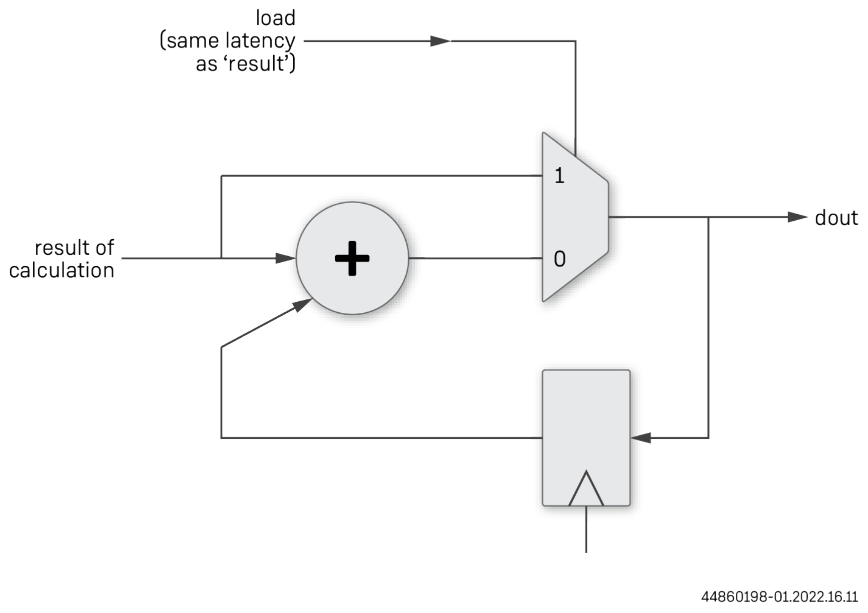


Figure 109: ***Accumulator With Load Signal***

Floating-Point Format

The input and output format of each operation is specified with two parameters, `fp_size` and `fp_exp_size`. Refer to Number Formats for an explanation of these two parameters.

Note

The selected format applies to both inputs and outputs. Internally, the actual multiplications and additions are always performed with fp24.

Output Status

Operations have a two bit status output. The interpretation is as follows.

Table 202: *Output Status Bits*

Status	Description
`2'b00`	Normal.
`2'b01`	Result is ± 0.0.
`2'b11`	Last operation had underflow, and thus, the result is ± 0.0.
`2'b10`	Result is ± infinity.

That a result is 0.0 or infinity can also be determined by inspecting the exponent field of the result. The status flags are an additional method to check the result.

When a result is 0.0, it can be because the result is mathematically 0 (e.g., x − x = 0) or because an underflow occurred. For instance, if dout = a × b + c, the underflow status refers to the addition. Underflow of the multiplication would merely result in dout = 0 + c, which itself has no underflow.

Note

Underflow refers to the last operation that produced the current output.

ACX_FP_ADD

The ACX_FP_ADD module computes A+B, with optional accumulation. Internal register stages can be enabled to allow for higher operating frequencies.

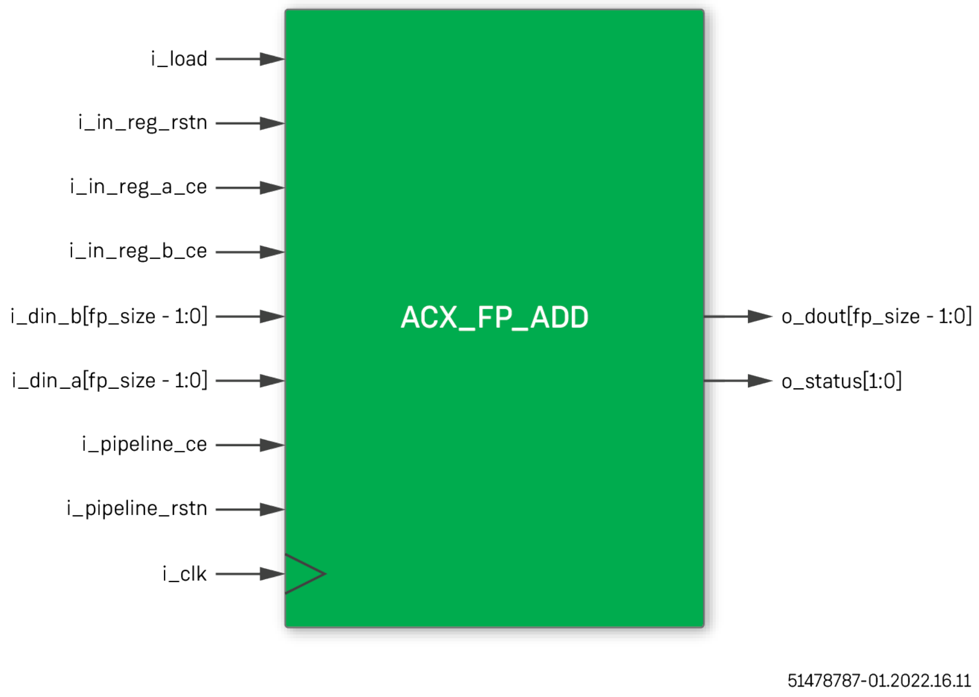


Figure 110: ***Floating-Point Adder With Optional Accumulate***

Parameters

Table 203: *ACX_FP_ADD Parameters*

Parameter	Supported Values	Default	Description
`fp_size`	16, 24	16	Width of floating point number. Supports fp24, fp16, and fp16e8.
`fp_exp_size`	5, 8	5	Size of floating-point exponent.
`subtract`	0, 1	0	0 – compute `i_din_a + i_din_b`. 1 – compute `i_din_a - i_din_b`.
`accumulate`	0, 1	0	0 – no accumulation: `dout = i_din_a ± i_din_b` (determined by the `subtract` parameter). 1 – accumulation: `dout` is the accumulated value. The start of accumulation is signaled by asserting `i_load=1`.
`in_reg_enable`	0, 1	0	0 – no input registers. 1 – `i_din_a` and `i_din_b` are registered. The input registers are controlled by the `i_in_reg_a_ce`, `i_in_reg_b_ce`, and `i_in_reg_rstn` inputs. Enabling the input registers adds one cycle of latency.
`pipeline_regs`	0–5	0	The number of pipeline registers, not counting the input register. The total latency is `pipeline_regs + in_reg_enable`.

Ports

Table 204: *ACX_FP_ADD Pin Descriptions*

Name	Direction	Description
`i_clk`	Input	Clock input. Used by the (optional) registers and accumulator.
`i_din_a[(fp_size-1):0]`	Input	'A' data input to adder.
`i_din_b[(fp_size-1):0]`	Input	'B' data input to adder.
`i_in_reg_a_ce`	Input	If `in_reg_enable=0` – ignored. If `in_reg_enable=1` – clock enable for `i_din_a`.
`i_in_reg_b_ce`	Input	If `in_reg_enable=0` – ignored. If `in_reg_enable=1` – clock enable for `i_din_b`.
`i_in_reg_rstn`	Input	If `in_reg_enable=0` – ignored. If `in_reg_enable=1` – synchronous active-low reset for input registers.
`i_pipeline_ce`	Input	If `pipeline_regs=0` – ignored. If `pipeline_regs>1` – clock enable for pipeline and accumulator registers.
`i_pipeline_rstn`	Input	If `pipeline_regs=0` – ignored. If `pipeline_regs>1` – synchronous active-low reset for pipeline and accumulator registers.
`i_load`	Input	If `accumulate=0` – ignored. If `accumulate=1` – resets the accumulator to `i_din_a ± i_din_b`, ignoring the previous value. This signal is internally pipelined to have the same latency as `i_din_a ± i_din_b`.
`o_dout[(fp_size-1):0]`	Output	Result of addition and accumulation.
`o_status[1:0]`(1)	Output	Error status of `o_dout`.

Table Notes

1. See Output Status for details.

Usage and Inference

ACX_FP_ADD cannot be inferred and must be directly instantiated. The specified floating point format applies to the inputs and output but, internally, the operations are performed with fp24.

Instantiation Templates

Verilog

```
// Verilog template for ACX_FP_ADD
ACX_FP_ADD #(
    .fp_size       (fp_size       ),
    .fp_exp_size   (fp_exp_size   ),
    .subtract      (subtract      ),
    .accumulate    (accumulate    ),
    .in_reg_enable (in_reg_enable ),
    .pipeline_regs (pipeline_regs )
) instance_name (
    .i_clk           (user_i_clk           ),
    .i_din_a         (user_i_din_a         ),
    .i_din_b         (user_i_din_b         ),
    .i_in_reg_a_ce   (user_i_in_reg_a_ce   ),
    .i_in_reg_b_ce   (user_i_in_reg_b_ce   ),
    .i_in_reg_rstn   (user_i_in_reg_rstn   ),
    .i_pipeline_ce   (user_i_pipeline_ce   ),
    .i_pipeline_rstn (user_i_pipeline_rstn ),
    .i_load          (user_i_load          ),
    .o_dout          (user_o_dout          ),
    .o_status        (user_o_status        )
);
```

VHDL

```
-- VHDL Component template for ACX_FP_ADD
component ACX_FP_ADD is
generic (
    fp_size             : integer := 16;
    fp_exp_size         : integer := 5;
    subtract            : integer := 0;
    accumulate          : integer := 0;
    in_reg_enable       : integer := 0;
    pipeline_regs       : integer := 0
);
port (
    i_clk               : in  std_logic;
    i_din_a             : in  std_logic_vector( fp_size-1 downto 0 );
    i_din_b             : in  std_logic_vector( fp_size-1 downto 0 );
    i_in_reg_a_ce       : in  std_logic;
    i_in_reg_b_ce       : in  std_logic;
    i_in_reg_rstn       : in  std_logic;
    i_pipeline_ce       : in  std_logic;
    i_pipeline_rstn     : in  std_logic;
    i_load              : in  std_logic;
    o_dout              : out std_logic_vector( fp_size-1 downto 0 );
    o_status            : out std_logic_vector( 1 downto 0 )
);
end component ACX_FP_ADD

-- VHDL Instantiation template for ACX_FP_ADD
instance_name : ACX_FP_ADD
generic map (
    fp_size             => fp_size,
    fp_exp_size         => fp_exp_size,
    subtract            => subtract,
    accumulate          => accumulate,
    in_reg_enable       => in_reg_enable,
    pipeline_regs       => pipeline_regs
)
port map (
    i_clk               => user_i_clk,
    i_din_a             => user_i_din_a,
    i_din_b             => user_i_din_b,
    i_in_reg_a_ce       => user_i_in_reg_a_ce,
    i_in_reg_b_ce       => user_i_in_reg_b_ce,
    i_in_reg_rstn       => user_i_in_reg_rstn,
    i_pipeline_ce       => user_i_pipeline_ce,
    i_pipeline_rstn     => user_i_pipeline_rstn,
    i_load              => user_i_load,
    o_dout              => user_o_dout,
    o_status            => user_o_status
);
```

ACX_FP_MULT

The ACX_FP_MULT module computes A × B, with optional accumulation. Internal register stages can be enabled to allow for higher operating frequencies.

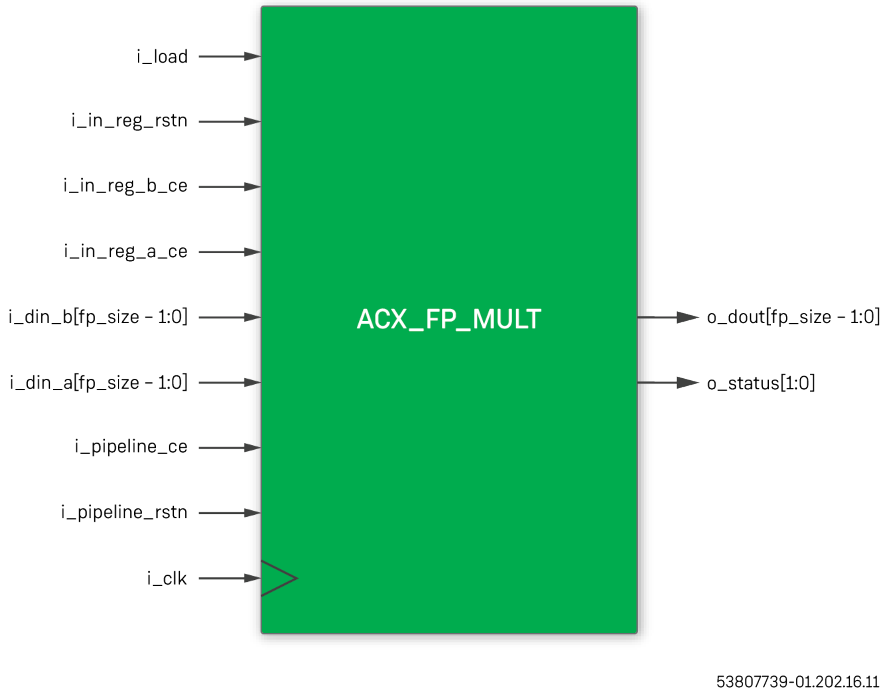


Figure 111: ***Floating-Point Multiplier With Optional Accumulate***

Parameters

Table 205: *ACX_FP_MULT Parameters*

Parameter	Supported Values	Default	Description
`fp_size`	16, 24	16	Width of floating-point number. Supports fp24, fp16, and fp16e8.
`fp_exp_size`	5, 8	5	Size of floating-point exponent.
`accumulate`	0, 1	0	0 – no accumulation: `dout = i_din_a × i_din_b`. 1 – accumulation: `dout` is the accumulated value. The start of accumulation is signaled by asserting `i_load=1`.
`in_reg_enable`	0, 1	0	0 – no input registers. 1 – `i_din_a` and `i_din_b` are registered. The input registers are controlled by the `i_in_reg_a_ce`, `i_in_reg_b_ce`, and `i_in_reg_rstn` inputs. Enabling the input registers adds one cycle of latency.
`pipeline_regs`	0–4	0	The number of pipeline registers, not counting the input register. The total latency is `pipeline_regs + in_reg_enable`.

Ports

Table 206: *ACX_FP_MULT Pin Descriptions*

Name	Direction	Description
`i_clk`	Input	Clock input, used for the (optional) registers and accumulator.
`i_din_a[(fp_size-1):0]`	Input	'A' data input to multiplier.
`i_din_b[(fp_size-1):0]`	Input	'B' data input to multiplier.
`i_in_reg_a_ce`	Input	If `in_reg_enable=0` – ignored. If `in_reg_enable=1` – clock enable for `i_din_a`.
`i_in_reg_b_ce`	Input	If `in_reg_enable=0` – ignored. If `in_reg_enable=1` – clock enable for `i_din_b`.
`i_in_reg_rstn`	Input	If `in_reg_enable=0` – ignored. If `in_reg_enable=1` – synchronous active-low reset for input registers.
`i_pipeline_ce`	Input	If `pipeline_regs=0` – ignored. If `pipeline_regs>1` – clock enable for pipeline and accumulator registers.
`i_pipeline_rstn`	Input	If `pipeline_regs=0` – ignored. If `pipeline_regs>1` – synchronous active-low reset for pipeline and accumulator registers.
`i_load`	Input	If `accumulate=0` – ignored. If `accumulate=1` – resets the accumulator to `i_din_a × i_din_b`, ignoring the previous value. This signal is internally pipelined to have the same latency as `i_din_a × i_din_b`.
`o_dout[(fp_size-1):0]`	Output	Result of multiplication and accumulation.
`o_status[1:0]`(1)	Output	Error status of `o_dout`.

Table Notes

1. See Output Status for details.

Usage and Inference

ACX_FP_MULT cannot be inferred and must be directly instantiated. The specified floating point format applies to the inputs and output but, internally, the operations are performed with fp24.

Instantiation Templates

Verilog

```
// Verilog template for ACX_FP_MULT
ACX_FP_MULT #(
    .fp_size       (fp_size       ),
    .fp_exp_size   (fp_exp_size   ),
    .accumulate    (accumulate    ),
    .in_reg_enable (in_reg_enable ),
    .pipeline_regs (pipeline_regs )
) instance_name (
    .i_clk           (user_i_clk           ),
    .i_din_a         (user_i_din_a         ),
    .i_din_b         (user_i_din_b         ),
    .i_in_reg_a_ce   (user_i_in_reg_a_ce   ),
    .i_in_reg_b_ce   (user_i_in_reg_b_ce   ),
    .i_in_reg_rstn   (user_i_in_reg_rstn   ),
    .i_pipeline_ce   (user_i_pipeline_ce   ),
    .i_pipeline_rstn (user_i_pipeline_rstn ),
    .i_load          (user_i_load          ),
    .o_dout          (user_o_dout          ),
    .o_status        (user_o_status        )
);
```

VHDL

```
-- VHDL Component template for ACX_FP_MULT
component ACX_FP_MULT is
generic (
    fp_size              : integer := 16;
    fp_exp_size          : integer := 5;
    accumulate           : integer := 0;
    in_reg_enable        : integer := 0;
    pipeline_regs        : integer := 0
);
port (
    i_clk                : in  std_logic;
    i_din_a              : in  std_logic_vector( fp_size-1 downto 0 );
    i_din_b              : in  std_logic_vector( fp_size-1 downto 0 );
    i_in_reg_a_ce        : in  std_logic;
    i_in_reg_b_ce        : in  std_logic;
    i_in_reg_rstn        : in  std_logic;
    i_pipeline_ce        : in  std_logic;
    i_pipeline_rstn      : in  std_logic;
    i_load               : in  std_logic;
    o_dout               : out std_logic_vector( fp_size-1 downto 0 );
    o_status             : out std_logic_vector( 1 downto 0 )
);
end component ACX_FP_MULT

-- VHDL Instantiation template for ACX_FP_MULT
instance_name : ACX_FP_MULT
generic map (
    fp_size              => fp_size,
    fp_exp_size          => fp_exp_size,
    accumulate           => accumulate,
    in_reg_enable        => in_reg_enable,
    pipeline_regs        => pipeline_regs
)
port map (
    i_clk                => user_i_clk,
    i_din_a              => user_i_din_a,
    i_din_b              => user_i_din_b,
    i_in_reg_a_ce        => user_i_in_reg_a_ce,
    i_in_reg_b_ce        => user_i_in_reg_b_ce,
    i_in_reg_rstn        => user_i_in_reg_rstn,
    i_pipeline_ce        => user_i_pipeline_ce,
    i_pipeline_rstn      => user_i_pipeline_rstn,
    i_load               => user_i_load,
    o_dout               => user_o_dout,
    o_status             => user_o_status
);
```

ACX_FP_MULT_PLUS

The ACX_FP_MULT_PLUS module computes A×B+C, with optional accumulation. Internal register stages can be enabled to allow for higher operating frequencies.

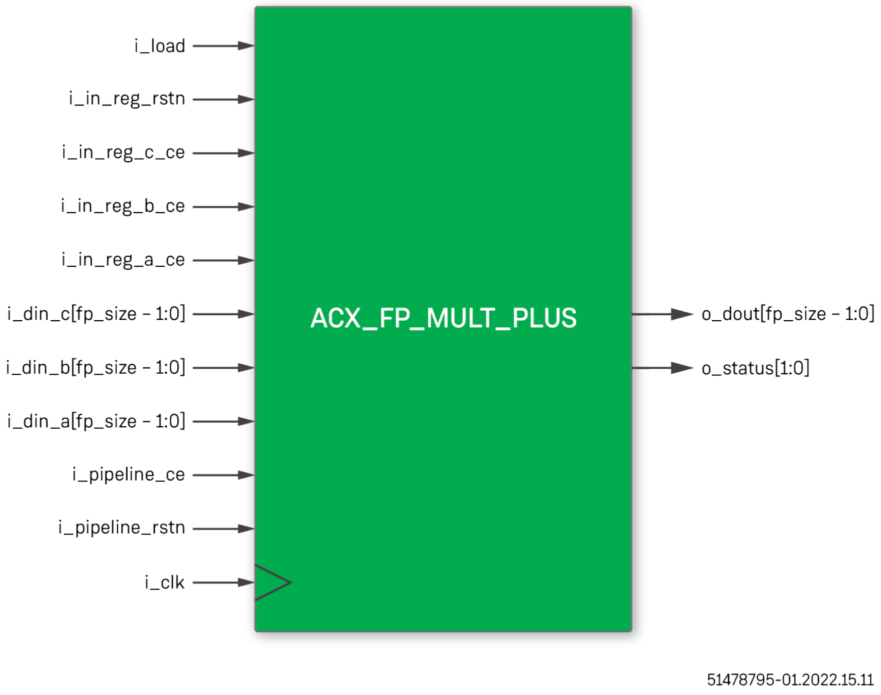


Figure 112: ***Floating-Point Multiplier Plus Adder With Optional Accumulate***

Parameters

Table 207: ***ACX_FP_MULT_PLUS Parameters***

Parameter	Supported Values	Default	Description
`fp_size`	16, 24	16	Width of floating-point number. Supports fp24, fp16, and fp16e8.
`fp_exp_size`	5, 8	5	Size of floating-point exponent.
`subtract`	0, 1	0	0 – compute `i_din_a × i_din_b + i_din_c`. 1 – compute `i_din_a × i_din_b - i_din_c`.
`accumulate`	0, 1	0	0 – no accumulation: `dout = i_din_a × i_din_b ± i_din_c`. 1 – accumulation: `dout` is the accumulated value. The start of accumulation is signaled by asserting `i_load=1`.
`in_reg_enable`	0, 1	0	0 – no input registers. 1 – `i_din_a`, `i_din_b`, and `i_din_c` are registered. The input registers are controlled by the `i_in_reg_a_ce`, `i_in_reg_b_ce`, `i_in_reg_c_ce`, and `i_in_reg_rstn` inputs. Enabling the input registers adds one cycle of latency.
`pipeline_regs`	0–5	0	The number of pipeline registers, not counting the input register. The total latency is `pipeline_regs + in_reg_enable`.

Ports

Table 208: *ACX_FP_MULT_PLUS Pin Descriptions*

Name	Direction	Description
`i_clk`	Input	Clock input. All inputs are registered on rising edge of `i_clk`. All outputs are synchronous to `i_clk`.
`i_din_a[(fp_size-1):0]`	Input	'A' data input to multiplier.
`i_din_b[(fp_size-1):0]`	Input	'B' data input to multiplier.
`i_din_c[(fp_size-1):0]`	Input	'C' data input direct to adder.
`i_in_reg_a_ce`	Input	If `in_reg_enable=0` – ignored. If `in_reg_enable=1` – clock enable for `i_din_a`.
`i_in_reg_b_ce`	Input	If `in_reg_enable=0` – ignored. If `in_reg_enable=1` – clock enable for `i_din_b`.
`i_in_reg_c_ce`	Input	If `in_reg_enable=0` – ignored. If `in_reg_enable=1` – clock enable for `i_din_c`.
`i_in_reg_rstn`	Input	If `in_reg_enable=0` – ignored. If `in_reg_enable=1` – synchronous active-low reset for input registers.
`i_pipeline_ce`	Input	If `pipeline_regs=0` – ignored. If `pipeline_regs>1` – clock enable for pipeline and accumulator registers.
`i_pipeline_rstn`	Input	If `pipeline_regs=0` – ignored. If `pipeline_regs>1` – synchronous active-low reset for pipeline and accumulator registers.
`i_load`	Input	If `accumulate=0` – ignored. If `accumulate=1` – resets the accumulator to `i_din_a × i_din_b ± i_din_c`, ignoring the previous value. This signal is internally pipelined to have the same latency as `i_din_a × i_din_b ± i_din_c`.
`o_dout[(fp_size-1):0]`	Output	Result of multiplication and accumulation.
`o_status[1:0]`[(1)]	Output	Error status of `o_dout`.

Table Notes

1. See Output Status for details.

Usage and Inference

ACX_FP_MULT_PLUS cannot be inferred and must be directly instantiated. The specified floating point format applies to the inputs and output but, internally, the operations are performed with fp24. The multiplication result A×B is rounded (to fp24) before being added to C. Thus, this is not the fusedMultiplyAdd operation defined in the IEEE-754 standard (which would avoid the intermediate rounding step).

Instantiation Templates

Verilog

```
// Verilog template for ACX_FP_MULT_PLUS
ACX_FP_MULT_PLUS #(
    .fp_size       (fp_size       ),
    .fp_exp_size   (fp_exp_size   ),
    .subtract      (subtract      ),
    .accumulate    (accumulate    ),
    .in_reg_enable (in_reg_enable ),
    .pipeline_regs (pipeline_regs )
) instance_name (
    .i_clk           (user_i_clk           ),
    .i_din_a         (user_i_din_a         ),
    .i_din_b         (user_i_din_b         ),
    .i_din_c         (user_i_din_c         ),
    .i_in_reg_a_ce   (user_i_in_reg_a_ce   ),
    .i_in_reg_b_ce   (user_i_in_reg_b_ce   ),
    .i_in_reg_c_ce   (user_i_in_reg_c_ce   ),
    .i_in_reg_rstn   (user_i_in_reg_rstn   ),
    .i_pipeline_ce   (user_i_pipeline_ce   ),
    .i_pipeline_rstn (user_i_pipeline_rstn ),
    .i_load          (user_i_load          ),
    .o_dout          (user_o_dout          ),
    .o_status        (user_o_status        )
);
```

VHDL

```
-- VHDL Component template for ACX_FP_MULT_PLUS
component ACX_FP_MULT_PLUS is
generic (
    fp_size              : integer := 16;
    fp_exp_size          : integer := 5;
    subtract             : integer := 0;
    accumulate           : integer := 0;
    in_reg_enable        : integer := 0;
    pipeline_regs        : integer := 0
);
port (
    i_clk                : in  std_logic;
    i_din_a              : in  std_logic_vector( fp_size  1 downto 0 );
    i_din_b              : in  std_logic_vector( fp_size  1 downto 0 );
    i_din_c              : in  std_logic_vector( fp_size  1 downto 0 );
    i_in_reg_a_ce        : in  std_logic;
    i_in_reg_b_ce        : in  std_logic;
    i_in_reg_c_ce        : in  std_logic;
    i_in_reg_rstn        : in  std_logic;
    i_pipeline_ce        : in  std_logic;
    i_pipeline_rstn      : in  std_logic;
    i_load               : in  std_logic;
    o_dout               : out std_logic_vector( fp_size  1 downto 0 );
    o_status             : out std_logic_vector( 1 downto 0 )
);
end component ACX_FP_MULT_PLUS

-- VHDL Instantiation template for ACX_FP_MULT_PLUS
instance_name : ACX_FP_MULT_PLUS
generic map (
    fp_size              => fp_size,
    fp_exp_size          => fp_exp_size,
    subtract             => subtract,
    accumulate           => accumulate,
    in_reg_enable        => in_reg_enable,
    pipeline_regs        => pipeline_regs
)
port map (
    i_clk                => user_i_clk,
    i_din_a              => user_i_din_a,
    i_din_b              => user_i_din_b,
    i_din_c              => user_i_din_c,
    i_in_reg_a_ce        => user_i_in_reg_a_ce,
    i_in_reg_b_ce        => user_i_in_reg_b_ce,
    i_in_reg_c_ce        => user_i_in_reg_c_ce,
    i_in_reg_rstn        => user_i_in_reg_rstn,
    i_pipeline_ce        => user_i_pipeline_ce,
    i_pipeline_rstn      => user_i_pipeline_rstn,
    i_load               => user_i_load,
    o_dout               => user_o_dout,
    o_status             => user_o_status
);
```

ACX_FP_MULT_2X

The ACX_FP_MULT_2X module is similar to ACX_FP_MULT, but uses a single ACX_MLP72 to compute two products in parallel, with optional accumulations. The two operations are:

- `dout_ab = i_din_a × i_din_b`
- `dout_cd = i_din_c × i_din_d`

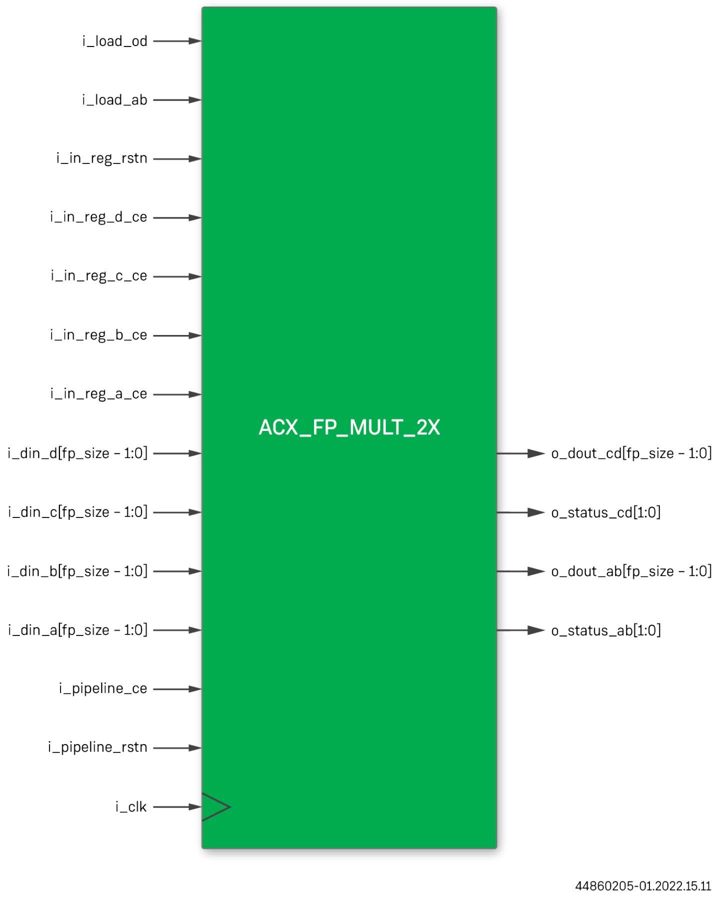


Figure 113: ***Twin Floating-Point Multipliers With Optional Accumulate***

Parameters

Table 209: *ACX_FP_MULT_2X Parameters*

Parameter	Supported Values	Default	Description
`fp_size`	16, 24	16	Width of floating-point number. Supports fp24, fp16, and fp16e8.
`fp_exp_size`	5, 8	5	Size of floating-point exponent.
`accumulate`	0, 1	0	0 – no accumulation: `dout_ab = i_din_a × i_din_b`, `dout_cd = i_din_c × i_din_d`. 1 – accumulation: `dout_ab` and `dout_cd` are the accumulated values. The start of accumulation is signaled by asserting `i_load_ab=1` or `i_load_cd=1`, respectively.
`in_reg_enable`	0, 1	0	0 – no input registers. 1 – `i_din_a`, `i_din_b`, `i_din_c` and `i_din_d` are registered. The input registers are controlled by the `i_in_reg_a_ce`, `i_in_reg_b_ce`, `i_in_reg_c_ce`, `i_in_reg_d_ce` and `i_in_reg_rstn` inputs. Enabling the input registers adds one cycle of latency.
`pipeline_regs`	0–4	0	The number of pipeline registers, not counting the input register. The total latency is `pipeline_regs + in_reg_enable`.

Ports

Table 210: *ACX_FP_MULT_2X Pin Descriptions*

Name	Direction	Description
i_clk	Input	Clock input, used for the (optional) registers and accumulator.
i_din_a[(fp_size-1):0]	Input	'A' data input to AB multiplier.
i_din_b[(fp_size-1):0]	Input	'B' data input to AB multiplier.
i_din_c[(fp_size-1):0]	Input	'C' data input to CD multiplier.
i_din_d[(fp_size-1):0]	Input	'D' data input to CD multiplier.
i_in_reg_a_ce	Input	If in_reg_enable=0 – ignored. If in_reg_enable=1 – clock enable for i_din_a.
i_in_reg_b_ce	Input	If in_reg_enable=0 – ignored. If in_reg_enable=1 – clock enable for i_din_b.
i_in_reg_c_ce	Input	If in_reg_enable=0 – ignored. If in_reg_enable=1 – clock enable for i_din_c.
i_in_reg_d_ce	Input	If in_reg_enable=0 – ignored. If in_reg_enable=1 – clock enable for i_din_d.
i_in_reg_rstn	Input	If in_reg_enable=0 – ignored. If in_reg_enable=1 – synchronous active-low reset for input registers.
i_pipeline_ce	Input	If pipeline_regs=0 – ignored. If pipeline_regs>1 – clock enable for pipeline and accumulator registers.
i_pipeline_rstn	Input	If pipeline_regs=0 – ignored. If pipeline_regs>1 – synchronous active-low reset for pipeline and accumulator registers.
i_load_ab	Input	If accumulate=0 – ignored. If accumulate=1 – resets the AB accumulator to i_din_a × i_din_b, ignoring the previous value. This signal is internally pipelined to have the same latency as i_din_a × i_din_b.
i_load_cd	Input	If accumulate=0 – ignored. If accumulate=1 – resets the CD accumulator to i_din_c × i_din_d, ignoring the previous value. This signal is internally pipelined to have the same latency as i_din_c × i_din_d.
o_dout_ab[(fp_size-1):0]	Output	Result of A × B multiplication and accumulation.
o_dout_cd[(fp_size-1):0]	Output	Result of C × D multiplication and accumulation.
o_status_ab[1:0](1)	Output	Error status of o_dout_ab.
o_status_cd[1:0](1)	Output	Error status of o_dout_cd.

Name	Direction	Description
Table Notes 1. See Output Status for details.		

Usage and Inference

ACX_FP_MULT_2X cannot be inferred and must be directly instantiated. The specified floating point format applies to the inputs and outputs but, internally, the operations are performed with fp24.

If `fp_size=24`, the four data inputs require 96 bits total. Since this is more than 72 bits, the ACX_MLP72 that performs the operation is used in wide input mode. In this mode, the adjacent ACX_BRAM72K site is used as route-through, meaning it is no longer available for BRAM placement. By contrast, if `fp_size=16`, only 64 input bits are needed and normal input mode is used.

Instantiation Templates

Verilog

```
// Verilog template for ACX_FP_MULT_2X
ACX_FP_MULT_2X #(
    .fp_size       (fp_size       ),
    .fp_exp_size   (fp_exp_size   ),
    .accumulate    (accumulate    ),
    .in_reg_enable (in_reg_enable ),
    .pipeline_regs (pipeline_regs )
) instance_name (
    .i_clk           (user_i_clk           ),
    .i_din_a         (user_i_din_a         ),
    .i_din_b         (user_i_din_b         ),
    .i_din_c         (user_i_din_c         ),
    .i_din_d         (user_i_din_d         ),
    .i_in_reg_a_ce   (user_i_in_reg_a_ce   ),
    .i_in_reg_b_ce   (user_i_in_reg_b_ce   ),
    .i_in_reg_c_ce   (user_i_in_reg_c_ce   ),
    .i_in_reg_d_ce   (user_i_in_reg_d_ce   ),
    .i_in_reg_rstn   (user_i_in_reg_rstn   ),
    .i_pipeline_ce   (user_i_pipeline_ce   ),
    .i_pipeline_rstn (user_i_pipeline_rstn ),
    .i_load_ab       (user_i_load_ab       ),
    .i_load_cd       (user_i_load_cd       ),
    .o_dout_ab       (user_o_dout_ab       ),
    .o_dout_cd       (user_o_dout_cd       ),
    .o_status_ab     (user_o_status_ab     ),
    .o_status_cd     (user_o_status_cd     )
);
```

VHDL

```
-- VHDL Component template for ACX_FP_MULT_2X
component ACX_FP_MULT_2X is
generic (
    fp_size              : integer := 16;
    fp_exp_size          : integer := 5;
    accumulate           : integer := 0;
    in_reg_enable        : integer := 0;
    pipeline_regs        : integer := 0
);
port (
    i_clk                : in  std_logic;
    i_din_a              : in  std_logic_vector( fp_size-1 downto 0 );
    i_din_b              : in  std_logic_vector( fp_size-1 downto 0 );
    i_din_c              : in  std_logic_vector( fp_size-1 downto 0 );
    i_din_d              : in  std_logic_vector( fp_size-1 downto 0 );
    i_in_reg_a_ce        : in  std_logic;
    i_in_reg_b_ce        : in  std_logic;
    i_in_reg_c_ce        : in  std_logic;
    i_in_reg_d_ce        : in  std_logic;
    i_in_reg_rstn        : in  std_logic;
    i_pipeline_ce        : in  std_logic;
    i_pipeline_rstn      : in  std_logic;
    i_load_ab            : in  std_logic;
    i_load_cd            : in  std_logic;
    o_dout_ab            : out std_logic_vector( fp_size-1 downto 0 );
    o_dout_cd            : out std_logic_vector( fp_size-1 downto 0 );
    o_status_ab          : out std_logic_vector( 1 downto 0 );
    o_status_cd          : out std_logic_vector( 1 downto 0 )
);
end component ACX_FP_MULT_2X

-- VHDL Instantiation template for ACX_FP_MULT_2X
instance_name : ACX_FP_MULT_2X
generic map (
    fp_size              => fp_size,
    fp_exp_size          => fp_exp_size,
    accumulate           => accumulate,
    in_reg_enable        => in_reg_enable,
    pipeline_regs        => pipeline_regs
)
port map (
    i_clk                => user_i_clk,
    i_din_a              => user_i_din_a,
    i_din_b              => user_i_din_b,
    i_din_c              => user_i_din_c,
    i_din_d              => user_i_din_d,
    i_in_reg_a_ce        => user_i_in_reg_a_ce,
    i_in_reg_b_ce        => user_i_in_reg_b_ce,
    i_in_reg_c_ce        => user_i_in_reg_c_ce,
    i_in_reg_d_ce        => user_i_in_reg_d_ce,
    i_in_reg_rstn        => user_i_in_reg_rstn,
    i_pipeline_ce        => user_i_pipeline_ce,
    i_pipeline_rstn      => user_i_pipeline_rstn,
    i_load_ab            => user_i_load_ab,
    i_load_cd            => user_i_load_cd,
    o_dout_ab            => user_o_dout_ab,
    o_dout_cd            => user_o_dout_cd,
    o_status_ab          => user_o_status_ab,
    o_status_cd          => user_o_status_cd
);
```

ACX_FP_MULT_ADD

The ACX_FP_MULT_ADD module computes (A×B) + (C×D), with optional accumulation. Internal register stages can be enabled to allow for higher operating frequencies.

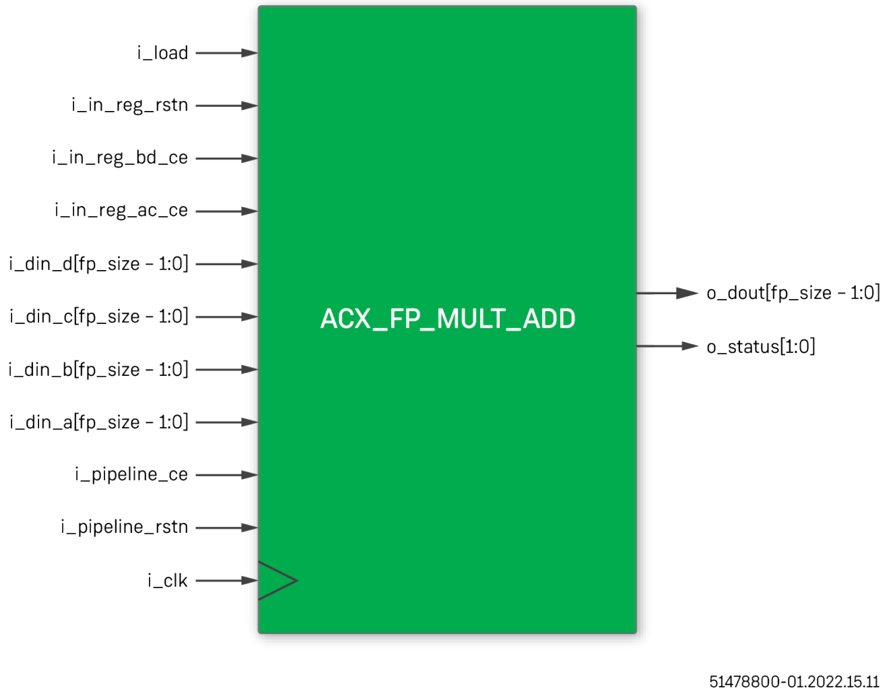


Figure 114: ***Twin Floating-Point Multiplies With Addition and Optional Accumulation***

Parameters

Table 211: *ACX_FP_MULT_ADD Parameters*

Parameter	Supported Values	Default	Description
`fp_size`	16, 24	16	Width of floating-point number. Supports fp24, fp16, and fp16e8.
`fp_exp_size`	5, 8	5	Size of floating-point exponent.
`subtract`	0, 1	0	0 – compute `(i_din_a × i_din_b) + (i_din_c × i_din_d)`. 1 – compute `(i_din_a × i_din_b) - (i_din_c × i_din_d)`.
`accumulate`	0, 1	0	0 – no accumulation: `dout = (i_din_a × i_din_b) ± (i_din_c × i_din_d)`. 1 – accumulation: `dout` is the accumulated value. The start of accumulation is signaled by asserting `i_load=1`.
`in_reg_enable`	0, 1	0	0 – no input registers. 1 – `i_din_a`, `i_din_b`, `i_din_c` and `i_din_d` are registered. The input registers are controlled by the `i_in_reg_ac_ce`, `i_in_reg_bd_ce` and `i_in_reg_rstn` inputs. Enabling the input registers adds one cycle of latency.
`pipeline_regs`	0–5	0	The number of pipeline registers not counting the input register. The total latency is `pipeline_regs + in_reg_enable`.

Ports

Table 212: ***ACX_FP_MULT_ADD Pin Descriptions***

Name	Direction	Description
`i_clk`	Input	Clock input, used for the (optional) registers and accumulator.
`i_din_a[(fp_size-1):0]`	Input	'A' data input to AB multiplier.
`i_din_b[(fp_size-1):0]`	Input	'B' data input to AB multiplier.
`i_din_c[(fp_size-1):0]`	Input	'C' data input to CD multiplier.
`i_din_d[(fp_size-1):0]`	Input	'D' data input to CD multiplier.
`i_in_reg_ac_ce`	Input	If `in_reg_enable=0` – ignored. If `in_reg_enable=1` – clock enable for `i_din_a` and `i_din_c`.
`i_in_reg_bd_ce`	Input	If `in_reg_enable=0` – ignored. If `in_reg_enable=1` – clock enable for `i_din_b` and `i_din_d`.
`i_in_reg_rstn`	Input	If `in_reg_enable=0` – ignored. If `in_reg_enable=1` – synchronous active-low reset for input registers.
`i_pipeline_ce`	Input	If `pipeline_regs=0` – ignored. If `pipeline_regs>1` – clock enable for pipeline and accumulator registers.
`i_pipeline_rstn`	Input	If `pipeline_regs=0` – ignored. If `pipeline_regs>1` – synchronous active-low reset for pipeline and accumulator registers.
`i_load`	Input	If `accumulate=0` – ignored. If `accumulate=1` – resets the accumulator to: `(i_din_a × i_din_b) ± (i_din_c × i_din_d)`, ignoring the previous value. This signal is internally pipelined to have the same latency as: `(i_din_a × i_din_b) ± (i_din_c × i_din_d)`.
`o_dout[(fp_size-1):0]`	Output	Result of multiplication and accumulation.
`o_status[1:0]`(1)	Output	Error status of `o_dout`.

Table Notes

1. See Output Status for details.

Usage and Inference

ACX_FP_MULT_ADD cannot be inferred and must be directly instantiated. The specified floating point format applies to the inputs and output but, internally, the operations are performed with fp24.

If `fp_size=24`, the four data inputs require 96 bits total. Since this is more than 72 bits, the ACX_MLP72 that performs the operation is used in wide input mode. In this mode, the adjacent ACX_BRAM72K site is used as route-through, meaning it is no longer available for BRAM placement. By contrast, if `fp_size=16`, only 64 input bits are needed, and normal input mode is used.

Instantiation Templates

Verilog

```
// Verilog template for ACX_FP_MULT_ADD
ACX_FP_MULT_ADD #(
    .fp_size       (fp_size       ),
    .fp_exp_size   (fp_exp_size   ),
    .subtract      (subtract      ),
    .accumulate    (accumulate    ),
    .in_reg_enable (in_reg_enable ),
    .pipeline_regs (pipeline_regs )
) instance_name (
    .i_clk           (user_i_clk           ),
    .i_din_a         (user_i_din_a         ),
    .i_din_b         (user_i_din_b         ),
    .i_din_c         (user_i_din_c         ),
    .i_din_d         (user_i_din_d         ),
    .i_in_reg_ac_ce  (user_i_in_reg_ac_ce  ),
    .i_in_reg_bd_ce  (user_i_in_reg_bd_ce  ),
    .i_in_reg_rstn   (user_i_in_reg_rstn   ),
    .i_pipeline_ce   (user_i_pipeline_ce   ),
    .i_pipeline_rstn (user_i_pipeline_rstn ),
    .i_load          (user_i_load          ),
    .o_dout          (user_o_dout          ),
    .o_status        (user_o_status        )
);
```

VHDL

```
-- VHDL Component template for ACX_FP_MULT_ADD
component ACX_FP_MULT_ADD is
generic (
    fp_size              : integer := 16;
    fp_exp_size          : integer := 5;
    subtract             : integer := 0;
    accumulate           : integer := 0;
    in_reg_enable        : integer := 0;
    pipeline_regs        : integer := 0
);
port (
    i_clk                : in  std_logic;
    i_din_a              : in  std_logic_vector( fp_size-1 downto 0 );
    i_din_b              : in  std_logic_vector( fp_size-1 downto 0 );
    i_din_c              : in  std_logic_vector( fp_size-1 downto 0 );
    i_din_d              : in  std_logic_vector( fp_size-1 downto 0 );
    i_in_reg_ac_ce       : in  std_logic;
    i_in_reg_bd_ce       : in  std_logic;
    i_in_reg_rstn        : in  std_logic;
    i_pipeline_ce        : in  std_logic;
    i_pipeline_rstn      : in  std_logic;
    i_load               : in  std_logic;
    o_dout               : out std_logic_vector( fp_size-1 downto 0 );
    o_status             : out std_logic_vector( 1 downto 0 )
);
end component ACX_FP_MULT_ADD

-- VHDL Instantiation template for ACX_FP_MULT_ADD
instance_name : ACX_FP_MULT_ADD
generic map (
    fp_size              => fp_size,
    fp_exp_size          => fp_exp_size,
    subtract             => subtract,
    accumulate           => accumulate,
    in_reg_enable        => in_reg_enable,
    pipeline_regs        => pipeline_regs
)
port map (
    i_clk                => user_i_clk,
    i_din_a              => user_i_din_a,
    i_din_b              => user_i_din_b,
    i_din_c              => user_i_din_c,
    i_din_d              => user_i_din_d,
    i_in_reg_ac_ce       => user_i_in_reg_ac_ce,
    i_in_reg_bd_ce       => user_i_in_reg_bd_ce,
    i_in_reg_rstn        => user_i_in_reg_rstn,
    i_pipeline_ce        => user_i_pipeline_ce,
    i_pipeline_rstn      => user_i_pipeline_rstn,
    i_load               => user_i_load,
    o_dout               => user_o_dout,
    o_status             => user_o_status
);
```

Chapter - 6: Memories

ACX_BRAMSDP (20-kb Simple Dual-Port Memory with Error Correction)

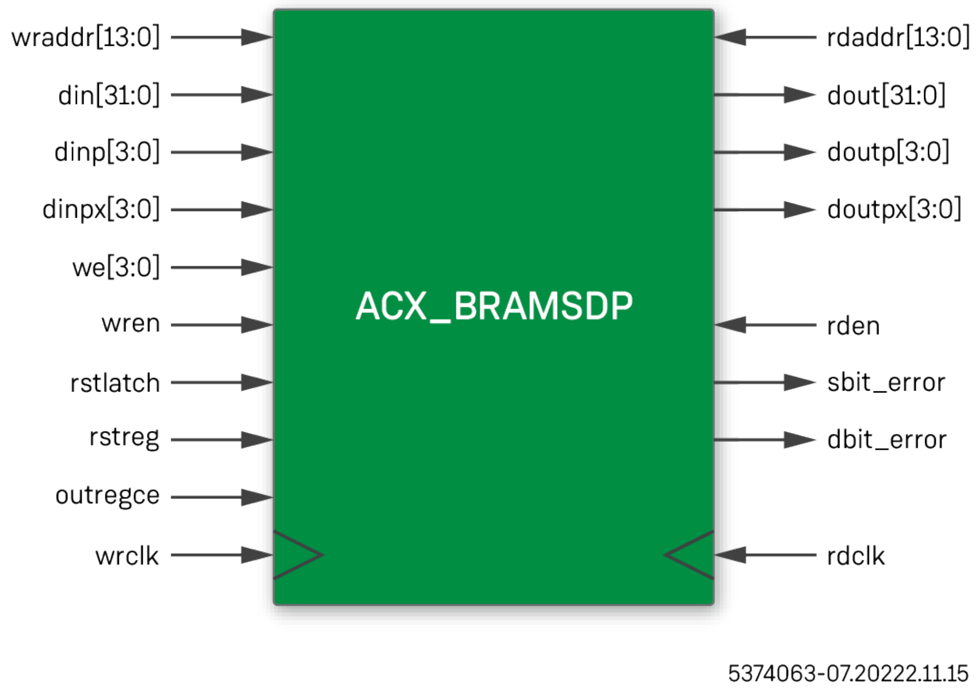


Figure 115: ***20-kb Simple Dual-Port Memory With Error Correction***

The block RAM (ACX_BRAMSDP) implements a 20kb simple-dual-port (SDP) memory block with one write port and one read port. Each port can be independently configured as follows:

- 512 × 40
- 512 × 36
- 512 × 32
- 1k × 20
- 1k × 18
- 1k × 16
- 2k × 10
- 2k × 9
- 2k × 8
- 4k × 5
- 4k × 4
- 8k × 2
- 16k × 1

The read and write operations are both synchronous. For higher performance operation, an additional output register can be enabled which causes an additional cycle of read latency. Write enable (we) controls provide 8-bit, 9-bit, or 10-bit byte enable control for port widths above 16 bits.

The initial value of the memory contents may be specified either with parameters or with a memory initialization file. The initial/reset values of the output registers may also be specified.

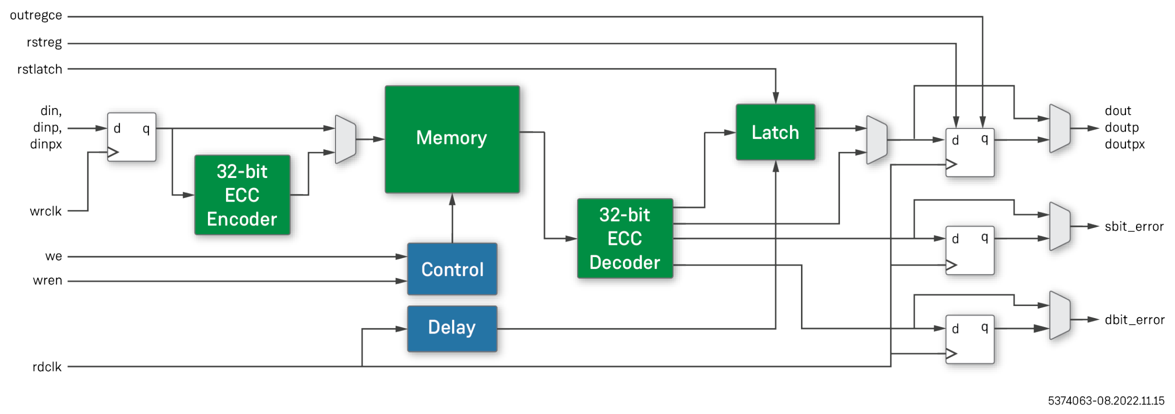


Figure 116: *ACX_BRAMSDP Block Diagram (Per Port)*

Table 213: ***ACX_BRAMSDP Pin Descriptions***

Name	Type	Description
`wrclk`, `rdclk`	Input	Write/read clock inputs. Read/write operations are fully synchronous and occur upon the active edge of `wrclk`/`rdclk` when the `wren`/`rden` signal is high. The active edge of `wrclk`/`rdclk` is determined by the `write_clock_polarity`/`read_clock_polarity` parameter.
`rden`	Input	Read port enable. Asserted to perform a read operation. If the `read_peval` parameter is `1'b0`, `rden` is active low, otherwise if `1'b1`, `rden` is active high.
`rdaddr[13:0]`	Input	Determines which memory location is being read. When the port width is greater than 1, the address must be top-justified, meaning that the low order address bits must be tied to 0.
`wren`	Input	Write port enable. Asserted to perform a write operation. If the `write_peval` parameter is `1'b0`, `wren` is active low, otherwise if `1'b1`, `wren` is active high.
`wraddr[13:0]`	Input	Determines which memory location is being written. When the port width is greater than 1, the address must be top-justified, meaning that the low order address bits must be tied to 0.
`we[3:0]`	Input	Write port byte-wide write enable. Each bit enables a 10-bit byte to be written to the memory block as follows: Byte writes are enabled on the write port when both the `wren` signal is asserted and the corresponding bit in the `we` signal is asserted. For write port widths <= 20, `we[3:2]` must be tied to `2'b00`. For write port width <= 10, `we[1]` must be tied to `we[0]`.
`din[31:0]`,	Input	Write port data input.
`dinp[3:0]`	Input	Write port parity input. May be used for data.
`dinpx[3:0]`	Input	Write port extended parity input. May be used for data.
`rstlatch`	Input	Output latch synchronous reset. When asserted, the value of the `read_srval` parameter is written to the output latch on the next active edge of `rdclk`.
`rstreg`	Input	Output register reset. The `sr_assertion_reg` parameter determines whether the reset is synchronous (default) or asynchronous, and the `reg_rstval` parameter determines whether it is active-high (default) or active-low. When reset is asserted, the output register is assigned the value of the `read_srval` parameter. The priority of the `rstreg` input relative to the clock enable input, `outregce`, is determined by the value of the `regce_priority` parameter.
`outregce`	Input	Output register clock enable (active-high). When the `en_out_reg` parameter is `1'b1`, de-asserting the `outregce` signal causes the BRAM output to retain the `dout`, `doutp`, and `doutpx` signals unchanged, independent of a read operation. When `en_out_reg` is 1'b0, the `outregce` input is ignored.
`dout[31:0]`	Output	Read port data output. For read operations, `dout` is updated with the memory contents addressed by `rdaddr` if the `rden` port enable is active.
`doutp[3:0]`	Output	Read port parity output. Behaves in the same manner as `dout` and is used when the `read_width` is set to 5, 9, 10, 18, 20, 36, or 40 bits.
`doutpx[3:0]`	Output	Read port extended parity output. Behaves in the same manner as `dout` and is used when the `read_width` is set to 10, 20 or 40 bits.
`sbit_error`	Output	Single-bit error (active-high). Asserted during a read operation when the `decoder_enable` parameter is `1'b1`, and a single-bit error is detected. In this case, the corrected word is output on the `dout` pins. The memory contents are not corrected by the error correction circuitry. The `sbit_error` signal is aligned with the associated read data word.

Name	Type	Description
dbit_error	Output	Dual-bit error (active-high). Asserted during a read operation when the decoder_enable parameter is 1'b1, and a dual-bit error is detected. In the case of a dual-bit error condition, the uncorrected read data word is output on the dout pins. The dbit_error signal is aligned with the associated read data word.

Table 214: *ACX_BRAMSDP Parameters*

Parameter	Defined Values	Default Value	Description
`read_width`	1, 2, 4, 5, 8, 9, 10, 16, 18, 20, 32, 36, 40	40	Sets the width of the read port.
`write_width`	1, 2, 4, 5, 8, 9, 10, 16, 18, 20, 32, 36, 40	40	Sets the width of the write port. May vary from the write port width, but it must be within the allowable combinations defined in Memory Organization and Data Input/Output Pin Assignments (see page 295).
`write_clock_polarity`, `read_clock_polarity`	`rise`, `fall`	`rise`	Used to set the active edge of the read and write clocks.
`write_peval`	`1'b0`, `1'b1`	`1'b1`	Defines the active level of the `wren` port. A value of `1'b0` sets active low, while `1'b1` sets active-high.
`read_peval`	`1'b0`, `1'b1`	`1'b1`	Defines the active level of the `rden` ports. A value of `1'b0` sets active low, `1'b1` sets active-high.
`latch_rstval`	`1'b0`, `1'b1`	`1'b1`	Defines the active level of the `rstlatch` input. A value of `1'b0` sets active low, while `1'b1` sets active high.
`en_out_reg`	`1'b0`, `1'b1`	`1'b0`	Determines whether the output register is enabled. A value of `1'b0` disables the output register and results in a read latency of one cycle, while `1'b1` enables the output register and results in a read latency of two cycles.
`reg_rstval`	`1'b0`, `1'b1`	`1'b1`	Defines the active level of the `rstreg` input. A value of `1'b0` sets `rstreg` active low, while `1'b1` sets active high.
`regce_priority`	`rstreg`, `regce`	`rstreg`	Defines the priority of the `outregce` clock enable input relative to the `rstreg` reset. • "`rstreg`" – allows the output register to be reset by asserting `rstreg` without requiring assertion of `outregce`. • "`regce`" – allows the output register to be reset only by asserting both `rstreg` and `outregce` together.
`read_initval`(1)	40-bit number	`40'h0`	When enabled, defines the power-up default value of the data on the output of the latch and output register. Assignment is dependent on the `read_width` parameter as shown in Table: initval, srval, and meminit File Mapping to Output Signals (see page 304).
`read_srval`(1)	40-bit number	`40'h0`	When enabled, defines the reset value of the data on the output of the latch and output register, when `rstlatch` and/or `rstreg` is asserted. Assignment is dependent on the `read_width` parameter as shown in Table: initval, srval, and meminit File Mapping to Output Signals (see page 304).
`sr_assertion_reg`	`clocked`, `un clocked`	`clocked`	Sets whether the assertion of the output register reset is synchronous or asynchronous with respect to the `rdclk` input. A value of "`clocked`" sets synchronous reset where the output register is reset at the next rising edge of the clock if `rstreg` is asserted. A value of "`unclocked`" sets asynchronous reset where the output register is reset immediately following the assertion of the `rstreg` input.
			Provides a mechanism to set the initial contents of the ACX_BRAMSDP memory.

Parameter	Defined Values	Default Value	Description
`mem_init_file`	<path to HEX file>	""	• If this parameter is defined, the BRAM is initialized with the values defined in the file pointed to by the parameter according to the format defined in Memory Initialization (see page 304). • If left at the default value (""), the initial contents are defined by the values of the `initd_00–initd_63`, `initp_0–initp_7`, and `initpx_0–initpx_7` parameters. • If the memory initialization parameters and the `mem_init_file` parameters are not defined, the contents of the BRAM remain uninitialized.
`initd_00–initd_63`	256-bit hex value	`256'hx`	The `initd_00` through `initd_63` parameters define the initial contents of the memory associated with `douta[15:0]` and `doutb[15:0]`. Each 256-bit parameter associated with the BRAM memory is defined in Memory Initialization (see page 304).
`initp_0–initp_7`	256-bit hex value	`256'hx`	The `initp_0` through `initp_7` parameters define the initial contents of the memory associated with `doutpa[1:0]` and `doutpb[1:0]`. Each 256-bit parameter associated with the BRAM memory is defined in Memory Initialization (see page 304).
`initpx_0–initpx_7`	256-bit hex value	`256'hx`	The `initpx_0` through `initpx_7` parameters define the initial contents of the memory associated with `doutpxa[1:0]` and `doutpxb[1:0]`. Each 256-bit parameter associated with the BRAM memory is defined in Memory Initialization (see page 304).
`encoder_enable`	`1'b0`, `1'b1`	`1'b0`	Determines if the ECC encoder circuitry is selected or bypassed. A value of `1'b1` enables the ECC encoder for normal operation, while `1'b0` disables the ECC encoder circuitry and allows the `dinp` and `dinpx` inputs to be connected directly to the underlying memory array.
`decoder_enable`	`1'b0`, `1'b1`	`1'b0`	Determines if the ECC decoder circuitry is selected or bypassed. A value of `1'b1` enables the ECC decoder for normal operation while `1'b0` disables the ECC decoder circuitry and allows the `doutp` and `doutpx` memory outputs to be driven directly from the underlying memory array.

Table Notes

1. Special Case for ECC Mode: This parameter has no effect when the ECC decoder is enabled (`decoder_enable == 1'b1`), and the BRAM data output register is disabled (`en_out_reg == 1'b0`). In this configuration, the BRAM output latch is bypassed, and the power-up default value of the data output is undefined. To enable read port init values and/or reset values in ECC mode, the output register must be enabled (`en_out_reg == 1'b1`).

Note

The ACE BRAM IP Configuration GUI and ACX_BRAM_GEN macros only support a single bit write enable (we) for the entire data word. Byte-wise write enables are not supported via the GUI or in Verilog macros. Access to the full capabilities of the BRAM is available by instantiating the ACX_BRAMSDP primitive directly.

Memory Organization and Data Input/Output Pin Assignments

The ACX_BRAMSDP block supports memory widths from one to forty bits wide. The width of the `din` data input is determined by the `write_width` parameter while the `dout` data output width is determined by the `read_width` parameter. The read port and write port widths may be different. There are some limitations of the port width assignments between the read and write width assignments, these limitations and the supported port width combinations are described in the following table. 'X' indicates a supported configuration.

Table 215: *Supported Width Combinations*

Read Width	Write Width												
	512 x 40	1k × 20	2k × 10	4k × 5	512 x 36	1k × 18	2k × 9	512 x 32	1k × 16	2k × 8	4k × 4	8k × 2	16k × 1
512 × 40	X	X	X	X	–	–	–	–	–	–	–	–	–
1k × 20	X	X	X	X	–	–	–	–	–	–	–	–	–
2k × 10	X	X	X	X	–	–	–	–	–	–	–	–	–
4k × 5	X	X	X	X	–	–	–	–	–	–	–	–	–
512 x 36	–	–	–	–	X	X	X	–	–	–	–	–	–
1k × 18	–	–	–	–	X	X	X	–	–	–	–	–	–
2k × 9	–	–	–	–	X	X	X	–	–	–	–	–	–
512 x 32	–	–	–	–	–	–	–	X	X	X	X	X	X
1k × 16	–	–	–	–	–	–	–	X	X	X	X	X	X
2k × 8	–	–	–	–	–	–	–	X	X	X	X	X	X
4k × 4	–	–	–	–	–	–	–	X	X	X	X	X	X
8k × 2	–	–	–	–	–	–	–	X	X	X	X	X	X
16k × 1	–	–	–	–	–	–	–	X	X	X	X	X	X

Data Widths Using Parity Pins

The ACX_BRAMSDP memory has three buses for both data in and data out; the respective `din` and `dout` interfaces, along with the `dinp`, `dinpx`, `doutp` and `doutpx` parity interfaces. When ECC is used, the parity interfaces are unused, as the ECC encoder and decoder make use of the respective memory pins for ECC operation. When ECC is disabled, the parity interfaces are assigned to the respective data buses as shown in the following table.

Table 216: ***Parity Pins Assignment, (Per Port)***

Data Width	dinpx/doutpx	dinp/doutp	din/dout
40	{data[39], data[29], data[19], data[9]}	{data[34], data[24], data[14], data[4]}	{data[38:35], data[33:30], data[28:25], data[23:20], data[18:15], data[13:10], data[8:5], data[3:0]}
36	–	{data[35], data[26], data[17], data[8]}	{data[34:27], data[25:18], data[16:9], data[7:0]}
32	–	–	data[31:0]
20	{2'b00, data[19], data[9]}	{2'b00, data[14], data[4]}	{16'h0, data[18:15], data[13:10], data[8:5], data[3:0]}
18	–	{2'b00, data[17], data[8]}	{16'h0, data[16:9], data[7:0]}
16	–	–	{16'h0, data[15:0]}
10	{3'b000, data[9]}	{3'b000, data[4]}	{24'h0, data[8:5], data[3:0]}
9	–	{3'b000, data[8]}	{24'h0, data[7:0]}
8	–	–	{24'h0, data[7:0]}
5	–	{3'b000, data[4]}	{28'h0, data[3:0]}
4	–	–	{28'h0, data[3:0]}
2	–	–	{30'h0, data[1:0]}
1	–	–	{31'h0, data[0]}

Caution!

Pay close attention to non power-of-two-sized data widths and how the data bits are assigned.

Address Bus Mapping

When the ACX_BRAMSDP is configured for memory depths of less than 16K entries, the address bus is assigned left justified, assigning the lower unused address bits to 0 as required. This is shown in the following table.

Table 217: ***ACX_BRAMSDP Address Bus Mapping (Per Port)***

Memory Organization	rdaddr/wraddr Pins	Address Pins Tied to 0
512 × 40	13:5	4:0
512 × 36	13:5	4:0
512 × 32	13:5	4:0
1k × 20	13:4	3:0
1k × 18	13:4	3:0
1k × 16	13:4	3:0
2k × 10	13:3	2:0
2k × 9	13:3	2:0
2k × 8	13:3	2:0
4k × 5	13:2	1:0
4k × 4	13:2	1:0
8k × 2	13:1	0
16k × 1	13:0	–

Warning

A common error is to assign the address bus incorrectly justified; it must be assigned *left-justified*, not right-justified.

Read and Write Operations

Timing Options

The BRAM has two options for interface timing, controlled by the `en_out_reg` parameter:

- Latched mode – when `en_out_reg` is `1'b0`, the port is in latched mode where the read address is registered and the stored data is latched into the output latches on the following clock cycle, providing a read operation with one cycle of latency.
- Registered mode – when `en_out_reg` is `1'b1`, the port is in registered mode where there is an additional register after the latch, supporting higher-frequency designs, providing a read operation with two cycles of latency.

Read Operation

Read operations are signaled by driving the `rdaddr` signal with the address to be read, and asserting the `rden` signal. The requested read data arrives on the `dout`, `doutp`, and `doutpx` signals on the following clock cycle or the cycle after, depending on the `en_out_reg` parameter value.

Table 218: ***Latched Mode BRAM Output Function Table (Assumes Rising-Edge Clock and Active-High Port Enable)***

Operation	rdclk	rstlatch	rden	dout
Hold	X	X	X	Hold previous value.
Reset latch	↑	1	X	`init_srval`
Hold	↑	0	0	Hold previous value.
Read	↑	0	1	`mem[rdaddr]`

Table 219: ***Registered Mode BRAM Output Function Table (Assumes Active-High Clock, Output Register Clock Enable, and Output Register Reset)***

Operation	regce_priority	rdclk	rstreg	outregce	dout
Hold	–	X	X	X	`dout_previous`
Reset Output	`"rstreg"`	↑	1	X	`read_srval`
Reset Output	`"regce"`	↑	1	1	`read_srval`
Hold	`"regce"`	↑	X	0	`dout_previous`
Hold	`"rstreg"` or `"regce"`	↑	0	0	`dout_previous`
Update Output	`"rstreg"` or `"regce"`	↑	0	1	Registered from latch output.

Write Operation

Write operations are signaled by asserting the `wren` signal and asserting the write enable (`we`) signal for the bytes to be written. The values of the `din`, `dinp`, and `dinpx` signals are stored in the memory array at the indicated address by the `wraddr` signal on the next active clock edge.

Simultaneous Memory Operations

Memory operations may be performed simultaneously from both sides of the memory; however, there is a restriction regarding memory collisions. A memory collision is defined as the condition where both of the ports access the same memory location(s) within the same clock cycle (both ports connected to the same clock), or within a fixed time window (if each port is connected to a different clock). If one of the ports is writing an address while the other port is reading the same address (qualified with overlapping write enables per bit), the write operation takes precedence, but the read data is invalid. The data may be reliably read on the next cycle if there is no longer a write collision.

Timing Diagrams

This section contains timing diagrams for both values of the `en_out_reg` parameter. The first timing diagram illustrates the behavior of the ACX_BRAMSDP instance with the output register disabled.

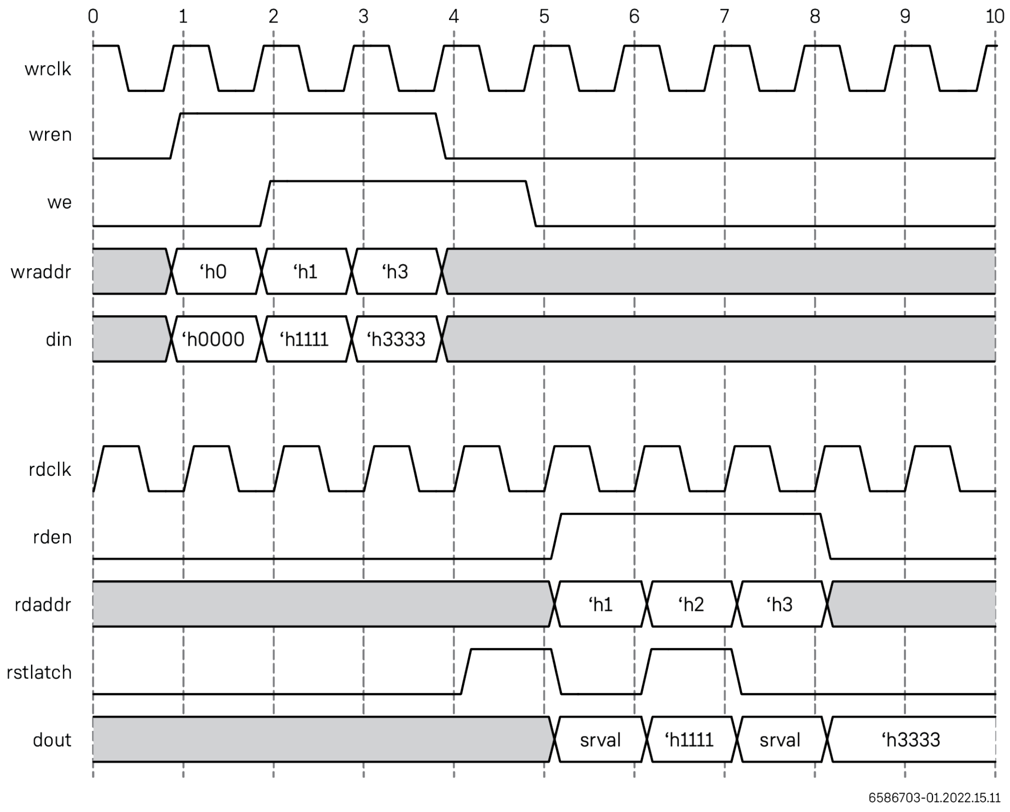


Figure 117: ***Latched Mode Read Timing Diagram***

The behavior of the ACX_BRAMSDP on each clock cycle of the preceding diagram is described in the following table, where each row represents a transaction that spans the clock cycles indicated.

Table 220: *ACX_BRAMSDP Timing Diagram Clock Cycle Behavior With Output Register Disabled*

Clock Cycle	Transaction	Description
1	No-op	`wren` is asserted but `we` is not asserted. Nothing is written to the memory array.
2	Write	`wren` and `we` are both asserted. Data on `din` is committed to `wraddr` location in the memory array.
3	Write	`wren` and `we` are both asserted. Data on `din` is committed to `wraddr` location in the memory array.
4	No-op	`wren` is not asserted. Asserted `we` is ignored and nothing is written to the memory array.
4–5	Read reset latch	`rstlatch` is asserted, causing the output to be set to `srval` as provided by the `read_srval` parameter on the next cycle.
5–6	Read	`rden` is asserted. The memory is read and presented on `dout` on the following cycle.
6–7	Read reset latch	`rden` is asserted. The memory is read. Since `rstlatch` is asserted, the output is reset to the `srval` as provided by the `read_srval` parameter.
7–8	Read	`rden` is asserted. The memory is read and presented on `dout` on the following cycle.
8–9	Hold	`rden` and `rstlatch` are both de-asserted. `dout` retains its previous value.

The second timing diagram illustrates the behavior of a ACX_BRAMSDP instance with the output register enabled.

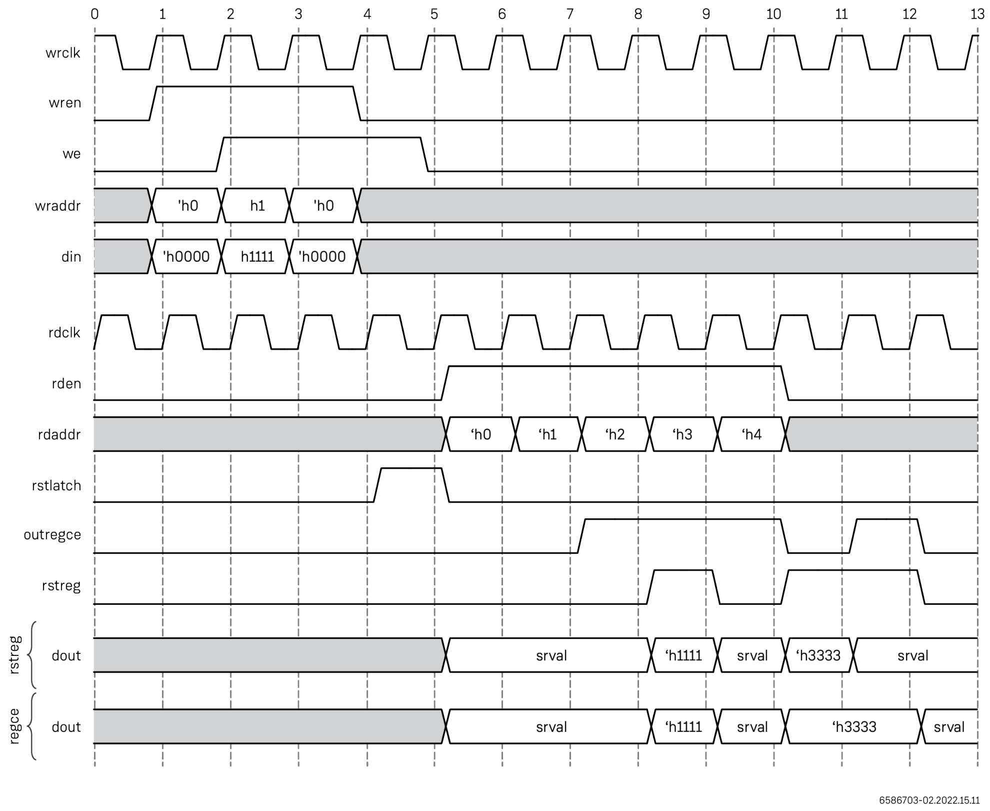


Figure 118: ***Registered Mode Read Timing Diagram***

The behavior of the ACX_BRAMSDP on each clock cycle of the preceding diagram is described in the following table, where each row represents a transaction that spans the clock cycles indicated.

Table 221: ***ACX_BRAMSDP Timing Diagram Clock Cycle Behavior With Output Register Enabled***

Clock Cycle	Transaction	Description
1	No-op	`wren` is asserted but `we` is not asserted. Nothing is written to memory.
2	Write	`wren` and `we` are both asserted. Data on `din` is committed to `wraddr` location in memory.
3	Write	`wren` and `we` are both asserted. Data on `din` is committed to `wraddr` location in memory.
4	No-op	`wren` is not asserted. The asserted `we` is ignored and nothing is written to memory.
3–5	Read reset latch	`rstlatch` is asserted on the second cycle, causing the output to be set to `srval` on the next cycle as provided by the `read_srval` parameter.
4–6	Hold	All of the control signals are de-asserted and the `dout` signals retain their previous value.
5–7	Hold	`rden` is asserted. Memory is read. Since `outregce` is de-asserted on the second cycle, `dout` retains its previous value.
6–8	Read	`rden` is asserted. Memory is read. Since `outregce` is asserted on the second cycle, `dout` provides the data that was just read from the memory array.
7–9	Read reset register	`rden` is asserted. Memory is read. Since `outregce` and `rstreg` are both asserted, `dout` is reset to the `read_srval` value instead of providing the data that was just read.
8–10	Read	`rden` is asserted. Memory is read. Since `outregce` is asserted on the second cycle, `dout` provides the data that was just read.
9–11	Register reset without `outregce`	`rden` is asserted. Memory is read. On the second cycle, `rstreg` is asserted and `outregce` is de-asserted. The output data is either unchanged or is set to the `srval` as provided by the `read_srval` parameter depending on the value of the `regce_priority` parameter. • If `regce_priority` is `"rstreg"`, asserting `rstreg` resets the output register independent of the `outregce` signal • If `regce_priority` is `"regce"`, both `rstreg` and `outregce` must be asserted to reset the output register
10–12	–	`outregce` and `rstreg` are both asserted on the second cycle, then `dout` is reset to the `read_srval` value.

Memory Initialization

Initializing With Parameters

The data portion of initial memory contents may be defined by setting the 64 256-bit parameters `initd_00` through `initd_63`. The data memory is organized as little-endian with bit 0 mapped to bit zero of parameter `initd_00` and bit 16383 mapped to bit 255 of parameter `initd_63`.

When a BRAM is configured with port widths of 9 or 18 bits, the parity portion of the initial memory contents may be defined by setting the eight 256-bit parameters `initp_0` through `initp_7`. The parity memory is also organized as little-endian with the first parity bit location mapped to bit 0 of `initp_0` and the last parity bit mapped to the bit 255 of `initp_7`.

When a BRAM is configured with port widths of 5, 10 or 20 bits, the parity and extended parity portions of the initial memory contents may be defined by setting the eight 256-bit parameters `initp_0` through `initp_7` and the eight 256-bit parameters `initpx_0` through `initpx_7`. The parity and extended parity memories are both organized as little-endian with the first parity bit location mapped to bit 0 of `initp_0`/`initpx_0` and the last parity bit mapped to bit 255 of `initp_7`/`initpx_7`.

Initializing With a Memory Initialization File

Alternatively, a BRAM may be initialized with a memory file by setting the `mem_init_file` parameter to the path of a memory initialization file. The file format must be hexadecimal entries separated by white space, where the white space is defined by spaces or line separation. Each entry is a hexadecimal number of width equal to the maximum of the `read_width` and `write_width` parameters.

A number entry may contain underscore (_) characters among the digits (i.e., `"A234_4567_33"`). Commenting is allowed beginning with a double-slash (`//`). C-like commenting is also allowed with the comment placed between `"/*"` and `"*/"` characters. The memory is initialized starting with the first entry of the file initializing the memory array starting with address zero and moving upward.

If `mem_init_file` is defined, the BRAM is initialized with the values in the file referenced by the `mem_init_file` parameter. If `mem_init_file` is left at the default value of "", the initial contents are defined by the values of the parameters `initd_00` through `initd_63`, `initp_0` through `initp_7`, and `initpx_0` through `initpx_7`. If neither the memory initialization parameters nor the `mem_init_file` parameters are defined, the contents of the BRAM remain uninitialized and unknown until the memory locations are written.

The following tables show how the init values in the `read_initval` and `read_srval` parameters and the memory initialization file entries map to `dout`, `doutp`, and `doutpx`:

Table 222: ***srval and initval to Output Signals Mapping for datawidth = 1, 2, 4, 8, 16, and 32***

initval	datawidth					
	32	16	8	4	2	1
`init[31:16]`	`dout[31:16]`	–				
`init[15:8]`	`dout[15:8]`		–			
`init[7:4]`	`dout[7:4]`			–		
`init[3:2]`	`dout[3:2]`				–	
`init[1]`	`dout[1]`					–
`init[0]`	`dout[0]`					

Table 223: ***srval and initval to Output Signals Mapping for datawidth = 9, 18, and 36***

initval	datawidth		
	36	18	9
`init[35]`	`doutp[3]`		
`init[34:27]`	`dout[31:24]`	–	
`init[26]`	`doutp[2]`		
`init[25:18]`	`dout[23:16]`		
`init[17]`	`doutp[1]`		–
`init[16:9]`	`dout[15:8]`		
`init[8]`	`doutp[0]`		
`init[7:0]`	`dout[7:0]`		

Table 224: ***srval and initval to Output Signals Mapping for datawidth = 5, 10, 20, and 40***

initval	datawidth			
	40	20	10	5
`init[39]`	`doutpx[3]`			
`init[38:35]`	`dout[31:28]`			
`init[34]`	`doutp[3]`			
`init[33:30]`	`dout[27:24]`	–		
`init[29]`	`doutpx[2]`			
`init[28:25]`	`dout[23:20]`			
`init[24]`	`doutp[2]`			
`init[23:20]`	`dout[19:16]`			
`init[19]`	`doutpx[1]`			
`init[18:15]`	`dout[15:12]`		–	
`init[14]`	`doutp[1]`			
`init[13:10]`	`dout[11:8]`			
`init[9]`	`doutpx[0]`			–
`init[8:5]`	`dout[7:4]`			
`init[4]`	`doutp[0]`			
`init[3:0]`	`dout[3:0]`			

ECC Modes of Operation

There are four modes of operation for the ACX_BRAMSDP defined by the `encoder_enable` and `decoder_enable` parameters as shown in the following table. The `write_width` and `read_width` parameters must both be set to 40 to enable any of these modes.

Table 225: *ACX_BRAMSDP ECC Modes of Operation*

encoder_enable	decoder_enable	ECC Operation Mode
`1'b0`	`1'b0`	ECC encoder and decoder disabled, standard ACX_BRAMSDP operation available.
`1'b0`	`1'b1`	ECC decode-only mode.
`1'b1`	`1'b0`	ECC encode-only mode.
`1'b1`	`1’b1`	Normal ECC encode/decode mode.

ECC Encode/Decode Operation Mode

The ECC encode/decode operation mode utilizes both the ECC encoder and the ECC decoder. The 32-bit user data is written into the ACX_BRAMSDP via the `din[31:0]` inputs. The ECC encoder generates the 7-bit error correction syndrome and writes it into the memory array alongside the data word via the parity (`dinp`) and extended parity (`dinpx`) inputs. During read operations, the ECC decoder reads the 32-bit user data and the 7-bit syndrome data to generate an error correction mask. The ECC decoder corrects any single-bit error and only detects, but does not correct, any dual-bit error. If the ECC decoder detects a single-bit error, it automatically corrects the error and places the corrected data on the `dout[31:0]` pins and asserts the `sbit_error` output. The memory location containing the error is not corrected. If the ECC decoder detects a dual-bit error, it places the uncorrected data on the `dout[31:0]` pins and asserts the `dbit_error` output one cycle after the the data word is read.

ECC Encode-Only Operation Mode

The ECC encode-only operation has the ECC encoder enabled and the ECC decoder disabled. This mode allows writing 32 bits of data while having the 7-bit error correction syndrome automatically written to the (`{dinpx[2:0],dinp[3:0]}`) bits of the memory array during write operations. Read operations provide the 32-bit user data and the error syndrome without correcting the data. Encode-only mode can be used as a building block to have error correction for off-chip memories.

ECC Decode-Only Operation Mode

The ECC decode-only operation has the ECC encoder disabled and the ECC decoder enabled. This mode bypasses the ECC encoder and allows writing 40-bit data directly into the memory array during write operations. Read operations place the 7-bit error correction syndrome on the (`{doutpx[2:0],doutp[3:0]}`) bits. The ECC decoder corrects any single-bit error and detects, but does not correct, any dual-bit error. If the ECC decoder detects a single-bit error, it automatically corrects the error and places the corrected data on the `dout[31:0]` pins and asserts the `sbit_error` output. The memory location containing the error is not corrected. If the ECC decoder detects a dual-bit error, it places the uncorrected data on the `dout[31:0]` pins and asserts the `dbit_error` output one cycle after the the data word is read. Decode-only mode can be used as a building block to have error correction for off-chip memories.

Using ACX_BRAMSDP as a Read-Only Memory (ROM)

The ACX_BRAMSDP macro can be used as a read-only memory (ROM) by providing memory initialization data via a file or parameters (as described in Memory Initialization (see page 304)), and tying the `wren` signal to its de-asserted value. All signals on the read-side of the ACX_BRAMSDP operate as described above. This configuration allows reading from the memory, but not writing to it.

Create an Instance

To create memories within a design, there are three available methods:

1. Infer the memory – this method provides the greatest code portability and is the recommended approach. An example follows of an ACX_BRAMSDP inference.
2. Directly instantiated – this method gives access to the full feature set of the memory. However, any code is less portable to other technology nodes. See Instantiation Template (see page 313).
3. ACE BRAM IP generator – use this tool to create the appropriate memory structure. Refer to the *ACE User Guide* (UG070) for details.

Inference Template

ACX_BRAMSDP Symmetric Inference

```
//---------------------------------------------------------------------------------
//
// Copyright (c) 2022  Achronix Semiconductor Corp.
// All Rights Reserved.
//
//
// This software constitutes an unpublished work and contains
// valuable proprietary information and trade secrets belonging
// to Achronix Semiconductor Corp.
//
// This software may not be used, copied, distributed or disclosed
// without specific prior written authorization from
// Achronix Semiconductor Corp.
//
// The copyright notice above does not evidence any actual or intended
// publication of such software.
//
//

//---------------------------------------------------------------------------------
// Design: BRAMSDP Symmetric Inference
//         An example to infer a symmetric BRAMSDP in Speedcore designs
//---------------------------------------------------------------------------------

`timescale 1ps / 1ps

module bram_sdp_symmetric
#(
    parameter       ADDR_WIDTH      = 11,
    parameter       DATA_WIDTH      = 9,
    parameter       INIT_FILE_NAME  = ""
)
(
    // Clocks and resets
    input  wire                     clk,

    // Enables
    input  wire                     we,

    // Address and data
    input  wire [ADDR_WIDTH-1:0]    wr_addr,
    input  wire [ADDR_WIDTH-1:0]    rd_addr,
    input  wire [DATA_WIDTH-1:0]    wr_data,

    // Output
    output reg  [DATA_WIDTH-1:0]    rd_data
);

localparam DATA_DEPTH = (2 ** ADDR_WIDTH);

reg [DATA_WIDTH-1:0]  mem_ram[DATA_DEPTH-1:0] /* synthesis syn_ramstyle = "block_ram"
"no_rw_check" */;

initial begin
    if (INIT_FILE_NAME != "")
        $readmemh(INIT_FILE_NAME, mem_ram);
end

// synthesis synthesis_off
reg addr_collision;
assign addr_collision = (rd_addr == wr_addr);
// synthesis synthesis_on

always @(posedge clk) begin
    // synthesis synthesis_off
    if (addr_collision && we)
        rd_data <= {DATA_WIDTH{1'bx}};
    else
    // synthesis synthesis_on
        rd_data <= mem_ram[rd_addr];

    if(we)
        mem_ram[wr_addr] <= wr_data;
end

endmodule : bram_sdp_symmetric
```

ACX_BRAMSDP Inference

```
//-------------------------------------------------------------------------------
//
// Copyright (c) 2022  Achronix Semiconductor Corp.
// All Rights Reserved.
//
//
// This software constitutes an unpublished work and contains
// valuable proprietary information and trade secrets belonging
// to Achronix Semiconductor Corp.
//
// This software may not be used, copied, distributed or disclosed
// without specific prior written authorization from
// Achronix Semiconductor Corp.
```

```
//
// The copyright notice above does not evidence any actual or intended
// publication of such software.
//
//

//------------------------------------------------------------------------------
// Design: BRAMSDP Inference
//         An example to infer a simple dual-port BRAM in Speedcore designs
//------------------------------------------------------------------------------

`timescale 1ps / 1ps

module bram_sdp
#(
    parameter       WRITE_ADDR_WIDTH = 9,
    parameter       READ_ADDR_WIDTH  = 11,
    parameter       WRITE_DATA_WIDTH = 32,
    parameter       READ_DATA_WIDTH  = 8,
    parameter       INIT_FILE_NAME   = ""
)
(
    // Clocks and resets
    input  wire                          clk,

    // Enables
    input  wire                          we,

    // Address and data
    input  wire [WRITE_ADDR_WIDTH-1:0]    wr_addr,
    input  wire [READ_ADDR_WIDTH-1:0]     rd_addr,
    input  wire [WRITE_DATA_WIDTH-1:0]    wr_data,

    // Output
    output reg  [READ_DATA_WIDTH-1:0]     rd_data
);

`define min(a,b) {(a) < (b) ? (a) : (b)}
`define max(a,b) {(a) > (b) ? (a) : (b)}
`define clamp(a, val, b) {((val) < (a)) ? (a) : (((val) > (b)) ? (b) : (val))}

localparam MIN_DATA_WIDTH = `min(WRITE_DATA_WIDTH, READ_DATA_WIDTH);
localparam MAX_DATA_WIDTH = `max(WRITE_DATA_WIDTH, READ_DATA_WIDTH);

localparam WIDTH_RATIO = MAX_DATA_WIDTH / MIN_DATA_WIDTH;

localparam WRITE_DATA_MULT = (WRITE_DATA_WIDTH < READ_DATA_WIDTH) ? 1 : WIDTH_RATIO;
localparam READ_DATA_MULT = (READ_DATA_WIDTH < WRITE_DATA_WIDTH) ? 1 : WIDTH_RATIO;

localparam WRITE_DEPTH = (2 ** WRITE_ADDR_WIDTH) * WRITE_DATA_MULT;
localparam READ_DEPTH = (2 ** READ_ADDR_WIDTH) * READ_DATA_MULT;

localparam MAX_DEPTH = `max(WRITE_DEPTH, READ_DEPTH);

reg [MIN_DATA_WIDTH-1:0]  mem_ram[MAX_DEPTH-1:0] /* synthesis syn_ramstyle = "block_ram"
"no_rw_check" */;

initial begin
    if (INIT_FILE_NAME != "")
```

```
        $readmemh(INIT_FILE_NAME, mem_ram);
end

// Generate bitmask for overlapping wr_addr bit(s) in the rd_addr word(s). We
// assign x to colliding bits and 0 otherwise, then apply the mask with xor.
// Note that A ^ x = x and A ^ 0 = A

// synthesis synthesis_off
reg [(MIN_DATA_WIDTH*MAX_DEPTH)-1:0] min_wr_bit_addr;
reg [(MIN_DATA_WIDTH*MAX_DEPTH)-1:0] max_wr_bit_addr;

reg [(MIN_DATA_WIDTH*MAX_DEPTH)-1:0] min_rd_bit_addr;
reg [(MIN_DATA_WIDTH*MAX_DEPTH)-1:0] max_rd_bit_addr;

reg [READ_DATA_WIDTH-1:0] read_collision_mask;

assign min_wr_bit_addr = wr_addr * WRITE_DATA_WIDTH;
assign max_wr_bit_addr = (wr_addr + 1) * WRITE_DATA_WIDTH - 1;

assign min_rd_bit_addr = rd_addr * READ_DATA_WIDTH;
assign max_rd_bit_addr = (rd_addr + 1) * READ_DATA_WIDTH - 1;

localparam PADDED_READ_DATA_WIDTH = READ_DATA_WIDTH + 2;

reg [PADDED_READ_DATA_WIDTH-1:0] padded_read_collision_mask;
reg [$clog2(PADDED_READ_DATA_WIDTH)-1:0] min_padded_read_collision_mask_bit_addr;
reg [$clog2(PADDED_READ_DATA_WIDTH)-1:0] max_padded_read_collision_mask_bit_addr;

assign min_padded_read_collision_mask_bit_addr = `clamp(min_rd_bit_addr, min_wr_bit_addr+1,
max_rd_bit_addr+2) - min_rd_bit_addr;
assign max_padded_read_collision_mask_bit_addr = `clamp(min_rd_bit_addr, max_wr_bit_addr+1,
max_rd_bit_addr+2) - min_rd_bit_addr;

assign padded_read_collision_mask = ((2 ** (max_padded_read_collision_mask_bit_addr -
min_padded_read_collision_mask_bit_addr + 1))-1) << min_padded_read_collision_mask_bit_addr;
assign read_collision_mask = {READ_DATA_WIDTH{1'bx}} & (padded_read_collision_mask
[READ_DATA_WIDTH:1] & {READ_DATA_WIDTH{we}});
// synthesis synthesis_on

genvar i;
generate
    if (WRITE_DATA_MULT <= 1) begin
        always @(posedge clk)
            if(we)
                mem_ram[wr_addr] <= wr_data;
    end
    else begin
        for (i=0; i < WRITE_DATA_MULT; i=i+1) begin : gen_write
            localparam write_stride = MIN_DATA_WIDTH*i;
            always @(posedge clk)
                if(we)
                    mem_ram[{wr_addr, i[$clog2(WRITE_DATA_MULT)-1:0]}] <= wr_data[(write_stride)+:
MIN_DATA_WIDTH];
        end
    end
endgenerate

genvar j;
generate
```

```
    if (READ_DATA_MULT <= 1) begin
        always @(posedge clk)
            // synthesis synthesis_off
            if (1)
                rd_data <= (mem_ram[rd_addr] ^ read_collision_mask);
            else
            // synthesis synthesis_on
                rd_data <= mem_ram[rd_addr];
    end
    else begin
        for (j=0; j < READ_DATA_MULT; j=j+1) begin : gen_read
            localparam read_stride = MIN_DATA_WIDTH*j;
            always @(posedge clk)
                // synthesis synthesis_off
                if (1)
                    rd_data[(read_stride)+:MIN_DATA_WIDTH] <= (mem_ram[{rd_addr, j[$clog2
(READ_DATA_MULT)-1:0]}] ^ read_collision_mask[(read_stride)+:MIN_DATA_WIDTH]);
                else
                // synthesis synthesis_on
                    rd_data[(read_stride)+:MIN_DATA_WIDTH] <= mem_ram[{rd_addr, j[$clog2
(READ_DATA_MULT)-1:0]}];
        end
    end
endgenerate

endmodule : bram_sdp
```

Instantiation Template

Verilog

```
ACX_BRAMSDP #(
.read_width(40),
.write_width(40),
.write_clock_polarity("rise"),
.en_out_reg(1'b0),
.regce_priority("rstreg"),
.write_peval(1'b1),
.read_peval(1'b1),
.reg_rstval(1'b1),
.latch_rstval(1'b1),
.read_initval(40'h0),
.read_srval(40'h0),
.read_clock_polarity("rise"),
.encoder_enable(1'b0),
.decoder_enable(1'b0),
.mem_init_file(""),
.initd_00(256'h0),
.initd_01(256'h0),
.initd_02(256'h0),
.initd_03(256'h0),
.initd_04(256'h0),
.initd_05(256'h0),
.initd_06(256'h0),
.initd_07(256'h0),
.initd_08(256'h0),
.initd_09(256'h0),
.initd_10(256'h0),
.initd_11(256'h0),
.initd_12(256'h0),
.initd_13(256'h0),
.initd_14(256'h0),
.initd_15(256'h0),
.initd_16(256'h0),
.initd_17(256'h0),
.initd_18(256'h0),
.initd_19(256'h0),
.initd_20(256'h0),
.initd_21(256'h0),
.initd_22(256'h0),
.initd_23(256'h0),
.initd_24(256'h0),
.initd_25(256'h0),
.initd_26(256'h0),
.initd_27(256'h0),
.initd_28(256'h0),
.initd_29(256'h0),
.initd_30(256'h0),
.initd_31(256'h0),
.initd_32(256'h0),
.initd_33(256'h0),
.initd_34(256'h0),
.initd_35(256'h0),
.initd_36(256'h0),
.initd_37(256'h0),
.initd_38(256'h0),
.initd_39(256'h0),
.initd_40(256'h0),
.initd_41(256'h0),
.initd_42(256'h0),
.initd_43(256'h0),
.initd_44(256'h0),
.initd_45(256'h0),
.initd_46(256'h0),
.initd_47(256'h0),
.initd_48(256'h0),
.initd_49(256'h0),
.initd_50(256'h0),
.initd_51(256'h0),
.initd_52(256'h0),
.initd_53(256'h0),
.initd_54(256'h0),
.initd_55(256'h0),
.initd_56(256'h0),
.initd_57(256'h0),
.initd_58(256'h0),
.initd_59(256'h0),
.initd_60(256'h0),
.initd_61(256'h0),
.initd_62(256'h0),
.initd_63(256'h0),
.initp_0(256'h0),
.initp_1(256'h0),
.initp_2(256'h0),
.initp_3(256'h0),
.initp_4(256'h0),
.initp_5(256'h0),
.initp_6(256'h0),
.initp_7(256'h0),
.initpx_0(256'h0),
.initpx_1(256'h0),
.initpx_2(256'h0),
.initpx_3(256'h0),
.initpx_4(256'h0),
.initpx_5(256'h0),
.initpx_6(256'h0),
.initpx_7(256'h0)
)
instance_name
(
.wraddr(user_wraddr),
.din(user_din),
.dinp(user_dinp),
.dinpx(user_dinpx),
.we(user_we),
.wren(user_wren),
.rstlatch(user_rstlatch),
.rstreg(user_rstreg),
.outregce(user_outregce),
.wrclk(user_wrclk),
.dout(user_dout),
.doutp(user_doutp),
.doutpx(user_doutpx),
.sbit_error(user_sbit_error),
.dbit_error(user_dbit_error),
.rdaddr(user_rdaddr),
.rdclk(user_rdclk),
.rden(user_rden)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
------------- DONE ACHRONIX LIBRARY ---------
-- Component Instantiation
ACX_BRAMSDP_instance_name : ACX_BRAMSDP
generic map (
read_width => 40,
write_width => 40,
write_clock_polarity => "rise",
en_out_reg => 0,
regce_priority => "rstreg",
write_peval => 1,
read_peval => 1,
reg_rstval => 1,
latch_rstval => 1,
read_initval => X"0000000000",
read_srval => X"0000000000",
write_clock_polarity => "rise",
encoder_enable => 0,
decoder_enable => 0,
mem_init_file => "",
initd_00 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_01 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_02 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_10 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_11 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_12 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_13 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_14 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_15 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_16 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_17 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_18 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_19 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_20 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_21 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_22 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_23 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_24 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_25 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_26 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_27 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_28 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_29 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_30 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_31 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_32 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_33 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_34 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_35 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_36 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_37 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_38 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_39 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_40 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_41 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_42 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_43 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_44 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_45 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_46 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_47 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_48 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_49 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_50 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_51 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_52 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_53 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_54 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_55 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_56 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_57 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_58 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_59 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_60 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_61 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_62 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_63 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_0 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_1 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_2 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_3 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_4 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_5 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_6 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_7 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_0 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_1 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_2 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_3 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_4 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_5 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_6 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_7 => X"0000000000000000000000000000000000000000000000000000000000000000")
port map (
wraddr => user_wraddr,
din => user_din,
dinp => user_dinp,
dinpx => user_dinpx,
we => user_we,
wren => user_wren,
rstlatch => user_rstlatch,
rstreg => user_rstreg,
outregce => user_outregce,
wrclk => user_wrclk,
dout => user_dout,
doutp => user_doutp,
doutpx => user_doutpx,
sbit_error => user_sbit_error,
dbit_error => user_dbit_error,
rdaddr => user_rdaddr,
rdclk => user_rdclk,
rden => user_rden
);
```

ACX_BRAMTDP (20-kb True Dual-Port Memory)

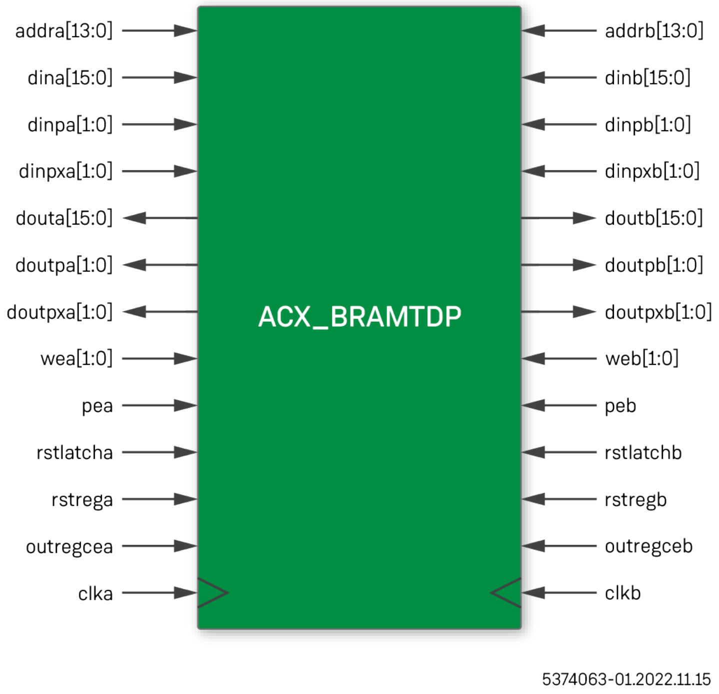


Figure 119: *20-kb True Dual-Port Memory*

The block RAM (ACX_BRAMTDP) implements a 20-kb true-dual-ported (TDP) memory block where each port can be independently configured with respect to size and function. The BRAM can be configured as a single-port (one R/W port), dual-port (two R/W ports with independent clocks), or ROM memory. Each memory port can be configured as follows:

- 1k × 20
- 1k × 18
- 1k × 16
- 2k × 10
- 2k × 9
- 2k × 8
- 4k × 5
- 4k × 4
- 8k × 2
- 16k × 1

The read and write operations are both synchronous. For higher performance operation, an additional output register can be enabled. Enabling the output register requires an additional cycle of read latency. Write Enable (wea/web) controls provide 8-bit, 9-bit, or 10-bit write granularity for port widths above 16 bits.

The initial value of the memory contents may be user specified from either parameters or a memory initialization file. The initial/reset values of the output registers may also be user specified. The `porta_write_mode/portb_write_mode` parameters define the behavior of the output data port during a write operation. When `porta_write_mode/portb_write_mode` is set to `"write_first"`, the `douta/doutb` port gets the value that was present on the `dina/dinb` port during each write operation. Setting `porta_write_mode/portb_write_mode` to `"no_change"` keeps the `douta/doutb` port unchanged during a write operation.

Note

The `"write_first"` mode requires that both the read and write ports of the same side must be set to the same width.

Conflict arises when the same memory cell is accessed by both ports within a narrow window and one or both ports are writing to memory. If this condition occurs, the contents of the memory for the colliding address is undefined, but no damage occurs to the memory.

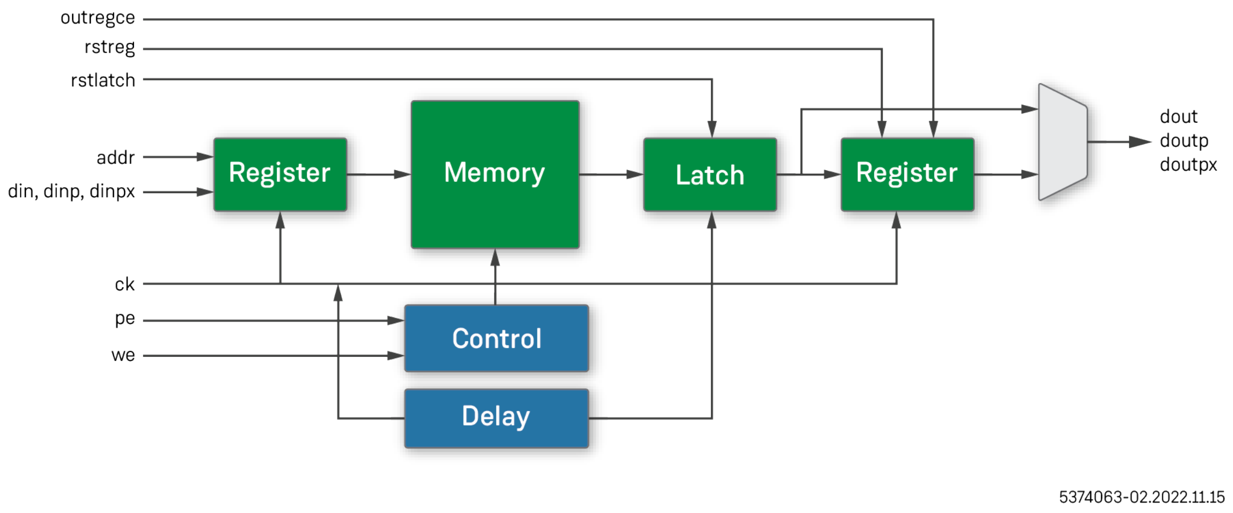


Figure 120: ***ACX_BRAMTDP Block Diagram (Per Port)***

Table 226: ***ACX_BRAMTDP Pin Descriptions***

Name	Type	Description
`clka`, `clkb`	Input	Port A (B) clock input. Read and write operations are fully synchronous to the active edge of `clka` (`clkb`) when the `pea` (`peb`) signal is high. The active edge of `clka` (`clkb`) is determined by the `porta_clock_polarity` (`portb_clock_polarity`) parameter.
`pea`, `peb`	Input	Port A (B) port enable. The `pea` (`peb`) signal must be asserted to perform a read or write operation. The `porta_peval` and `portb_peval` parameters determine whether they are active-high (default) or active-low.
`addra[13:0]`, `addrb[13:0]`	Input	Port A (B) address input. The `addra` (`addrb`) signal determines which memory location is being written to or read from. When the port width is greater than 1, the low-order address bits must be tied to 0. See Table: BRAM Address Bus Mapping (Per Port). (see page 327)
`wea[1:0]`, `web[1:0]`	Input	Port A (B) write enable. Each bit of a port write enable input enables an 8-bit, 9-bit, or 10-bit byte to be written to memory, depending on the `porta_write_width` (`portb_write_width`) parameter value. A write to memory occurs when both the corresponding `wea` (`web`) bit is high and the port enable `pea` (`peb`) signal is active. If `wea` (`web`) is inactive while the `pea` (`peb`) is active, a read operation occurs, and the output is updated with the contents of the addressed memory cells. For data widths of 16 or larger, `we[1]` corresponds to `din[15:8]`, `dinp[1]`, and `dinpx[1]`, while `we[0]` corresponds to `din[7:0]`, `dinp[0]`, and `dinpx[0]`. For data widths less than 16, `we[0]` and `we[1]` must be the same.
`dina[15:0]`, `dinb[15:0]`	Input	Port A (B) data input.
`dinpa[1:0]`, `dinpb[1:0]`	Input	Port A (B) additional data input. Used to extend `dina`/`dinb`.
`dinpxa[1:0]`, `dinpxb[1:0]`	Input	Port A (B) extended data input. Used to further extend `dina`/`dinb`.
`outregcea`, `outregceb`	Input	Port A (B) output register clock enable (active-high). When the `porta_en_out_reg` (`portb_en_out_reg`) parameter is set, de-asserting the `outregcea` (`outregceb`) signal causes the BRAM output to keep the `douta`, `doutpa`, and `doutpxa` (`doutb`, `doutpb`, and `doutpxb`) signals unchanged, independent of a read or write operation.
`rstlatcha`, `rstlatchb`	Input	Port A (B) output latch synchronous reset. When `rstlatcha` (`rstlatchab`) is asserted, the value of `{porta_read_width{1'b0}}` (`{portb_read_width {1'b0}}`) is written to the port A (B) output latch upon the next active edge of `clka` (`clkb`). The `porta_latch_rstval` and `portb_latch_rstval` parameters determine whether they are active-high (default) or active-low.
		Port A (B) output register reset. The `porta_sr_assertion_reg` (`portb_sr_assertion_reg`) parameter determines if the reset is synchronous (default) or asynchronous, and the `porta_reg_rstval` (`portb_reg_rstval`)

Name	Type	Description
`rstrega`, `rstregb`	Input	parameter determines if the reset is active-high (default) or active-low. When reset is asserted, the port A (B) output register is assigned the value of the `porta_srval` (`portb_srval`) parameter. The priority of `rstrega` (`rstregb`) relative to the clock enable input `outregcea` (`outregceb`) is determined by the value of the `porta_regce_priority` (`portb_regce_priority`) parameter.
`douta[15:0]`, `doutb[15:0]`	Output	Port A (B) data output. During read operations, the `douta` (`doutb`) outputs are updated with the memory contents addressed by `addra` (`addrb`) if the `pea` (`peb`) port enable is active and `wea` (`web`) inputs are low. For write operations, the behavior of the `douta` (`doutb`) outputs depends on the `porta_write_mode` (`portb_write_mode`) parameter.
`doutpa[1:0]`, `doutpb[1:0]`	Output	Port A (B) addtional data output. The port A (B) `doutpa` (`doutpb`) output behaves the same as outputs `douta` (`doutb`) and is used when the `porta_read_width` (`portb_read_width`) is set to 5, 9, 10, 18, or 20 bits.
`doutpxa[1:0]`, `doutpxb[1:0]`	Output	Port A (B) extended data output. The port A (B) `doutpxa` (`doutpxb`) extended data output behaves the same as outputs `douta` (`doutb`) and is used when the `porta_read_width` (`portb_read_width`) is set to 10 or 20 bits.

Table 227: ***ACX_BRAMTDP Parameters***

Parameter	Defined Values	Default Value	Description
`porta_read_width`, `portb_read_width`	1, 2, 4, 5, 8, 9, 10, 16, 18, 20	20	Sets the read width for port A (B).
`porta_write_width`, `portb_write_width`	1, 2, 4, 5, 8, 9, 10, 16, 18, 20	20	Sets the write width for port A (B). The read port width may vary from the write port width, but it must be within the allowable combinations defined in Memory Organization and Data I/O Pin Assignments (see page 325).
`porta_write_mode`, `portb_write_mode`	`"write_first"`, `"read_first"`(1), `"no_change"`	`"write_first"`	Defines the response of the port A (B) output to write operations. The output in a write response appears at the `douta` (`doutb`) output with the same timing as a read operation. The modes are: • `"write_first"` – the data present on the `dina` (`dinb`) input during the write operation appears on the output of port A (B) for words in which the write enable bit in `wena` (`wenb`) is asserted. The output data for words in which the write enable bit is de-asserted is undefined. This mode is only supported when the `porta_read_width` (`portb_read_width`) and `porta_write_width` (`portb_write_width`) parameters of the port are the same. • `"read_first"` - The data previously stored at the specified write address appears on the output of port A (B) • `"no_change"` – `douta` (`doutb`) remains unchanged during write operations
`porta_clock_polarity`, `portb_clock_polarity`	`"rise"`, `"fall"`	`"rise"`	Sets the active edge of the port A (B) clock.
`porta_peval`, `portb_peval`	`1'b0`, `1'b1`	`1'b1`	Defines the active level of the `pea`(`peb`) port enable input. A value of `1'b0` sets active low, while `1'b1` sets active high.
`porta_latch_rstval`, `portb_latch_rstval`	`1'b0`, `1'b1`	`1'b1`	Defines the active level of the `rstlatcha` (`rstlatchb`) input. A value of `1'b0` sets active low, while `1'b1` sets active high.
`porta_en_out_reg`, `portb_en_out_reg`	`1'b0`, `1'b1`	`1'b0`	Determines whether the port A (B) output register is enabled. A value of `1'b0` disables the output register and results in a read latency of one cycle, while `1'b1` enables the output register and results in a read latency of two cycles.
`porta_reg_rstval`, `portb_reg_rstval`	`1'b0`, `1'b1`	`1'b1`	Defines the active level of the `rstrega` (`rstregb`) input. A value of `1'b0` sets active low, while a value of `1'b1` sets active high.
`porta_regce_priority`, `portb_regce_priority`	`"rstreg"`, `"regce"`	`"rstreg"`	Defines the priority of the `outregcea` (`outregceb`) clock enable input relative to the `rstrega` (`rstregb`) reset: • `"rstreg"` – allows the port A (B) output register to be reset by asserting `rstrega` (`rstregb`) without requiring `outregcea` (`outregceb`) to be asserted • `"regce"` – allows the port A (B) output register to be reset by only asserting both `rstrega` (`rstregb`) and `outregcea` (`outregceb`)
`porta_initval`, `portb_initval`	20-bit hex number	`20'h0`	Defines the power-up default value of the data on the output of port A (B) latch and output register, if enabled. The 20-bit parameter assignment is dependent on the `porta_read_width` (`portb_read_width`) parameter as shown in Table: initval, srval, and meminit File Mapping to Output Signals (see page 337) .
`porta_srval`, `portb_srval`	20-bit hex number	`20'h0`	Defines the reset value of the data on the output port A (B) latch and output register, if enabled, when `rstlatcha` (`rstlatchb`) and/or `rstrega` (`rstregb`) is asserted. The 20-bit parameter assignment is dependent on the `porta_read_width` (`portb_read_width`) parameter as shown in Table: initval, srval, and meminit File Mapping to Output Signals (see page 337).

Parameter	Defined Values	Default Value	Description
`porta_sr_assertion_reg`, `portb_sr_assertion_reg`	`"clocked"`, `"unclocked"`	`"clocked"`	Sets whether the assertion of the reset of the port A (B) output register is synchronous or asynchronous with respect to the `clka` (`clkb`) input. A value of `"clocked"` corresponds to a synchronous reset where the port A (B) output register is reset on the next rising edge of the clock if `rstrega` (`rstregb`) is asserted. A value of `"unclocked"` corresponds to an asynchronous reset where the port A (B) output register is reset immediately following the assertion of the `rstrega` (`rstregb`) input.
`mem_init_file`	<path to HEX file>	`""`	Provides a mechanism to set the initial contents of the BRAM memory. If defined, the BRAM is initialized with the values defined in the file pointed to by the parameter according to the format defined in Memory Initialization (see page 336). If left at the default value, the initial contents are defined by the values of the `initd_00–initd_63`, `initp_0–initp_7`, and the `initpx_0–initpx_7` parameters. If the memory initialization parameters and the `mem_init_file` parameters are not defined, the contents of the BRAM remain uninitialized.
`initd_00–initd_63`	256-bit hex number	`256'hx`	Define the initial contents of the memory associated with `douta[15:0]` and `doutb[15:0]`. Each 256-bit parameter is associated with the BRAM as defined in Memory Initialization (see page 336).
`initp_0–initp_7`	256 bit hex number	`256’hx`	Define the initial contents of the memory associated with `doutpa[1:0]` and `doutpb[1:0]`. Each 256-bit parameter is associated with the BRAM is as defined in Memory Initialization (see page 336).
`initpx_0–initpx_7`	256 bit hex number	`256'hx`	Define the initial contents of the memory associated with `doutpxa[1:0]` and `doutpxb[1:0]`. Each 256-bit parameter is associated with the BRAM is defined in Memory Initialization (see page 336).

Table Notes

1. ACX_BRAMTDP supports `"read-first"` mode only when directly instantiating the ACX_BRAMTDP primitive. Synthesis is not able to infer a "read-first" mode from RTL, and this mode is not supported through the IP configuration GUI. If a memory with this behavior is described by behavioral RTL, a warning is issued during synthesis, and a register file is synthesized.

Note

The ACE BRAM IP Configuration GUI and ACX_BRAM_GEN macros only support a single-bit write enable (we) for the entire data word. Byte-wise write enables are not supported via the GUI or in Verilog macros. Non-zero reset values are similarly not supported. Access to the full capabilities of the BRAM is available by instantiating the ACX_BRAMTDP primitive directly.

Memory Organization and Data I/O Pin Assignments

The BRAM supports memory widths from one bit to twenty bits. The width of the `dina` (`dinb`) data input is determined by the `porta_write_width` (`portb_write_width`) parameter while the width of the `douta` (`doutb`) data output is determined by the `porta_read_width` (`portb_read_width`) parameter. Port A width may be different than the port B width, and the width of each read port may be set differently from the width of each write port. The supported port width combinations are as described in the following table. "X" indicates a supported configuration.

Table 228: ***Supported Port Width Combinations***

Port A Read Width	Port A Write Width, Port B Read Width, Port B Write Width									
	1k × 20	2k × 10	4k × 5	1k × 18	2k × 9	1k × 16	2k × 8	4k × 4	8k × 2	16k × 1
1k × 20	X	X	X	–	–	–	–	–	–	–
2k × 10	X	X	X	–	–	–	–	–	–	–
4k × 5	X	X	X	–	–	–	–	–	–	–
1k × 18	–	–	–	X	X	–	–	–	–	–
2k × 9	–	–	–	X	X	–	–	–	–	–
1k × 16	–	–	–	–	–	X	X	X	X	X
2k × 8	–	–	–	–	–	X	X	X	X	X
4k × 4	–	–	–	–	–	X	X	X	X	X
8k × 2	–	–	–	–	–	X	X	X	X	X
16k × 1	–	–	–	–	–	X	X	X	X	X

Data Widths Using Extended Data Interfaces

The ACX_BRAMTDP memory has three buses for both data in and data out:

1. The `din` and `dout` interfaces.
2. The `dinp` and `dinpx` extended data interfaces.
3. The `doutp` and `doutpx` extended data interfaces

The extended interfaces are used to support the wide range of data bus widths shown in Supported Port Width Combinations (see page 325). The extended interfaces are assigned to the respective data buses as shown in the following table.

Table 229: ***Extended Data Interface Assignment, (Per Port)***

Data Width	dinpx/doutpx	dinp/doutp	din/dout
20	`{data[19], data[9]}`	`{data[14], data[4]}`	`{data[18:15], data[13:10], data[8:5], data[3:0]}`
18	–	`{data[17], data[8]}`	`{data[16:9], data[7:0]}`
16	–	–	`{data[15:0]}`
10	`{1'b0, data[9]}`	`{1'b0, data[4]}`	`{8'h0, data[8:5], data[3:0]}`
9	–	`{1'b0, data[8]}`	`{8'h0, data[7:0]}`
8	–	–	`{8'h0, data[7:0]}`
5	–	`{1'b0, data[4]}`	`{12'h0, data[3:0]}`
4	–	–	`{12'h0, data[3:0]}`
2	–	–	`{14'h0, data[1:0]}`
1	–	–	`{15'h0, data[0]}`

Caution!

Pay close attention to non power-of-two sized data widths and how the data bits are assigned.

Address Bus Mapping

When the port width is greater than 1, the memory address must be left-justified to the most-significant bit (MSB) of the `addra` (`addrb`) input, meaning that the low-order address bits must be tied to 0. The following table shows the address bits that must be tied to zero for the various memory organization options.

Table 230: *ACX_BRAMTDP Address Bus Mapping (Per Port)*

Memory Organization	Used Address bits	Tied to 0 Address bits
1k × 20	13:4	3:0
1k × 18	13:4	3:0
1k × 16	13:4	3:0
2k × 10	13:3	2:0
2k × 9	13:3	2:0
2k × 8	13:3	2:0
4k × 5	13:2	1:0
4k × 4	13:2	1:0
8k × 2	13:1	0
16k × 1	13:0	–

Warning

A common error is to assign the address bus incorrectly justified. It must be assigned *left-justified*, not right-justified.

Read and Write Operations

Timing Options

The BRAM has two options for interface timing, controlled by the `porta_en_out_reg` (`portb_en_out_reg`) parameter:

- Latched mode – when `porta_en_out_reg` (`portb_en_out_reg`) is `1'b0`, the port is in latched mode. In this mode, the read address is registered, and the stored data is latched into the output latches on the following clock cycle, providing a read operation with one cycle of latency.
- Registered mode – when `porta_en_out_reg` (`portb_en_out_reg`) is `1'b1`, the port is in registered mode. In this mode, there is an additional register after the latch, supporting higher-frequency designs and providing a read operation with two cycles of latency.

Read Operation

Read operations are signaled by driving the `addra` (`addrb`) signal with the address to be read, asserting the `pea` (`peb`) port enable signal and not asserting the `wea` (`web`) write enable signal. The requested read data arrives on the `douta` (`doutb`), `doutpa` (`doutpb`), and `doutpxa` (`doutpxb`) signals on the following clock cycle or the cycle after, depending on the `porta_en_out_reg` (`portb_en_out_reg`) parameter.

Write Operation

Write operations are signaled by driving the `dina` (`dinb`), `dinpa` (`dinpb`), and `dinpxa` (`dinpxb`) signals with the data to be written, `addra` (`addrb`) with the address to write to, asserting the `pea` (`peb`) port enable signal, and asserting the `wea` (`web`) write enable signal. The input data is stored in the memory array at the indicated address on the next active clock edge.

There are two options for the behavior of the data output signals during a write operation, as controlled by the `porta_write_mode` (`portb_write_mode`) parameter:

- `"write_first"` – the write data is reflected on the `douta` (`doutb`), `doutpa` (`doutpb`), and `doutpxa` (`doutpxb`) signals.
- `"no_change"` – the `douta` (`doutb`), `doutpa` (`doutpb`), and `doutpxa` (`doutpxb`) signals remain unchanged during a write operation on port A (B).

Table 231: ***Latched Mode BRAM Output Function (Rising-Edge Clock and Active-High Port Enable)***

Operation	porta_write_mode (portb_write_mode)	clka (clkb)	rstlatcha (rstlatchb)	pea (peb)	wea (web)	douta (doutb)
Hold	`"write_first"` or `"no_change"`	X	X	X	X	Hold previous value.
Reset Latch	`"write_first"` or `"no_change"`	↑	1	X	X	`porta_srval` (`portb_srval`)
Hold	`"write_first"` or `"no_change"`	↑	0	0	X	Hold previous value.
Read	`"write_first"` or `"no_change"`	↑	0	1	0	`mem[addra]` (`mem[addrb]`)
Write	`"write_first"`	↑	0	1	1	`dina` (`dinb`)
Write	`"no_change"`	↑	0	1	1	Hold previous value.

Table 232: ***Registered Mode BRAM Output Function (Rising-Edge Clock and Active-high Port Enable)***

Operation	regce_priority	rdclk	rstreg	outregce	dout
Hold	–	X	X	X	`douta_previous` (`doutb_previous`)
Reset Output	`"rstreg"`	↑	1	X	`porta_srval` (`portb_srval`)
Reset Output	`"regce"`	↑	1	1	`porta_srval` (`portb_srval`)
Hold	`"regce"`	↑	X	0	`douta_previous` (`doutb_previous`)
Hold	`"rstreg"` or `"regce"`	↑	0	0	`douta_previous` (`doutb_previous`)
Update Output	`"rstreg"` or `"regce"`	↑	0	1	Latch output.

Table 233: ***Port a/b Registered Mode BRAM Output Function (Rising-Edge Clock and Active-high Port Enable)***

Operation	porta_regce_priority (portb_regce_priority)	rstrega (rstregb)	outregcea (outregceb)	clka (clkb)	douta (doutb)
Hold	X	X	X	X	`douta_previous` (`doutb_previous`)
Hold	`rstreg`	0	0	↑	`douta_previous` (`doutb_previous`)
Update output	`rstreg`	0	1	↑	`latcha_output` (`latchb_output`)
Reset output	`rstreg`	1	X	↑	`porta_srval` (`portb_srval`)
Hold	`regce`	X	0	↑	`douta_previous` (`doutb_previous`)
Update output	`regce`	0	1	↑	`latcha_output` (`latchb_output`)
Reset output	`regce`	1	1	↑	`porta_srval` (`portb_srval`)

Simultaneous Memory Operations

Memory operations may be performed simultaneously from both sides of the memory; however, there is a restriction with memory collisions. A memory collision is defined as the condition where both of the ports access the same memory location within the same clock cycle (both ports connected to the same clock), or within a fixed time window (if each port is connected to a different clock). Simultaneous read operations to the same memory location by both ports is allowed and produces valid data on each of the ports. If one of the ports is writing an address while the other port is reading the same address, the write operation occurs, but the read data is invalid. The data may be reliably read on the next cycle if there is no longer a write collision. If both ports write the same memory location(s) at the same time, the memory contents for that memory address are invalid. While simultaneously writing the same address from both ports invalidate the data, no damage to the hardware occurs.

Note

For the special case of the BRAM having both ports configured for `"write_first"` mode, a write-write collision corrupts the memory contents, but the correct data is seen at both output ports. In this case, the data corruption is not noticed by the circuit until the the corrupted memory location is later reread.

Timing Diagrams

The timing diagrams for the two `porta_en_out_reg` (`portb_en_out_reg`) parameter values follow. The first timing diagram illustrates the behavior of a ACX_BRAMTDP port with the output register disabled.

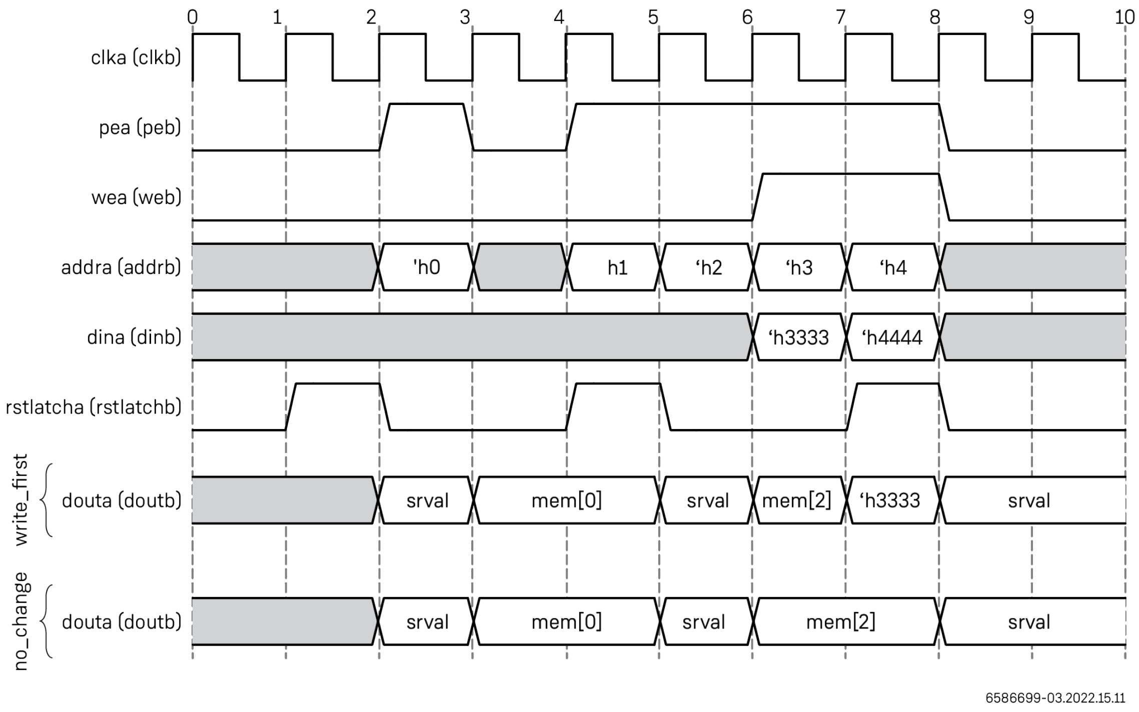


Figure 121: ***Latched Mode Timing Diagram***

The following table describes the behavior of the ACX_BRAMTDP on each clock cycle of the diagram, where each row represents a transaction that spans the two indicated clock cycles.

Table 234: *ACX_BRAMTDP Timing Diagram Clock Cycle Behavior With Output Register Disabled*

Clock Cycle	Transaction	Description
0–1	Hold	None of the control signals are asserted. The output remains unchanged.
1–2	Reset latch	The `rstlatcha` (`rstlatchb`) signal is asserted. `douta` (`doutb`) is set to the `srval` as provided by the `porta_srval` (`portb_srval`) parameter.
2–3	Read	`pea` (`peb`) is asserted and `wea` (`web`) is de-asserted. The memory is read and `douta` (`doutb`) is set to the data read from the addressed location.
3–4	Hold	None of the control signals are asserted. The output remains unchanged.
4–5	Read with reset latch	Read operation, with the except that `rstlatcha` (`rstlatchb`) is asserted, causing the output to be set to the value provided by the `porta_srval` (`portb_srval`) parameter.
5–6	Read	`pea` (`peb`) is asserted and `wea` (`web`) is de-asserted. The memory is read and `douta` (`doutb`) is set to the data read from the addressed location.
6–7	Write	`pea` (`peb`) and `wea` (`web`) are both asserted. The data on the `dina` (`dinb`) pins is committed to memory. If `porta_write_mode` (`portb_write_mode`) is `"write_first"`, the output data reflects the data provided on the `dina` (`dinb`) port. If `"no_change"`, the output data remains unchanged.
7–8	Write with reset latch	`pea` (`peb`) and `wea` (`web`) are both asserted. the data on the `dina` (`dinb`) pins is committed to memory. Since `rstlatcha` (`rstlatchb`) is asserted, the output register is set to the value provided by the `porta_srval` (`portb_srval`) parameter.
8–9	Hold	None of the control signals are asserted. The output remains unchanged.

The second timing diagram illustrates the behavior of a ACX_BRAMTDP port with the output register enabled.

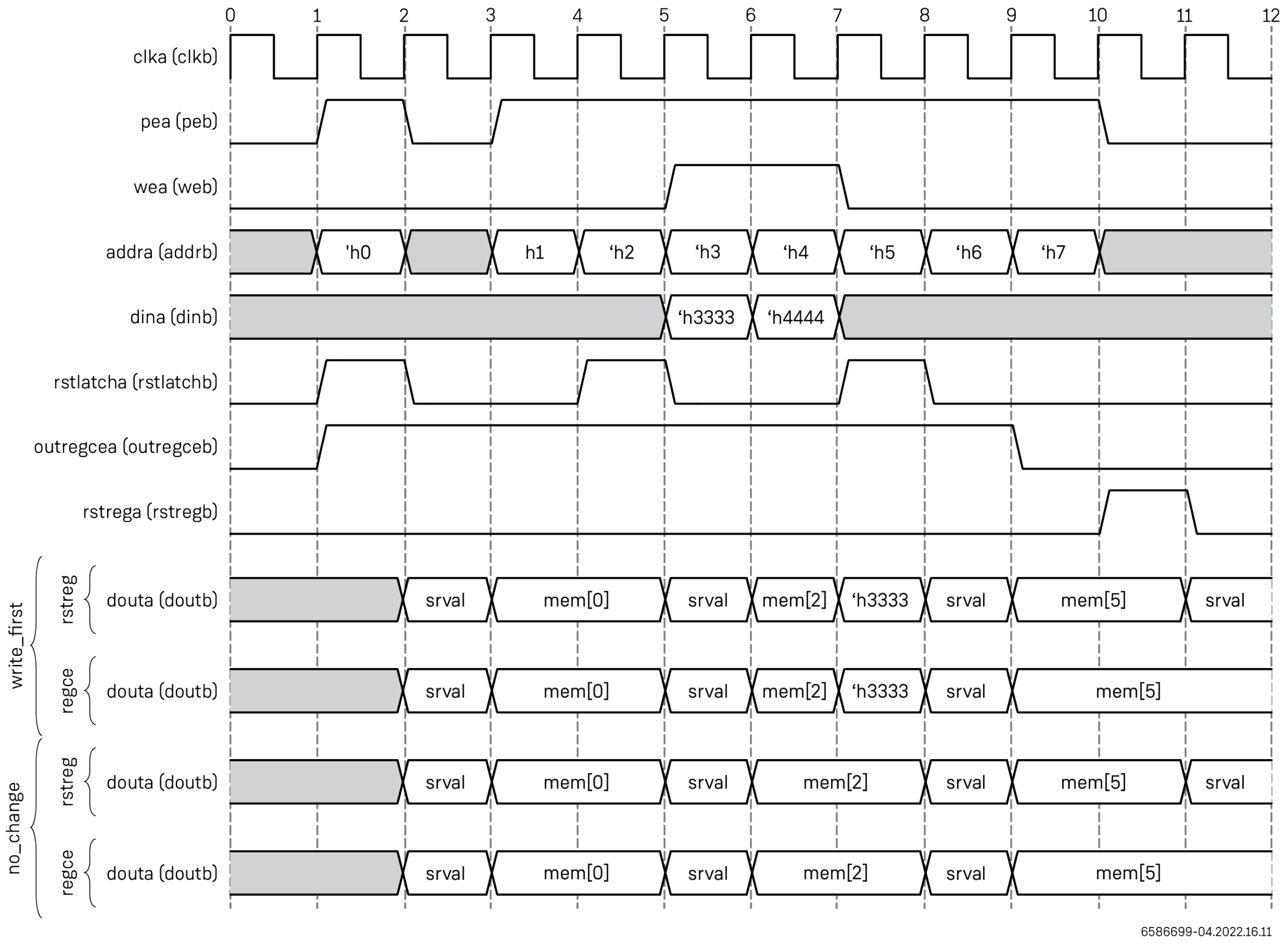


Figure 122: ***Registered Mode Timing Diagram***

The following table describes the behavior of the ACX_BRAMTDP on each clock cycle, where each line represents a single transaction that spans the three indicated clock cycles.

Table 235: ***ACX_BRAMTDP Timing Diagram Clock Cycle Behavior With Output Register Enabled***

Clock Cycle	Transaction	Description
0–2	Reset latch	None of the control signals are asserted. This is neither a read nor a write. On cycle 1, `rstlatch` is asserted and the output is set to the value provided by the `porta_srval` (`portb_srval`) parameter.
1–3	Read	`pea` (`peb`) is asserted and `wea` (`web`) is de-asserted. The memory is read and `douta` (`doutb`) is set two cycles later to the data read from the addressed location.
2–4	Hold	None of the control signals are asserted. The output remains unchanged.
3–5	Read with reset latch	Read operation, with the exception that `rstlatcha` (`rstlatchb`) is asserted on the second cycle of the transaction, causing the output to be set to the value provided by the `porta_srval` (`portb_srval`) parameter.
4–6	Read	`pea` (`peb`) is asserted and `wea` (`web`) is de-asserted. The memory is read and `douta` (`doutb`) is set to the data read from the addressed location.
5–7	Write	`pea` (`peb`) and `wea` (`web`) are both asserted. The data on the `dina` (`dinb`) pins is committed to memory. If `porta_write_mode` (`portb_write_mode`) is `"write_first"`, the output data (cycle 7) reflects the data provided on the `dina` (`dinb`) port. If `"no_change"`, the output data remains unchanged.
6–8	Write with reset latch	`pea` (`peb`) and `wea` (`web`) are both asserted. the data on the `dina` (`dinb`) pins is committed to memory. Since `rstlatcha` (`rstlatchb`) is asserted, on the second cycle the output register is set to the value provided by the `porta_srval` (`portb_srval`) parameter.
7–9	Read	`pea` (`peb`) is asserted and `wea` (`web`) is de-asserted. The memory is read and `douta` (`doutb`) is set to the data read from the addressed location.
8–10	Hold	`pea` (`peb`) is asserted and `wea` (`web`) is de-asserted. The memory is read. On the second cycle (cycle 9), `outregcea` (`outregceb`) is de-asserted. The output data is unchanged from the previous cycle.
9–11	Reset register	`pea` (`peb`) is asserted and `wea` (`web`) is de-asserted. the memory is read. On the second cycle (cycle 9), `rstrega` (`restregb`) is asserted, and `outregcea` (`outregceb`) is de-asserted. The output data is either unchanged, or is set to the value provided by the `porta_srval` (`portb_srval`) parameter, depending on whether the `rstrega` (`rstregb`) signal has priority over the `outregcea` (`outregceb`) signal, as determined by the the value of the `porta_regce_priority` (`portb_regce_priority`) parameter. If `"rstreg"`, asserting `rstrega` (`rstregb`) resets the output register independent of the `outregcea` (`outregceb`) signal. If `"regce"`, both `rstrega` (`rstregb`) and `outregcea` (`outregceb`) must be asserted to reset the output register.

Memory Initialization

The contents of the memory array can be initialized at power-up with one of the following two methods. This initialization is only performed when the FPGA is configured after power-up. The memory is not re-initialized when user logic is reset.

Initializing With Parameters

The data portion of initial memory contents may be defined by setting the 64 256-bit parameters, `initd_00` through `initd_63`. The data memory is organized as little-endian with bit 0 mapped to bit zero of parameter `initd_00` and bit 16383 mapped to bit 255 of parameter `initd_63`.

When the BRAM is configured with port widths of 9 or 18 bits, the parity portion of the initial memory contents may be defined by setting the eight 256-bit parameters, `initp_0` through `initp_7`. The parity memory is also organized as little-endian with the first parity bit location mapped to bit 0 of `initp_0` and the last parity bit mapped to the bit 255 of `initp_7`.

When the BRAM is configured with port widths of 5, 10 or 20 bits, the parity and extended parity portions of the initial memory contents may be defined by setting the eight 256-bit parameters `initp_0` through `initp_7` and the eight 256-bit parameters `initpx_0` through `initpx_7`. The parity and extended parity memories are both organized as little-endian with the first parity bit location mapped to bit 0 of `initp_0/initpx_0` and the last parity bit mapped to bit 255 of `initp_7/initpx_7`.

Initializing With a Memory Initialization File

Alternatively, the BRAM may be initialized with a memory file by setting the `mem_init_file` parameter to the path of a memory initialization file. The file format must be hexadecimal entries separated by white space, where the white space is defined by spaces or line separation. Each entry is a hexadecimal number of width equal to the maximum of the `porta_read_width`, `porta_write_width`, `portb_read_width`, and `portb_write_width` parameters.

A number entry may contain underscore (_) characters within the digits (i.e., `"A234_4567_33"`). Commenting is allowed beginning with a double-slash (`//`). C-like commenting is also allowed with the comment placed between `"/*"` and `"*/"` characters. The memory is initialized starting with the first entry of the file initializing the memory array starting with address zero and moving upward.

If `mem_init_file` is defined, the BRAM is initialized with the values in the file referenced by the `mem_init_file` parameter. If `mem_init_file` is left at the default value of "", the initial contents are defined by the values of the parameters `initd_00` through `initd_63`, `initp_0` though `initp_7` and `initpx_0` through `initpx_7`. If neither the memory initialization parameters nor the `mem_init_file` parameters are defined, the contents of the BRAM remain uninitialized and unknown until the memory locations are written.

The following tables show how the init values in the `porta_initval` (`portb_initval`), `porta_srval` (`portb_srval`) parameters and the memory initialization file entries map to `douta` (`doutb`), `doutpa` (`doutpb`), and `doutpx` (`doutpxb`):

Table 236: ***srval and initval to Output Signals Mapping for datawidth = 1, 2, 4, 8, and 16***

initval	datawidth				
	16	8	4	2	1
`init[15:8]`	`dout[15:8]`	–			
`init[7:4]`	`dout[7:4]`		–		
`init[3:2]`	`dout[3:2]`			–	
`init[1]`	`dout[1]`				–
`init[0]`	`dout[0]`				

Table 237: ***srval and initval to Output Signals Mapping for datawidth = 9 and 18***

initval	datawidth	
	18	9
`init[17]`	`doutp[1]`	–
`init[16:9]`	`dout[15:8]`	
`init[8]`	`doutp[0]`	
`init[7:0]`	`dout[7:0]`	

Table 238: ***srval and initval to Output Signals Mapping for datawidth = 5, 10, and 20***

initval	datawidth		
	20	10	5
`init[19]`	`doutpx[1]`	–	
`init[18:15]`	`dout[15:12]`		
`init[14]`	`doutp[1]`		
`init[13:10]`	`dout[11:8]`		
`init[9]`	`doutpx[0]`		–
`init[8:5]`	`dout[7:4]`		
`init[4]`	`doutp[0]`		
`init[3:0]`	`dout[3:0]`		

Create an Instance

To create memories within a design, there are three available methods:

1. Infer the memory – this method provides the greatest code portability and is the recommended approach. The following are examples of ACX_BRAMTDP (single port) and ACX_BRAMTDP (dual port) inference.
2. Directly instantiated – this method gives access to the full feature set of the memory. However, any code is less portable to other technology nodes. See Instantiation Templates (see page 344)
3. Use the ACE BRAM IP generator to create the appropriate memory structure – Refer to *ACE User Guide* (UG070) for details.

Inference Templates

ACX_BRAMTDP Single-Port Inference

```
//---------------------------------------------------------------------------------
//
// Copyright (c) 2022  Achronix Semiconductor Corp.
// All Rights Reserved.
//
//
// This software constitutes an unpublished work and contains
// valuable proprietary information and trade secrets belonging
// to Achronix Semiconductor Corp.
//
// This software may not be used, copied, distributed or disclosed
// without specific prior written authorization from
// Achronix Semiconductor Corp.
//
// The copyright notice above does not evidence any actual or intended
// publication of such software.
//
//

//---------------------------------------------------------------------------------
// Design: BRAMTDP Single-Port Inference
//         An example to infer a single-ported BRAMTDP (1 shared R/W port)
//         in Speedcore designs
//---------------------------------------------------------------------------------

`timescale 1ps / 1ps

module bram_tdp_single_port
#(
    parameter       ADDR_WIDTH     = 10,
    parameter       DATA_WIDTH     = 8,
    parameter       INIT_FILE_NAME = "",
    parameter       READ_MODE      = "NO_CHANGE"
)
(
    // Clocks and resets
    input  wire                     clk,

    // Enables
    input  wire                     we,

    // Address and data
    input  wire [ADDR_WIDTH-1:0]    addr,
    input  wire [DATA_WIDTH-1:0]    wr_data,

    // Output
    output reg  [DATA_WIDTH-1:0]    rd_data
);

localparam DATA_DEPTH = (2 ** ADDR_WIDTH);

reg [DATA_WIDTH-1:0]  mem_ram[DATA_DEPTH-1:0] /* synthesis syn_ramstyle = "block_ram"
"no_rw_check" */;
```

```
initial begin
    if (INIT_FILE_NAME != "")
        $readmemh(INIT_FILE_NAME, mem_ram);
end

always @(posedge clk) begin
    if(we) begin
        mem_ram[addr] <= wr_data;
        if (READ_MODE == "WRITE_FIRST") begin
            rd_data <= wr_data;
        end
    end
    else begin
        rd_data <= mem_ram[addr];
    end
end

endmodule : bram_tdp_single_port
```

ACX_BRAMTDP Symmetric Dual-Port Inference

```
//--------------------------------------------------------------------------------
//
// Copyright (c) 2022  Achronix Semiconductor Corp.
// All Rights Reserved.
//
//
// This software constitutes an unpublished work and contains
// valuable proprietary information and trade secrets belonging
// to Achronix Semiconductor Corp.
//
// This software may not be used, copied, distributed or disclosed
// without specific prior written authorization from
// Achronix Semiconductor Corp.
//
// The copyright notice above does not evidence any actual or intended
// publication of such software.
//
//

//--------------------------------------------------------------------------------
// Design: BRAMTDP Symmetric Inference
//         An example to infer a symmetric true dual-port BRAM in Speedcore designs
//--------------------------------------------------------------------------------

`timescale 1ps / 1ps

module bram_tdp_symmetric
#(
    parameter       ADDR_WIDTH     = 10,
    parameter       DATA_WIDTH     = 16,
    parameter       INIT_FILE_NAME   = "",
    parameter       READ_MODE      = "WRITE_FIRST"
)
(
    // Clocks and resets
    input  wire                     clk_a,
    input  wire                     clk_b,

    // Enables
    input  wire                     we_a,
    input  wire                     we_b,

    // Address and data
    input  wire [ADDR_WIDTH-1:0]    addr_a,
    input  wire [ADDR_WIDTH-1:0]    addr_b,
    input  wire [DATA_WIDTH-1:0]    wr_data_a,
    input  wire [DATA_WIDTH-1:0]    wr_data_b,

    // Output
    output reg  [DATA_WIDTH-1:0]    rd_data_a,
    output reg  [DATA_WIDTH-1:0]    rd_data_b
);
localparam DATA_DEPTH = (2 ** ADDR_WIDTH);
```

```
reg [DATA_WIDTH-1:0]  mem_ram[DATA_DEPTH-1:0] /* synthesis syn_ramstyle = "block_ram"
"no_rw_check" */;

initial begin
    if (INIT_FILE_NAME != "")
        $readmemh(INIT_FILE_NAME, mem_ram);
end

// synthesis synthesis_off
reg addr_collision;
assign addr_collision = (addr_a == addr_b);
// synthesis synthesis_on

always @(posedge clk_a) begin
    if(we_a) begin
        // synthesis synthesis_off
        if (addr_collision && we_b)
            mem_ram[addr_a] <= {DATA_WIDTH{1'bx}};
        else
        // synthesis synthesis_on
            mem_ram[addr_a] <= wr_data_a;

        if (READ_MODE == "WRITE_FIRST") begin
            rd_data_a <= wr_data_a;
        end
    end
    else begin
        // synthesis synthesis_off
        if (addr_collision && we_b)
            rd_data_a <= {DATA_WIDTH{1'bx}};
        else
        // synthesis synthesis_on
            rd_data_a <= mem_ram[addr_a];
    end
end

always @(posedge clk_b) begin
    if(we_b) begin
        // synthesis synthesis_off
        if (addr_collision && we_a)
            mem_ram[addr_b] <= {DATA_WIDTH{1'bx}};
        else
        // synthesis synthesis_on
            mem_ram[addr_b] <= wr_data_b;

        if (READ_MODE == "WRITE_FIRST") begin
            rd_data_b <= wr_data_b;
        end
    end
    else begin
        // synthesis synthesis_off
        if (addr_collision && we_a)
            rd_data_b <= {DATA_WIDTH{1'bx}};
        else
        // synthesis synthesis_on
            rd_data_b <= mem_ram[addr_b];
    end
end
```

```
endmodule : bram_tdp_symmetric
```

Instantiation Templates

Verilog

```
ACX_BRAMTDP #(
.porta_read_width(20),
.porta_write_width(20),
.porta_write_mode("write_first"),
.porta_clock_polarity("rise"),
.porta_en_out_reg(1'b0),
.porta_regce_priority("rstreg"),
.porta_peval(1'b1),
.porta_reg_rstval(1'b1),
.porta_latch_rstval(1'b1),
.porta_initval(20'h0),
.porta_srval(20'h0),
.portb_read_width(20),
.portb_write_width(20),
.portb_write_mode("write_first"),
.portb_clock_polarity("rise"),
.portb_en_out_reg(1'b0),
.portb_regce_priority("rstreg"),
.portb_peval(1'b1),
.portb_reg_rstval(1'b1),
.portb_latch_rstval(1'b1),
.portb_initval(20'h0),
.portb_srval(20'h0),
.mem_init_file(""),
.initd_00(256'h0),
.initd_01(256'h0),
.initd_02(256'h0),
.initd_03(256'h0),
.initd_04(256'h0),
.initd_05(256'h0),
.initd_06(256'h0),
.initd_07(256'h0),
.initd_08(256'h0),
.initd_09(256'h0),
.initd_10(256'h0),
.initd_11(256'h0),
.initd_12(256'h0),
.initd_13(256'h0),
.initd_14(256'h0),
.initd_15(256'h0),
.initd_16(256'h0),
.initd_17(256'h0),
.initd_18(256'h0),
.initd_19(256'h0),
.initd_20(256'h0),
.initd_21(256'h0),
.initd_22(256'h0),
.initd_23(256'h0),
.initd_24(256'h0),
.initd_25(256'h0),
.initd_26(256'h0),
.initd_27(256'h0),
.initd_28(256'h0),
.initd_29(256'h0),
.initd_30(256'h0),
.initd_31(256'h0),
.initd_32(256'h0),
.initd_33(256'h0),
.initd_34(256'h0),
.initd_35(256'h0),
.initd_36(256'h0),
.initd_37(256'h0),
.initd_38(256'h0),
.initd_39(256'h0),
.initd_40(256'h0),
.initd_41(256'h0),
.initd_42(256'h0),
.initd_43(256'h0),
.initd_44(256'h0),
.initd_45(256'h0),
.initd_46(256'h0),
.initd_47(256'h0),
.initd_48(256'h0),
.initd_49(256'h0),
.initd_50(256'h0),
.initd_51(256'h0),
.initd_52(256'h0),
.initd_53(256'h0),
.initd_54(256'h0),
.initd_55(256'h0),
.initd_56(256'h0),
.initd_57(256'h0),
.initd_58(256'h0),
.initd_59(256'h0),
.initd_60(256'h0),
.initd_61(256'h0),
.initd_62(256'h0),
.initd_63(256'h0),
.initp_0(256'h0),
.initp_1(256'h0),
.initp_2(256'h0),
.initp_3(256'h0),
.initp_4(256'h0),
.initp_5(256'h0),
.initp_6(256'h0),
.initp_7(256'h0),
.initpx_0(256'h0),
.initpx_1(256'h0),
.initpx_2(256'h0),
.initpx_3(256'h0),
.initpx_4(256'h0),
.initpx_5(256'h0),
.initpx_6(256'h0),
.initpx_7(256'h0))
instance_name (.addra(user_addra),
.dina(user_dina),
.dinpa(user_dinpa),
.dinpxa(user_dinpxa),
.wea(user_wea),
.pea(user_pea),
.rstlatcha(user_rstlatcha),
.rstrega(user_rstrega),
.outregcea(user_outregcea),
.clka(user_clka),
.douta(user_douta),
.doutpa(user_doutpa),
.doutpxa(user_doutpxa),
.addrb(user_addrb),
.dinb(user_dinb),
.dinpb(user_dinpb),
.dinpxb(user_dinpxb),
.web(user_web),
.peb(user_peb),
.rstlatchb(user_rstlatchb),
.rstregb(user_rstregb),
.outregceb(user_outregceb),
.clkb(user_clkb),
.doutb(user_doutb),
.doutpb(user_doutpb),
.doutpxb(user_doutpxb));
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
------------- DONE ACHRONIX LIBRARY ---------
-- Component Instantiation
ACX_BRAMTDP_instance_name : ACX_BRAMTDP
generic map (
porta_read_width => 20,
porta_write_width => 20,
porta_write_mode => "write_first",
porta_clock_polarity => "rise",
porta_en_out_reg => 0,
porta_regce_priority => "rstreg",
porta_peval => 1,
porta_reg_rstval => 1,
porta_latch_rstval => 1,
porta_initval => X"00000",
porta_srval => X"00000",
portb_read_width => 20,
portb_write_width => 20,
portb_write_mode => "write_first",
portb_clock_polarity => "rise",
portb_en_out_reg => 0,
portb_regce_priority => "rstreg",
portb_peval => 1,
portb_reg_rstval => 1,
portb_latch_rstval => 1,
portb_initval => X"00000",
portb_srval => X"00000",
mem_init_file => "",
initd_00 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_01 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_02 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_10 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_11 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_12 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_13 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_14 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_15 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_16 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_17 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_18 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_19 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_20 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_21 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_22 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_23 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_24 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_25 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_26 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_27 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_28 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_29 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_30 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_31 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_32 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_33 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_34 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_35 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_36 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_37 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_38 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_39 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_40 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_41 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_42 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_43 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_44 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_45 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_46 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_47 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_48 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_49 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_50 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_51 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_52 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_53 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_54 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_55 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_56 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_57 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_58 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_59 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_60 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_61 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_62 => X"0000000000000000000000000000000000000000000000000000000000000000",
initd_63 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_0 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_1 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_2 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_3 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_4 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_5 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_6 => X"0000000000000000000000000000000000000000000000000000000000000000",
initp_7 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_0 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_1 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_2 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_3 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_4 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_5 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_6 => X"0000000000000000000000000000000000000000000000000000000000000000",
initpx_7 => X"0000000000000000000000000000000000000000000000000000000000000000")
port map (
addra => user_addra ,
dina => user_dina ,
dinpa => user_dinpa ,
dinpxa => user_dinpxa ,
wea => user_wea ,
pea => user_pea ,
rstlatcha => user_rstlatcha ,
rstrega => user_rstrega ,
outregcea => user_outregcea ,
clka => user_clka ,
douta => user_douta ,
doutpa => user_doutpa ,
doutpxa => user_doutpxa ,
addrb => user_addrb ,
dinb => user_dinb ,
dinpb => user_dinpb ,
dinpxb => user_dinpxb ,
web => user_web ,
peb => user_peb ,
rstlatchb => user_rstlatchb ,
rstregb => user_rstregb ,
outregceb => user_outregceb ,
clkb => user_clkb ,
doutb => user_doutb ,
doutpb => user_doutpb ,
doutpxb => user_doutpxb);
```

ACX_BRAMFIFO (20-kb FIFO Memory with Optional Error Correction)

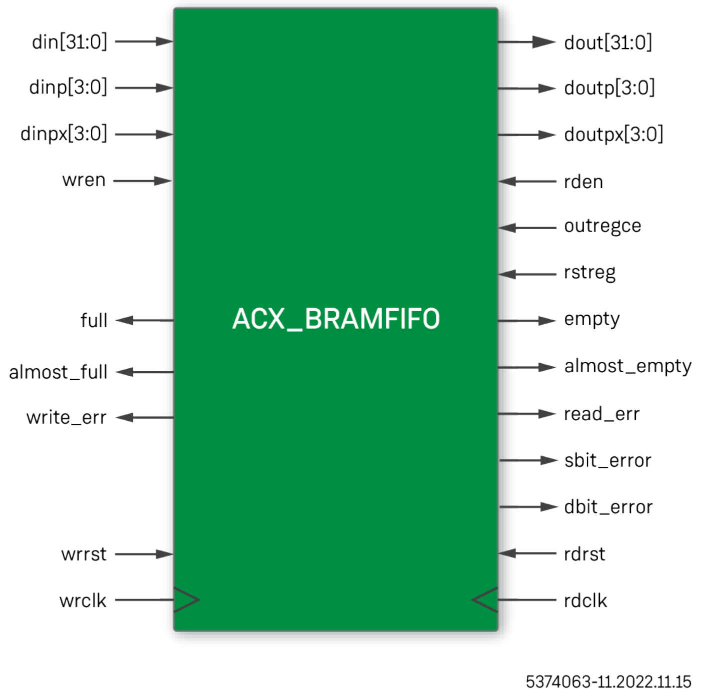


Figure 123: ***20Kb FIFO Memory With Optional Error Correction***

The BRAMFIFO component implements a 20Kb FIFO memory block using the embedded BRAM blocks with dedicated pointer and flag logic. The BRAMFIFO can be configured to support a variety of widths and depths, ranging from 512-word depth with 40-bit data to 16k depth with 1-bit data. Additionally, the read and write ports may be set to different widths. The read and write clocks may be either synchronous or asynchronous with respect to each other. If the user read and write clocks are the same, the `sync_mode` parameter may be set to `1'b1` to enable faster and synchronous generation of the status flags and FIFO pointer outputs.

When the `write_width` and `read_width` parameters are both set to 40, the error correction code (ECC) logic may be enabled to allow single-bit error correction with single and double-bit error detection on 32 bits of data. The embedded error correction encoder generates seven parity bits and stores them alongside each 32-bit word written into the memory. During the read operations, the error correction decoder reads the seven parity bits and the 32 data bits to provide error correction for all single-bit errors and error detection without correction for all two-bit errors.

An optional output register, complete with reset and clock enable inputs, is available to increase the speed of memory accesses. The use of the output register incurs a single additional cycle of read latency.

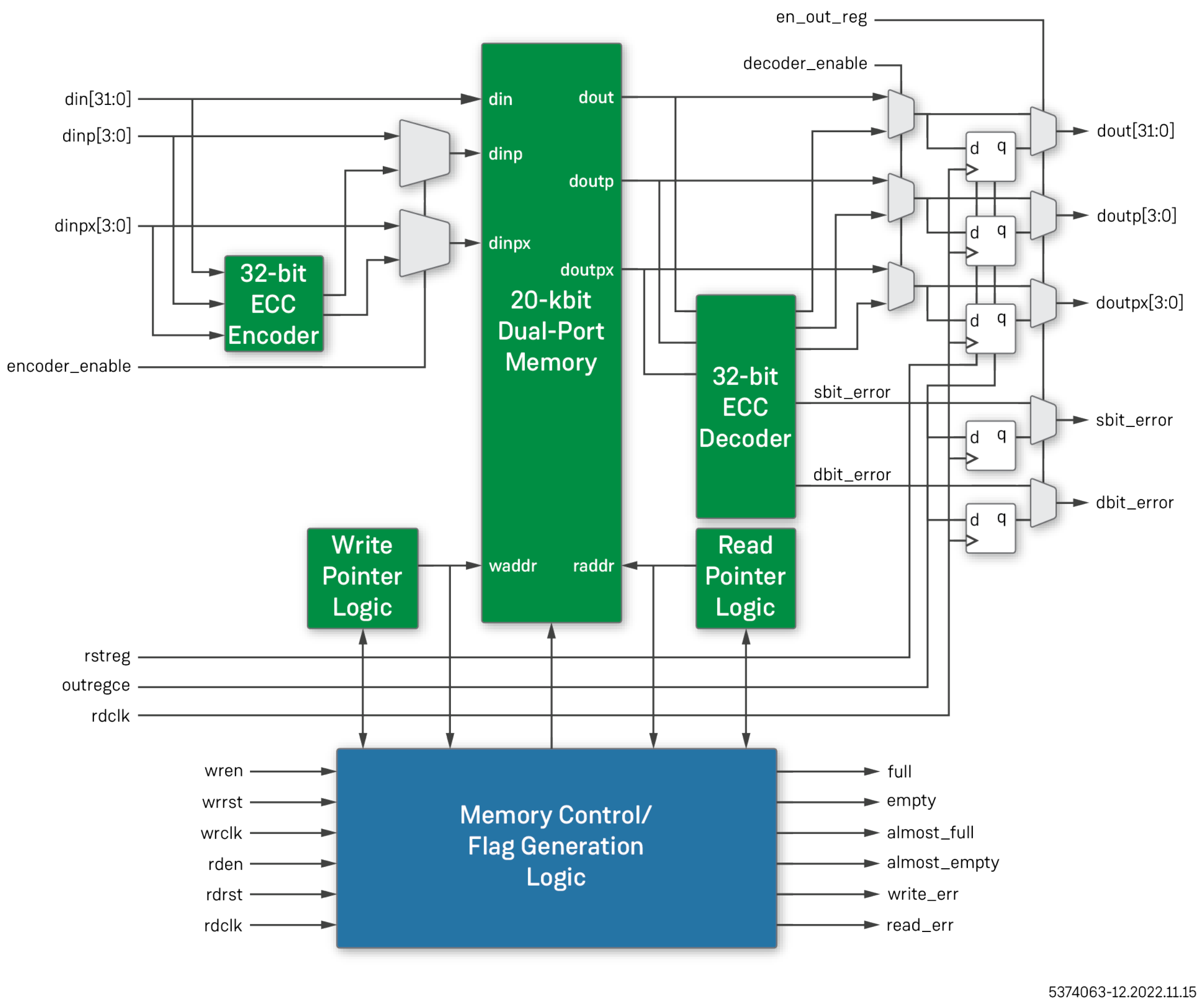


Figure 124: ***BRAMFIFO Block Diagram***

Table 239: ***BRAMFIFO Pin Description***

Name	Type	Clock Domain	Description
Write Interface			
`wrrst`	Input	`prog`	Write port FIFO reset (programmable, default active-high). Resets the FIFO to clear the read-side and/or write-side logic. The contents of the output register, if enabled, are not affected by the `wrrst` signal.
`wrclk`	Input	`wrclk`	Write clock (rising edge).
`wren`	Input	`wrclk`	Write enable signal. When asserted, data is written to the next unused memory location in the FIFO, at the next active edge of the write clock, as long as `full` is not also asserted. The `wren_polarity_sel` parameter determines whether `wren` is active-high (default) or active-low.

Name	Type	Clock Domain	Description
`din[31:0]`	Input	`wrclk`	Write port data input.
`dinp[3:0]`	Input	`wrclk`	Write port parity input (may be used for data). Used if the write data width is 9, 10, 18, 20, 36, or 40.
`dinpx[3:0]`	Input	`wrclk`	Write port extended parity input (may be used for data). Used if the write data width is 10, 20, or 40.
`full`	Output	`wrclk`	Full flag (active-high). Asserted when no more memory locations are available in the FIFO.
`almost_full`	Output	`wrclk`	Almost full flag (active-high). Asserted when fewer memory locations are available in the FIFO than the value of the `afull_offset` parameter.
`write_err`	Output	`wrclk`	Write error flag (active-high). Asserted after attempting to write to a full FIFO.
Read Interface			
`rdrst`	Input	`prog`	Read port FIFO reset (programmable, default active-high). Asserting `rdrst` resets the FIFO to clear the read-side and/or write-side logic. The contents of the output register, if enabled, are not affected by the `wrrst` signal.
`rdclk`	Input	`rdclk`	Read clock (rising edge).
`rden`	Input	`rdclk`	Read enable. Causes the the next data element to be read from the FIFO at the next active edge of the clock, if the empty signal is not asserted. The `rden_polarity_sel` parameter determines whether `rden` is active-high (default) or active-low.
`outregce`	Input	`rdclk`	Output register clock enable (active-high). When the output register is enabled, controls when the output data from the FIFO is presented on the `dout`, `doutp`, and `doutpx` ports. In most cases, this input should be held high. When the output register is enabled, the `outregce` signal should always be asserted during a read operation.
`rstreg`	Input	`rdclk`	Output register reset. The `sr_assertion_reg` parameter determines whether the reset is synchronous (default) or asynchronous, and the `reg_rstval` parameter determines whether `rstreg` is active-high (default) or active-low. When reset is asserted, the output register is assigned the value of the `reg_srval` parameter.
`dout[31:0]`	Output	`rdclk`	Read port `dout` output.
`doutp[3:0]`	Output	`rdclk`	Read port parity output (used for data). Used if the read data width is 9, 10, 18, 20, 36, or 40.
`doutpx[3:0]`	Output	`rdclk`	Read port extended parity output (used for data). Used if the read data width is 10, 20, or 40.

Name	Type	Clock Domain	Description
`empty`	Output	`rdclk`	Empty flag (active-high). Asserted whenever there is less than one word of data available to be read.
`almost_empty`	Output	`rdclk`	Almost-empty flag (active-high). Asserted when there are fewer words to be read than the value of the `aempty_offset` parameter.
`read_err`	Output	`rdclk`	Read-error flag (active-high). Asserted after attempting to read from an empty FIFO.
`sbit_error`	Output	`rdclk`	Single-bit error (active-high). The `sbit_error` signal is asserted during a read operation when a single-bit error is detected and the corrected word is output on the `dout` pins. Memory contents are not corrected by the error correction logic. The `sbit_error` signal is aligned with the associated read data word.
`dbit_error`	Output	`rdclk`	Dual-bit error (active-high). Asserted during a read operation when a dual-bit error is detected. In the case of a dual-bit error condition, the uncorrected read data word is output on the `dout` pins. The `dbit_error` signal is asserted one cycle after the associated read data word.

Table 240: *BRAMFIFO Parameters*

Parameter	Defined Values	Default Value	Description
`sync_mode`	`1'b0`, `1'b1`	`1'b0`	Bypasses the synchronization logic between the read and write ports. For use when the `wrclk` and `rdclk` clock inputs are connected to the same clock. Reduces the latency through the FIFO and provides faster de-assertion of the status flags (empty, full, etc.). If the read and write clocks are connected to different clock sources, the synchronization logic must be used, and `sync_mode` must be set to `1'b0`.
`write_width`	1, 2, 5, 4, 8, 9, 10, 16, 18, 20, 32, 36, 40	40	Defines the width of the data input bus. The Width Versus FIFO Depth (see page 359) table shows the maximum depth of the FIFO for each valid value of the `write_width` parameter.
`read_width`	1, 2, 5, 4, 8, 9, 10, 16, 18, 20, 32, 36, 40	40	Defines the width of the data output bus. The allowed value is subject to the combinations defined in the Valid Read Width Versus Write Width Combinations per Port (see page 357) table. The Width Versus FIFO Depth (see page 359) table shows the depth of the FIFO for each value of the `write_width` parameter.
`fwft`	`1'b0`, `1'b1`	`1'b0`	Defines whether the FIFO is in first-word-fall-through mode. Only effects the availability of the first word written to the FIFO when empty. Operation of the two modes is the same after the first read operation. May only be set to `1'b1` when the `sync_mode` parameter is set to `1'b0`. The two settings operate as follows: • **`1'b1`** – the first value written to the FIFO appears at the `dout` (and `doutp`, `doutxp` if applicable) output without the need to perform a read operation. • **`1'b0`** – the first data word written to the FIFO is available at the FIFO output one `rdclk` clock cycle after the first read operation.
`en_out_reg`	`1'b0`, `1'b1`	`1'b1`	Enables the FIFO output register. A value of `1'b0` disables the output register. When enabled, there is an additional cycle of latency for each read operation. The `en_out_reg` signal may only be `1'b0` when the FIFO is in single clock mode (sync_mode = `1'b1`).
`reg_initval`(1)	40-bit hexadecimal number	`40'h0`	Defines the power-up default value of the output register data, if enabled.
`reg_srval`(1)	40-bit hexadecimal number	`40'h0`	If the output register is enabled, defines the reset value of the output register data when `rstreg` is asserted.

Parameter	Defined Values	Default Value	Description
reg_rstval	1'b0, 1'b1	1'b1	Defines the active level of the rstreg input: • 1'b0 – sets active low. • 1'b1 – sets active high.
sr_assertion_reg	clocked, unclocked	clocked	Sets whether the assertion of the output register reset is synchronous or asynchronous with respect to rdclk. A value of clocked corresponds to a synchronous reset where the output register is reset upon the next rising edge of the clock if rstreg is asserted. A value of unclocked corresponds to an asynchronous reset where the output register is reset immediately following the assertion of rstreg.
wrrst_input_mode(2)	2'b00, 2'b01, 2'b10, 2'b11	2'b10	Defines how the write pointer is reset.
rdrst_input_mode(2)	2'b00, 2'b01, 2'b10, 2'b11	2'b10	Defines how the read pointer is reset.
wrrst_rstval(3)	1'b0, 1'b1	1'b1	Defines the active level of the wrrst input.
rdrst_rstval(3)	1'b0, 1'b1	1'b1	Defines the active level of the rdrst input.
afull_offset	15-bit hexadecimal number	15'h0004	Defines the word depth at which the almost_full output changes. almost_full may be used to determine the number of FIFO blind writes that can be made without monitoring the full flag. For example, if the afull_offset parameter is set to 15'h0004 and the almost_full signal is de-asserted, there are at least five empty FIFO locations. All five words may be written without overflowing the FIFO and causing assertion of write_err. The almost_full Flag Assertion Based on afull_offset Parameter (see page 362) table provides details.
aempty_offset	15-bit hexadecimal number	15'h0004	Defines the word depth at which almost_empty changes. almost_empty may be used to determine the number of blind FIFO reads that can be performed without monitoring the empty flag. For example, if the aempty_offset parameter is set to 15'h0004 and almost_empty is de-asserted, there are at least five words in the FIFO. All five words may be read without FIFO underflow, causing read_err to be asserted. Refer to the almost_empty Flag Assertion Based on afull_offset Parameter (see page 362) table.

Parameter	Defined Values	Default Value	Description
wren_polarity_sel	1'b0, 1'b1	1'b1	Determines the active level of wren. When set to 1'b0, the wren input is active-low. When set to 1'b1 it is active-high.
rden_polarity_sel	1'b0, 1'b1	1'b1	Determines the active level of rden. When set to 1'b0 the rden input is active-low. When set to 1'b1 it is active-high.
encoder_enable	1'b0, 1'b1	1'b1	Defines whether the ECC encoder logic is enabled or bypassed. When set to 1'b1, enables the ECC encoder for normal operation. When set to 1'b0, disables the ECC encoder logic and allows the din, dinp, and dinpx inputs to be connected directly to the memory write port.
decoder_enable	1'b0, 1'b1	1'b1	Defines whether the ECC decoder logic is enabled or bypassed. When set to 1'b1, enables the ECC decoder for normal operation. When set to 1'b0, disables the ECC decoder logic and allows the dout, doutp, and doutpx memory outputs to be routed to the output ports without error correction.

Table Notes

1. This 40-bit parameter assignment is dependent on the read_width parameter as shown in the srval & initval to Output Signals Mapping (see page 366) table.
2. The reset may be synchronized outside the FIFO or internal to the FIFO, and can be reset by the wrrst input or the rdrst input. Refer to FIFO Reset (see page 360) for details.
3. A value of 1'b0 sets active low, 1'b1 sets active high.

Memory Organization and Data Pin Assignments

The BRAMFIFO block supports port widths from one to forty bits. The widths of the `din`, `dinp`, and `dinpx` inputs are determined by the `write_width` parameter while the widths of the `dout`, `doutp`, and `doutpx` outputs are determined by the `read_width` parameter. The width of the read port may differ from the width of the write port. The Valid Read Width Versus Write Width Combinations per Port (see page 357) table shows the legal combinations of read and write widths. "X" indicates a supported configuration.

Table 241: *Valid Read Width Versus Write Width Combinations Per Port*

	Write Width												
Read Width	**512 × 40**	**1k × 20**	**2k × 10**	**4k × 5**	**512 × 36**	**1k × 18**	**2k × 9**	**512 × 32**	**1k × 16**	**2k × 8**	**4k × 4**	**8k × 2**	**16k × 1**
512 × 40	X	X	X	X	–	–	–	–	–	–	–	–	–
1k × 20	X	X	X	X	–	–	–	–	–	–	–	–	–
2k × 10	X	X	X	X	–	–	–	–	–	–	–	–	–
4k × 5	X	X	X	X	–	–	–	–	–	–	–	–	–
512 × 36	–	–	–	–	X	X	X	–	–	–	–	–	–
1k × 18	–	–	–	–	X	X	X	–	–	–	–	–	–
2k × 9	–	–	–	–	X	X	X	–	–	–	–	–	–
512 × 32	–	–	–	–	–	–	–	X	X	X	X	X	X
1k × 16	–	–	–	–	–	–	–	X	X	X	X	X	X
2k × 8	–	–	–	–	–	–	–	X	X	X	X	X	X
4k × 4	–	–	–	–	–	–	–	X	X	X	X	X	X
8k × 2	–	–	–	–	–	–	–	X	X	X	X	X	X
16k × 1	–	–	–	–	–	–	–	X	X	X	X	X	X

Data Widths Using Parity Pins

The ACX_BRAMFIFO memory has three buses for both data in and data out consisting of the respective `din` and `dout` interfaces, along with the `dinp`, `dinpx`, `doutp` and `doutpx` parity interfaces. When ECC is used, the parity interfaces are unused, allowing the ECC encoder and decoder to make use of the respective memory pins for ECC operation. When ECC is disabled, the parity interfaces are assigned to the respective data buses as shown in the following table.

Table 242: ***Parity Pins Assignment Per Port***

Data Width	dinpx/doutpx	dinp/doutp	din/dout
40	`{data[39], data[29], data[19], data[9]}`	`{data[34], data[24], data[14], data[4]}`	`{data[38:35], data[33:30],data[28:25], data[23:20], data[18:15], data[13:10],data[8:5], data[3:0]}`
36	–	`{data[35], data[26], data[17], data[8]}`	`{data[34:27], data[25:18],data[16:9], data[7:0]}`
32	–	–	`data[31:0]`
20	`{2'b00, data[19], data[9]}`	`{2'b00, data[14], data[4]}`	`{16'h0, data[18:15],data[13:10], data[8:5],data[3:0]}`
18	–	`{2'b00, data[17], data[8]}`	`{16'h0, data[16:9], data[7:0]}`
16	–	–	`{16'h0, data[15:0]}`
10	`{3'b000, data[9]}`	`{3'b000, data[4]}`	`{24'h0, data[8:5], data[3:0]}`
9	–	`{3'b000, data[8]}`	`{24'h0, data[7:0]}`
8	–	–	`{24'h0, data[7:0]}`
5	–	`{3'b000, data[4]}`	`{28'h0, data[3:0]}`
4	–	–	`{28'h0, data[3:0]}`
2	–	–	`{30'h0, data[1:0]}`
1	–	–	`{31'h0, data[0]}`

Caution!

Pay close attention to non power-of-two sized data widths and how the data bits are assigned.

Read and Write Depth

The FIFO write depth is the number of elements that can be written to the input side of an empty FIFO before the FIFO is full. Similarly, the FIFO read depth is the number of elements that can be read from a full FIFO before the FIFO is empty. The effective FIFO depth for the read or write port is determined by the width of each port. The following table shows the effective read depth and write depth as determined by the `read_width` and `write_width` parameters.

Table 243: ***Port Width Versus FIFO Depth***

write_width/read_width Parameter Value	Write/Read Depth fwft = 1'b0	Write/Read Depth fwft = 1'b1
40	512	513
36	512	513
32	512	513
20	1024	1025
18	1024	1025
16	1024	1025
10	2048	2049
9	2048	2049
8	2048	2049
5	4096	4097
4	4096	4097
2	8192	8193
1	16384	16385

FIFO Operation

The BRAMFIFO operations are described in detail in this section.

FIFO Reset

A FIFO reset is performed by asserting the `rdrst` and/or `wrrst` inputs as described in FIFO Resets (see page 380), causing the internal FIFO state to be reset such that the FIFO is empty. After a reset, it is not possible to retrieve any data contained in the FIFO prior to the reset. The entire FIFO is available to be accept new data.

FIFO Write

A FIFO write is performed by asserting the `wren` input when the FIFO is not full. Asserting `wren` causes the data present on the `din`, `dinp`, and `dinpx` inputs (depending on the data width) to be stored in the FIFO for later retrieval with a read operation. If a write operation fills the last remaining location in the FIFO, the `full` signal is asserted on the following clock cycle. If `wren` is asserted when the FIFO is full, the write fails, and `write_error` is asserted on the next clock cycle.

FIFO Read

A FIFO read is performed by asserting the `rden` input when the FIFO is not empty. Asserting the `rden` presents the next data word from the FIFO memory array on the `dout`, `doutp` and `doutpx` outputs (depending on the data width). Data is always read in the same order as it was written and is no longer stored in the FIFO after it has been read. If the last remaining location in the FIFO is read, the `empty` signal is asserted on the following clock cycle. If `rden` is asserted when the FIFO is empty, the read fails, and `read_error` is asserted on the next clock cycle.

FIFO Status Signals

The following table describes the FIFO status signals output by the BRAMFIFO component.

Table 244: ***FIFO Pointers and Status Flag Clock Domain Assignments***

Status Signal	Clock Domain	Description
empty	rdclk	Asserted when either the FIFO is reset or all data has been read from the FIFO. This flag is synchronous to the rdclk domain. Asserting rden when empty is asserted does not change the contents of the FIFO nor does it affect the data output, but does cause the read_err output to be asserted in the following rdclk cycle. When sync_mode is 1'b0, meaning that read and write ports are not in the same clock domain, a few clock cycles are required after writing data to the FIFO before empty is de-asserted. The empty signal is always asserted immediately when the FIFO becomes empty.
almost_empty	rdclk	Asserted when there are aempty_offset or fewer words remaining in the FIFO (refer to the almost_empty Flag Assertion Based on afull_offset Parameter (see page 362) table). The almost_empty flag may be used to determine the number of reads that can be performed without causing FIFO underflow and assertion of rd_err. For example, if the aempty_offset parameter is 15'h0004, and the almost_empty flag is not asserted, at least five words remain in the FIFO. When sync_mode is 1'b0, meaning that the read and write ports are not in the same clock domain, a few clock cycles are required after writing data to the FIFO before almost_empty is de-asserted. almost_empty is always asserted immediately when aempty_offset words remain in the FIFO.
read_err	rdclk	Asserted in the cycle following assertion of rden while the FIFO is empty.
full	wrclk	Asserted whenever all FIFO locations are in use. Asserting wren when full is asserted does not change the FIFO contents and causes the write_err output to be asserted in the following wrclk clock cycle. The din inputs are ignored in this case. When sync_mode is 1'b0, meaning that read and write ports are not in the same clock domain, a few clock cycles are required after reading FIFO data before full is de-asserted. full is always asserted immediately when the FIFO becomes full.
almost_full	wrclk	Asserted when afull_offset or fewer unused FIFO locations remain. almost_full may be used to determine the number of writes that can be performed without causing FIFO overflow and assertion of write_err. For example, if afull_offset is 15'h00004, and almost_full is not asserted, there are at least five empty FIFO locations. When sync_mode is 1'b0, meaning that the read and write ports are not in the same clock domain, a few clock cycles are required after reading from the FIFO before almost_full is de-asserted. almost_full is always asserted immediately when afull_offset locations remain in the FIFO.
write_err	wrclk	Asserted in the cycle following wren assertion while the FIFO is full.

The following two tables describe the conditions for asserting and de-asserting `almost_full` and `almost_empty`.

Table 245: ***almost_full Flag Assertion Based on afull_offset Parameter***

sync_mode	fwft	almost_full Assertion Condition	almost_full De-assertion Condition
`1'b0`	`1'b0`	`afull_offset` or fewer empty FIFO locations remain.	At least (`afull_offset` + 1) empty FIFO locations remain.
`1'b0`	`1'b1`		
`1'b1`	`1'b0`		
`1'b1`	`1'b1`	Illegal combination.	

Table 246: ***almost_empty Flag Assertion Based on afull_offset Parameter***

sync_mode	fwft	almost_empty Assertion Condition	almost_empty De-assertion Condition
`1'b0`	`1’b0`	`aempty_offset` or fewer FIFO words remain.	At least (`aempty_offset` + 1) FIFO words present.
`1'b0`	`1'b1`	(`aempty_offset` + 1) or fewer FIFO words remain.	At least (`aempty_offset` + 1) FIFO words present, plus the word fallen through to output.
`1'b1`	`1'b0`	`aempty_offset` or fewer FIFO words remain.	At least (`aempty_offset` + 1) FIFO words present.
`1'b1`	`1'b1`	Illegal combination.	

Before flag calculations can be made, the flag logic must ensure that both pointers are in the same clock domain as the flag for which the calculation is performed. The write pointer and read pointer synchronizers transfer each of the pointers into the other clock domain. A given pointer is synchronized to the opposite clock domain using a series of registers. The transfer of a pointer through these registers adds additional delay to the flag calculation. The following table shows the versions of the pointers used for flag calculations.

Table 247: ***Pointers Used for FIFO Flag Calculations***

Flag	Flag Calculation Write Pointer	Flag Calculation Read Pointer
`empty`	Synchronized Write Pointer	Read Pointer
`almost_empty`		
`full`	Write Pointer	Synchronized Read Pointer
`almost_full`		

The `empty` flag is computed from the synchronized write and read pointers. The write pointer incurs an additional delay of two `rdclk` cyles before being used to calculate the `empty` flag. Therefore, the `empty` flag does not transition from empty to non-empty state for a minimum of two `rdclk` cycles after the first write to the FIFO occurs. A similar delay occurs for the `almost_empty` flag. Also, for the `full` and `almost_full` flags, there are two `wrclk` cycles of delay in the actual FIFO status due to the synchronized read pointer. For an asynchronous FIFO, the calculation of the flags does not immediately reflect the state of the FIFO (not typical behavior for an asynchronous FIFO). A synchronous FIFO has only a single clock, so no clock domain crossing is required. A synchronous FIFO has the advantage that the flag calculation is immediate and precise.

FIFO Operational Modes

The BRAMFIFO is a highly configurable IP component supporting a number of modes of operation, including either synchronous or asynchronous (dual-clock) operation:

- Synchronous – the same clock must be connected to the `wrclk` and `rdclk` inputs, and there cannot be a phase offset between them.
- Asynchronous – two different clocks can be connected to the `wrclk` and `rdclk` inputs. The BRAMFIFO does not require any phase or frequency relationship between the two clocks. The two clock inputs are treated as being completely asynchronous to one another. There is no requirement regarding the relative frequencies of the two clocks. Either clock can be faster or slower than the other.

Synchronous Operation

The synchronous FIFO mode is selected by setting the `sync_mode` parameter to `1'b1`. In synchronous mode, there is no latency in updating the `empty` and `almost_empty` signals after a write operation, or updating the `full` and `almost_full` signals after a read operation. This lack of latency means that the status outputs always represent the exact state of the FIFO.

In this mode, first-word-fall-through (fwft, described in Asynchronous Operation (see page 372)) is not supported, and the `fwft` parameter must be `1'b0`.

Optional Output Register

An optional output register may be enabled at the output of the FIFO to improve the clock-to-out timing when in single clock mode (`sync_mode` = `1'b1`). Enabling the output register adds an a additional cycle of latency to the output data for each read operation. It should be considered as an optional pipeline stage at the data output of the FIFO. The timing of the `empty`, `almost_empty`, `full`, and `almost_full` signals are not changed when the output register is enabled.

The output register is enabled by setting the `en_out_reg` parameter to `1'b1`. The output register has independent clock enable (`outregce`) and synchronous reset (`rstreg`) inputs. The output register may be configured to have an active-high or active-low reset input as determined by the `reg_rstval` parameter. When `rstreg` is asserted, the value of the `reg_srval` is placed on the output of the register at the next active edge of `rdclk`. The initial power-up value of the output register is defined by the `reg_initval` parameter. The following table shows the functions of the optional output register and assumes the following:

- Active-high `rdclk`
- Active-high `outregce`
- Active-high `rstreg`

Table 248: ***Optional Output Register Function Table***

Operation	rstreg	outregce	rdclk	dout
Hold	X	X	X	`dout_previous`
Reset output	1	X	↑	`reg_srval`
Hold	0	0	↑	`dout_previous`
Update output	0	1	↑	`fifo_output`

When the output register is enabled, the `dout`, `doutp`, and `doutpx` signals take on the value specified in the `reg_initval` parameter when the FPGA is first configured. When the input `rstreg` is asserted, the `dout`, `doutp`, and `doutpx` signals take on the value specified in the `reg_srval` parameter. The following tables show how the `reg_init` and `reg_srval` parameters map to `dout`, `doutp`, and `doutpx`.

Table 249: ***srval and initval to Output Signals Mapping for datawidth = 1, 2, 4, 8, 16, and 32***

initval	datawidth					
	32	16	8	4	2	1
`init[31:16]`	`dout[31:16]`	–				
`init[15:8]`	`dout[15:8]`		–			
`init[7:4]`	`dout[7:4]`			–		
`init[3:2]`	`dout[3:2]`				–	
`init[1]`	`dout[1]`					–
`init[0]`	`dout[0]`					

Table 250: ***srval and initval to Output Signals Mapping for datawidth = 9, 18, and 36***

initval	datawidth		
	36	18	9
`init[35]`	`doutp[3]`	–	
`init[34:27]`	`dout[31:24]`		
`init[26]`	`doutp[2]`		
`init[25:18]`	`dout[23:16]`		
`init[17]`	`doutp[1]`		–
`init[16:9]`	`dout[15:8]`		
init[8]	`doutp[0]`		
`init[7:0]`	`dout[7:0]`		

Table 251: ***srval and initval to Output Signals Mapping for datawidth = 5, 10, 20, and 40***

initval	datawidth			
	40	20	10	5
`init[39]`	`doutpx[3]`	–		
`init[38:35]`	`dout[31:28]`			
`init[34]`	`doutp[3]`			
`init[33:30]`	`dout[27:24]`			
`init[29]`	`doutpx[2]`			
`init[28:25]`	`dout[23:20]`			
`init[24]`	`doutp[2]`			
`init[23:20]`	`dout[19:16]`			
`init[19]`	`doutpx[1]`		–	
`init[18:15]`	`dout[15:12]`			
`init[14]`	`doutp[1]`			
`init[13:10]`	`dout[11:8]`			
`init[9]`	`doutpx[0]`			–
`init[8:5]`	`dout[7:4]`			
`init[4]`	`doutp[0]`			
`init[3:0]`	`dout[3:0]`			

Timing Diagrams

The following diagram shows the operation of the FIFO in synchronous mode when the FIFO is empty, and the `aempty_offset` parameter is 3. This diagram assumes that all signals not shown, such as `rdrst` and `wrrst`, are not asserted.

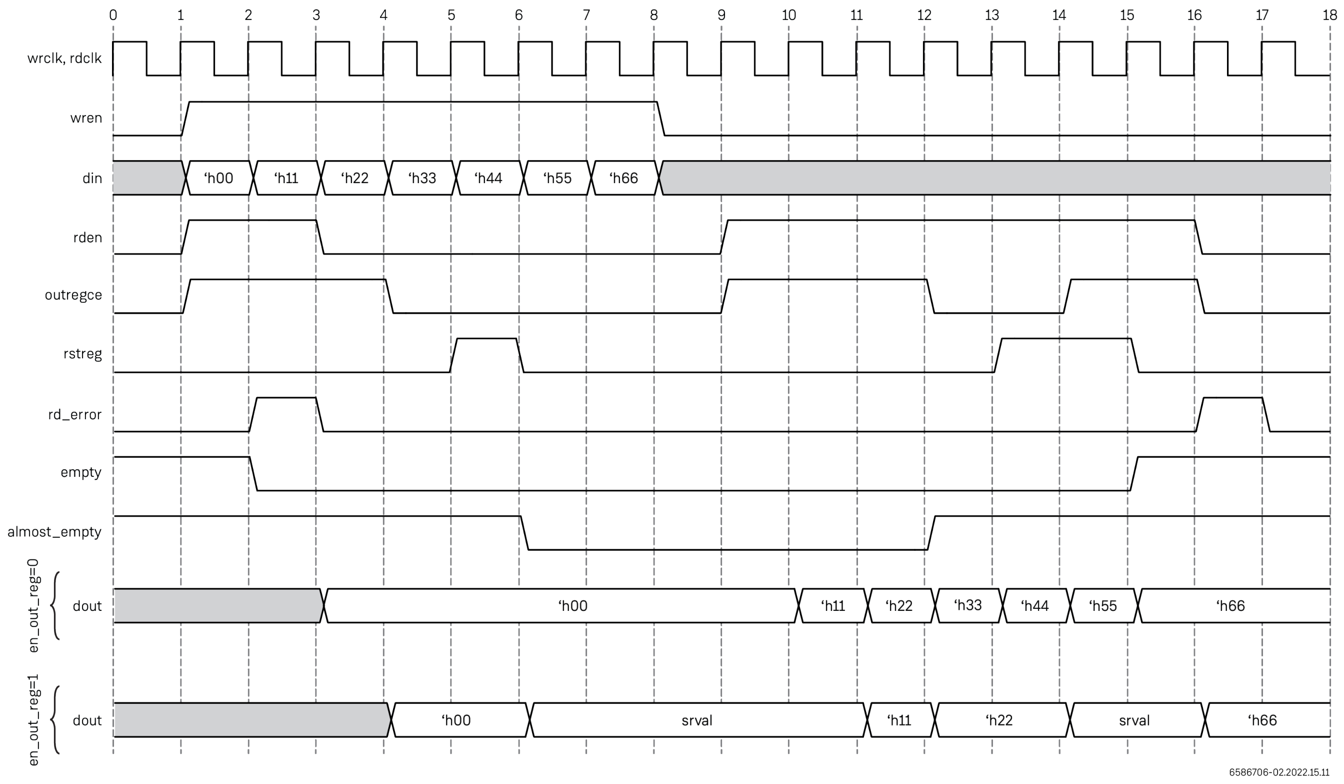


Figure 125: *Synchronous Mode Empty FIFO Timing Diagram*

The events of each clock cycle in the preceding diagram are described in the following table.

Table 252: *Synchronous Mode Empty FIFO Timing Diagram Events*

Event	Description
1	The `wren` signal is asserted, writing the first data word to the FIFO, causing `empty` to be de-asserted on the following clock cycle (FIFO is no longer empty). Simultaneously, `rden` is asserted, indicating a read attempt from the FIFO. Since the FIFO is still empty, `rd_err` is asserted on the following clock cycle, and the `dout` output does not change.
2	The `wren` signal is asserted, writing the second data word to the FIFO. Simultaneously, `rden` is asserted, reading the first data word from the FIFO. The data arrives on the `dout` output on the following cycle or the one after, depending on the `en_out_reg` parameter.
3	The `wren` signal is asserted, writing the third data word to the FIFO. The `rden` signal is not asserted in this cycle, so nothing is read from the FIFO. Asserting `outregce` when `rden` is de-asserted has no effect.
4	The `wren` signal is asserted, writing the fourth data word to the FIFO.
5	The `wren` signal is asserted, writing the fifth data word to the FIFO, leaving four words in the FIFO (the first word has already been read). The number of words is greater than the `aempty_offset` value of 3, so `almost_empty` is de-asserted on the following clock cycle. Simultaneously, `rstreg` is asserted, resetting the value of the output register on the following clock cycle to the value provided by the `reg_srval` parameter, if enabled. Enabling the output register has no effect on `aempty_offset` timing. If the output register is not enabled, `rstreg` has no effect.
6	The `wren` signal is asserted, writing the sixth data word to the FIFO.
7	The `wren` signal is asserted, writing the seventh data word to the FIFO.
8	No control signals are asserted.
9	The `rden` signal is asserted, reading the second data word from the FIFO. The data arrives on `dout` on the following cycle or the one after, depending on the `en_out_reg` parameter.
10	The `rden` signal is asserted, reading the third data word from the FIFO. The data arrives on `dout` on the following cycle or the one after, depending on the `en_out_reg` parameter.
11	The `rden` signal is asserted, reading the fourth data word from the FIFO, `outregce` is de-asserted on the following cycle. Since this leaves only three words in the FIFO, the `almost_full` signal is asserted on the next clock cycle. • If the `en_out_reg` parameter is `1'b0`, data arrives on `dout` on the following cycle, and de-asserting `outregce` has no effect. • If the `en_out_reg` parameter is `1'b1`, de-asserting `outregce` causes the output register to hold the previous value instead of the data just read from the FIFO. Even though the data was not present on the `dout` pins, it has been read from the FIFO, and it can not be read again.
	The `rden` signal is asserted, reading the fifth data word from the FIFO. `rstreg` is asserted on the following cycle.

Event	Description
12	• If the `en_out_reg` parameter is `1'b0`, the data arrives on `dout` on the following cycle, and the assertion of `rstreg` has no effect. • If the `en_out_reg` parameter is `1'b1`, asserting `rstreg` causes the output register to receive the value provided in the `reg_srval` parameter. Even though the data was not presented on `dout`, it was read from the FIFO, and it can not be read again.
13	The `rden` signal is asserted, reading the sixth data word from the FIFO. The signals `outregce` and `rstreg` are both asserted on the following cycle. • If the `en_out_reg` parameter is `1'b0`, the data arrives on `dout` on the following cycle, and `outregce` and `rstreg` have no effect. • If the `en_out_reg` parameter is `1'b1`, asserting `rstreg` causes the output register to receive the value provided in the `reg_srval` parameter regardless of the `outregce` value. Even though the data was not presented on `dout`, it was read from the FIFO, and it can not be read again.
14	The `rden` signal is asserted, reading the seventh and last data word from the FIFO. The data arrives on `dout` on the following cycle or the one after, depending on the `en_out_reg` parameter. Since the FIFO is empty, the `empty` signal is asserted on the next cycle. • If `en_out_reg` is `1'b0`, the data arrives on `dout` on the following cycle, and `outregce` and `rstreg` have no effect. • If `en_out_reg` parameter is `1'b1`, asserting `rstreg` causes the output register to receive the value provided in the `reg_srval` parameter, regardless of the `outregce` value. Even though the data was not presented on `dout`, it was read from the FIFO, and it can not be read again.
15	The `rden` signal is asserted even though the FIFO is empty. `read_error` is asserted on the following clock edge, and the FIFO contents are unchanged.

The following diagram shows the operation of the FIFO in synchronous mode, starting when there are 5 locations remaining in the FIFO, and the `afull_offset` parameter is 3. This diagram assumes that all signals not shown, such as `rdrst` and `wrrst`, are de-asserted, and that the `en_out_reg` parameter is `1'b0`. If `en_out_reg` was `1'b1`, the `dout` signal would be delayed by one cycle.

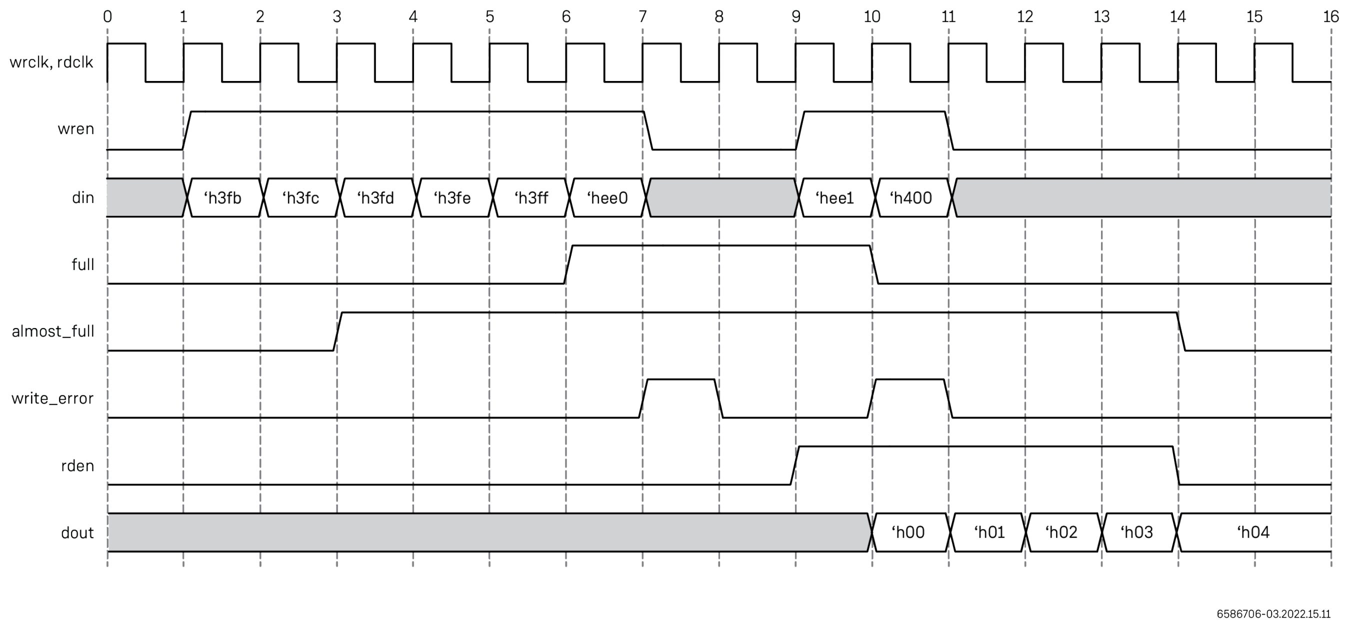


Figure 126: ***Synchronous Mode Full FIFO Timing Diagram***

The events of each clock cycle in the preceding diagram are described in the following table.

Table 253: ***Synchronous Mode Full FIFO Timing Diagram Events***

Event	Description
1–5	The `wren` signal is asserted, writing a data word to the FIFO. After the second write, only three locations are free, so `almost_full` is asserted on the next clock cycle. The fifth write fills the last FIFO element, and the `full` signal is asserted on the following clock cycle.
6	The `wren` signal is asserted. Since the FIFO is already full, the write operation does not take place, and `write_error` is asserted on the following clock cycle.
7–8	No operation.
9	The `wren` and `rden` signals are both asserted at the same time as both a read and a write operation is desired. Since the `full` signal is asserted, the write fails, and `write_error` is asserted on the following cycle. The read is successful, and the output data is presented on `dout` on the following cycle.
10	The `wren` and `rden` signals are both asserted at the same time, and the input word is written to the FIFO while the next output word is read and presented on `dout`. Since `full` is not asserted, both operations are successful.
11–13	The `rden` signal is asserted, and the next output data is read from the FIFO and presented on `dout`. After the third read, there are more than three unused locations in the FIFO, so `almost_full` is de-asserted on the next cycle.
14	The `rden` signal is not asserted, so the output remains constant.

Asynchronous Operation

When the FIFO is configured for asynchronous operation (`sync_mode` = `1'b0`), no phase or frequency relationship is assumed between between the write and the read clocks. The two clock inputs are treated as being completely asynchronous to one another. There is no requirement regarding the relative frequencies of the two clocks. Either clock can be faster or slower than the other.

Compared to synchronous mode, asynchronous mode causes an additional delay when updating `empty` and `almost_empty` after a write operation, or updating `full` and `almost_full` after a read operation, as it takes time for the status to cross safely from one clock domain to the other. All status signals are asserted without delay having only their de-assertion requiring additional time. For asynchronous operation, the `en_out_reg` parameter must be set to `1'b1`.

When using the FIFO with two clocks, the first-word-fall-through (fwft) parameter controls when data is made available on the output signals:

- `fwft` = `1'b0` (request mode) – when in request mode, asserting `rden` requests that the data be presented on the `dout` pins on the following cycle. This mode is identical to when `sync_mode` = `1'b1`, and the clocks are synchronous to one another. In this mode, the output of the FIFO remains unchanged after the first write in the empty state. After the first write operation, the `empty` flag is de-asserted, indicating that data is available. The FIFO is read by asserting `rden` and the first word written is then available at the outputs on the next `rdclk` cycle. Each subsequent read operation updates the FIFO outputs with the next stored data word if it is available (`empty` = `0`).

- fwft = 1'b1 (acknowledge mode) – when in acknowledge mode, the FIFO behaves as a first-word-fall-through FIFO, meaning that when empty, the first data word written to the FIFO is presented on the output pins as soon as possible, without waiting for `wren` to be asserted. After a reset (or after the last word has been read), the FIFO is in an empty state as indicated by assertion of `empty`. The output of the FIFO is updated after the next write and `empty` is de-asserted indicating that data is available to be read. Asserting `rden` effectively acknowledges the output data currently on the `dout` pins, allowing the FIFO to move to the next data word if not empty. Each subsequent read operation updates the outputs with the next stored data word if available (`empty` flag = `1'b0`). First-word-fall-through mode has the effect of making the FIFO one element deeper.

Timing Diagrams

The following diagram shows the operation of the FIFO in asynchronous mode when the FIFO is empty, and the `aempty_offset` parameter is 3. This diagram assumes that all signals not shown, such as `rdrst` and `wrrst`, are de-asserted.

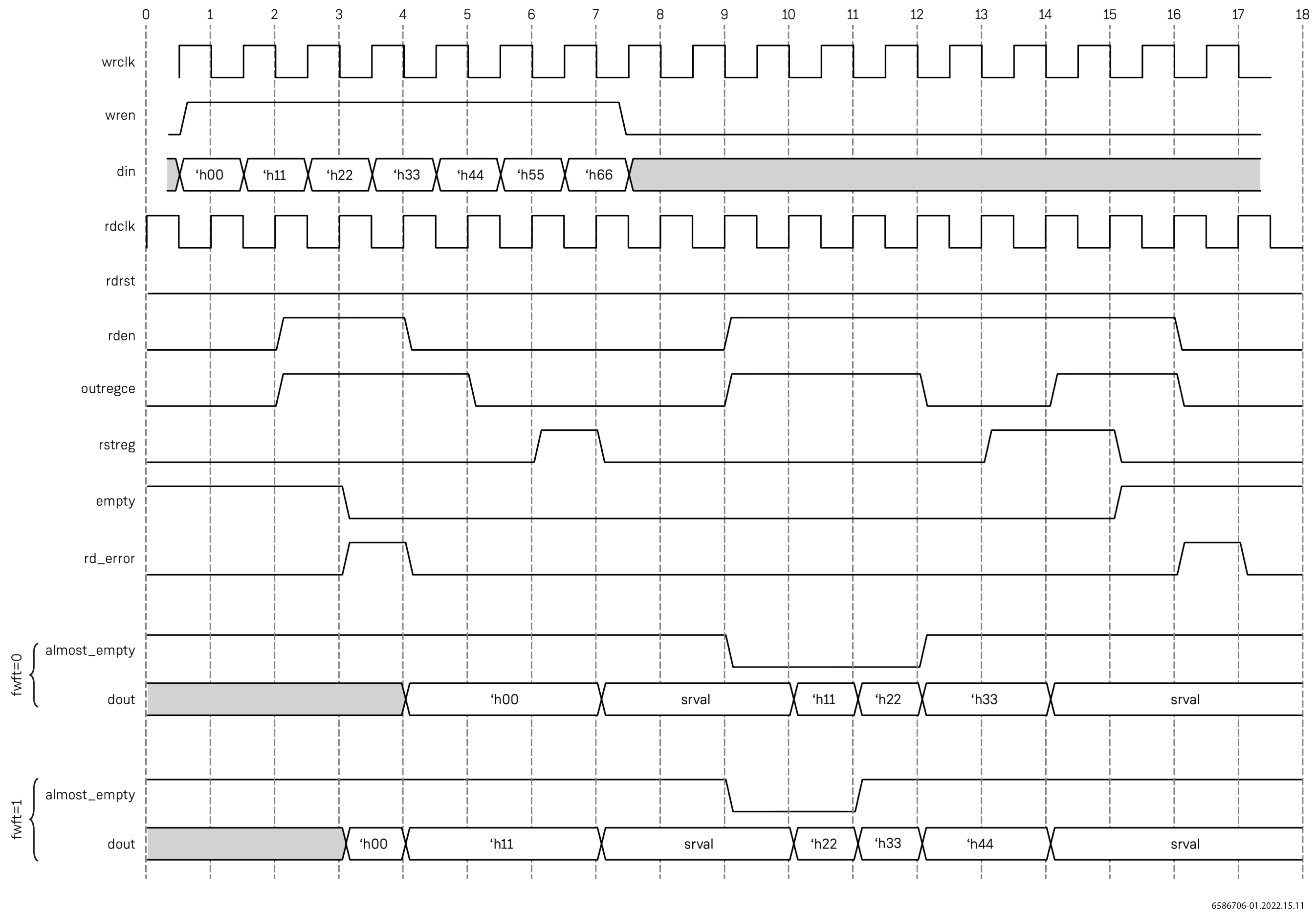


Figure 127: ***Asynchronous Mode Empty FIFO Timing Diagram***

The events of each clock cycle in the preceding digram are described in the following table.

Table 254: *Asynchronous Mode Empty FIFO Timing Diagram Events*

Event	Description
0–6	The `wren` signal is asserted synchronous to `wrclk`, writing seven data words to the FIFO. • Two or three clock cycles after the first write, `empty` is de-asserted synchronous to `rdclk`. If `fwft = 1'b1`, the first data word is presented on `dout` when `empty` is de-asserted. • After the fifth write, the FIFO has four words (since the first word has already been read). The amount of words is greater than the aempty_offset value of 3, so almost_empty is asserted two or three clock cycles later, synchronous to `rdclk`.
2	The `rden` signal is asserted indicating that a read from the FIFO is desired. Since the `empty` output remains asserted, the read fails, and `rd_err` is asserted on the following clock cycle. The data on `dout` does not change.
3	The `rden` signal is also asserted, reading the first data word from the FIFO. • If `fwft = 1'b0`, the data arrives on `dout` on the following clock cycle. • If `fwft = 1'b1`, the first data word on `dout` is replaced by the second data word.
4	The `rden` signal is not asserted in this cycle. Nothing is read from the FIFO. Asserting `outregce` when `rden` is de-asserted has no effect.
6	The `rstreg` signal is asserted, resetting the value of the output register to that provided by the `reg_srval` parameter. The effect of `rstreg` is not dependent on the `fwft` parameter.
7–8	No control signals are asserted.
9	The `rden` signal is asserted, reading the second data word from the FIFO. • If `fwft = 1'b0`, the data arrives on `dout` on the following cycle. • If `fwft = 1'b1`, the previous data word on `dout` is replaced by the next data word.
10	The `rden` signal is asserted, reading the third data word from the FIFO. • If `fwft = 1'b0`, the data arrives on `dout` on the following cycle. • If `fwft = 1'b1`, the previous data word on `dout` is replaced by the next data word, leaving only four more words in the FIFO. `almost_empty` is de-asserted.
11	The `rden` signal is asserted, reading the fourth data word from the FIFO. • If `fwft = 1'b0`, the data arrives on `dout` on the following cycle, leaving only four more words in the FIFO. `almost_empty` is de-asserted. • If `fwft = 1'b1`, the previous data word on `dout` is replaced by the next data word.
12	The `rden` signal is asserted, reading the fifth data word from the FIFO. The `outregce` signal is de-asserted, causing the output register to hold the previous value instead of providing the data just read. Even though the data is not presented on `dout`, it has been read from the FIFO and cannot be read again.

Event	Description
13	The `rden` signal is asserted, reading the sixth data word from the FIFO. asserting `rstreg` causes the output register to receive the value provided in the `reg_srval` parameter. Even though the data is not presented on `dout`, it has been read from the FIFO and cannot be read again.
14	The `rden` signal is asserted, reading the seventh and last data word from the FIFO. The `outregce` and `rstreg` signals are both asserted. asserting `rstreg` causes the output register to receive the value provided in the `reg_srval` parameter, regardless of the `outregce` value. Since the FIFO is empty, the `empty` signal is asserted on the next `rdclk` cycle. Even though the data was not presented on `dout`, it has been read from the FIFO, and it can not be read again.
15	The `rden` signal is asserted even though the FIFO is empty. The `rd_error` signal is asserted on the following clock edge, and the FIFO contents are unchanged.

The following diagram shows the operation of the FIFO in asynchronous mode, starting when there are five locations remaining in the FIFO where the `afull_offset` parameter is 3. This diagram assumes that all signals not shown, such as `rdrst` and `wrrst`, are de-asserted, and the `en_out_reg` parameter is `1'b1`.

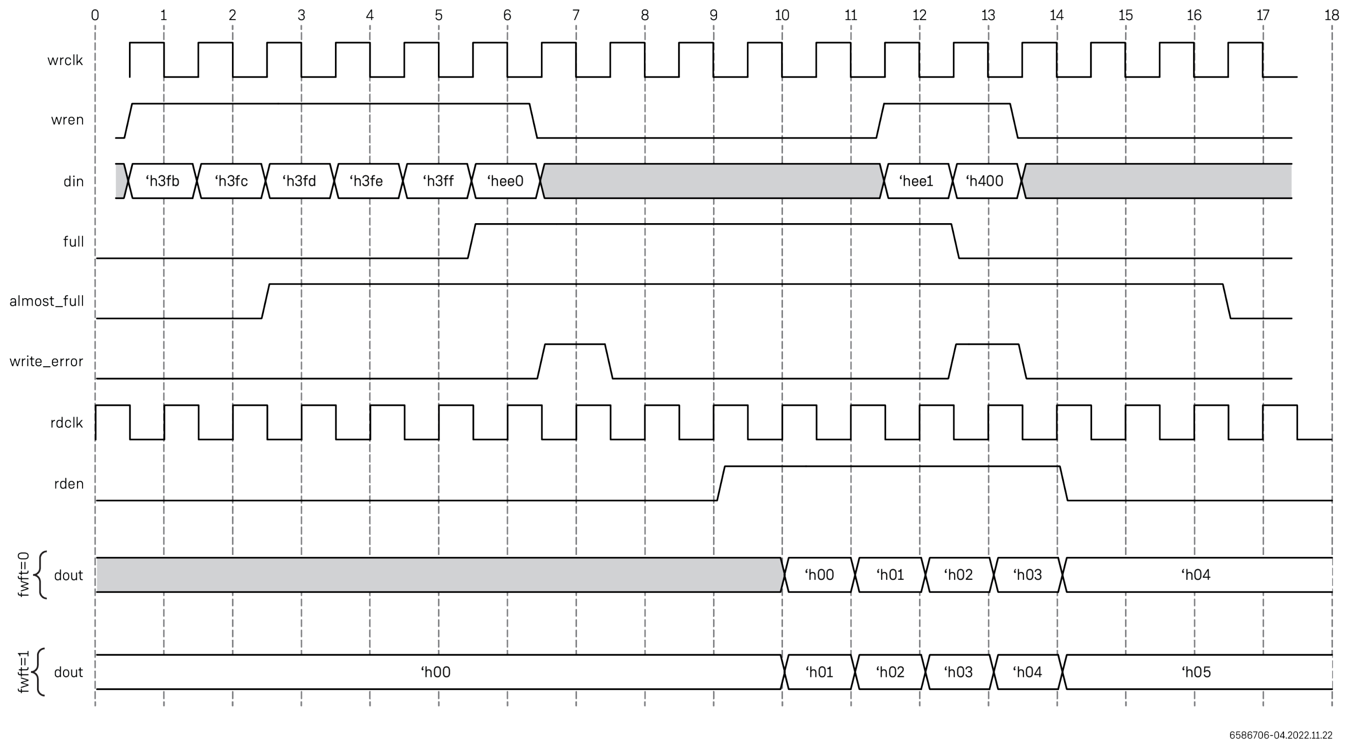


Figure 128: ***Asynchronous Mode Full FIFO Timing Diagram***

The events of each clock cycle in the preceding diagram are described in the following table.

Table 255: ***Asynchronous Mode Full FIFO Timing Diagram Events***

Event	Description
1–5	The `wren` signal is asserted, writing five data words to the FIFO. • After the second write, the FIFO has only three locations free, so `almost_full` is asserted on the next clock cycle. • After the fifth write, the last FIFO element has been used, and `full` is asserted on the following clock cycle.
6	The `wren` signal is asserted. Since the FIFO is already full, the write operation does not take place, and `write_error` is asserted on the following clock cycle.
7–8	No operation.
9	The `rden` signal is asserted, and the next output data is read from the FIFO and presented on `dout`. Two or three cycles later, `full` is de-asserted synchronously to `wrclk`. • If `fwft = 1'b0`, the first data arrives on `dout` on the following cycle. • If `fwft = 1'b1`, the first data word remains present on the output since it was first written. This data word is replaced by the next data word being read from the FIFO.
10	The `rden` signal is asserted, and the next output data is read from the FIFO and presented on `dout`.
11	The `rden` signal is asserted synchronously to `rdclk` and `wren` is asserted synchronously to `wrclk`, meaning that the both a read and write operation is desired. Since `full` is asserted, the write fails, and `write_error` is asserted on the following `wrclk` cycle. The read is successful, and the output data is updated on the following `rdclk` cycle.
12	The `rden` signal is asserted synchronously to `rdclk` and `wren` is asserted synchronously to `wrclk`. The input word is written to the FIFO while the next output word is read from the FIFO and presented on `dout`. Since `full` is not asserted, both operations are successful. Now there are more than three unused locations in the FIFO, so `almost_full` is de-asserted two or three cycles later, synchronously to `wrclk`.
13	The `rden` signal is asserted, and the next output data is read from the FIFO and presented on `dout`.

Mixed-Width Modes

The BRAMFIFO allows the read port width to be different than the write port width. The port widths affect how the signals `empty`, `almost_empty`, `almost_full`, and `full` are asserted. The signals `empty` and `almost_empty` are relative to the read data width and are only de-asserted when one or more `aempty_offset` read operations can be performed at the read data width. Similarly, the `full` and `almost_full` signals are relative to the write data width and are only de-asserted when one or more `afull_offset` write operations can be performed at the write data width.

To illustrate, the following diagram shows the operation of the FIFO in synchronous mode, starting when the FIFO is empty, and the `aempty_offset` parameter is 3. The write data width is 4 bits, and the read data width is 8 bits. This diagram assumes that all signals not shown, such as `rdrst` and `wrrst`, are de-asserted, and the `en_out_reg` parameter is `1'b0`. If the `en_out_reg` parameter had been set to `1'b1`, `dout` would be delayed by one cycle.

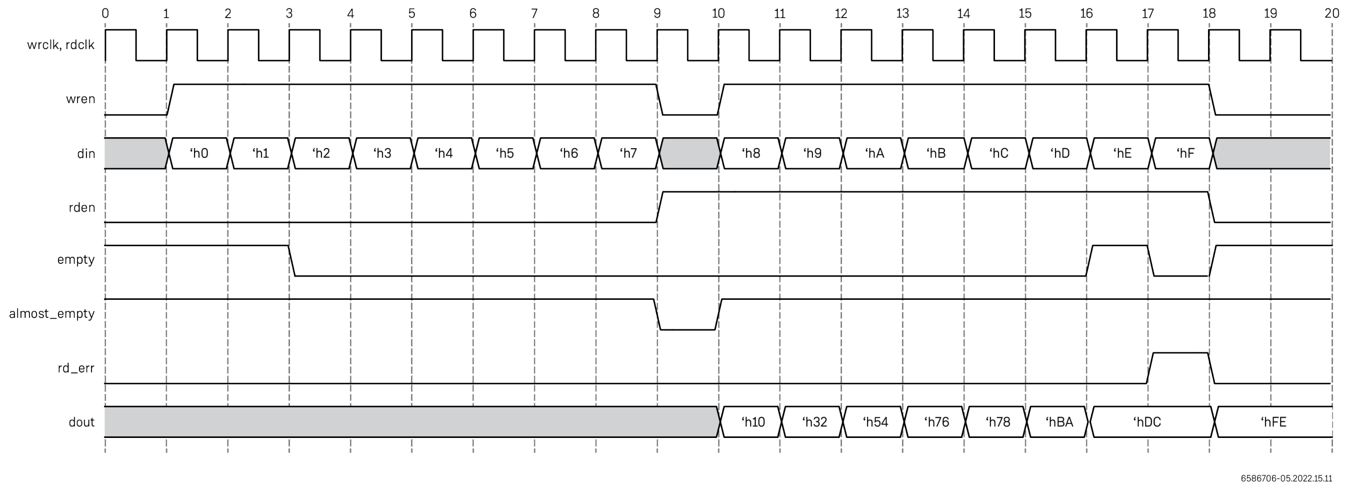


Figure 129: ***Synchronous Mode Mixed-Width FIFO Operation Timing Diagram***

The events of each clock cycle in the preceding diagram are described in the following table.

Table 256: ***Synchronous Mode Mixed-Width FIFO Operation Timing Diagram Events***

Event	Description
1-8	The `wren` signal is asserted, writing a total of eight 4-bit words to the FIFO.
3	The `empty` signal is de-asserted after two 4-bit writes have been completed, providing enough data for a single 8-bit read.
9	The `almost_empty` signal is de-asserted after eight 4-bit writes to the FIFO since there are now four 8-bit words available to the read side of the FIFO, which is larger than the `aempty_offset` value of 3.
9-17	The `rden` signal is asserted, causing the data to be read from the FIFO in 8-bit words. Each read returns data that was written in two write operations.
10	The `almost_empty` signal is asserted again since there are no longer four 8-bit words available to be read from the FIFO.
10-17	The `wren` signal is asserted, writing a total of eight 4-bit words to the FIFO.
16	The `empty` signal is asserted since at this time the FIFO contains only 4 bits of data, and there is no longer enough data for a complete 8-bit read operation.
17	The `rd_err` signal is asserted as a result of `rden` assertion while empty on the previous clock cycle.
18	The `empty` signal is asserted as a result of the last data word being read from the FIFO.

FIFO Resets

Several FIFO reset options are available. The basic option allows the FIFO to be reset without the need to synchronize the reset signals externally. The reset synchronization performed within the component requires two clock cycles. A lower-latency reset can be achieved using one of the following advanced reset modes.

For the basic FIFO reset, do the following:

1. Set both the `wrrst_input_mode` and `rdrst_input_mode` parameters to `2'b11`, the default setting.
2. Connect the user reset signal to both the `wrrst` and `rdrst` input pins.
3. To reset the FIFO, assert the reset signal for a minimum of three cycles of the slower of `wrclk` and `rdclk` which performs the following:
    - Resets the internal write and read pointers
    - Sets the `empty` and `almost_empty` flags
    - Clears the `full` and `almost_full` flags
4. Do not attempt to read or write the FIFO while the reset is asserted or before three cycles after the de-assertion of the reset signal

For basic FIFO operation, the parameters associated with reset should remain at their default settings as shown in the Reset Parameter Mapping (see page 382) table.

Advanced FIFO Reset Modes

The BRAMFIFO provides several reset options from either the read or write clock domains. The reset may be either sychronous with respect to the read and/or write clock domains or the internal reset synchronization logic may be enabled to synchronize the reset inputs. The capability for synchronous resets is provided to allow the fastest response between the reset assertion and when the FIFO is ready to be written. Internal to the FPGA, the read and write pointers have synchronous reset inputs. The BRAMFIFO macro provides the necessary logic to perform the synchronization of pointer resets without requiring the implementation of synchronization logic external to the FIFO.

The write pointer reset logic is configured via the `wrrst_input_mode` parameter while reset logic for the read pointer is configured via the `rdrst_input_mode` parameter. The reset operation of the read and write pointers is configured independently so that in addition to optionally synchronizing the reset inputs, each side of the FIFO can be configured to respond to one or both of the `wrrst` or `rdrst` input signals.

It is important to ensure that the resets are synchronized to the proper clock domain. If the read or write pointer synchronizers are bypassed, synchronization of `rdrst` and `wrrst` must be performed external to the FIFO. The following figure shows the block diagram of the FIFO reset selection logic, with a table that describes the behavior of each mode. The logic to configure the read and write pointer resets is identical.

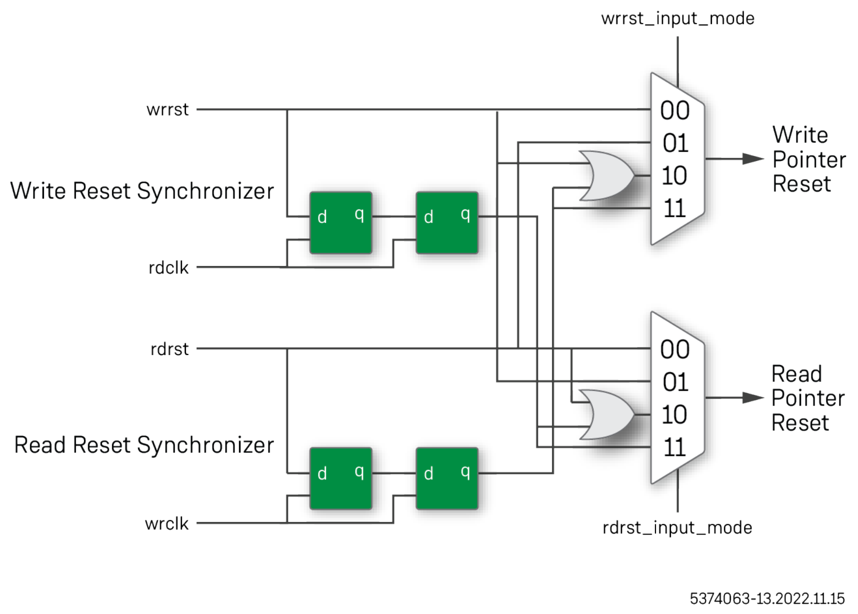


Figure 130: ***Read and Write Pointer Reset Input Selection Block Diagram***

Table 257: *wrrst_input_mode (rdrst_input_mode) Parameter Mapping*

wrrst_input_mode (rdrst_input_mode)	Write-side (Read-side) Reset Selected Input
`2'b00`	`wrrst` (`rdrst`) resets the write (read) interface logic. The `wrrst` (`rdrst`) signal must be synchronized to the `wrclk` (`rdclk`) clock external to the FIFO.
`2'b01`	`rdrst` (`wrrst`) resets the write (read) interface logic. The `rdrst` (`wrrst`) signal must be synchronized to the `wrclk` (`rdclk`) clock external to the FIFO.
`2'b10`	`wrrst` (`rdrst`) is ORed with the internally synchronized `rdrst` (`wrrst`) input to reset the write (read) interface logic. `wrrst` and `rdrst` must be synchronous to `wrclk` and `rdclk`, respectively.
`2'b11`	`rdrst` (`wrrst`) is internally synchronized and is then used to reset the write (read) interface logic.

Table Notes

1. For a synchronous (single clock) FIFO, the lowest reset latency is achieved by selecting reset input mode `2'b00` on both interfaces and connecting the same reset wire to both the wrrst and rdrst inputs.
2. For an asynchronous FIFO, the lowest reset latency is achieved by using a reset input that is synchronous to `wrclk`. The write pointer should be configured with a synchronous reset (`wrrst_input_mode = 2'b00`), and the read pointer should be configured to synchronize the `wrrst` input (`rdrst_sync_mode = 2'b11`). The user reset signal should be connected to `wrrst`, and `rdrst` should be de-asserted.
3. When resetting the FIFO, the reset input(s) should be held asserted for at least three clock cycles of the slowest clock domain.

The following table describes some example reset mode use cases.

Table 258: ***Reset Usage Model for wrrst and rdrst Inputs***

wrrst_input_mode	rdrst_input_mode	Description
`2'b11`	`2'b00`	For an asynchronous FIFO, a single reset in the `rdclk` domain resets both read and write pointers, with the FIFO synchronizing the write pointer logic. The user reset should be connected to the `rdrst` input. The `wrrst` signal should be de-asserted.
`2'b00`	`2'b11`	For an asynchronous FIFO, a single reset in the `wrclk` domain resets both read and write pointers, with the FIFO synchronizing the read pointer logic. The user reset should be connected to the `wrrst` input. The `rdrst` signal should be de-asserted.
`2'b11`	`2'b11`	For an asynchronous FIFO, a single asynchronous reset resets both the read and write pointers. The user reset should be connected to both the `wrrst` and `rdrst` inputs.
`2'b10`	`2'b10`	For an asynchronous FIFO, `wrrst` or `rdrst` resets both the read and write pointers. Each reset input must be synchronous to its own clock domain, and is synchronized by the FIFO for the other clock domain.
`2'b00`	`2'b00`	For a synchronous FIFO, `wrrst` resets the write pointer, and `rdrst` resets the read pointer. Both reset inputs must be synchronized externally to the FIFO single clock domain.

Error Detection and Correction

There are four modes of operation for the BRAMFIFO defined by the `encoder_enable` and `decoder_enable` parameters described in the following table. The ECC encoder and decoder can only be used if both the read width and the write width are 40.

Table 259: ***BRAMFIFO ECC Modes of Operation***

encoder_enable	decoder_enable	BRAMECC Operation Mode
`1'b0`	`1'b0`	ECC mode disabled, standard BRAMSDP operation is available.
`1'b0`	`1'b1`	ECC decode-only mode.
`1'b1`	`1'b0`	ECC encode-only mode.
`1'b1`	`1'b1`	Normal ECC encode/decode mode

ECC Encode/Decode Mode

The ECC encode/decode mode uses both the ECC encoder and the ECC decoder. 32-bit user data is written into the memory via the `din[31:0]` inputs. The ECC encoder generates the 7-bit error correction syndrome and writes it into the memory alongside the data word, using the parity (`dinp`) and extended parity (`dinpx`) bit positions. During read operations, the ECC decoder reads the 32-bit user data and the 7-bit syndrome to generate an error correction mask.

The ECC decoder corrects any single-bit error and detects, but does not correct, any dual-bit error. If the ECC decoder detects a single-bit error, it automatically corrects the error, places the corrected data on the `dout[31:0]` pins, and asserts the `sbit_error` flag. The memory location containing the error is not corrected. If the ECC decoder detects a dual-bit error, it places the uncorrected data on the `dout[31:0]` pins and asserts the `dbit_error` flag one cycle after the the data word is read.

ECC Encode-Only Mode

In the ECC encode-only mode, the ECC encoder is enabled and the ECC decoder is disabled. This mode allows writing the user 32-bit data while the 7-bit error correction syndrome is calculated by the FIFO. The syndrome is presented on `doutpx[2:0]` and `doutp[3:0]`. Read operations allow the 32-bit user data and the error syndrome to be read directly out of the memory without correcting the data. The encode-only mode can be used as a building block to provide error correction for off-chip memories.

ECC Decode-Only Mode

In the ECC decode-only mode, the ECC encoder is disabled and the ECC decoder enabled. This mode bypasses the ECC encoder and allows writing 40-bit data directly into the FIFO during write operations. Read operations use the memory `doutp[3:0]` and `doutpx[2:0]` locations as a 7-bit syndrome for error correction. The ECC decoder corrects any single-bit error and detects, but does not correct, any dual-bit error.

If the ECC decoder detects a single-bit error, it automatically corrects the error and places the corrected data on the `dout[31:0]` pins as well as asserts the `sbit_error` flag. The memory location containing the error is not corrected. If the ECC decoder detects a dual-bit error, it places the uncorrected data on the `dout[31:0]` pins and assert the `dbit_error` flag one clock cycle after the the data word is read. The decode-only mode can be used as a building block to provide error correction for off-chip memories.

Instantiation Template

Verilog

```
BRAMFIFO #(
.sync_mode(1'b0),
.read_width(40),
.write_width(40),
.fwft(1'b0),
.en_out_reg(1'b0),
.reg_initval(40'h0),
.reg_srval(40'h0),
.reg_rstval(1'b1),
.wrrst_input_mode(2'b11),
.rdrst_input_mode(2'b11),
.wrrst_rstval(1'b1),
.rdrst_rstval(1'b1),
.afull_offset(15'h4),
.aempty_offset(15'h4),
.wren_polarity_sel(1'b1),
.rden_polarity_sel(1'b1),
.encoder_enable(1'b0),
.decoder_enable(1'b0)
) instance_name (
.wrclk(user_wrclk),
.wrrst(user_wrrst),
.wren(user_wren),
.din(user_din),
.dinp(user_dinp),
.dinpx(user_dinpx),
.full(user_full),
.almost_full(user_almost_full),
.write_err(user_write_err),
.rdclk(user_rdclk),
.rdrst(user_rdrst),
.rden(user_rden),
.rstreg(user_rstreg),
.outregce(user_outregce),
.dout(user_dout),
.doutp(user_doutp),
.doutpx(user_doutpx),
.empty(user_empty),
.almost_empty(user_almost_empty),
.read_err(user_read_err),
.sbit_error(user_sbit_error),
.dbit_error(user_dbit_error)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
------------- DONE ACHRONIX LIBRARY ---------
-- Component Instantiation
instance_name : BRAMFIFO
generic map (
 sync_mode  => 0,
 read_width  => 40,
 write_width  => 40,
 fwft  => 0,
 en_out_reg  => 0,
 reg_initval  => 0,
 reg_srval  => 0,
 reg_rstval  => 1,
 wrrst_input_mode  => 11,
 rdrst_input_mode  => 11,
 wrrst_rstval  => 1,
 rdrst_rstval  => 1,
 afull_offset  => 4,
 aempty_offset  => 4,
 wren_polarity_select  => 1,
 rden_polarity_select  => 1,
 encoder_enable  => 0,
 decoder_enable  => 0)
port map (
 wrclk => user_wrclk,
 wrrst => user_wrrst,
 wren => user_wren,
 din => user_din,
 dinp => user_dinp,
 dinpx => user_dinpx,
 full => user_full,
 almost_full => user_almost_full,
 write_err => user_write_err,
 rdclk => user_rdclk,
 rdrst => user_rdrst,
 rden => user_rden,
 rstreg => user_rstreg,
 outregce => user_outregce,
 dout => user_dout,
 doutp => user_doutp,
 doutpx => user_doutpx,
 empty => user_empty,
 almost_empty => user_almost_empty,
 read_err => user_read_err,
 sbit_error => user_sbit_error,
 dbit_error => user_dbit_error
);
```

ACX_BRAM72K_SDP (72-kb Simple Dual-Port Memory with Error Correction)

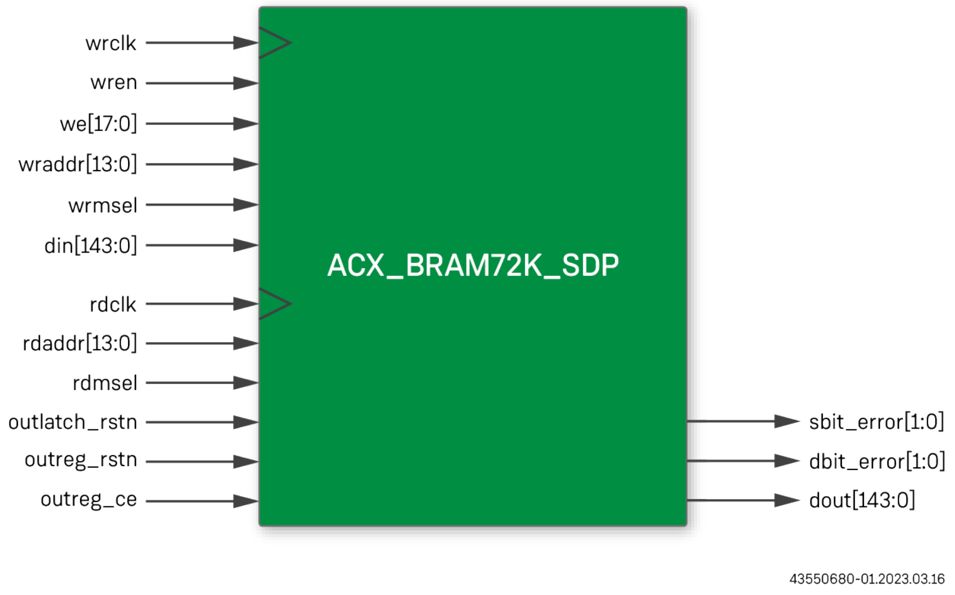


Figure 131: ***ACX_BRAM72K_SDP Logic Symbol***

The ACX_BRAM72K_SDP block RAM primitive implements a 72-Kb simple dual-port (SDP) memory block with one write port and one read port. Each port can be independently configured with respect to bit-width. Both ports can be configured as any one of 512 × 144, 512 × 128, 1024 × 72, 1024 × 64, 2048 × 36, 2048 × 32, 4096 × 18, 4096 × 16, 8192 × 9, 8192 × 8, or 16384 × 4, (depth × data width). The read and write operations are both synchronous.

For higher performance operation, an additional output register can be enabled at the cost of an additional cycle of read latency.

When writing, there is one write enable bit (`we[]`) for each 8 or 9 bits of input data, depending on the byte_width parameter.

The initial value of the memory contents may be user-specified from either parameters or a memory initialization file.

The following block diagram shows the data flow through the ECC modules, memories, and optional output registers.

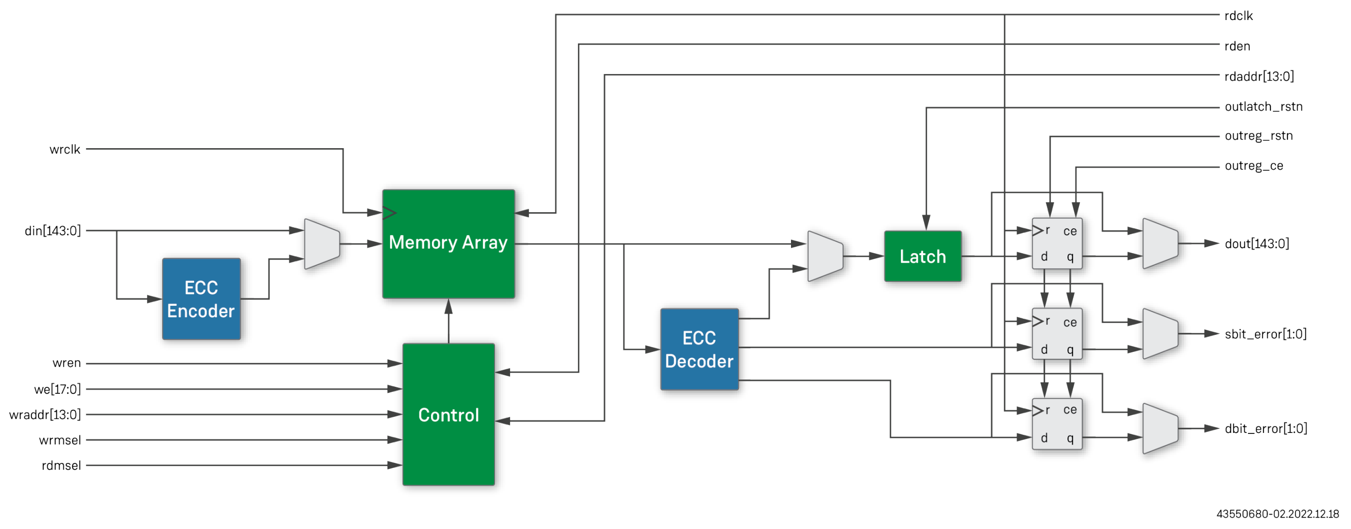


Figure 132: *ACX_BRAM72K_SDP Block Diagram*

Parameters

Table 260: *ACX_BRAM72K_SDP Parameters*

Parameter	Supported Values	Default Value	Description
`read_width`(1)	4, 8, 9, 16, 18, 32, 36, 64, 72, 128, 144	72	Data width of read port. Read port widths of 36 or narrower are not supported for `write_width` settings of 72 or 144.
`write_width`(1)	4, 8, 9, 16, 18, 32, 36, 64, 72, 128, 144	72	Data width of write port.
`rdclk_polarity`	"rise", "fall"	"rise"	Determines whether the `rdclk` signal uses the falling or rising edge: "rise" – rising edge. "fall" – falling edge.
`wrclk_polarity`	"rise", "fall"	"rise"	Determines whether the `wrclk` signal uses the falling or rising edge: "rise" – rising edge. "fall" – falling edge.
`outreg_enable`	0, 1	0	Determines whether the output register is enabled: 0 – disables the output register and results in a read latency of one cycle. 1 – enables the output register and results in a read latency of two cycles.
`outreg_sr_assertion`	"clocked", "unclocked"	"clocked"	Determines whether the assertion of the output register reset is synchronous or asynchronous with respect to the `rdclk` input. "clocked" – synchronous reset. The output register is reset upon the next rising edge of the clock when `outreg_rstn` is asserted. "unclocked" – asynchronous reset. The output register is reset immediately following the assertion of the `outreg_rstn` input.
`byte_width`(2)	8, 9	9	Determines whether the the `we[]` signal applies as 8-bit bytes or 9-bit bytes: • The `byte_width=8` setting is required for `read_width` and `write_width` settings of 4, 8, 16, 32, 64 or 128. The 144-bit `din[]` signal should be viewed as eighteen 8-bit bytes. During a write operation, `we[17:0]` selects which of the 8-bit bytes to be written, where `we[0]` implies that `din[7:0]` is written to memory, and `we[17]` implies that `din[143:136]` is written. • The `byte_width=9` setting is required for `read_width` and `write_width` settings of 9, 18 or 36. The 144-bit `din[]` signal should be viewed as sixteen 9-bit bytes. During a write operation, `we[7:0]` selects which of the lower 9-bit bytes to be written and `we[16:9]` selects which of the higher 9-bit bytes to be written, where `we[0]` implies that `din[8:0]` is written to memory, and `we[16]` implies that `din[143:135]` is written. In this mode, `we[8]` and `we[17]` are ignored.
`mem_init_file`	Path to HEX file	""	Provides a mechanism to set the initial contents of the ACX_BRAM72K_SDP memory: • If the `mem_init_file` parameter is defined, the BRAM is initialized with the values defined in the file pointed to by the `mem_init_file` parameter according to the format defined in Memory Initialization (see page 399). • If the `mem_init_file` is left at the default value of "", the initial contents are defined by the values of the `initd_0` through `initd_1023` parameters. • If the memory initialization parameters and the `mem_init_file` parameters are not defined, the contents of the BRAM remain uninitialized.
`initd_0–initd_1023`	72 bit hex number	`72'hX`	The `initd_0` through `initd_1023` parameters define the initial contents of the memory associated with `dout[71:0]` as defined in Memory Initialization (see page 399).
`ecc_encoder_enable`	0, 1	0	Determines if the ECC encoder circuitry is enabled. A value of 1 is only supported for a write width of 64 or 128: 0 – disables the ECC encoder. 1 – enables the ECC encoder such that `din[71:64]` and `din[143:136]` are ignored and bits [71:64] and [143:136] of the memory array are populated with ECC bits.

Parameter	Supported Values	Default Value	Description
`ecc_decoder_enable`	0, 1	0	Determines if the ECC decoder circuitry is enabled. A value of 1 is only supported for a read width of 64 or 128: 0 – disables the ECC decoder. 1 – enables the ECC decoder.
`read_remap`	0, 1	0	Enable read port to be remapped: 0 - disable remap. In `byte_mode=8`, the port presents up to 1024 locations. 1 - enable remap. With `read_width=4, 8, 16, 32` or `64`, when `rdmsel=1'b1` and `rdaddr[11]=1'b0`, the port presents up to 1152 locations, reading the higher order data bits as extended memory address locations. Refer to Advanced Modes (see page 405) for full details.
`write_remap`	0, 1	0	Enable write port to be remapped: 0 - disable remap. In `byte_mode=8`, the port presents up to 1024 locations. 1 - enable remap. With `write_width=4, 8, 16, 32` or `64`, when `wrmsel=1'b1` and `wraddr[11]=1'b0`, the port presents up to 1152 locations, writing the extended memory address locations to the higher order data bits. Refer to Advanced Modes (see page 405) for full details.

Table Notes

1. Setting `read_width` or `write_width` to 128 or 144 consumes the adjacent MLP site by using it as a route-through to accommodate the transfer of wide data.
2. Write and read port widths of 72 or 144 are allowed to use either `byte_width` 8 or 9.

Ports

Table 261: ***ACX_BRAM72K_SDP Pin Descriptions***

Name	Direction	Description
`wrclk`	Input	Write clock input. Write operations are fully synchronous and occur upon the active edge of the `wrclk` clock input when `wren` is asserted. The active edge of `wrclk` is determined by the `wrclk_polarity` parameter.
`wren`	Input	Write port enable. Assert `wren` high to perform a write operation.
`we[17:0]`	Input	Write enable mask. There is one bit of `we[]` for each byte of `din` (byte width can be set to either 8 or 9 bits). Asserting each `we[]` bit causes the corresponding byte of `din` to be written to memory. When using 72-bit width or smaller, only the lower 9 bits must be connected.
`wraddr[13:0]`	Input	The `wraddr` signal determines which memory location is being written to. See the following write port address and data bus mapping tables for details.
`wrmsel`	Input	Write support for advanced modes. Used in conjunction with `wraddr[11]` to set the following modes, {`wrmsel`, `wraddr[11]`}: `1'b0`, `1'bx` – normal mode. BRAM write-side operation. `1'b1`, `1'b0` – remap depth mode. 9-bit bytes remapped to 8-bit bytes. `1'b1`, `1'b1` – reserved. Refer to Advanced Modes (see page 405) for full details of the operation.
`din[143:0]`	Input	The `din` signal determines the data to write to the memory array during a write operation. See the following write port address and data bus mapping tables for details.
`rdclk`	Input	Read clock input. Read operations are fully synchronous and occur upon the active edge of the `rdclk` input when the `rden` signal is asserted. The active edge of `rdclk` is determined by the `rdclk_polarity` parameter.
`rden`	Input	Read port enable. Assert `rden` high to perform a read operation.
`rdaddr[13:0]`	Input	The `rdaddr` signal determines which memory location to read from. See the following read port address and data bus mapping tables for details.
`rdmsel`	Input	Read support for advanced modes. Used in conjunction with `rdaddr[11]` to set the following modes, {`rdmsel`, `rdaddr[11]`}: `1'b0`, `1'bx` – normal mode. BRAM read-side operation. `1'b1`, `1'b0` – remap mode. 9-bit bytes remapped to 8-bit bytes. `1'b1`, `1'b1` – reserved. Refer to Advanced Modes (see page 405) for full details of the operation.
`outlatch_rstn`	Input	Output latch synchronous reset. When `outlatch_rstn` is asserted low, the value of the output latches are reset to 0.
`outreg_rstn`	Input	Output register synchronous reset. When `outreg_rstn` is asserted low, the value of the output registers are reset to 0.
`outreg_ce`	Input	Output register clock enable (active high). When `outreg_enable=1`, de-asserting `outreg_ce` causes the BRAM to keep the `dout` signal unchanged, independent of a read operation. When `outreg_enable=0`, `outreg_ce` input is ignored.
`dout[143:0]`	Output	Read port data output. For read operations, the `dout` output is updated with the memory contents addressed by `rdaddr` if the `rden` port enable is active. See the following read port address and data bus mapping tables for details.

Name	Direction	Description
`sbit_error[1:0]`(1)	Output	Single-bit error (active high). The `sbit_error` signal is asserted during a read operation when `ecc_decoder_enable=1` and a single-bit error is detected. In this case, the corrected word is output on the `dout` pins. The memory contents are not corrected by the error correction circuitry. The `sbit_error` signal is aligned with the associated read data word. When using 64-bit width, only `sbit_error[0]` should be used. `sbit_error[1]` is unused.
`dbit_error[1:0]`(1)	Output	Dual-bit error (active high). The `dbit_error` signal is asserted during a read operation when `ecc_decoder_enable=1` and two or more bit errors are detected. In the case of two or more bit errors, the uncorrected read data word is output on the `dout` pins. The `dbit_error` signal is aligned with the associated read data word. When using 64-bit width, only `dbit_error[0]` should be used. `dbit_error[1]` is unused.

Table Notes

1. ECC modes are only applicable with read and write widths of 64 and 128 bits. In these modes, bits [71:64] and [143:136] of the memory array are used to store the ECC parity bits. If ECC is enabled with other `read_width` settings, the respective data input and output on these memory array bits are ignored. Refer to ECC Modes of Operation (see page 401) for full details of ECC operation and configuration.

Memory Organization and Data Input/Output Pin Assignments

Supported Width Combinations

The ACX_BRAM72K_SDP block supports a variety of memory width combinations, as shown in the following table.

Table 262: *ACX_BRAM72K_SDP Supported Data Widths*

	Write Data Width										
Read Data Width	**144**	**72**	**36**	**18**	**9**	**128**	**64**	**32**	**16**	**8**	**4**
144	✓	✓	✓	✓	✓		✓ (w)(1)				
72	✓	✓	✓	✓	✓		✓ (w)(1)				
36			✓	✓	✓		✓ (w)(1)				
18			✓	✓	✓		✓ (w)(1)				
9			✓	✓	✓		✓ (w)(1)				
128						✓	✓	✓	✓	✓	✓
64	✓ (r)(1)	✓ (r)(1)	✓ (r)(1)	✓ (r)(1)	✓ (r)(1)	✓	✓	✓	✓	✓	✓
32						✓	✓	✓	✓	✓	✓
16						✓	✓	✓	✓	✓	✓
8						✓	✓	✓	✓	✓	✓
4						✓	✓	✓	✓	✓	✓

Table Notes

1. Requires remap mode:
   (w) – `write_remap=1'b1`.
   (r) – `read_remap=1'b1`.

Write Data Port Usage

Table 263: ***ACX_BRAM72K_SDP Write Port Address and Data Bus Mapping***

Write Port Configuration	Data Input Assignment	Write Word Address Assignment

144 × 512	`din[143:0] <= user_din[143:0]`	`wraddr[13:5] <= user_wraddr[8:0]`
		`wraddr[4:0] <= 5'b0`

128 × 512	`din[143:136] <= 8'b0`
	`din[135:72] <= user_din[127:64]`
	`din[71:64] <= 8'b0`
	`din[63:0] <= user_din[63:0]`
		`wraddr[13:5] <= user_wraddr[8:0]`
		`wraddr[4:0] <= 5'b0`

72 × 1024	`din[143:72] <= 72'b0`
	`din[71:0] <= user_din[71:0]`
		`wraddr[13:4] <= user_wraddr[9:0]`
		`wraddr[3:0] <= 4'b0`

64 × 1024	`din[143:64] <= 80'b0`
	`din[63:0] <= user_din[63:0]`
		`wraddr[13:4] <= user_wraddr[9:0]`
		`wraddr[3:0] <= 4'b0`

36 × 2048	`din[143:36] <= 108'b0`
	`din[35:0] <= user_din[35:0]`
		`wraddr[13:3] <= user_wraddr[10:0]`
		`wraddr[2:0] <= 3'b0`

32 × 2048	`din[143:32] <= 112'b0`
	`din[31:0] <= user_din[31:0]`
		`wraddr[13:3] <= user_wraddr[10:0]`
		`wraddr[2:0] <= 3'b0`

18 × 4096	`din[143:18] <= 126'b0`
	`din[17:0] <= user_din[17:0]`
		`wraddr[13:2] <= user_wraddr[11:0]`
		`wraddr[1:0] <= 2'b0`

16 × 4096	`din[143:16] <= 128'b0`
	`din[15:0] <= user_din[15:0]`
		`wraddr[13:2] <= user_wraddr[11:0]`
		`wraddr[1:0] <= 2'b0`

9 × 8192	`din[143:9] <= 135'b0`
	`din[8:0] <= user_din[8:0]`
		`wraddr[13:1] <= user_wraddr[12:0]`
		`raddr[0] <= 1'b0`

8 × 8192	`din[143:8] <= 136'b0`
	`din[7:0] <= user_din[7:0]`
		`wraddr[13:1] <= user_wraddr[12:0]`
		`wraddr[0] <= 1'b0`

4 × 16384	`din[143:4] <= 140'b0`
	`din[3:0] <= user_din[3:0]`
		`wraddr[13:0] <= user_wraddr[13:0]`

Table 264: ***ACX_BRAM72K_SDP Read Port Address and Data Bus Mapping***

Read Port Configuration	Data Output Assignment	Read Word Address Assignment

144 × 512	`user_dout[143:0] <= dout[143:0]`	`rdaddr[13:5] <= user_rdaddr[8:0]`
		`rdaddr[4:0] <= 5'b0`

128 × 512	`user_dout[127:64] <= dout[135:72]`
	`user_dout[63:0] <= dout[63:0]`
		`rdaddr[13:5] <= user_rdaddr[8:0]`
		`rdaddr[4:0] <= 5'b0`

72 × 1024	`user_dout[72:0] <= dout[72:0]`	`rdaddr[13:4] <= user_rdaddr[9:0]`
		`rdaddr[3:0] <= 4'b0`

64 × 1024	`user_dout[63:0] <= dout[63:0]`	`rdaddr[13:4] <= user_rdaddr[9:0]`
		`rdaddr[3:0] <= 4'b0`

36 × 2048 (1)	`user_dout[35:0] <= dout[35:0]`	`rdaddr[13:3] <= user_rdaddr[10:0]`
		`rdaddr[2:0] <= 3'b0`

32 × 2048 (1)	`user_dout[31:0] <= dout[31:0]`	`rdaddr[13:3] <= user_rdaddr[10:0]`
		`rdaddr[2:0] <= 3'b0`

18 × 4096 (1)	`user_dout[17:0] <= dout[17:0]`	`rdaddr[13:2] <= user_rdaddr[11:0]`
		`rdaddr[1:0] <= 2'b0`

16 × 4096 (1)	`user_dout[15:0] <= dout[15:0]`	`rdaddr[13:2] <= user_rdaddr[11:0]`
		`rdaddr[1:0] <= 2'b0`

9 × 8192 (1)	`user_dout[8:0] <= dout[8:0]`	`rdaddr[13:1] <= user_rdaddr[12:0]`
		`rdaddr[0] <= 1'b0`

8 × 8192 (1)	`user_dout[7:0] <= dout[7:0]`	`rdaddr[13:1] <= user_rdaddr[12:0]`
		`rdaddr[0] <= 1'b0`

4 × 16384 (1)	`user_dout[3:0] <= dout[3:0]`	`rdaddr[13:0] <= user_rdaddr[13:0]`

Table Notes

1. Not supported for `write_width` setting of 72 or 144 because `read_width` is 36 bits or less.

Read and Write Operations

Timing Options

The ACX_BRAM72K_SDP has two options for interface timing, controlled by the outreg_enable parameter:

- Latched mode – when `outreg_enable=0`, the port is in latched mode. In latched mode, the read address is registered and the stored data is latched into the output latches on the following clock cycle providing a read operation with one cycle of latency.
- Registered mode – when `outreg_enable=1`, the port is in registered mode. In registered mode, there is an additional register after the latch to support higher-frequency designs providing a read operation with two cycles of latency.

Read Operation

Read operations are signaled by driving the `rdaddr[]` signal with the address to be read and asserting the `rden` signal. The requested read data arrives on the `dout[]` signal on the following clock cycle or the cycle after depending on the `outreg_enable` parameter value.

Table 265: ***ACX_BRAM72K_SDP Latched Mode Output Function Table***

Operation	`rdclk`	`outlatch_rstn`	`rden`	`dout[]`
Reset latch	↑	0	X	0
Hold	↑	1	0	Hold previous value
Read	↑	1	1	`mem[rdaddr]`

Table Notes

- Operation assumes rising-edge clock and active-high port enable, otherwise previous value is held.

Table 266: ***ACX_BRAM72K_SDP Registered Mode Output Function Table***

Operation	`rdclk`	`outreg_rstn`	`outregce`	`dout[]`
Reset Output	↑	0	1	0
Hold	↑	1	0	Previous `dout[]`
Update Output	↑	1	1	Registered from latch output

Table Notes

- Operation assumes active-high clock, output register clock enable, and output register reset, otherwise previous `dout[]` is held.

Write Operation

Write operations are signaled by asserting the `wren` signal. The value of the `din[]` signal is stored in the memory array at the address indicated by the `wraddr[]` signal on the next active clock edge.

Simultaneous Memory Operations

Memory operations may be performed simultaneously from both sides of the memory. However, there is a restriction regarding memory collisions. A memory collision is defined as the condition where both ports access the same memory location(s) within the same clock cycle (both ports connected to the same clock), or within a fixed time window (if each port is connected to a different clock). If one of the ports is writing an address while the other port is reading the same address (qualified with overlapping write enables per bit), the write operation takes precedence, but the read data is invalid. The data may be reliably read on the next cycle if there is no longer a write collision.

Timing Diagrams

The following timing diagram illustrates the behavior of a ACX_BRAM72K_SDP instance with the output register both disabled and enabled via the `outreg_enable` parameter.

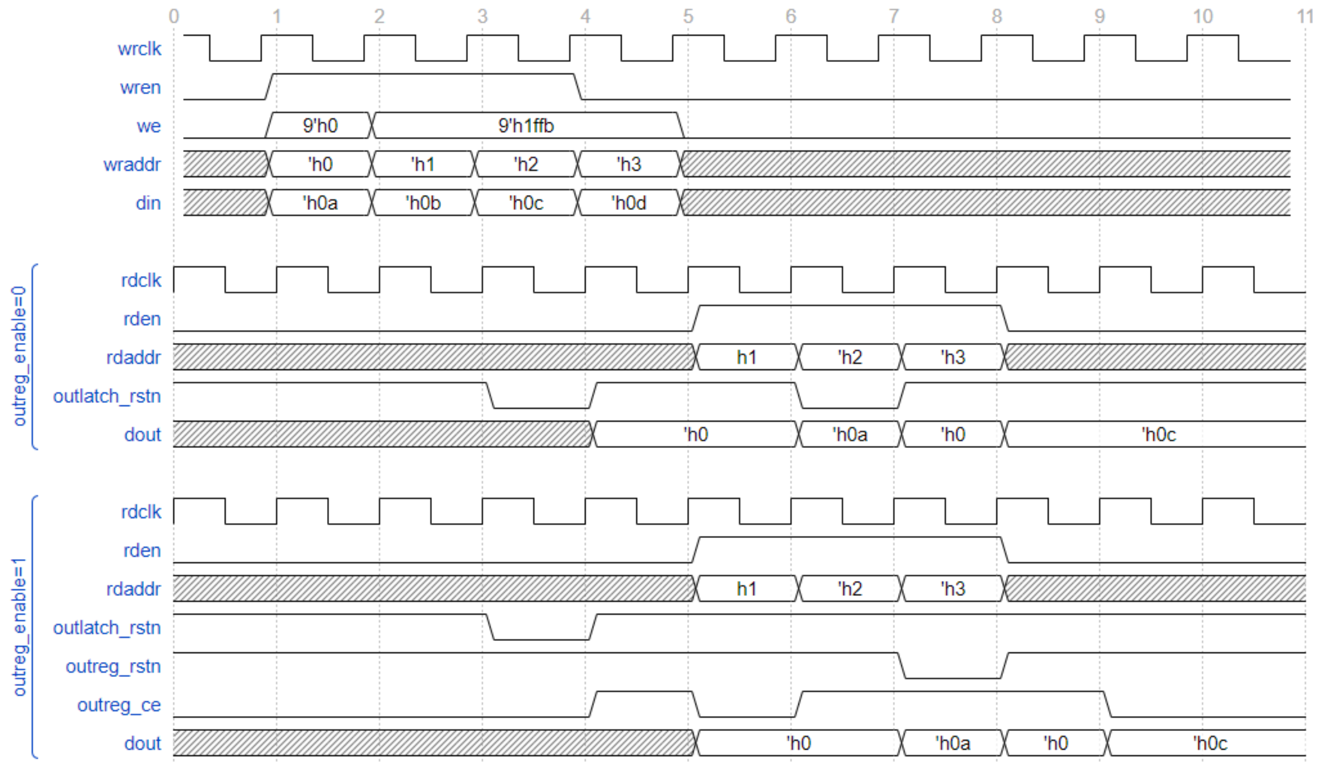


Figure 133: *ACX_BRAM72K_SDP Timing Diagram*

The behavior of the ACX_BRAM72K_SDP on each clock cycle of the preceding diagram is described in the following table.

Table 267: *ACX_BRAM72K_SDP Timing Diagram Events*

Event	Transaction	Description
Write Clock		
1	No-Op	`wren` is asserted but `we` is not asserted. Nothing is written to the memory array.
2–4	Write	`wren` and `we` are both asserted. Data on `din[]` is committed to the `wraddr[]` location.
Read clock		
4	Read Reset latch	`outlatch_rstn` is asserted, causing the output of the latch to be set to 0. `outreg_enable = 0` – the data is reset to zero on the following cycle. `outreg_enable = 1` – the output of the latch is reset to zero on the following cycle. The value is visible at the output of the memory on the second cycle because `outreg_ce` is asserted.
6	Read	`rden` is asserted. The memory is read from the memory array. `outreg_enable = 0` – the value is output on the following cycle. `outreg_enable = 1` – the value is output two cycles later, because `outreg_ce` is asserted on the next cycle.
7	Read with latch/register reset	`rden` is asserted. The memory is read from the memory array. `outreg_enable = 0` – `dout[]` is set to 0 since `outlatch_rstn` is asserted. `outreg_enable = 1` – `dout[]` is set to 0 after two cycles since `outreg_rstn` is asserted on the following cycle.
8	Read	`rden` is asserted. The memory is read from the memory array. `outreg_enable = 0` – the value is output on the following cycle. `outreg_enable = 1` – the value is output two cycles later, because `outreg_ce` is asserted on the next cycle.
7–8	Read	`rden` is asserted. The memory is read from the memory array and presented on `dout[]` on the following cycle.
8–9	Hold	`rden` and `outlatch_rstn` are both de-asserted. `dout[]` retains its previous value.

Memory Initialization

Initializing with Parameters

The data portion of initial memory contents may be defined by setting the 1024 72-bit parameters `initd_0` through `initd_1023`. The data memory is organized as little-endian with bit zero mapped to bit zero of parameter `initd_0` and bit 73727 mapped to bit 71 of parameter `initd_1023`.

Initializing with Memory Initialization File

A ACX_BRAM72K_SDP may alternatively be initialized with a memory file by setting the `mem_init_file` parameter to the path of a memory initialization file. The file format must be hexadecimal entries separated by white space where the white space is defined by spaces or line separation. Each number is a hexadecimal number of width equal to 72 bits.

The ACX_BRAM72K_SDP memory organization is configured with the `byte_width` parameter as either `byte_width=8` or `byte_width=9`. For read and write data widths, the `mem_init_file` contains 1024 lines with 72 bits of init data per line, organized as follows:

Table 268: ***9-bit Byte Mode (byte_width == 9)***

		Bits							
Line in `mem_init_file`	**Corresponding `initd_*` Parameter**	**71:63**	**62:54**	**53:45**	**44:36**	**35:27**	**26:18**	**17:9**	**8:0**
1st line	`initd_0`	9byte7	9byte6	9byte5	9byte4	9byte3	9byte2	9byte1	9byte0
2nd line	`initd_1`	9byte15	9byte14	9byte13	9byte12	9byte11	9byte10	9byte9	9byte8
...	...	...	...	...	...	...	...	...	...
1024th line	`initd_1023`	9byte8191	9byte8190	9byte8189	9byte8188	9byte8187	9byte8186	9byte8185	9byte8184

Table 269: ***8-bit Byte Mode (byte_width == 8)***

		Bits							
Line in `mem_init_file`	**Corresponding `initd_*` Parameter**	**71**	**70:63**	**62**	**61:54**	**53**	**52:45**	**44**	**43:36**
		35	**34:27**	**26**	**25:18**	**17**	**16:9**	**8**	**7:0**
1st line	`initd_0`	`1'b0`	byte7	`1'b0`	byte6	`1'b0`	byte5	`1'b0`	byte4
		`1'b0`	byte3	`1'b0`	byte2	`1'b0`	byte1	`1'b0`	byte0
2nd line	`initd_1`	`1'b0`	byte15	`1'b0`	byte14	`1'b0`	byte13	`1'b0`	byte12
		`1'b0`	byte11	`1'b0`	byte10	`1'b0`	byte9	`1'b0`	byte8
...	...	...	...	...	...	...	...	...	...
1024th line	`initd_1023`	`1'b0`	byte8191	`1'b0`	byte8190	`1'b0`	byte8189	`1'b0`	byte8188
		`1'b0`	byte8187	`1'b0`	byte8186	`1'b0`	byte8185	`1'b0`	byte8184

Table 270: ***8-bit Byte Mode (byte_width == 8, write_width is 72 or 144)***

Line in mem_init_file	Corresponding initd_* Parameter	Bits							
		71	70:63	62	61:54	53	52:45	44	43:36
		35	34:27	26	25:18	17	16:9	8	7:0
1st line	initd_0	byte8[7]	byte7	byte8[6]	byte6	byte8[5]	byte5	byte8[4]	byte4
		byte8[3]	byte3	byte8[2]	byte2	byte8[1]	byte1	byte8[0]	byte0
2nd line	initd_1	byte17[7]	byte16	byte17[6]	byte15	byte17[5]	byte14	byte17[4]	byte13
		byte17[3]	byte12	byte17[2]	byte11	byte17[1]	byte10	byte17[0]	byte9
...	...	...	...	...	...	...	...	...	...
1024th line	initd_1023	byte9215[7]	byte9214	byte9215[6]	byte9213	byte9215[5]	byte9212	byte9215[4]	byte9211
		byte9215[3]	byte9210	byte9215[2]	byte9209	byte9215[1]	byte9208	byte9215[0]	byte9207

A number entry can contain underscore (_) characters among the digits, for example, A234_4567_33. Commenting is allowed following a double-slash (//) through to the end of the line. C-like commenting is also allowed where the characters between the /* and */ are ignored. The memory is initialized starting with the first entry of the file initializing the memory array starting with address zero, moving upward.

If `mem_init_file` is defined, the ACX_BRAM72K_SDP is initialized with the values in the file referenced by the `mem_init_file` parameter. If the `mem_init_file` paramter is left at the default value of "", the initial contents are defined by the values of the `initd_0` through `initd_1023` parameters. If neither the memory initialization parameters nor the `mem_init_file` parameters are defined, the contents of a BRAM remain uninitialized and the contents are unknown until the memory locations are written.

ECC Modes of Operation

There are four modes of operation for a ACX_BRAM72K_SDP defined by the `enable_ecc_encoder` and `enable_ecc_decoder` parameters shown in the table below.

Table 271: ***ACX_BRAM72K_SDP ECC Modes of Operation***

enable_ecc_encoder	enable_ecc_decoder	ECC Operation Mode
0	0	ECC encoder and decoder disabled. Standard ACX_BRAM72K_SDP operation available.
0	1	ECC decode-only mode. Applies only to `read_width` of 64 or 128.
1	0	ECC encode-only mode. Applies only to `write_width` of 64 or 128.
1	1	Normal ECC encode/decode mode. Applies only to `read_width` and `write_width` of 64 or 128.

ECC Encode/Decode Operation Mode

The ECC encode/decode operation mode utilizes both the ECC encoder and the ECC decoder. The 64-bit user data is written into a ACX_BRAM72K_SDP via the `din[63:0]` inputs. The ECC encoder generates the 8-bit error correction syndrome and writes it into the memory array bits `[71:64]`. During read operations, the ECC decoder reads the 64-bit user data and the 8-bit syndrome data to generate an error correction mask. The ECC decoder corrects any single-bit error and only detects, but does not correct, any dual-bit error.

If the ECC decoder detects a single-bit error, it corrects the error and places the corrected data on the `dout[63:0]` pins and asserts the `sbit_error` output. The memory location containing the error is not corrected.

If the ECC decoder detects a dual-bit error, it places the uncorrected data on the `dout[63:0]` pins and asserts the `dbit_error` output. The `sbit_error` and `dbit_error` outputs are asserted aligned with the output data.

ECC Encode-Only Operation Mode

The ECC encode-only operation has the ECC encoder enabled and the ECC decoder disabled. This mode allows writing 64 bits of data with the 8-bit error correction syndrome automatically written to bits `[71:64]` of the memory array during write operations. Read operations provide the 64-bit user data and the error syndrome without correcting the data. Encode-only mode can be used as a building block to provide error correction for off-chip memories.

ECC Decode-Only Operation Mode

The ECC decode-only operation has the ECC encoder disabled and the ECC decoder enabled. This mode bypasses the ECC encoder and allows writing 72-bit data directly into the memory array during write operations. If the ECC decoder detects a single-bit error, it corrects the error and places the corrected data on the `dout[63:0]` pins and asserts the `sbit_error` output. The memory location containing the error is not corrected. If the ECC decoder detects a dual-bit error, it places the uncorrected data on the `dout[63:0]` pins and asserts the `dbit_error` output one cycle after the the data word is read. For read operations in this mode, `dout [71:64]` is unknown. Decode-only mode can be used as a building block to provide error correction for off-chip memories.

Additional Requirements for ECC Mode With ACE GUI Memory Generator

When initializing memory with the ACE GUI Memory Generator, there are additional requirements when ECC mode is enabled:

1. If the **Enable ECC Encoder** box is checked, the `write_width`/`read_width` parameters must be 64 or 128.
2. If the **Enable ECC Encoder** box is checked and the **Memory Initialization File** is defined, each line of the memory initialization file must be:
   - 72 bits if the `write_width` is 64 (8 bits of parity and bits `[63:0]` of data)
   - 144 bits if the `write_width` is 128 (8 bits of parity and bits `[127:64]` of data, or 8 bits of parity and bits `[63:0]` of data)

   If it is chosen to initialize the memory, not only must the data bits be initialized, but the parity bits must also be assigned. The parity information is required in the memory initialization file so that if the initialized values are read from memory, the error flags are not set. Eight parity bits are required to be generated for each 64 bits of user data, and must be placed in the top eight bits of each 72-bit segment of the initialization words.

   - If `write_width` = 64, eight parity bits are assigned to `mem_init_word` bits `[71:64]`, generated from `user_initialization` bits `[63:0]`
   - If `write_width` = 128, eight parity bits are assigned to `mem_init_word` bits `[71:64]`, generated from `user_initialization` bits `[63:0]`, and eight parity bits are assigned to `mem_init_word` bits `[143:136]`, generated from `user_initialization` bits `[127:64]`

   The parity bits are generated according to the following Verilog equations.

Note

The same parity equations are used for each segment of 64 `user_initialization` bits. The `i_din[]` references are with respect to the index into each 64-bit data segment.

ECC Parity Equations

```
assign parity[0] = i_din[ 0] ^ i_din[ 1] ^ i_din[ 3] ^ i_din[ 4] ^ i_din[ 6] ^ i_din[ 8] ^ i_din
[10] ^ i_din[11] ^ i_din[13] ^ i_din[15] ^ i_din[17] ^ i_din[19] ^ i_din[21] ^ i_din[23] ^ i_din
[25] ^ i_din[26] ^ i_din[28] ^ i_din[30] ^ i_din[32] ^ i_din[34] ^ i_din[36] ^ i_din[38] ^ i_din
[40] ^ i_din[42] ^ i_din[44] ^ i_din[46] ^ i_din[48] ^ i_din[50] ^ i_din[52] ^ i_din[54] ^ i_din
[56] ^ i_din[57] ^ i_din[59] ^ i_din[61] ^ i_din[63];

assign parity[1] = i_din[ 0] ^ i_din[ 2] ^ i_din[ 3] ^ i_din[ 5] ^ i_din[ 6] ^ i_din[ 9] ^ i_din
[10] ^ i_din[12] ^ i_din[13] ^ i_din[16] ^ i_din[17] ^ i_din[20] ^ i_din[21] ^ i_din[24] ^ i_din
[25] ^ i_din[27] ^ i_din[28] ^ i_din[31] ^ i_din[32] ^ i_din[35] ^ i_din[36] ^ i_din[39] ^ i_din
[40] ^ i_din[43] ^ i_din[44] ^ i_din[47] ^ i_din[48] ^ i_din[51] ^ i_din[52] ^ i_din[55] ^ i_din
[56] ^ i_din[58] ^ i_din[59] ^ i_din[62] ^ i_din[63];

assign parity[2] = i_din[ 1] ^ i_din[ 2] ^ i_din[ 3] ^ i_din[ 7] ^ i_din[ 8] ^ i_din[ 9] ^ i_din
[10] ^ i_din[14] ^ i_din[15] ^ i_din[16] ^ i_din[17] ^ i_din[22] ^ i_din[23] ^ i_din[24] ^ i_din
[25] ^ i_din[29] ^ i_din[30] ^ i_din[31] ^ i_din[32] ^ i_din[37] ^ i_din[38] ^ i_din[39] ^ i_din
[40] ^ i_din[45] ^ i_din[46] ^ i_din[47] ^ i_din[48] ^ i_din[53] ^ i_din[54] ^ i_din[55] ^ i_din
[56] ^ i_din[60] ^ i_din[61] ^ i_din[62] ^ i_din[63];

assign parity[3] = i_din[ 4] ^ i_din[ 5] ^ i_din[ 6] ^ i_din[ 7] ^ i_din[ 8] ^ i_din[ 9] ^ i_din
[10] ^ i_din[18] ^ i_din[19] ^ i_din[20] ^ i_din[21] ^ i_din[22] ^ i_din[23] ^ i_din[24] ^ i_din
[25] ^ i_din[33] ^ i_din[34] ^ i_din[35] ^ i_din[36] ^ i_din[37] ^ i_din[38] ^ i_din[39] ^ i_din
[40] ^ i_din[49] ^ i_din[50] ^ i_din[51] ^ i_din[52] ^ i_din[53] ^ i_din[54] ^ i_din[55] ^ i_din
[56];

assign parity[4] = i_din[11] ^ i_din[12] ^ i_din[13] ^ i_din[14] ^ i_din[15] ^ i_din[16] ^ i_din
[17] ^ i_din[18] ^ i_din[19] ^ i_din[20] ^ i_din[21] ^ i_din[22] ^ i_din[23] ^ i_din[24] ^ i_din
[25] ^ i_din[41] ^ i_din[42] ^ i_din[43] ^ i_din[44] ^ i_din[45] ^ i_din[46] ^ i_din[47] ^ i_din
[48] ^ i_din[49] ^ i_din[50] ^ i_din[51] ^ i_din[52] ^ i_din[53] ^ i_din[54] ^ i_din[55] ^ i_din
[56];

assign parity[5] = i_din[26] ^ i_din[27] ^ i_din[28] ^ i_din[29] ^ i_din[30] ^ i_din[31] ^ i_din
[32] ^ i_din[33] ^ i_din[34] ^ i_din[35] ^ i_din[36] ^ i_din[37] ^ i_din[38] ^ i_din[39] ^ i_din
[40] ^ i_din[41] ^ i_din[42] ^ i_din[43] ^ i_din[44] ^ i_din[45] ^ i_din[46] ^ i_din[47] ^ i_din
[48] ^ i_din[49] ^ i_din[50] ^ i_din[51] ^ i_din[52] ^ i_din[53] ^ i_din[54] ^ i_din[55] ^ i_din
[56];

assign parity[6] = i_din[57] ^ i_din[58] ^ i_din[59] ^ i_din[60] ^ i_din[61] ^ i_din[62] ^ i_din
[63];

assign parity[7] = i_din[ 0] ^ i_din[ 1] ^ i_din[ 2] ^ i_din[ 3] ^ i_din[ 4] ^ i_din[ 5] ^ i_din[
6] ^ i_din[ 7] ^ i_din[ 8] ^ i_din[ 9] ^ i_din[10] ^ i_din[11] ^ i_din[12] ^ i_din[13] ^ i_din
[14] ^ i_din[15] ^ i_din[16] ^ i_din[17] ^ i_din[18] ^ i_din[19] ^ i_din[20] ^ i_din[21] ^ i_din
[22] ^ i_din[23] ^ i_din[24] ^ i_din[25] ^ i_din[26] ^ i_din[27] ^ i_din[28] ^ i_din[29] ^ i_din
[30] ^ i_din[31] ^ i_din[32] ^ i_din[33] ^ i_din[34] ^ i_din[35] ^ i_din[36] ^ i_din[37] ^ i_din
[38] ^ i_din[39] ^ i_din[40] ^ i_din[41] ^ i_din[42] ^ i_din[43] ^ i_din[44] ^ i_din[45] ^ i_din
[46] ^ i_din[47] ^ i_din[48] ^ i_din[49] ^ i_din[50] ^ i_din[51] ^ i_din[52] ^ i_din[53] ^ i_din
[54] ^ i_din[55] ^ i_din[56] ^ i_din[57] ^ i_din[58] ^ i_din[59] ^ i_din[60] ^ i_din[61] ^ i_din
[62] ^ i_din[63] ^ parity[0] ^ parity[1] ^ parity[2] ^ parity[3] ^ parity[4] ^ parity[5] ^ parity
[6];
```

Parity `[7:0]` is user-assigned to either `mem_init_word[143:136]` or `mem_init_word[71:64]` depending on the specific 64-bit group of `user_initialization` bits.

Note

The `byte_en` inputs are ignored when the **Enable ECC Encoder** box is checked.

Using ACX_BRAM72K_SDP as a Read-Only Memory (ROM)

The ACX_BRAM72K_SDP macro can be used as a read-only memory (ROM) by providing memory initialization data via a file or parameters (as described in Memory Initialization (see page 399)) and tying the `wren` signal to its de-asserted value. All signals on the read-side of the ACX_BRAM72K_SDP operate as described above. This configuration allows the reading from the memory, but not writing to it.

Advanced Modes

The ACX_BRAM72K_SDP supports two advanced modes that allow for remapping of the address space within the memory to be accessed when in 8-bit byte mode and, additionally, for control of the tightly-coupled LRAM within the ACX_MLP72, (refer to ACX_MLP72 LRAM).

The advanced modes are enabled in the read and write sides by asserting the `wrmsel` and `rdmsel` inputs respectively. When asserted, `wrmsel` and `rdmsel` are combined with `wraddr[11]` and `rdaddr[11]` respectively to configure the write and read side advanced mode.

Remap Mode

(`wrmsel/rdmsel=1'b1`, `wraddr[11]/rdaddr[11]=1'b0`)

The ACX_BRAM72K_SDP is natively configured as a 72x1024 bit memory, with 9-bit bytes. However, access to the memory using traditional 8-bit byte access might be required, for example, when transferring data to and from the NAPs or directly with the interface IP, the majority of which is configured for 8-bit bytes. In order to assist with the conversion between these two formats, the ACX_BRAM72K_SDP uniquely offers a remap mode which allows either of the two ports to operate in an 8-bit byte mode, but with the ability to still access the full memory contents. This is achieved by the memory presenting an extended addressing depth, the extra 128 addresses contain the memory content from the higher bits of the 72 bit memory array. In this mode, the memory supports 1024 + 128 = 1152 addresses at 64-bit width.

Note

If 8-bit byte mode is required for both ports, the memory can be conventionally configured using the `read_width` and `write_width` parameters set to either 4, 8, 16, 32 or 64. However, in this mode, the extended addresses are not available and the memory only supports a maximum depth of 1024 words.

To enable the remap mode for either port, the respective parameter, `write_remap` and `read_remap` must be set to `1'b1`.

With the appropriate parameter enabled, `wrmsel/rdmsel=1'b1`, and `wraddr[11]/rdaddr[11]=1'b0`, the relevant ACX_BRAM72K_SDP port operates as a 1152 x 64-bit memory. This mode remaps the extra data bits between the full width of 72 bits and the reduced width of 64 bits, and arranging them as extended memory locations. With `wraddr[11]/rdaddr[11]` set to `1'b0`, the further address bits `wraddr[10:4]/rdaddr[10:4]` are used to access the additional 128 words of memory.

Note

(`wrmsel/rdmsel=1'b1`, `wraddr/rdaddr[11]=1'b1`) is a reserved mode and not supported by ACX_BRAM72K_SDP.

Inference

The ACX_BRAM72K_SDP is inferrable using RTL constructs commonly used to infer synchronous and RAMs and ROMs, with a variety of clock enable and reset schemes and polarities. The ECC functionality is not inferrable. All control inputs can be inferred as active low by placing an inverter in the netlist before the control input.

To ensure a BRAM is inferred, as opposed to an LRAM, use the following synthesis attributes in the memory declaration.

Verilog

```
// Infer BRAM memory array. Will create memory using ACX_BRAM72K_SDP set to a maximum width of 72-
bit
logic [DATA_WIDTH-1:0] mem [(2**ADDR_WIDTH)-1:0] /* synthesis syn_ramstyle = "block_ram" */;


// Alternatively infer wide BRAM memory array with ACX_BRAM72K_SDP primitives set to 144-bit width
logic [DATA_WIDTH-1:0] mem [(2**ADDR_WIDTH)-1:0] /* synthesis syn_ramstyle = "large_ram" */;
```

Example Template

```
//--------------------------------------------------------------------------------
//
// Copyright (c) 2021 Achronix Semiconductor Corp.
// All Rights Reserved.
//
// This Software constitutes an unpublished work and contains
// valuable proprietary information and trade secrets belonging
// to Achronix Semiconductor Corp.
//
// Permission is hereby granted to use this Software including
// without limitation the right to copy, modify, merge or distribute
// copies of the software subject to the following condition:
//
// The above copyright notice and this permission notice shall
// be included in in all copies of the Software.
//
// The Software is provided “as is” without warranty of any kind
// expressed or implied, including but not limited to the warranties
// of merchantability fitness for a particular purpose and non-infringement,
// in no event shall the copyright holder be liable for any claim,
// damages, or other liability for any damages or other liability,
// whether an action of contract, tort or otherwise, arising from,
// out of or in connection with the Software
//
//
//--------------------------------------------------------------------------------
// Design:  SDP memory inference
//          Decides between BRAM and LRAM based on the requested size
//          Restriction that read and write ports must be of the same dimensions
//--------------------------------------------------------------------------------

`timescale 1ps / 1ps
```

```
module sdpram_infer
#(
    parameter       ADDR_WIDTH     = 0,
    parameter       DATA_WIDTH     = 0,
    parameter       OUT_REG_EN     = 0,
    parameter       INIT_FILE_NAME = ""
)
(
    // Clocks and resets
    input  wire                     wr_clk,
    input  wire                     rd_clk,

    // Enables
    input  wire                     we,
    input  wire                     rd_en,
    input  wire                     rstreg,

    // Address and data
    input  wire [ADDR_WIDTH-1:0]    wr_addr,
    input  wire [ADDR_WIDTH-1:0]    rd_addr,
    input  wire [DATA_WIDTH-1:0]    wr_data,

    // Output
    output reg  [DATA_WIDTH-1:0]    rd_data
);

    // Determine if size is small enough for an LRAM
    localparam MEM_LRAM = ( ((DATA_WIDTH <= 36)  && (ADDR_WIDTH <= 6)) ||
                            ((DATA_WIDTH <= 72)  && (ADDR_WIDTH <= 5)) ||
                            ((DATA_WIDTH <= 144) && (ADDR_WIDTH <= 4)) ) ? 1 : 0;

    localparam WIDE_BRAM = (DATA_WIDTH > 72) ? 1 : 0;

    // Define combinatorial and registered outputs from memory array
    logic [DATA_WIDTH-1:0]  rd_data_int;
    logic [DATA_WIDTH-1:0]  rd_data_reg;
    logic                   read_collision;
    always @(posedge rd_clk)
        if (~rstreg)
            rd_data_reg <= {DATA_WIDTH{1'b0}};
        else
            rd_data_reg <= rd_data_int;

    // Need a generate block to apply the appropriate syn_ramstyle to the memory array
    // Rest of the the code has to be within the generate block to access that variable
    generate if ( MEM_LRAM == 1) begin : gb_lram

        logic [DATA_WIDTH-1:0] mem [(2**ADDR_WIDTH)-1:0] /* synthesis syn_ramstyle = "logic" */;

        // If an initialisation file exists, then initialise the memory
        if ( INIT_FILE_NAME != "" ) begin : gb_init
            initial
                $readmemh( INIT_FILE_NAME, mem );
        end

        // Writing.  Inference does not currently support byte enables
        // Also generate the signals to detect if there is a memory collision
        logic [ADDR_WIDTH-1:0]  wr_addr_d;
        always @(posedge wr_clk)
```

```
            if( we ) begin
                mem[wr_addr] <= wr_data;
                wr_addr_d    <= wr_addr;
            end

        // LRAM only supports the WRITE_FIRST mode.  So if rd_addr = wr_addr then
        // write takes priority and read value is invalid
        // The value from the array is combinatorial, (this is different than for BRAM)
        // Write address is effective on the cycle it is writing to the memory, (i.e. it is
registered)
        assign read_collision = (wr_addr_d == rd_addr);

        assign rd_data_int = (read_collision) ? {DATA_WIDTH{1'bx}} : mem[rd_addr];

    end
    else if ( WIDE_BRAM == 1 ) begin : gb_wide_bram

        logic [DATA_WIDTH-1:0] mem [(2**ADDR_WIDTH)-1:0] /* synthesis syn_ramstyle = "large_ram"
*/;

        // If an initialisation file exists, then initialise the memory
        if ( INIT_FILE_NAME != "" ) begin : gb_init
            initial
                $readmemh( INIT_FILE_NAME, mem );
        end

        // Writing.  Inference does not currently support byte enables
        always @(posedge wr_clk)
            if( we )
            begin
                mem[wr_addr] <= wr_data;
            end

        // BRAM supports WRITE_FIRST mode only, (write takes precedence over read)
        // Calculate if there will be a collision
        // write takes priority and read value is invalid
        // Both wr_addr and rd_addr have registered operations on the memory array
        assign read_collision = (wr_addr == rd_addr) && we;

        always @(posedge rd_clk)
            if( rd_en )
            begin
                // Read collisions cannot be modelled in synthesis, so use solely in simulation
                // synthesis synthesis_off
                if( read_collision )
                    rd_data_int <= {ADDR_WIDTH{1'bx}};
                else
                // synthesis synthesis_on
                    rd_data_int <= mem[rd_addr];
            end
    end
    else
    begin : gb_bram

        logic [DATA_WIDTH-1:0] mem [(2**ADDR_WIDTH)-1:0] /* synthesis syn_ramstyle = "block_ram"
*/;

        // If an initialisation file exists, then initialise the memory
        if ( INIT_FILE_NAME != "" ) begin : gb_init
```

```
            initial
                $readmemh( INIT_FILE_NAME, mem );
        end

        // Writing.  Inference does not currently support byte enables
        always @(posedge wr_clk)
            if( we )
            begin
                mem[wr_addr] <= wr_data;
            end

        // BRAM supports WRITE_FIRST mode only, (write takes precedence over read)
        // Calculate if there will be a collision
        // write takes priority and read value is invalid
        // Both wr_addr and rd_addr have registered operations on the memory array
        assign read_collision = (wr_addr == rd_addr) && we;

        always @(posedge rd_clk)
            if( rd_en )
            begin
                // Read collisions cannot be modelled in synthesis, so use solely in simulation
                // synthesis synthesis_off
                if( read_collision )
                    rd_data_int <= {ADDR_WIDTH{1'bx}};
                else
                // synthesis synthesis_on
                    rd_data_int <= mem[rd_addr];
            end
    end
    endgenerate

    // Select output based on whether output register is enabled
    assign rd_data = (OUT_REG_EN) ? rd_data_reg : rd_data_int;

endmodule : sdpram_infer
```

Instantiation Template

Verilog

```
ACX_BRAM72K_SDP #(
    .byte_width             (        9),
    .read_width             (       72),
    .write_width            (       72),
    .rdclk_polarity         (   "rise"),
    .wrclk_polarity         (   "rise"),
    .read_remap             (        0),
    .write_remap            (        0),
    .outreg_enable          (        1),
    .outreg_sr_assertion    ("clocked"),
    .ecc_encoder_enable     (        0),
    .ecc_decoder_enable     (        0),
    .mem_init_file          (       ""),
    .initd_0                (        0),
    <...>
    .initd_1023             (        0)
) instance_name (
    .wrclk                  (user_wrclk         ),
    .din                    (user_din           ),
    .we                     (user_we            ),
    .wren                   (user_wren          ),
    .wraddr                 (user_wraddr        ),
    .wrmsel                 (user_wrmsel        ),
    .rdclk                  (user_rdclk         ),
    .rden                   (user_rden          ),
    .rdaddr                 (user_rdaddr        ),
    .rdmsel                 (user_rdmsel        ),
    .outlatch_rstn          (user_outlatch_rstn ),
    .outreg_rstn            (user_outreg_rstn   ),
    .outreg_ce              (user_outreg_ce     ),
    .dout                   (user_dout          ),
    .sbit_error             (user_sbit_error    ),
    .dbit_error             (user_dbit_error    )
);
```

VHDL

```
-- VHDL Component template for ACX_BRAM72K_SDP
component ACX_BRAM72K_SDP is
generic (
    byte_width                : integer := 9;
    ecc_decoder_enable        : integer := 0;
    ecc_encoder_enable        : integer := 0;
    initd_0                   : integer := X"x";
    <...>
    initd_1023                : integer := X"x";
    mem_init_file             : string := "";
    outreg_enable             : integer := 0;
    outreg_sr_assertion       : string := "clocked";
    rdclk_polarity            : string := "rise";
    read_remap                : integer := 0;
    read_width                : integer := 72;
    wrclk_polarity            : string := "rise";
    write_remap               : integer := 0;
    write_width               : integer := 72
);
port (
    wrclk                     : in  std_logic;
    rdclk                     : in  std_logic;
    din                       : in  std_logic_vector( 143 downto 0 );
    we                        : in  std_logic_vector( 17 downto 0 );
    wren                      : in  std_logic;
    wraddr                    : in  std_logic_vector( 13 downto 0 );
    wrmsel                    : in  std_logic;
    rden                      : in  std_logic;
    rdaddr                    : in  std_logic_vector( 13 downto 0 );
    rdmsel                    : in  std_logic;
    outreg_rstn               : in  std_logic;
    outlatch_rstn             : in  std_logic;
    outreg_ce                 : in  std_logic;
    sbit_error                : out std_logic_vector( 1 downto 0 );
    dbit_error                : out std_logic_vector( 1 downto 0 );
    dout                      : out std_logic_vector( 143 downto 0 )
);
end component ACX_BRAM72K_SDP

-- VHDL Instantiation template for ACX_BRAM72K_SDP
instance_name : ACX_BRAM72K_SDP
generic map (
    byte_width                => byte_width,
    ecc_decoder_enable        => ecc_decoder_enable,
    ecc_encoder_enable        => ecc_encoder_enable,
    initd_0                   => initd_0,
    <...>
    initd_1023                => initd_1023,
    mem_init_file             => mem_init_file,
    outreg_enable             => outreg_enable,
    outreg_sr_assertion       => outreg_sr_assertion,
    rdclk_polarity            => rdclk_polarity,
    read_remap                => read_remap,
    read_width                => read_width,
    wrclk_polarity            => wrclk_polarity,
    write_remap                 => write_remap,
    write_width                 => write_width
)
port map (
    wrclk                       => user_wrclk,
    rdclk                       => user_rdclk,
    din                         => user_din,
    we                          => user_we,
    wren                        => user_wren,
    wraddr                      => user_wraddr,
    wrmsel                      => user_wrmsel,
    rden                        => user_rden,
    rdaddr                      => user_rdaddr,
    rdmsel                      => user_rdmsel,
    outreg_rstn                 => user_outreg_rstn,
    outlatch_rstn               => user_outlatch_rstn,
    outreg_ce                   => user_outreg_ce,
    sbit_error                  => user_sbit_error,
    dbit_error                  => user_dbit_error,
    dout                        => user_dout
);
```

ACX_BRAM72K_FIFO (72-kb FIFO Memory with Optional Error Correction)

The ACX_BRAM72K_FIFO implements a 72kb FIFO. Each port width can be independently configured and each port can use different clock domains. For higher performance operation, an additional output register can be enabled.

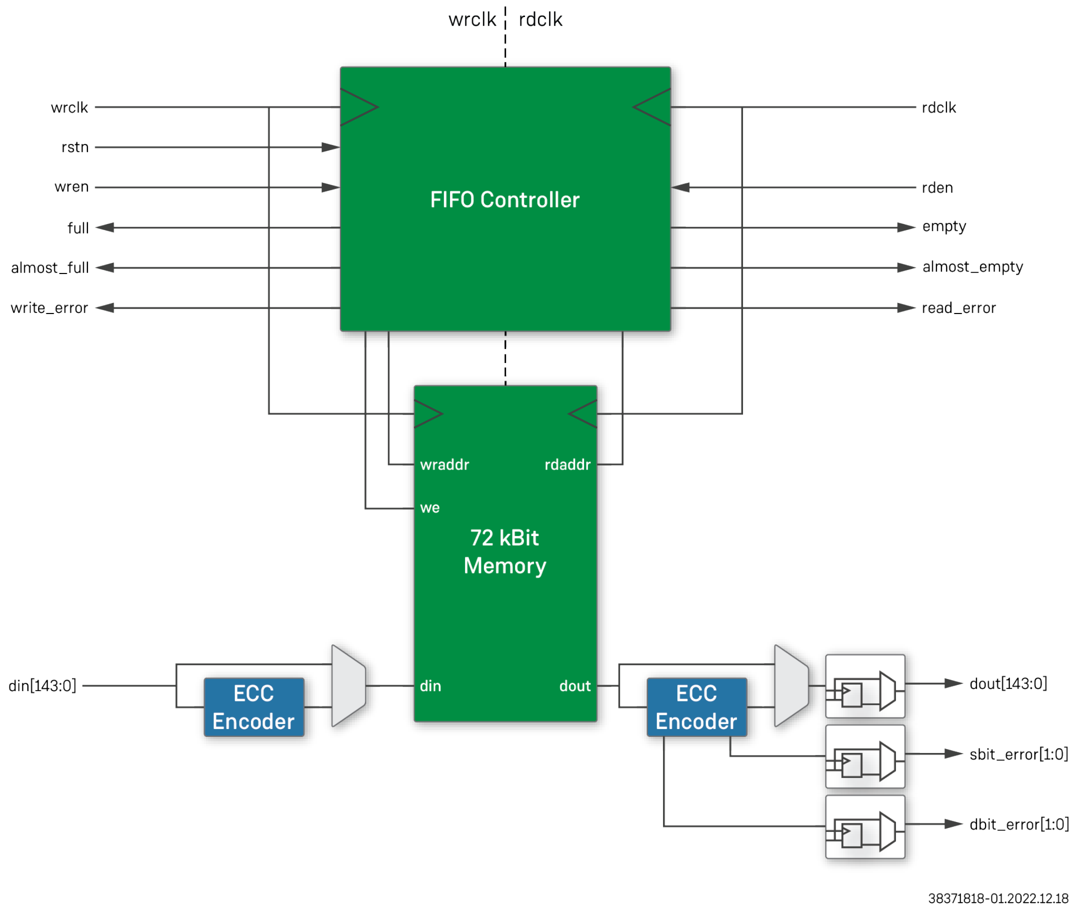


Figure 134: *ACX_BRAM72K_FIFO Block Diagram*

Parameters

Table 272: *ACX_BRAM72K_FIFO Parameters*

Parameter	Supported Values	Default Value	Description
`read_width`(1)	4, 8, 9, 16, 18, 32, 36, 64, 72, 128, 144	72	Controls the width of the read port.
`write_width`(1)	4, 8, 9, 16, 18, 32, 36, 64, 72, 128, 144	72	Controls the width of the write port.
`rdclk_polarity`	"rise", "fall"	"rise"	Detemines the clock edge used by the `rdclk` signal: "rise" – rising edge. "fall" – falling edge.
`wrclk_polarity`	"rise", "fall"	"rise"	Determines the clock edge used by the `wrclk` signal: "rise" – rising edge. "fall" – falling edge.
`outreg_enable`	0, 1	1	Controls whether the output register is enabled: 0 – disables the output register and results in a read latency of one cycle. 1 – enables the output register and results in a read latency of two cycles.
`sync_mode`	0,1	0	Controls whether the FIFO operates in synchronous or asynchronous mode. In synchronous mode, the two input clocks must be driven by the same clock input, and the pointer synchronization logic is bypassed, leading to lower latency for flag assertion. 0 – asynchronous mode. 1 – synchronous mode.
`afull_threshold`	`0–14'h3FFF`	`14'h10`	Defines the word depth at which the `almost_full` output changes. The `almost_full` signal may be used to determine the number of blind writes to the FIFO made without monitoring the `full` flag. For example, if the `afull_threshold` parameter is set to `14'h0004` and the `almost_full` signal is de-asserted, at least five empty locations exist in the FIFO. All five words may be written without overflowing the FIFO and causing assertion of `write_error`.
`aempty_threshold`(2)	`0–14'h3FFF`	`14'h10`	Defines the word depth at which the `almost_empty` output changes. May be used to determine the number of blind reads from the FIFO performed without monitoring the `empty` flag. For example, if the `aempty_threshold` parameter is set to `14'h0004` and the `almost_empty` flag is de-asserted, at least five words exist in the FIFO which may be read without underflowing the FIFO and causing assertion of `read_error`.
`fwft_mode`(3)	0, 1	0	First-word fall through (FWFT). Controls the behavior of data at the output of the FIFO relative to `rden`: 0 – data is presented at the output of the FIFO after `rden` is asserted when `outreg_enable = 1`. 1 – data is presented at the output of the FIFO as soon as it is available and coincident with the de-assertion of `empty (outreg_enable = 0)`. Data is held until `rden` is asserted. If `outreg_enable = 1`, an additional one `rdclk` cycle of latency results causing the `empty` flag to precede the output data by one `rdclk` cycle and should be externally delayed if flag alignment is required.
`ecc_encoder_enable`(4)	0, 1	0	Enables the ECC encoder which calculates the ECC syndrome and stores it in memory in data bits [71:64]. When enabled, `din[71:64]` is ignored: 0 – ECC encoder is disabled. 1 – ECC encoder is enabled.
`ecc_decoder_enable`(5)	0, 1	0	Enables the ECC decoder which uses the ECC syndrome in bits [71:64] to correct any single-bit error and detect any 2-bit error: 0 – ECC decoder is disabled. 1 – ECC decoder is enabled.

Parameter	Supported Values	Default Value	Description

Table Notes

1. Parameters `read_width/write_width` settings of 128 and 144 consume the adjacent MLP site by using it as a route through for the higher order bits of the respective data buses.
2. `aempty_threshold` does not consider the `fwft_mode` or `outreg_enable`. If `outreg_enable = 1`, then there are `aempty_threshold + 1` entries available when `almost_empty` is deasserted. If `fwft_mode = 1`, there are `aempty_threshold-1` entries available when `almost_empty` is asserted.
3. FWFT mode is not supported when the FIFO is in synchronous mode (`sync_mode = 1`) while the output register is enabled (`outreg_enable = 1`).
4. ECC encoding is only supported when `write_width = 64` or `write_width = 128`.
5. ECC decoding is only supported when `read_width = 64` or `read_width = 128`.

Ports

Table 273: *ACX_BRAM72K_FIFO Pin Descriptions*

Name	Direction	Description
`rstn`	Input	Asynchronous reset input. Resets the entire FIFO.
`wrclk`	Input	Write clock input. Write operations are fully synchronous and occur upon the active edge of the `wrclk` input when `wren` is asserted. The active edge of `wrclk` is determined by `wrclk_polarity`.
`wren`	Input	Write port enable. Assert `wren` high to write data to the FIFO.
`din[143:0]`	Input	Write port data input. Input data (`data_in`) should be aligned as follows: `write_width = 144`: `din = data_in`. `write_width = 128`: `din = {8'h0, data_in[127:64], 8'h0, data_in[63:0]}`. `write_width < 128`: `din[write_width-1:0] = data_in` (remaining `din` upper bits should be tied to `1'b0`).
`full`	Output	Asserted high when the FIFO is full.
`almost_full`	Output	Asserted high when remaining space in the FIFO is equal to or less than `afull_threshold`.
`write_error`	Output	Asserted the cycle after a write to the FIFO when the FIFO is already full.
`rdclk`	Input	Read clock input. Read operations are fully synchronous and occur upon the active edge of the `rdclk` input when the `rden` signal is asserted. The active edge of `rdclk` is determined by `rdclk_polarity`.
`rden`	Input	Read port enable. Assert `rden` high to perform a read operation.
`empty`(1)	Output	Asserted high when the FIFO is empty.
`almost_empty`(2)	Output	Asserted high when the FIFO has less than, or equal to `aempty_threshold` words remaining.
`read_error`	Output	Asserted the cycle after a FIFO read when the FIFO is already empty.
`sbit_error[1:0]`	Output	Asserted high when the data on `dout` includes a single-bit error that was corrected.
`dbit_error[1:0]`	Output	Asserted high when the data on `dout` includes an error or errors that were not corrected.
`dout[143:0]`(3)	Output	Read port data output. The output data, `data_out`, is aligned as follows (the organization is the same as `din` and `data_in`): `read_width = 144`: `dout = data_out`. `read_width = 128`: `dout = {8'hX, data_out[127:64], 8'hX, data_out[63:0]}`. `read_width < 128`: `dout[read_width-1:0] = data_out` (remaining `dout` upper bits present as `1'bX`.

Table Notes

1. When operating in synchronous mode (`sync_mode` = 1), the falling transition of `empty` is delayed by one cycle. `empty` remains asserted for the cycle after the last entry in the FIFO is read.
2. When operating in synchronous mode (`sync_mode` = 1), the falling transition of `almost_empty` is delayed by one cycle. `almost_empty` remains asserted for a cycle after `aempty_threshold` is reached.
3. For `data_out` bits marked X, these present as X in simulation and on silicon the values are undefined.

Read and Write Operations

Write Operation

Write operations are signaled by asserting the `wren` signal. The value of `din` is stored to the next available FIFO location on the rising edge of `wrclk` whenever `wren` is asserted, and `full` is de-asserted.

Read Operation

Read operations are signaled by asserting the `rden` signal. The next FIFO location contents are transferred to the output latches on the rising edge of `rdclk` whenever `rden` is asserted and `empty` is de-asserted. If `outreg_enable = 1`, the FIFO contents are available on `dout` on the following rising edge of `rdclk`.

First Word Fall Through (FWFT)

The FIFO operates in a first word fall through mode, where the first word written to the FIFO is presented on the output before `rden` is asserted, for the following configurations:

- `fwft_mode = 0` – FIFO operates as FWFT when `outreg_enable = 0`. With `outreg_enable = 1`, the first word is output on the rising edge after `rden` is asserted.
- `fwft_mode = 1` – FIFO always operates as FWFT, with the first word output either on the following rising edge of `rdclk` (`outreg_enable = 0`) or the third rising edge of `rdclk` (`output_enable = 1`) after the first word is written to the FIFO.

Output Latch and Register

Table 274: ***ACX_BRAM72K_FIFO Output Function Table for Latched Mode***

Operation [(1)]	`rdclk`	`outlatch_rstn`	`rden`	`dout`
Reset latch	↑	0	X	0
Hold	↑	1	0	Hold previous value.
Read	↑	1	1	Next FIFO value.

Table Notes

1. This function assumes rising-edge clock and active-high port enable, otherwise the previous value is held.

Table 275: ***ACX_BRAM72K_FIFO Output Function Table for Registered Mode***

Operation [(1)]	`rdclk`	`outreg_rstn`	`outregce`	`dout`
Reset Output	↑	0	1	0
Hold	↑	1	0	Previous `dout[]`.
Update Output	↑	1	1	Registered from latch output.

Table Notes

1. This function assumes active-high clock, output register clock enable, and output register reset, otherwise the previous `dout[]` is held.

Timing Diagrams

Synchronous Mode

Data output, `dout`, timing for all combinations of `outreg_enable` and `fwft_mode` is shown in the following waveform.

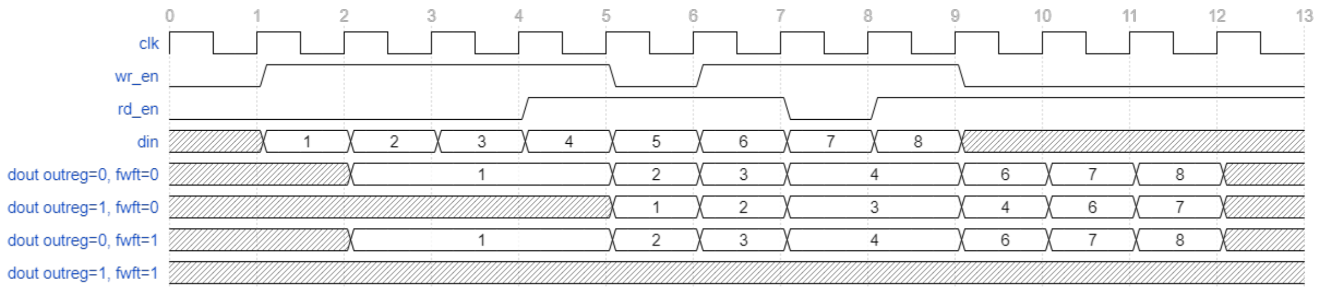


Figure 135: *Output Timing with `sync_mode = 1`*

Asynchronous Mode

Data output, `dout`, timing for all combinations of `outreg_enable` and `fwft_mode` is shown in the following waveform.

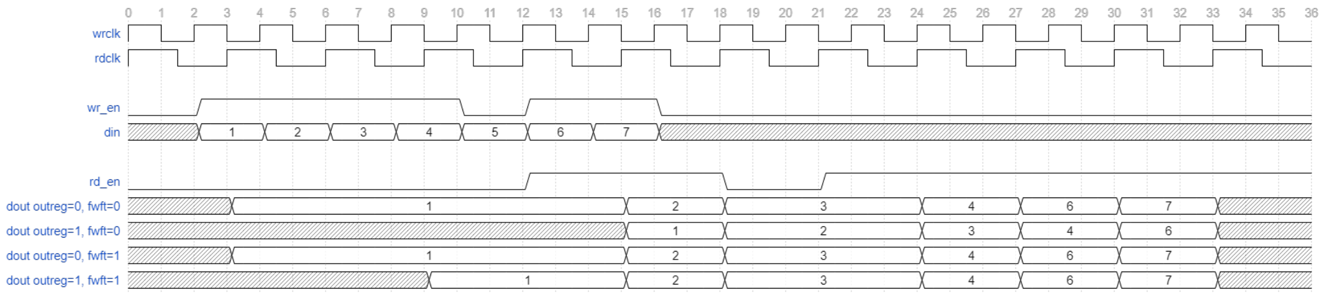


Figure 136: *Output Timing with `sync_mode = 0`*

Inference

The ACX_BRAM72K_FIFO is not inferrable.

Instantiation Template

Verilog

```
ACX_BRAM72K_FIFO #(
    .aempty_threshold    (aempty_threshold),
    .afull_threshold    (afull_threshold),
    .ecc_decoder_enable    (ecc_decoder_enable),
    .ecc_encoder_enable    (ecc_encoder_enable),
    .fwft_mode             (fwft_mode),
    .outreg_enable         (outreg_enable),
    .rdclk_polarity         (rdclk_polarity),
    .read_width             (read_width),
    .sync_mode             (sync_mode),
    .wrclk_polarity         (wrclk_polarity),
    .write_width         (write_width)
) instance_name (
    .din                 (din),
    .wrclk                 (wrclk),
    .rdclk                 (rdclk),
    .wren                 (wren),
    .rden                 (rden),
    .rstn                 (rstn),
    .dout                 (dout),
    .sbit_error             (sbit_error),
    .dbit_error             (dbit_error),
    .almost_full         (almost_full),
    .full                 (full),
    .almost_empty         (almost_empty),
    .empty                 (empty),
    .write_error         (write_error),
    .read_error             (read_error)
);
```

VHDL

```
-- VHDL Component template for ACX_BRAM72K_FIFO
component ACX_BRAM72K_FIFO is
generic (
    aempty_threshold         : std_logic_vector( 14 downto 0 ) := X"0010";
    afull_threshold          : std_logic_vector( 14 downto 0 ) := X"0010";
    ecc_decoder_enable       : integer := 0;
    ecc_encoder_enable       : integer := 0;
    fwft_mode                : integer := 0;
    outreg_enable            : integer := 0;
    rdclk_polarity           : string := "rise";
    read_width               : integer := 72;
    sync_mode                : integer := 0;
    wrclk_polarity           : string := "rise";
    write_width              : integer := 72
);
port (
    din                      : in  std_logic_vector( 143 downto 0 );
    wrclk                    : in  std_logic;
    rdclk                    : in  std_logic;
    wren                     : in  std_logic;
    rden                     : in  std_logic;
    rstn                     : in  std_logic;
    dout                     : out std_logic_vector( 143 downto 0 );
    sbit_error               : out std_logic_vector( 1 downto 0 );
    dbit_error               : out std_logic_vector( 1 downto 0 );
    almost_full              : out std_logic;
    full                     : out std_logic;
    almost_empty             : out std_logic;
    empty                    : out std_logic;
    write_error              : out std_logic;
    read_error               : out std_logic
);
end component ACX_BRAM72K_FIFO;



-- VHDL Instantiation template for ACX_BRAM72K_FIFO
instance_name : ACX_BRAM72K_FIFO
generic map (
    aempty_threshold         => aempty_threshold,
    afull_threshold          => afull_threshold,
    ecc_decoder_enable       => ecc_decoder_enable,
    ecc_encoder_enable       => ecc_encoder_enable,
    fwft_mode                => fwft_mode,
    outreg_enable            => outreg_enable,
    rdclk_polarity           => rdclk_polarity,
    read_width               => read_width,
    sync_mode                => sync_mode,
    wrclk_polarity           => wrclk_polarity,
    write_width              => write_width
)
port map (
    din                      => user_din,
    wrclk                    => user_wrclk,
    rdclk                    => user_rdclk,
    wren                        => user_wren,
    rden                        => user_rden,
    rstn                        => user_rstn,
    dout                        => user_dout,
    sbit_error                  => user_sbit_error,
    dbit_error                  => user_dbit_error,
    almost_full                 => user_almost_full,
    full                        => user_full,
    almost_empty                => user_almost_empty,
    empty                       => user_empty,
    write_error                 => user_write_error,
    read_error                  => user_read_error
);
```

ACX_LRAM (4096-bit (128x32) Simple-Dual-Port Memory)

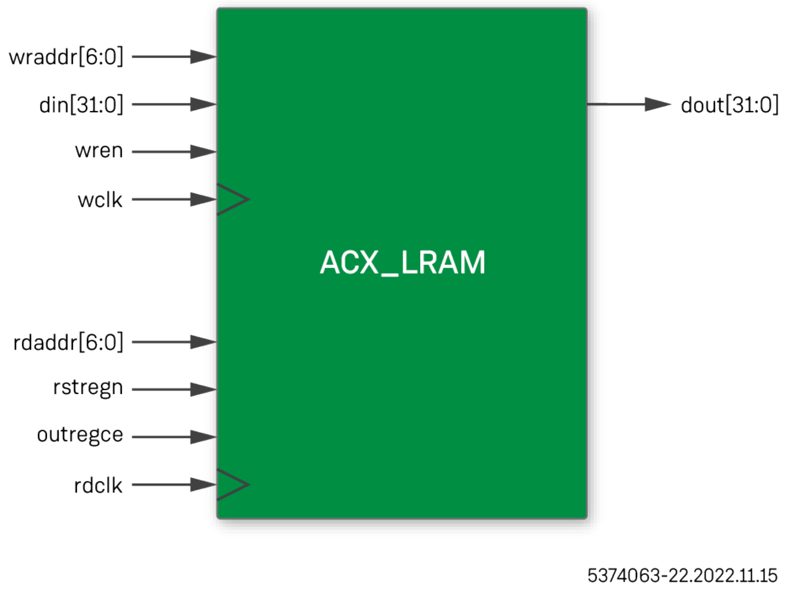


Figure 137: ***4096-bit (128 × 32) Simple-Dual-Port Memory***

The Logic RAM (ACX_LRAM) implements a 4096-bit memory block with one write port and one read port. The ACX_LRAM can be configured as either a 128 × 32 simple dual-port (1 write port, 1 read port) RAM or a 128 × 32 single port (1 read/write port) RAM. The ACX_LRAM has a synchronous write port. The read port is asynchronous and has an optional output register. This memory block is distributed in the FPGA fabric.

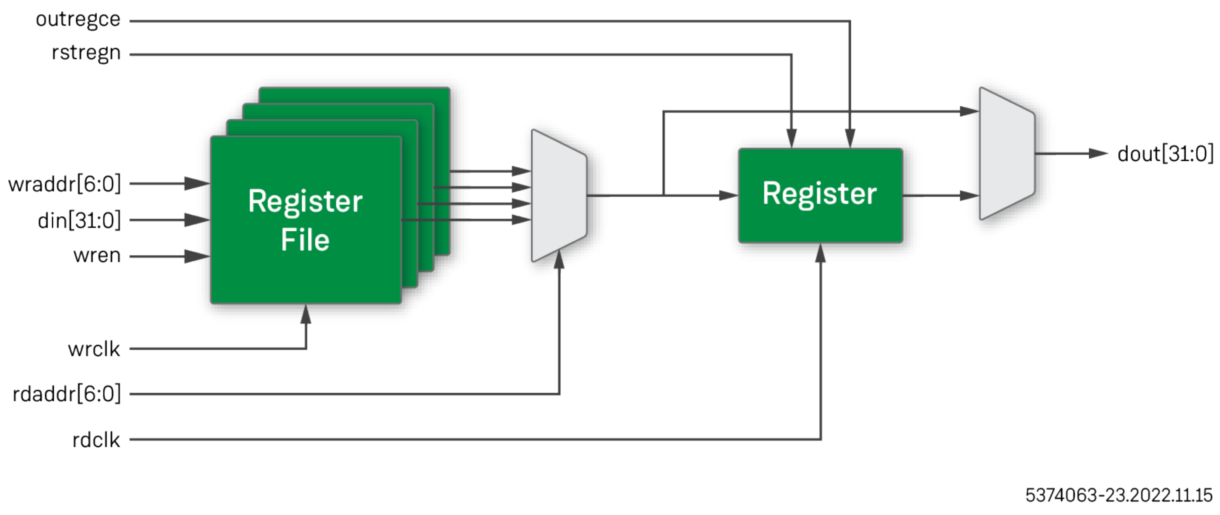


Figure 138: ***ACX_LRAM Block Diagram***

Table 276: ***ACX_LRAM Pin Descriptions***

Name	Type	Description
`wraddr[6:0]`	Input	Write port address input.
`din[31:0]`	Input	Write port data input.
`wren`	Input	Write port enable (active-high). When asserted, the data present on `din[31:0]` is written to the location addressed by `wraddr[6:0]` at the next active edge of `wrclk`.
`wrclk`	Input	Write port clock (programmable, default rising edge).
`rdaddr[6:0]`	Input	Read port address input.
`rstregn`	Input	Read port output register reset (active-low). The `sr_assertion` parameter determines whether the reset is synchronous (default) or asynchronous. When asserted, the read port output register is assigned the value of the `reg_rstval` parameter. The priority of the `rstregn` input relative to the read port output register clock enable (`outregce`) input is determined by the value of the `regce_priority` parameter. The `rstregn` signal only resets the read port output register. It does not reset the memory contents.
`outregce`	Input	Read port output register clock enable (active-high).
`rdclk`	Input	Read port clock (programmable, default rising edge).
`dout[31:0]`	Output	Read port data output. Configured to be either synchronous or asynchronous as determined by the `reg_dout` parameter. If `reg_dout` is `1'b0`, `dout[31:0]` reads the contents of the memory addressed by `raddr[6:0]` onto its pins. If the `reg_dout` parameter is `1'b1`, the `dout[31:0]` output is driven by the contents of the memory addressed by `raddr[6:0]` at the next active edge of `rclk` if the read port output clock enable input is high.

Table 277: ***ACX_LRAM Parameters***

Parameter	Defined Values	Default Value	Description
`write_clock_polarity`	`rise`, `fall`	`rise`	Sets the active edge of `wrclk`. A value of `rise` corresponds to an active rising edge assignment while `fall` corresponds to an active falling edge assignment.
`read_clock_polarity`	`rise`, `fall`	`rise`	Sets the active edge of `rdclk`. A value of `rise` corresponds to an active rising edge assignment while `fall` corresponds to an active falling edge assignment.
`reg_dout`	`1'b0`, `1'b1`	`1'b0`	Defines whether the read port output register is used or bypassed. `1'b0` bypasses the register while `1'b1` enables the register. Enabling the output register incurs an additional cycle of latency for the read operation.

Parameter	Defined Values	Default Value	Description
`reg_initval`	32-bit binary or hexadecimal value	`32'h0`	Defines the power-up default value of the read port output register.
`reg_rstval`	32-bit binary or hexadecimal value	`32'h0`	Defines the value assigned to the read port output register when the `rstregn` input is asserted low and there is an active edge on `rdclk`.
`regce_priority`	`rstreg`, `regce`	`rstreg`	Defines the priority of the `outregce` clock enable input relative to the `rstregn` reset input during its assertion on the read port output register. Setting `regce_priority` to `rstreg` allows set/reset of the read port output register to occur at the next active edge of `rdclk` without requiring the `outregce` clock enable input to be active. Setting `regce_priority` to `regce` requires the `regce` clock enable input to be high for the reset operation to occur at the next active edge of `rdclk`.
`sr_assertion`	`clocked`, `unclocked`	`clocked`	Sets whether the assertion of the output register reset is synchronous or asynchronous with respect to the `rdclk` input. A value of `clocked` sets synchronous reset where the output register is reset on the next rising edge of the clock if `rstregn` is asserted. A value of `unclocked` sets asynchronous reset where the output register is reset immediately following the assertion of the `rstregn` input.
`mem_init_00`–`mem_init_15`	256-bit hexadecimal value	`256'hx`	The `mem_init_00` through `mem_init_15` parameters define the initial contents of the memory. Each of the 16 256-bit parameters is associated with the 4096-bit LRAM memory as defined in LRAM Memory Initialization (see page 428).
`mem_init_file`	<path to HEX file>	""	Provides a mechanism to set the initial contents of the LRAM memory. If defined, the LRAM is initialized with the values defined in the file specified by the `mem_init_file` parameter according to the format defined in LRAM Memory Initialization (see page 428). If `mem_init_file` is the default value (""), the initial contents are defined by the value of the `mem_init` parameter. If the `mem_init_nn` and `mem_init_file` parameters are not defined, the contents of the LRAM are also undefined.

Simultaneous Memory Operations

Memory operations may be performed simultaneously from both sides of the memory, however there is a restriction with memory collisions. A memory collision is defined as the condition where both of the ports access the same memory address within the same clock cycle (with both ports connected to the same clock), or a window less than one clock cycle of the faster clock (with each port connected to a different clock). The definition of a memory collision depends on whether or not the read port output register is enabled.

If the read port output register in not enabled (`reg_dout = 1'b0`), a memory collision is defined by reading the same address the cycle after a write command has occurred. If the read port output register is enabled (`reg_dout = 1'b1`), a memory collision is defined by reading the same address two cycles after a write command has occurred. If a memory collision occurs, the write to memory is valid, but the read data might be incorrect.

Timing Diagram

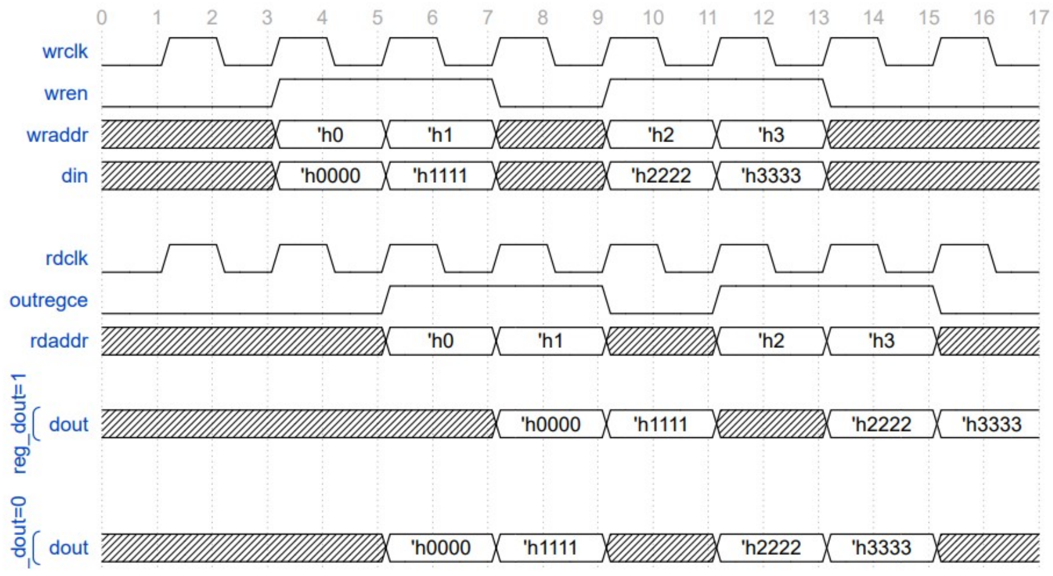


Figure 139: ***LRAM4K SDP Timing Diagram***

ACX_LRAM Memory Initialization

By default, the contents of the LRAM memory are undefined. If the initial contents are to be defined, they may be assigned from either a file specified to by the `mem_init_file` parameter or assigned from the values of the `mem_init_00` through `mem_init_15` parameters.

The memory is organized as little-endian with bit zero mapped to bit zero of the `mem_init_00` parameter and bit 4095 mapped to bit 255 of the `mem_init_15` parameter.

The ACX_LRAM memory block may alternatively be initialized with a memory file by setting the `mem_init_file` parameter to the path of a memory initialization file. The file format in the latter case is defined by hexadecimal entries separated by white space, where the white space can be spaces or line separation. Each number is written as a 32-bit hexadecimal number. Commenting is allowed with text following a double-slash ("//") through to the end of the line. C-like commenting is also allowed where the characters between "/*" and "*/" are ignored. The memory is initialized starting with the first entry of the file initializing the memory array at address zero, moving upward. Each line consists of a hexadecimal number representing the entry itself.

Using ACX_LRAM as a Read-Only Memory (ROM)

The ACX_LRAM memory can be used as a read-only memory (ROM) by providing memory initialization data with a file or via parameters (as described LRAM Memory Initialization (see page 428)), and tying the `wren` signal to its de-asserted value. All signals on the read-side of the ACX_LRAM operate as described above. This configuration allows reading from the memory, but not writing to it.

Create an Instance

To create an ACX_LRAM instance within a design, there are three available methods:

1. Infer the memory – this method provides the greatest code portability and is the recommended approach. Examples follow of how to infer an ACX_LRAM with an output register.
2. Directly instantiated – this method gives access to the full feature set of the memory. However, any code is less portable to other technology nodes. See Instantiation Template (see page 430)
3. ACE LRAM IP generator – Refer to the *ACE User Guide* (UG070) for details.

Inference Template

The following examples show how to infer an ACX_LRAM with an output register.

ACX_LRAM with Output Register

```
`timescale 1 ps / 1 ps
module ACX_LRAM_infer_meminitfile_16t (rdaddr, outregce, rstregn, rdclk, dout,
                wraddr, din, wren, wrclk);

// read port inputs & outputs
input  [6:0] rdaddr;
input        outregce;
input        rstregn;
input        rdclk;
output [31:0] dout;
// write port inputs & outputs
input  [6:0] wraddr;
input  [31:0] din;
input        wren;
input        wrclk;
// read port local variables
reg [31:0] dout_reg;

// 128x32 memory array
reg [31:0] mem_array [0:127] /* synthesis syn_ramstyle="logic_ram" */;
initial begin
        $readmemb("/<absolute_path>/memfile.txt", mem_array);
end
// read port
always @(posedge rdclk)
       if (outregce)
          if (~rstregn)
            dout_reg <= 10'b0;
          else
            dout_reg <= mem_array[rdaddr];
    else
         dout_reg <= dout_reg;
 assign dout = dout_reg;
// write port
always @(posedge wrclk)
  if (wren)
    mem_array[wraddr] <= din;
endmodule
```

Instantiation Template

Verilog

```
ACX_LRAM #(
    .write_clock_polarity   ("rise"),
    .read_clock_polarity    ("rise"),
    .reg_dout               (1'b1),
    .reg_initval            (32'h00),
    .reg_rstval             (32'h00),
    .regce_priority         ("rstreg"),
    .sr_assertion           ("clocked"),
    .mem_init_00            (256'h0),
    .mem_init_01            (256'h0),
    .mem_init_02            (256'h0),
    .mem_init_03            (256'h0),
    .mem_init_04            (256'h0),
    .mem_init_05            (256'h0),
    .mem_init_06            (256'h0),
    .mem_init_07            (256'h0),
    .mem_init_08            (256'h0),
    .mem_init_09            (256'h0),
    .mem_init_10            (256'h0),
    .mem_init_11            (256'h0),
    .mem_init_12            (256'h0),
    .mem_init_13            (256'h0),
    .mem_init_14            (256'h0),
    .mem_init_15            (256'h0),
    .mem_init_file            ("lram_init.hex")
) instance_name (
    .wrclk                  (lram_wrclk),
    .wren                   (lram_wren),
    .wraddr                 (lram_wraddr),
    .din                    (lram_wrdata),
    .rdclk                  (lram_rdclk),
    .rdaddr                 (lram_rdaddr),
    .outregce               (lram_rdenable),
    .rstregn                (lram_rstregn),
    .dout                   (lram_rddata)
);
```

ACX_LRAMFIFO (LRAM-Based 128-Word FIFO Memory)

The ACX_LRAMFIFO implements a 128-word deep by n-bit wide FIFO memory block utilizing embedded LRAM blocks and LUTs. The ACX_LRAMFIFO can be configured to support a variety of widths in increments of one bit. The read and write clocks may be either synchronous or asynchronous with respect to each other. If the user read and write clocks are from the same source, the `ptr_sync_mode` parameter may be set to `1'b1` to enable lower-latency synchronous generation of the status flags.

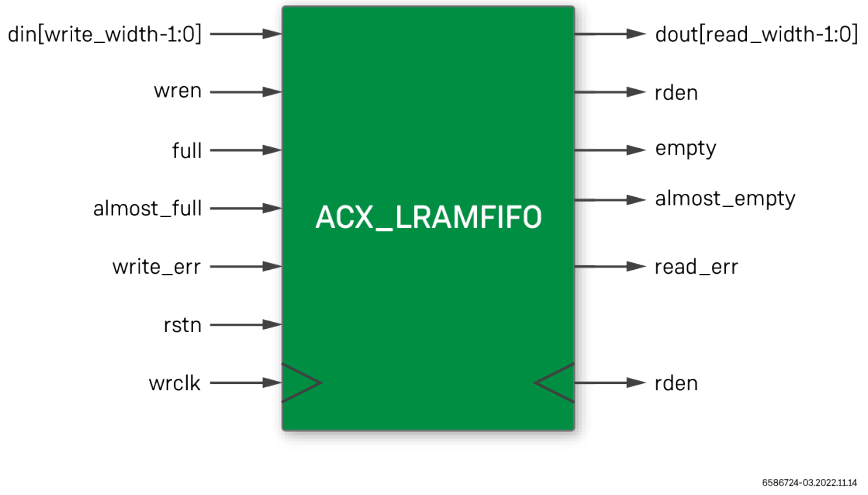


Figure 140: *ACX_LRAMFIFO Symbol*

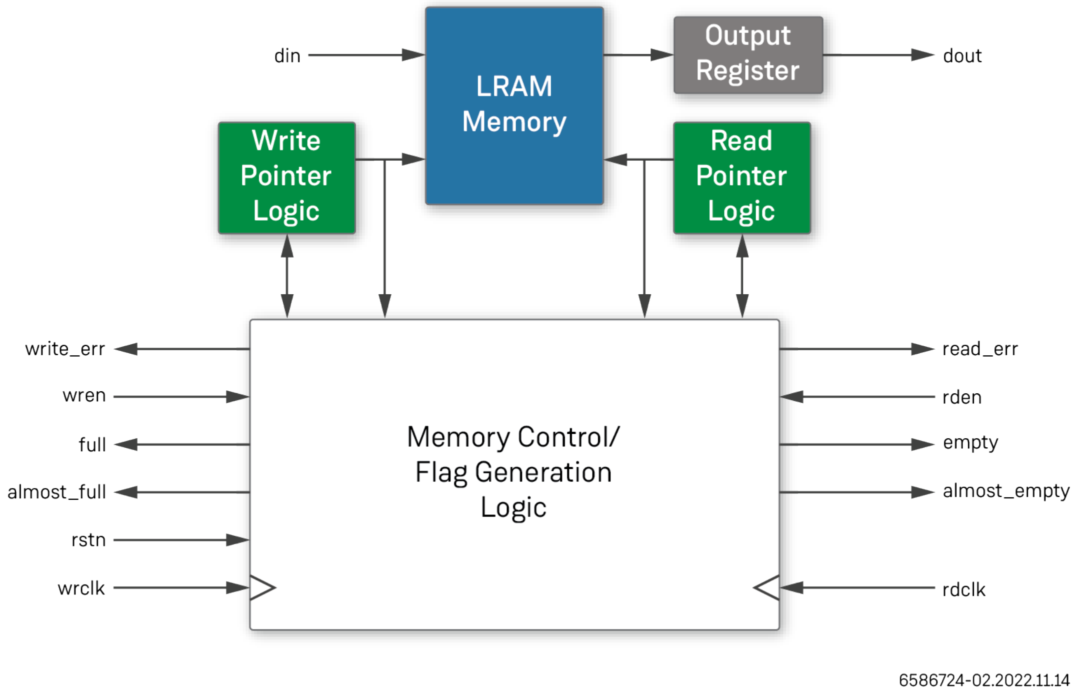


Figure 141: *ACX_LRAMFIFO Block Diagram*

Table 278: *ACX_LRAMFIFO Pin Description*

Name	Type	Clock Domain	Description
`rstn`	Input	programmable	FIFO reset (active-low). Asserted low resets the FIFO to clear both the read and write pointers and set the FIFO to the empty condition.
Write Interface			
`wrclk`	Input	`wrclk`	Write clock (rising edge based).
`wren`	Input	`wrclk`	Write enable (active-high). Data is written into the FIFO at the next activewrite clock edge when `wren` is driven high, if the `full` flag is not asserted.
`din[width-1:0]`	Input	`wrclk`	Write port data input.
`full`	Output	`wrclk`	Full flag (active-high).
`almost_full`	Output	`wrclk`	Almost-full flag (active-high).
`write_err`	Output	`wrclk`	Write error flag (active-high).
Read Interface			
`rdclk`	Input	`rdclk`	Read clock (rising edge based).
`rden`	Input	`rdclk`	Read enable (active-high). Data is read from the FIFO at the next active edge of the read clock when `rden` is driven high, if the `empty` flag is not asserted.
`dout[width-1:0]`	Output	`rdclk`	Read port data output.
`empty`	Output	`rdclk`	Empty flag (active-high).
`almost_empty`	Output	`rdclk`	Almost-empty flag (active-high).
`read_err`	Output	`rdclk`	Read error flag (active-high).

Parameters

Table 279: *ACX_LRAMFIFO Parameters*

Parameter	Defined Values	Default Value	Description
`read_width`, `write_width`	32, 64, 96, 128, 160, 192, 224, 256	32	Define the width of the FIFO data input and output buses. Must have the same value and must be a multiple of 32 bits.
`read_depth`, `write_depth`	4, 8, 16, 32, 64, 128	128	Define the depth of the FIFO, which may be up to 128 locations. Choosing a depth less than 128 locations allows a smaller implementation of the FIFO controller logic. Must have the same value, and must be a power of 2.
`ptr_sync_mode`	`1'b0`, `1'b1`	`1'b0`	Bypasses the synchronization circuitry between the read and write ports, for use when the `wrclk` and `rdclk` inputs are connected to the same source. Reduces the latency through the FIFO and provides faster de-assertion of the status flags (`empty`, `full`, etc.). If the read and write clocks are connected to different sources, the synchronization circuitry must be used, and `ptr_sync_mode` must be set to `1'b0`.
`rst_sync_mode`	`1'b0`, `1'b1`	`1'b0`	Bypasses the reset synchronization circuit. When the `rst_sync_mode` parameter is set to `1'b0`, both the read and write pointer resets utilize the reset synchronizer logic. When the `rst_sync_mode` parameter is set to `1'b1`, the `rstn` input must be synchronous to the `wrclk`/`rdclk` driving the FIFO. If the read and write clocks are connected to different sources, the synchronization logic must be used, and `rst_sync_mode` must be set to `1'b0`.
`afull_offset`	8-bit hexadecimal number	`8'h04`	Defines the word depth at which the `almost_full` output changes. The `almost_full` signal may be used to determine the number of blind writes to the FIFO that can occur without monitoring the `full` flag. For example, if `afull_offset` is set to `8'h04` and the `almost_full` signal is de-asserted, there are at least five empty locations in the FIFO. All five words may be written without overflowing the FIFO and causing `w rite_err` to be asserted. `afull_offset` must be smaller than the `wr ite_depth` value.
`aempty_offset`	8-bit hexadecimal number	`8'h04`	Defines the word depth at which the `almost_empty` output changes. The `almost_empty` signal may be used to determine the number of blind reads from the FIFO that can occur without monitoring the empty flag. For example, if `aempty_offset` is set to `8'h04` and the almost_empty flag is de-asserted, there are at least five words in the FIFO. All five words may be read without underflowing the FIFO and causing the `read_err` flag to be asserted. `aempty_offset` must be smaller than the `read_depth` value.
`fwft_mode`	`1'b0`, `1'b1`	`1'b0`	Defines whether the FIFO is in first-word-fall-through mode. This parameter only effects the availability of the first word written to the FIFO when empty. Operation of the two modes is the same after the first read operation. `fwft_mode` may only be set to `1'b1` when `ptr_sync_ mode` is set to `1'b0`. • If `fwft_mode` is `1'b1`, the first value written to the FIFO appears at `dout` (and `doutp`, `doutxp` if applicable) without having to perform a read operation. `hold_output` must be `1'b1` when `fw ft_mode` is `1'b1`. • If `fwft_mode` is `1'b0`, the first data word written to the FIFO is available at the FIFO output one `rdclk` cycle after the first read operation.

Parameter	Defined Values	Default Value	Description
`hold_output`	`1'b0`, `1'b1`	`1'b1`	Controls the read output value. When `hold_output` is set to `1'b1`, the read output holds its value until the next read. When `hold_output` is set to `1'b0`, the read output data is valid for one clock cycle after the read and then becomes invalid, giving a slight performance advantage in the circuit. Only disable this option if the user design can reliably pull the data from the output within one clock cycle after the read. `hold_output` must be `1'b1` when `fwft_mode` is `1'b1`.
`prevent_overunderflow`	`1'b0`, `1'b1`	`1'b1`	Enabling this option prevents data/pointer corruption caused by reading or writing the FIFO when empty or full, respectively. Disabling this safety check allows the FIFO to run faster, but results in data corruption if reading from the FIFO when empty or writing to the FIFO when full.

FIFO Operation

This section describes the operations of ACX_LRAMFIFO.

FIFO Reset

A FIFO reset is performed by asserting the `rstn` input signal for a minimum of four clock cycles of the slower of either `wrclk` or `rdclk`, causing the FIFO internal state to be reset such that the FIFO is empty. After a reset, it is not possible to retrieve any of the data contained in the FIFO before the reset occurred. The entire FIFO is available to be written with new data.

FIFO Write

A FIFO write is performed by asserting the `wren` input when the FIFO is not full. Asserting `wren` causes the data present on the `din` inputs to be stored in the FIFO to be retrieved later with a read operation. If a write operation fills the last remaining location in the FIFO, the `full` signal is asserted on the following clock cycle. If `wren` is asserted when the FIFO is full, the write fails, and `write_error` is asserted on the next clock cycle.

FIFO Read

A FIFO read is performed by asserting the `rden` input when the FIFO is not empty. Asserting `rden` causes the next data word from the FIFO memory array to be presented on the `dout` output. Data is always read in the same order in which it was written and is no longer stored in the FIFO when it has been read. If a read operation empties the last remaining FIFO location, the `empty` signal is asserted on the following clock cycle. If `rden` is asserted when the FIFO is empty, the read fails, and `read_error` is asserted on the next clock cycle.

FIFO Status Signals

The following table describes the signals output by the ACX_LRAMFIFO component to communicate the status of the FIFO.

Table 280: ***FIFO Pointers and Status Flag Clock Domain Assignments***

Status Signal	Clock Domain	Description
`empty`	`rdclk`	Asserted whenever the FIFO does not have data available to read. Asserted when either the FIFO is reset or all data has been read from the FIFO. The `empty` flag is synchronous to the `rdclk` domain. Asserting `rden` when `empty` is asserted does not change the contents of the FIFO in any way and does not affect the data output, but does cause the `read_err` output to be asserted in the following `rdclk` cycle. When `ptr_sync_mode` is `1'b0`, meaning that the read and write ports are not on the same clock domain, it takes a few clock cycles after writing data into the FIFO before `empty` is de-asserted. `empty` is always asserted immediately when the FIFO becomes empty.
`almost_empty`	`rdclk`	Asserted when there are `aempty_offset` or fewer words remaining in the FIFO. May be used to determine the number of reads that can be performed without causing the FIFO to underflow and `rd_err` to be asserted. For example, if `aempty_offset` is `8'h04`, and `almost_empty` is not asserted, at least five words remain in the FIFO. When `ptr_sync_mode` is `1'b0`, meaning the read and write ports are not in the same clock domain, it takes a few clock cycles after writing data into the FIFO before `almost_empty` is de-asserted. This signal is always asserted immediately when `aempty_offset` words remain.
`read_err`	`rdclk`	Asserted in the cycle following assertion of `rden` while the FIFO is empty.
`full`	`wrclk`	Asserted whenever all of the locations of the FIFO are in use. Asserting `wren` when `full` is asserted does not change the contents of the FIFO in any way and causes the `write_err` output to be asserted in the following `wrclk` cycle. The `din` inputs are ignored in this case. When `ptr_sync_mode` is `1'b0`, meaning the read and write ports are not in the same clock domain, it takes a few clock cycles after reading data from the FIFO before `full` is de-asserted. `full` is always asserted immediately when the FIFO becomes full.
`almost_full`	`wrclk`	Asserted when `afull_offset` or fewer unused locations remain in the FIFO. May be used to determine the number of writes that can be performed without causing the FIFO to overflow and `write_err` to be asserted. For example, if `afull_offset` is `8'h04`, and `almost_full` is not asserted, at least five empty locations remain in the FIFO. When `ptr_sync_mode` is `1'b0`, meaning the read and write ports are not in the same clock domain, it takes a few clock cycles after reading data from the FIFO before `almost_full` is de-asserted. This signal is always asserted immediately when `afull_offset` locations remain.
`write_err`	`wrclk`	Asserted in the cycle following assertion of `wren` while the FIFO is full.

Status Signals in Asynchronous mode

Before flag calculations can be made, the status signal generation logic ensures that both pointers are in the same clock domain as the status signal for which the calculation is performed. Write and read pointer synchronizers are used to transfer each of the pointers into the other clock domain. In order to synchronize a given pointer to the opposite clock domain, a series of registers are used, adding additional delay to the flag calculation. The status signal generation logic ensures that `full` and `almost_full` are asserted on the write clock domain immediately after the write that causes their assertion. The read that causes their de-assertion takes a few clock cycles to propagate. Likewise, `empty` and `almost_empty` are asserted on the read clock domain immediately after the read that causes their assertion, while the write that causes their de-assertion requires a few cycles to propagate across the synchronization logic.

The versions of the pointers used for flag calculations are shown in the following table.

Table 281: ***Pointers Used for FIFO Flag Calculations***

Flag	Write	Read
`empty`	Synchronized write pointer.	Read pointer.
`almost_empty`		
`full`	Write pointer.	Synchronized read pointer.
`almost_full`		

Status Signals in First-Word Fall Through Mode

First-word fall through (fwft) mode is implemented by placing an additional register at the output of the FIFO to present data to the user before `rden` is asserted. The ACX_LRAMFIFO can be thought of as popping data from the underlying FIFO into the output register whenever the output register is not occupied. This final register stage effectively adds one additional storage element to the FIFO and affects the generation of the status signals, as described in the following sections.

full and almost_full

The `full` and `almost_full` signals serve to prevent the user from overflowing the FIFO, by both indicating when the FIFO cannot accept additional data and when there is only a user-configurable number of spaces remaining, respectively.

In the case of a small FIFO and/or with a write clock frequency faster than the read frequency, it is possible to fill the FIFO to the almost full threshold, or even completely full, before the read-side logic has moved the first element of data from the underlying FIFO into the output register. In this case, `almost_full` or `full` may be asserted as the underlying FIFO fills, and then automatically de-asserted as the first element is moved to the output register, without ever having performed a read. This behavior is intentional and guarantees that a user design adhering to the `full` and/or `almost_full` signals overflows the FIFO, even while the first data element is moving to the output. This behavior also implies that in the absence of transient effects, `almost_full` is asserted when there are `afull_offset` + 1 empty spaces in the ACX_LRAMFIFO.

empty and almost_empty

The purpose of the `empty` and `almost_empty` signals are to prevent underflowing the FIFO, by indicating when the FIFO is truly empty and when there is only a user-configurable number of data elements remaining, respectively.

The generation of the `empty` signal is based on whether or not valid data is being presented to the user design by the output register and can always be used to indicate when the output data is valid. The implementation of the `almost_empty` flag uses the underlying FIFO fill level to determine its status. As a result, `almost_empty` is asserted when there are less than `aempty_offset` data elements in the underlying FIFO, or less than (`aempty_offset` + 1) elements in the ACX_LRAMFIFO (including the output register).

If the system is designed so that the FIFO is only drained when the fill level is over a given threshold, `aempty_offset` must be set to one less than the desired threshold, to account for the output register not being included in the `almost_empty` calculation.

FIFO Operational Modes

The ACX_LRAMFIFO is a highly configurable IP component that supports a number of modes of operation, including either synchronous or asynchronous (dual-clock) operation:

- Synchronous – the same clock must be connected to the `wrclk` and `rdclk` inputs, and there cannot be a phase offset between them.
- Asynchronous – two different clocks can be connected to the `wrclk` and `rdclk` inputs. The LRAM FIFO does not require any phase or frequency relationship between the two clocks whatsoever; it treats the two clock inputs as being completely asynchronous to one another. There is no requirement regarding the relative frequencies of the two clocks. Either clock can be faster or slower than the other.

Synchronous Operation

The synchronous FIFO mode is selected by setting the `ptr_sync_mode` parameter to `1'b1`. In synchronous mode, there is no latency in updating the `empty` and `almost_empty` signals after a write operation, or updating the `full` and `almost_full` signals after a read operation. This lack of latency means that the status outputs always represent the exact state of the FIFO.

In this mode, first-word-fall-through (described below) is not supported, and the `fwft` parameter must be `1'b0`.

Timing Diagrams

The following diagram shows the operation of the FIFO in asynchronous mode when the FIFO is empty, where `aempty_offset` = 3. This diagram assumes that all signals not shown, such as `rstn`, are de-asserted.

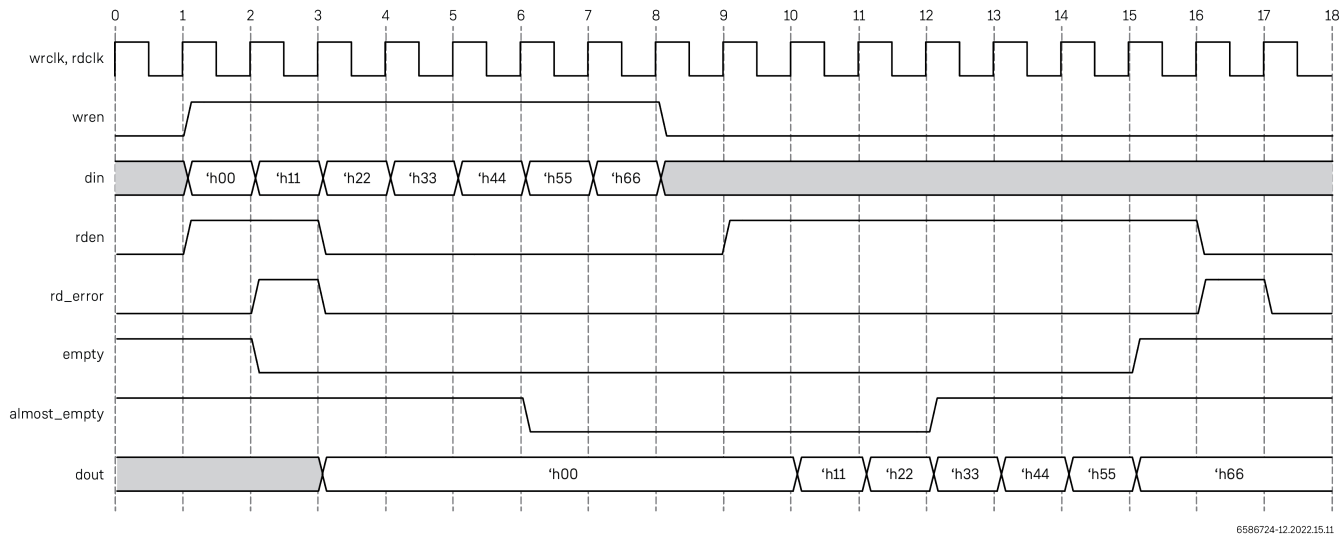


Figure 142: ***Synchronous Mode Empty FIFO Timing Diagram***

The events of each clock cycle in the preceding diagram are described in the following table.

Table 282: *Synchronous Mode Empty FIFO Timing Diagram Events*

Event	Description
1	`wren` is asserted, writing the first data word to the FIFO, causing `empty` to be de-asserted on the following clock cycle since the FIFO is no longer empty. At the same time, `rden` is asserted, indicating an attempt to read from the FIFO. Since the FIFO remains empty, `rd_err` is asserted on the following clock cycle, and the data output dout does not change.
2	`wren` is asserted, writing the second data word into the FIFO. At the same time, `rden` is asserted, reading the first data word from the FIFO. The data arrives on `dout` on the following cycle.
3	`wren` is asserted, writing the third data word into the FIFO. `rden` is not asserted in this cycle, so nothing is read from the FIFO.
4	`wren` is asserted, writing the fourth data word to the FIFO.
5	`wren` is asserted, writing the fifth data word to the FIFO, leaving four words in the FIFO (since the first word has already been read). The number of words is greater than the `aempty_offset` value of 3, so `almost_empty` is de-asserted on the following clock cycle.
6	`wren` is asserted, writing the sixth data word to the FIFO.
7	`wren` is asserted, writing the seventh data word to the FIFO.
8	No control signals are asserted.
9	`rden` is asserted, reading the second data word from the FIFO. The data arrives on `dout` on the following cycle.
10	`rden` is asserted, reading the third data word from the FIFO. The data arrives on `dout` on the following cycle.
11	`rden` is asserted, reading the fourth data word from the FIFO. Since only three words remain in the FIFO, the `almost_full` signal is asserted on the next clock cycle. The data arrives on `dout` on the following cycle.
12	`rden` is asserted, reading the fifth data word from the FIFO. The data arrives on `dout` on the following cycle.
13	`rden` is asserted, reading the sixth data word from the FIFO. The data arrives on `dout` on the following cycle.
14	`rden` is asserted, reading the seventh and last data word from the FIFO. The data arrives on `dout` on the following cycle. Since the FIFO is empty, the `empty` signal is asserted on the next cycle.
15	`rden` is asserted, even though the FIFO is empty. `read_error` is asserted on the following clock edge, and the FIFO contents are unchanged.

The following diagram shows the operation of the FIFO in synchronous mode, starting when there are five locations remaining in the FIFO, where the `afull_offset` parameter is 3. This diagram assumes that all signals not shown, such as `rstn`, are de-asserted, and that the `ptr_sync_mode` parameter is `1'b1`. If the `ptr_sync_mode` was `1'b0`, `dout` would be delayed by one cycle.

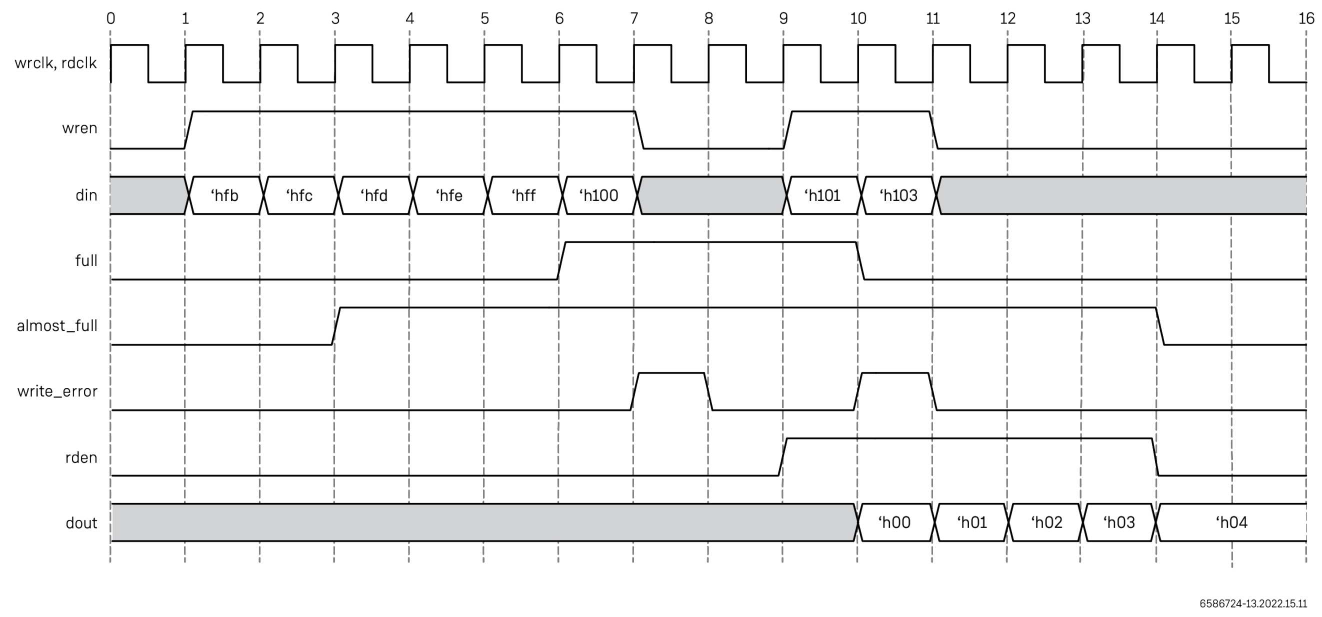


Figure 143: ***Synchronous Mode Full FIFO Timing Diagram***

The events of each clock cycle in the preceding diagram are described in the following table.

Table 283: ***Synchronous Mode Full FIFO Timing Diagram Events***

Event	Description
1-5	`wren` is asserted, writing a data word to the FIFO. After the second write, only thee locations are free, so `almost_full` is asserted on the next clock cycle. The fifth write fills up the last element and the `full` signal is asserted on the following clock cycle.
6	`wren` is asserted. Since the FIFO is already full, the write operation does not take place, and `write_error` is asserted on the following clock cycle.
7-8	No operation.
9	`wren` and `rden` are both asserted at the same time as both a read and a write operation are to be performed. Since `full` is asserted, the write fails, and `write_error` is asserted on the following cycle. The read is successful, and the output data is presented on `dout` on the following cycle.
10	`wren` and `rden` are both asserted at the same time, and the input word is written while the next output word is read and presented on `dout`. Since `full` is not asserted, both operations are successful.
11-13	`rden` is asserted, and the next output data is read and presented on `dout`. After the third read, more than three unused locations remain in the FIFO, so `almost_full` is de-asserted on the next cycle.
14	`rden` is not asserted, so the output remains constant.

Asynchronous Operation

When the FIFO is configured as an asynchronous FIFO (`ptr_sync_mode` = `1'b0`), no phase or frequency relationship is assumed between the write and the read clocks; the ACX_LRAMFIFO treats the two clock inputs as being completely asynchronous to one another. There is no requirement regarding the relative frequencies of the two clocks. Either clock can be faster or slower than the other.

Compared to synchronous mode, asynchronous mode causes additional delay when updating `empty` and `almost_empty` after a write operation, or updating `full` and `almost_full` after a read operation, as it takes time for the status to cross safely from one clock domain to the other. All status signals are asserted without delay; only their de-assertion requires additional time. For asynchronous operation, the `ptr_sync_mode` parameter must be set to `1'b0`.

When using the FIFO with two clocks, the first-word fall-through (`fwft`) parameter controls when data is made available on the output signals:

- `fwft` = 1'b0 (request mode) – When the `fwft` parameter is `1'b0`, the FIFO is in request mode. Asserting `rden` requests that the data be presented on the `dout` pins on the following cycle. This mode is identical to when the FIFO has `ptr_sync_mode` = `1'b1`, and the clocks are synchronous to one another. In this mode, the output of the FIFO remains unchanged after the first write to a FIFO in the empty state. After the first write operation, the `empty` flag is de-asserted, indicating that data is present in the FIFO and may be read. The FIFO must be read by asserting `rden`, and the first word written into the FIFO is available at the FIFO outputs on the next `rdclk` clock cycle. Each subsequent read operation updates the FIFO outputs with the next stored data word if it is available (`empty` = 0).
- `fwft` = 1'b1 (acknowledge mode) - When the `fwft` parameter is `1'b1`, the FIFO behaves as a first-word-fall-through FIFO, meaning that when the FIFO is empty, the first data word written to the FIFO is presented on the output pins as soon as possible, without waiting for `rden` to be asserted. After a reset (or after the last word has been read from the FIFO) the FIFO is in an empty state as indicated by assertion of `empty`. The output of the FIFO is updated after the next write to the FIFO, and `empty` is de-asserted indicating that there is data in the FIFO that may be read. Asserting `rden` effectively acknowledges the output data currently on the `dout` pins, allowing the FIFO to move to the next data word if not empty. Each subsequent read operation updates the FIFO outputs with the next stored data word if it is available (`empty` = `1'b0`). First-word fall-through mode effectively makes the FIFO one element deeper.

Timing Diagrams

The following diagram shows the operation of the FIFO in asynchronous mode when the FIFO is empty, where `aempty_offset` = 3. This diagram assumes that all signals not shown, such as `rstn`, are de-asserted.

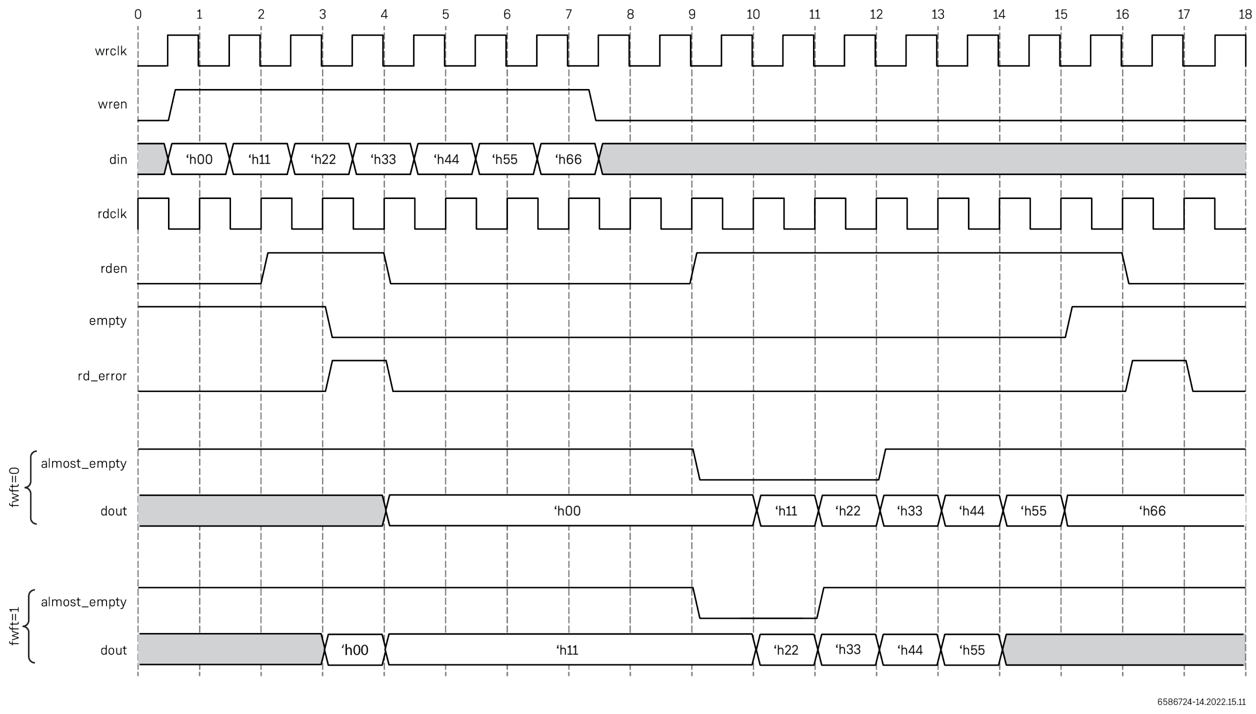


Figure 144: ***Asynchronous Mode Empty FIFO Timing Diagram***

The events of each clock cycle in the preceding diagram are described in the following table.

Table 284: *Asynchronous Mode Empty FIFO Timing Diagram Events*

Event	Description
0-6	`wren` is asserted synchronous to `wrclk`, writing seven data words to the FIFO. • Two or three clock cycles after the first write, `empty` is de-asserted synchronous to `rdclk`. If `fwft` = `1'b1`, the first data is presented on `dout` when `empty` is de-asserted. • After the fifth write, four words remain in the FIFO (since the first word has already been read). The amount of words is greater than `aempty_offset` (3), so `almost_empty` is asserted two or three clock cycles later, synchronous to `rdclk`.
2	`rden` is asserted indicating an attempt to read from the FIFO. Since the `empty` output remains asserted, the read fails, and `rd_err` is asserted on the following clock cycle. The data on `dout` does not change.
3	`rden` is also asserted, reading the first data word from the FIFO. • `fwft` = `1'b0` – the data arrives on `dout` on the following cycle. • `fwft` = `1'b1` – the first data word on `dout` is replaced by the second data word.
4	`rden` is not asserted in this cycle. Nothing is read from the FIFO.
6-8	No control signals are asserted.
9	`rden` is asserted, reading the second data word from the FIFO. • `fwft` = `1'b0` – the data arrives on `dout` on the following cycle. • `fwft` = `1'b1` – the previous data word on `dout` is replaced by the next data word.
10	`rden` is asserted, reading the third data word from the FIFO. • `fwft` = `1'b0` – the data arrives on `dout` on the following cycle. • `fwft` = `1'b1` – the previous data word on `dout` is replaced by the next data word, with only four more words remaining in the FIFO. `almost_empty` is de-asserted.
11	`rden` is asserted, reading the fourth data word from the FIFO. • `fwft` = `1'b0` – the data arrives on `dout` on the following cycle, with only four more words remaining in the FIFO. `almost_empty` is de-asserted. • `fwft` = `1'b1` – the previous data word on `dout` is replaced by the next data word.
12	`rden` is asserted, reading the fifth data word from the FIFO.
13	`rden` is asserted, reading the sixth data word from the FIFO.
14	`rden` is asserted, reading the seventh and last data word from the FIFO. Since the FIFO is empty, `empty` is asserted on the next `rdclk` cycle.
15	`rden` is asserted even though the FIFO is empty. `rd_error` is asserted on the following clock edge, and the FIFO contents are unchanged.

The following diagram shows the operation of the FIFO in asynchronous mode, starting when there are five locations remaining in the FIFO where `afull_offset` is 3. This diagram assumes that all signals not shown, such as `rstn`, are de-asserted, and `ptr_sync_mode` is `1'b0`.

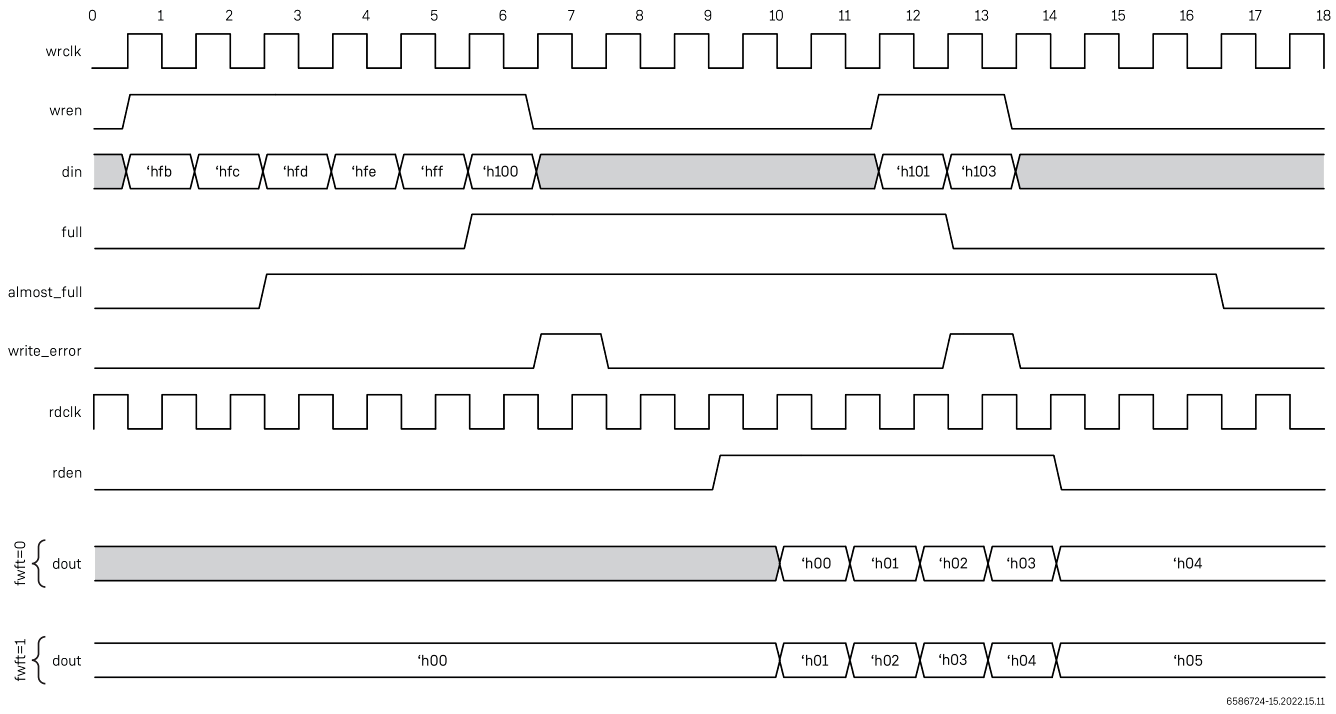


Figure 145: ***Asynchronous Mode Full FIFO Timing Diagram***

The events of each clock cycle in the preceding diagram are described in the following table.

Table 285: *Asynchronous Mode Full FIFO Timing Diagram Events*

Event	Description
1-5	`wren` is asserted, writing five data words to the FIFO. • After the second write, only three locations remain, so `almost_full` is asserted on the next clock cycle. • After the fifth write, the last element in the FIFO has been used, and `full` is asserted on the following clock cycle.
6	`wren` is asserted. Since the FIFO is already full, the write operation does not take place, and `write_error` is asserted on the following clock cycle.
7-8	No operation.
9	`rden` is asserted, and the next output data is read and presented on `dout`. Two or Three cycles later, `full` is de-asserted synchronous to `wrclk`. • `fwft = 1'b0` – the first data arrives on `dout` on the following cycle. • `fwft = 1'b1` – the first data has been present on the output since it was first written. This data is replaced by the next data being read from the FIFO.
10	`rden` is asserted, and the next output data is read from the FIFO and presented on `dout`.
11	`rden` is asserted synchronous to `rdclk` and `wren` is asserted synchronous to `wrclk`, meaning that both a read and write are to be performed. Since `full` is asserted, the write fails, and `write_error` is asserted on the following `wrclk` cycle. The read is successful, and the output data is updated on the following `rdclk` cycle.
12	`rden` is asserted synchronous to `rdclk` and `wren` is asserted synchronous to `wrclk`, The input word is written to the FIFO while the next output word is read from the FIFO and presented on `dout`. Since `full` is not asserted, both operations are successful. Now more than three unused locations remain in the FIFO, so `almost_full` is de-asserted two or three cycles later, synchronous to `wrclk`.
13	`rden` is asserted, and the next output data is read and presented on `dout`.

Instantiation Template

Verilog

```
ACX_LRAMFIFO #(
    .ptr_sync_mode    (1'b0),
    .read_width        (32),
    .write_width    (32),
    .read_depth        (128),
    .write_depth    (128),
    .fwft_mode        (1'b0),
    .afull_offset    (8'h4),
    .aempty_offset    (8'h4),
    .hold_output    (1'b0)
) instance_name (
    .rstn            (user_rstn),
    .wrclk            (user_wrclk),
    .wren            (user_wren),
    .din            (user_din),
    .full            (user_full),
    .almost_full    (user_almost_full),
    .write_err        (user_write_err),
    .rdclk            (user_rdclk),
    .rden            (user_rden),
    .dout            (user_dout),
    .empty            (user_empty),
    .almost_empty    (user_almost_empty)
);
```

VHDL

```
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.core.all;
------------- DONE ACHRONIX LIBRARY ---------
-- Component Instantiation
instance_name : ACX_LRAMFIFO
generic map
(
 ptr_sync_mode    => 0,
 read_width      => 32,
 write_width      => 32,
 read_depth      => 128,
 write_depth      => 128,
 fwft_mode      => 0,
 afull_offset      => 4,
 aempty_offset  => 4,
 hold_output     => 0
)
port map
(
 rstn              => user_rstn,
 wrclk              => user_wrclk,
 wren              => user_wren,
 din              => user_din,
 full              => user_full,
 almost_full     => user_almost_full,
 write_err          => user_write_err,
 rdclk              => user_rdclk,
 rden              => user_rden,
 dout              => user_dout,
 empty              => user_empty,
 almost_empty     => user_almost_empty,
 read_err         => user_read_err
);
```

ACX_LRAM2K_FIFO

The ACX_LRAM2K_FIFO implements a 2Kb FIFO, configured as either 72 bits wide by 32 words deep, or 36 bits wide by 64 words deep. Each port width can be independently configured and on different clock domains. For higher performance operation, an additional output register can be enabled. Enabling the output register causes an additional cycle of read latency.

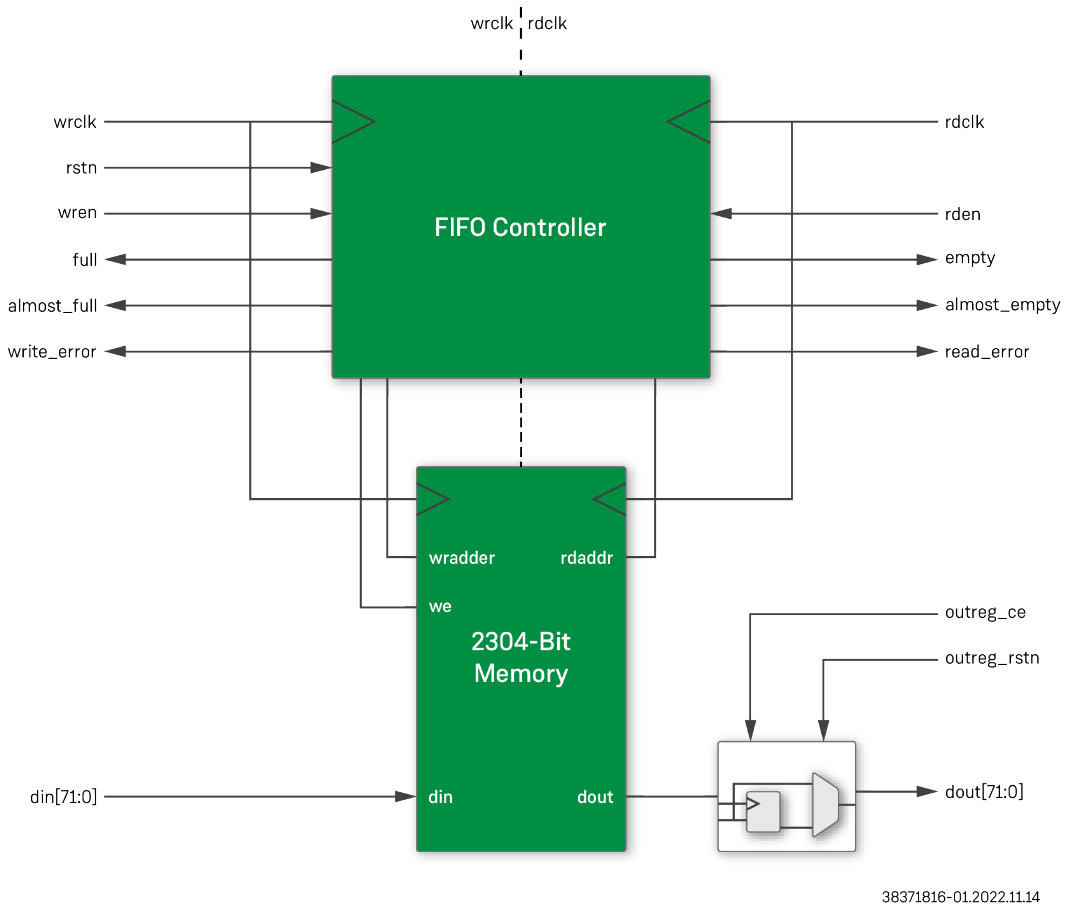


Figure 146: ***ACX_LRAM2K_FIFO Block Diagram***

Parameters

Table 286: *ACX_LRAM2K_FIFO Parameters*

Parameter	Supported Values	Default Value	Description
`read_width`	36, 72	72	Controls the width of the read port. Can be different from `write_width`: `read_width` = 72 – depth = 32 words. `read_width` = 36 – depth = 64 words.
`write_width`	36, 72	72	Controls the width of the write port. Can be different from `read_width`: `write_width` = 72 – depth = 32 words. `write_width` = 36 – depth = 64 words.
`rdclk_polarity`	"rise", "fall"	"rise"	Controls whether the `rdclk` signal uses the falling or the rising edge: "rise" – rising edge. "fall" – falling edge.
`wrclk_polarity`	"rise", "fall"	"rise"	Controls whether the `wrclk` signal uses the falling or the rising edge: "rise" – rising edge. "fall" – falling edge.
`outreg_enable`	0, 1	1	Controls whether the output register is enabled: 0 – disables the output register and results in a read latency of one cycle. 1 – enables the output register and results in a read latency of two cycles. Only effective when `fwft_mode` = 0. When `fwft_mode` = 1, the output defaults to `outreg_enable` = 0.
`sync_mode`	0, 1	0	Controls whether the FIFO operates in synchronous or asynchronous mode: 0 – asynchronous mode. 1 – synchronous mode. In synchronous mode, the two input clocks must be driven by the same clock input and pointer synchronization logic is bypassed resulting in lower latency for flag assertion.
`afull_threshold`	`0–6'h3F`	`6'h4`	The `afull_threshold` parameter defines the word depth at which the `almost_full` output changes. The `almost_full` signal may be used to determine the number of blind writes to the FIFO that can be issued without monitoring the `full` flag. For example, if the `afull_threshold` parameter is set to `6'h04` and the `almost_full` signal is de-asserted, there are at least five empty locations in the FIFO. All five words may be written without overflowing the FIFO and causing `write_error` to be asserted.
`aempty_threshold`	`0–6'h3F`	`6'h4`	The `aempty_threshold` parameter defines the word depth at which the `almost_empty` output changes. The `almost_empty` signal may be used to determine the number of blind reads from the FIFO that can be performed without monitoring the `empty` flag. For example, if the `aempty_threshold` parameter is set to `6'h04` and the `almost_empty` flag is de-asserted, there are at least five words in the FIFO. All five words may be read without underflowing the FIFO and causing the `read_error` flag to be asserted.
`fwft_mode`	0, 1	0	First-word fall through. Controls the behavior of data at the output of the FIFO relative to `rden`: 0 – first word data is presented at the output of the FIFO on the rising edge of `wrclk` except for `sync_mode` = 0 and `output_enable` = 1. For `sync_mode` = 1 and `output_enable` = 1, data is present one cycle later. 1 – first word data is presented at the output of the FIFO on the rising edge of `wrclk` in all modes. `outreg_enable` has no effect when `fwft_mode` = 1.

Ports

Table 287: *ACX_LRAM2K_FIFO Pin Descriptions*

Name	Direction	Description
`rstn`	Input	Asynchronous reset input. This signal resets the entire FIFO.
`wrclk`	Input	Write clock input. Write operations are fully synchronous and occur upon the active edge of the `wrclk` input when `wren` is asserted. The active edge of `wrclk` is determined by the `wrclk_polarity` parameter.
`wren`	Input	Write port enable. Assert `wren` high to write data to the FIFO.
`din[71:0]`	Input	Write port data input. When `write_width` is less than 72, the input data must be assigned from `din[0]` upwards (right justified).
`full`	Output	Asserted high when the FIFO is full.
`almost_full`	Output	Asserted high when remaining space in the FIFO is less than, or equal to, `afull_threshold`.
`write_error`	Output	Asserted the cycle after a write to the FIFO when the FIFO is already full.
`rdclk`	Input	Read clock input. Read operations are fully synchronous and occur upon the active edge of the `rdclk` input when the `wren` signal is asserted. The active edge of `rdclk` is determined by `rdclk_polarity` parameter.
`rden`	Input	Read port enable. Assert `rden` high to perform a read operation.
`outreg_rstn`	Input	Output register synchronous reset. When `outreg_rstn` is asserted low, the value of the output register is reset to 0.
`outreg_ce`	Input	Active-high output register clock enable. When `outreg_enable` = 1, de-asserting `outreg_ce` causes the LRAM to hold the `dout[]` signal unchanged, independent of a read operation. When `outreg_enable` = 0, the `outreg_ce` input is ignored.
`empty`	Output	Asserted high when the FIFO is empty.
`almost_empty`	Output	Asserted high when the FIFO contains less than, or equal to, `aempty_threshold` words.
`read_error`	Output	Asserted on the cycle after a read request to the FIFO when the FIFO is already empty.
`dout[71:0]`	Output	Read port data output. If `read_width` is less than 72, the output data is assigned from `dout[0]` upwards, (right justified).

Read and Write Operations

Write Operation

Write operations are signaled by asserting the `wren` signal. The value of `din` is stored to the next available FIFO location on the rising edge of `wrclk` whenever `wren` is asserted, and `full` is deasserted.

Read Operation

Read operations are signaled by asserting the `rden` signal. The next FIFO location contents are latched to the output latches on the rising edge of `rdclk` whenever `rden` is asserted and `empty` is deasserted. If `outreg_enable` = 1 and `fwft_mode` = 0, the FIFO contents are available on `dout` on the following rising edge of `rdclk`.

First Word Fall Through (FWFT)

The FIFO operates in a first-word fall-through mode (where the first word written to the FIFO is presented on the output before `rden` is asserted) for the following configurations:

- `fwft_mode` = 0 and `sync_mode` = 1 – FIFO natively operates as FWFT. With `outreg_enable` = 1, the first word takes an additional cycle of `rdclk` to be present on the output.
- `fwft_mode` = 0 and `sync_mode` = 0 – FIFO operates as FWFT when `outreg_enable` = 0.
- `fwft_mode` = 1 – FIFO operates as FWFT. `outreg_enable` has no effect and the next data is output on the rising edge of `rdclk` when `rden` is asserted.

Output Timing

The ACX_LRAM2K_FIFO has two options for interface timing controlled by the `outreg_enable` parameter:

- Latched mode – `outreg_enable` = 0. In latched mode, when the FIFO contents are read, the data is latched into the output latches on the rising edge of `rdclk`, providing a read operation with one cycle of latency.
- Registered mode – `outreg_enable` = 1. In registered mode, there is an additional register after the latch supporting higher-frequency designs and providing a read operation with two cycles of latency.

Table 288: ***ACX_LRAM2K_FIFO Output Function Table for Latched Mode***

Operation (1)	rdclk	outlatch_rstn	rden	dout[]
Hold	X	X	X	Hold previous value
Reset latch	↑	0	X	0
Hold	↑	1	0	Hold previous value
Read	↑	1	1	Next FIFO entry

Table Notes

1. Operation assumes rising-edge clock and active-high port enable.

Table 289: *ACX_LRAM2K_FIFO Output Function Table for Registered Mode*

Operation [(1)]	rdclk	outreg_rstn	outregce	dout[]
Hold	X	X	X	Previous `dout[]`
Reset Output	↑	0	1	0
Hold	↑	1	0	Previous `dout[]`
Update Output	↑	1	1	Registered from latch output

Table Notes

1. Operation assumes active-high clock, output register clock enable, and output register reset.

Timing Diagrams

Synchronous Mode

Data output, `dout`, timing for all combinations of `outreg_enable` and `fwft_mode` is shown in the following waveform.

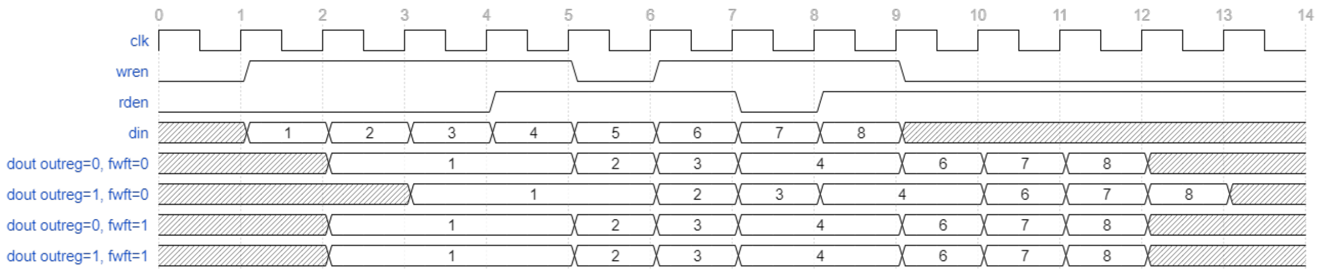


Figure 147: ***Output Timing With sync_mode = 1***

Asynchronous Mode

Data output, `dout`, timing for all combinations of `outreg_enable` and `fwft_mode` is shown in the following waveform.

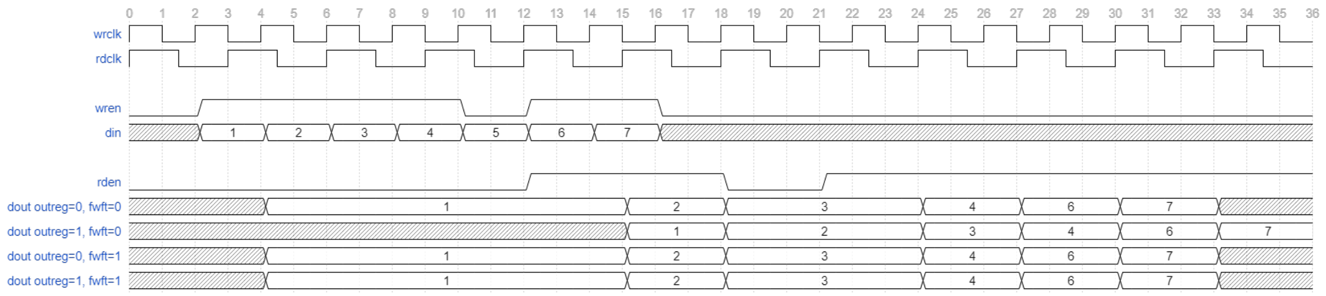


Figure 148: ***Output Timing With sync_mode = 0***

Inference

The ACX_LRAM2K_FIFO is not inferrable.

Instantiation Templates

Verilog

```
ACX_LRAM2K_FIFO #(
    .aempty_threshold    (aempty_threshold),
    .afull_threshold     (afull_threshold),
    .fwft_mode           (fwft_mode),
    .outreg_enable       (outreg_enable),
    .rdclk_polarity      (rdclk_polarity),
    .read_width          (read_width),
    .sync_mode           (sync_mode),
    .wrclk_polarity      (wrclk_polarity),
    .write_width         (write_width)
) instance_name (
    .din                 (din),
    .rstn                (rstn),
    .wrclk               (wrclk),
    .rdclk               (rdclk),
    .wren                (wren),
    .rden                (rden),
    .outreg_rstn         (outreg_rstn),
    .outreg_ce           (outreg_ce),
    .dout                (dout),
    .almost_full         (almost_full),
    .full                (full),
    .almost_empty        (almost_empty),
    .empty               (empty),
    .write_error         (write_error),
    .read_error          (read_error)
);
```

VHDL

```
-- VHDL Instantiation template for ACX_LRAM2K_FIFO
instance_name : ACX_LRAM2K_FIFO
generic map (
    aempty_threshold         => aempty_threshold,
    afull_threshold          => afull_threshold,
    fwft_mode                => fwft_mode,
    outreg_enable            => outreg_enable,
    rdclk_polarity           => rdclk_polarity,
    read_width               => read_width,
    sync_mode                => sync_mode,
    wrclk_polarity           => wrclk_polarity,
    write_width              => write_width
)
port map (
    din                      => user_din,
    rstn                     => user_rstn,
    wrclk                    => user_wrclk,
    rdclk                    => user_rdclk,
    wren                     => user_wren,
    rden                     => user_rden,
    outreg_rstn              => user_outreg_rstn,
    outreg_ce                => user_outreg_ce,
    dout                     => user_dout,
    almost_full              => user_almost_full,
    full                     => user_full,
    almost_empty             => user_almost_empty,
    empty                    => user_empty,
    write_error              => user_write_error,
    read_error               => user_read_error
);
```

Chapter - 7: JTAG TAP Controller Functions

The JTAG interface (IEEE Standard 1149.1) is a serial interface commonly used for device testing. This interface is much simpler than others such as PCIe or Ethernet but has a significantly lower bandwidth. However, for applications with low-throughput requirements, this simplicity is an advantage as it greatly reduces the time needed for bring-up.

Achronix devices have a built in JTAG interface with the following uses:

- Traditional device testing, such as with boundary scan
- Programming the device with a configuration bitstream
- A generic communication interface to a user design mapped to a Speedcore instance.

This section focuses on the latter application.

The built in JTAG controller is called a TAP controller, which is defined by the JTAG standard. The interface between a TAP controller and a Speedcore instance is referred to as the JTAP interface. While the core has only one JTAP interface in a Speedcore instance, the JTAP library enables multiplexing this interface between different parts of the user design.

The following figure shows the components of a system using the JTAP interface.

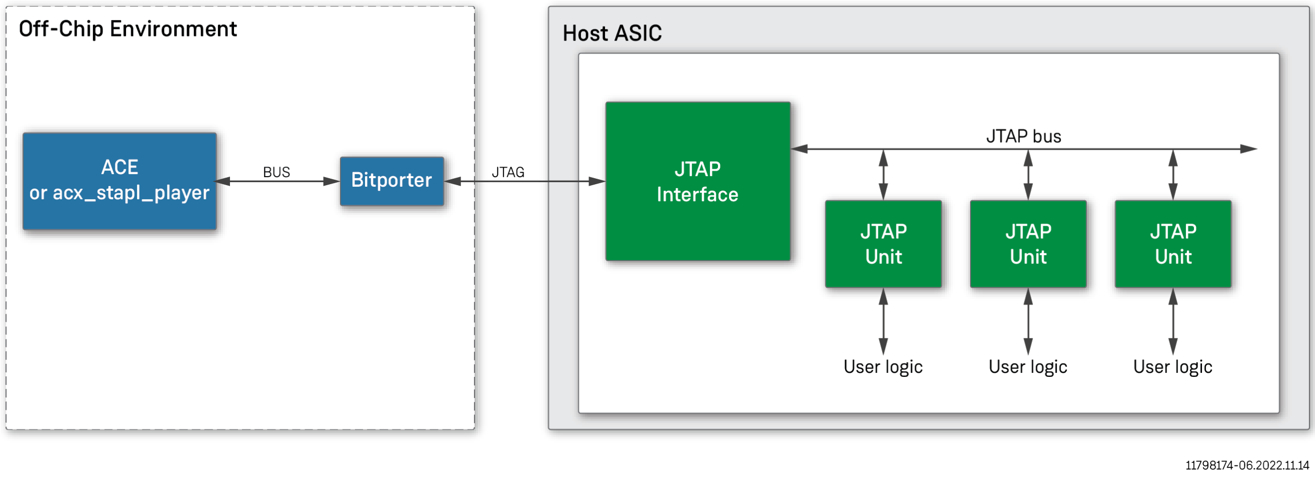


Figure 149: ***JTAG System Overview***

A common way of communicating over a JTAG interface is with the STAPL language. ACE includes a STAPL interpreter, `acx_stapl_player`, accessible as a stand-alone program or with the `run_stapl_action` ACE command. The ACE STAPL player accesses the JTAG interface through a Bitporter device, or through an FTDI FT2232H interface cable. Other JTAG-compliant software and hardware may be substituted for the off-chip environment.

In the Speedcore case, there are two types of instances:

1. One JTAP interface instance (ACX_JTAP_INTERFACE).
2. Any number of JTAP units.

The Speedcore library has two variants of JTAP unit:

1. The ACX_JTAP_REG_UNIT with a parallel user interface.
2. The ACX_JTAP_UNIT with a serial user interface.

These JTAP units are independent of each other and share the JTAP interface as described in the following section.

To access the macros described in this section, the JTAP library must be included:

```
`include "speedster<technology>/common/speedster<technology>_jtap.v"
```

`<technology>` is replaced with the target technology library name (i.e., `16t`).

ACX_JTAP_INTERFACE

The ACX_JTAP_INTERFACE includes the hard TAP controller and must be connected directly to the top-level JTAG ports without IPIN or OPIN instances. The macro also includes a 6-bit unit ID register used to select between multiple connected JTAP units, each having a unique identifying ID. To use the JTAP interface, a design must have one (and only one) ACX_JTAP_INTERFACE instance.

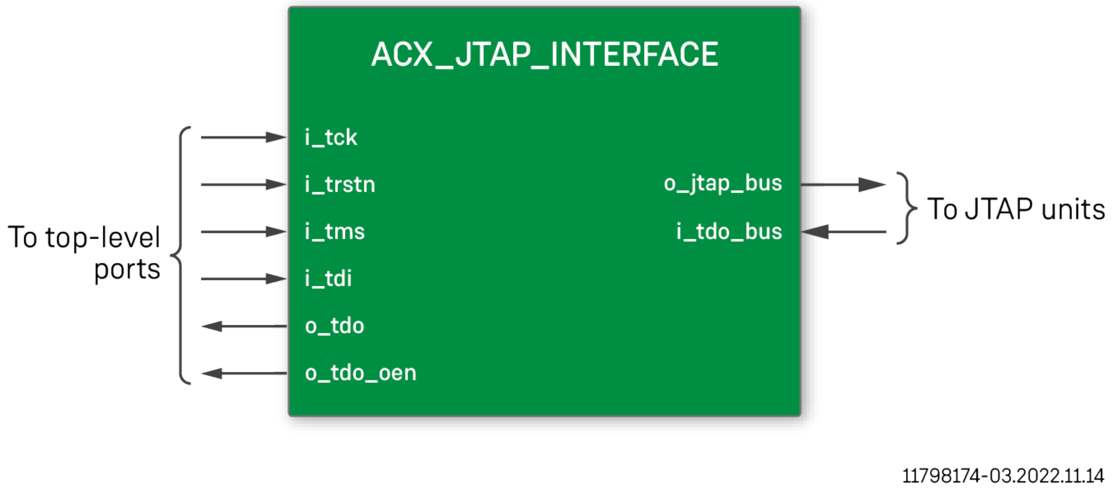


Figure 150: *ACX_JTAP_INTERFACE Pins*

Ports

Table 290: *ACX_JTAP_INTERFACE Pins*

Pin Name	Direction	Description
JTAG Pins		
`i_tck`	Input	JTAG test clock.
`i_trstn`	Input	JTAG test active-low reset.
`i_tdi`	Input	JTAG test data in.
`i_tms`	Input	JTAG test mode select.
`o_tdo`	Output	JTAG test data out.
`o_tdo_oen`(1)	Output	Active-low output enable for `o_tdo`
JTAP Bus Pins		
`o_jtap_bus`	Output	Output to JTAP units. Abstract type named `jtap_bus_tp`.
`i_tdo_bus`	Input	Input from `o_tdo_bus` of JTAP units.

Table Notes

1. The `o_tdo_oen` signal only exists in Speedcore products to be combined with `o_tdo` to drive a tri-state pad. Achronix stand-alone FPGAs already include the tri-state pad to drive `o_tdo`.

Connection to the JTAP Bus

The `o_jtap_bus` output is a Verilog struct of type `jtap_bus_tp` combining several wires (as defined in `speedster<technology>_jtap.v`). The struct `o_jtap_bus` fans out to all JTAP units. Simply use the struct by name as illustrated in the following code snippet (no need for concern with the contents).

```
jtap_bus_tp jtap_bus;

ACX_JTAP_INTERFACE x_jtap_interface (
    ...
    .o_jtap_bus(jtap_bus)
);

ACX_JTAP_REG_UNIT x_jtap_unit (
    .i_jtap_bus(jtap_bus)
    ...
);
```

Each JTAP unit has a single-bit signal, `o_tdo_bus`, that must be connected to the input `i_tdo_bus` of the ACX_JTAP_INTERFACE. There are two methods to make this connection:

1. Multiple units can be chained together by connecting the `o_tdo_bus` of one unit to the `i_tdo_bus` of another.
2. Multiple `o_tdo_bus` signals can be ORed together.

The following figure illustrates both methods.

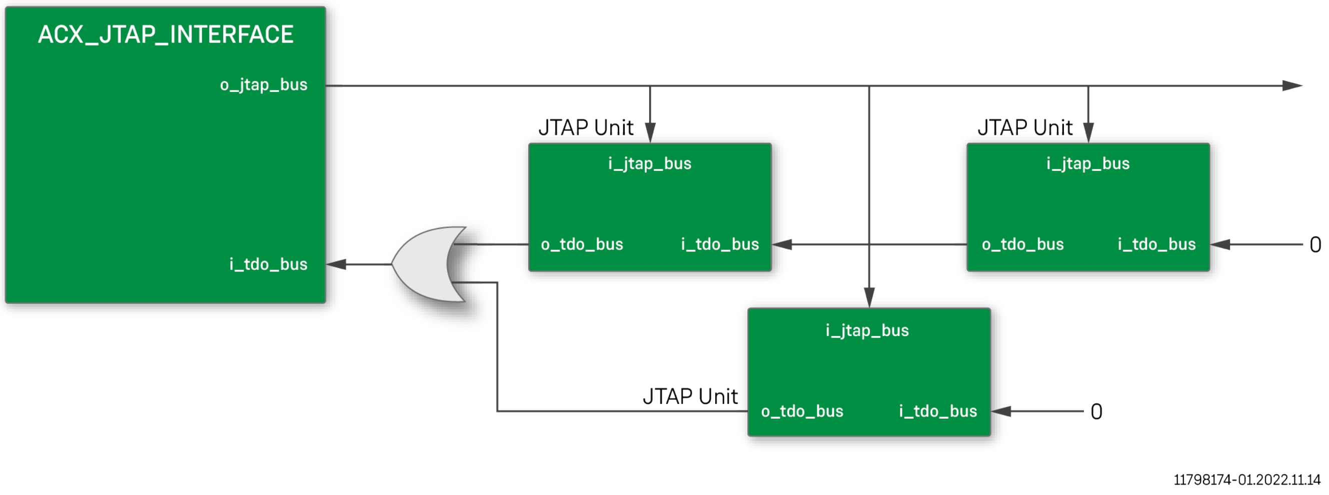


Figure 151: *JTAP Bus Example*

ACX_JTAP_REG_UNIT

ACX_JTAP_REG_UNIT connects to the JTAP bus and presents a parallel interface to the user design. Each JTAP unit (ACX_JTAP_REG_UNIT or ACX_JTAP_UNIT) has a unique, user-selected unit ID. The off-chip environment specifies the ID of the unit to be selected. The unit control outputs are only asserted when the unit is selected.

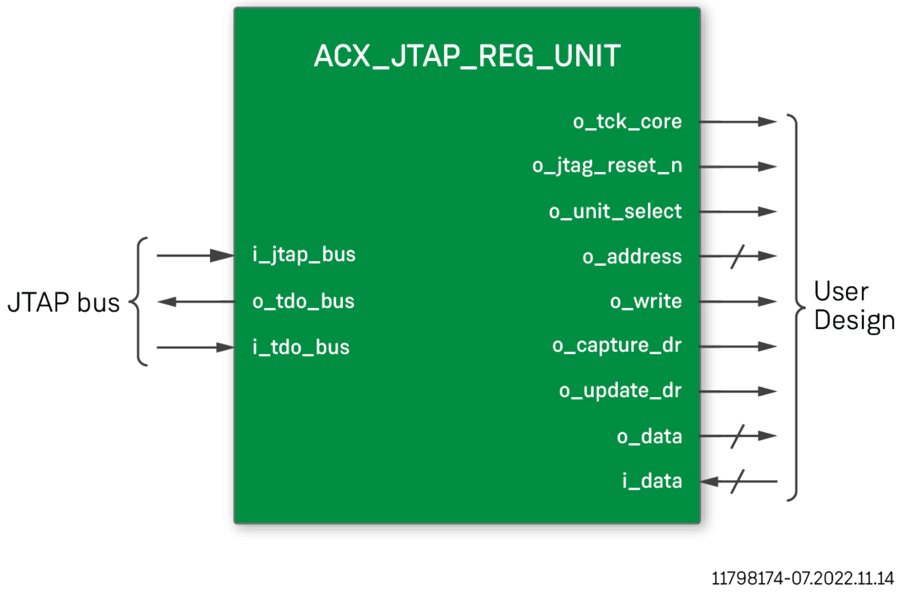


Figure 152: *ACX_JTAP_REG_UNIT Pins*

Parameters

Table 291: *ACX_JTAP_REG_UNIT Parameters*

Parameter	Default Value	Description
`UNIT_ID[5:0]`	1	Unique unit ID. ID 0 is reserved for Snapshot.
`ADDR_WIDTH`	0	Number of address bits, if any.
`ADDR_INC`	0	Address increment amount, if any.
`INPUT_DELAY`	0	Extra delay between `o_capture_dr` and sampling of `i_data`.
`DATA_WIDTH`	32	Number of data bits.

Ports

Table 292: *ACX_JTAP_REG_UNIT Pins*

Signal	Direction	Description
JTAP Bus		
`i_jtap_bus`	Input	JTAP bus input, driven by the ACX_JTAP_INTERFACE. Abstract type named `jtap_bus_tp`.
`i_tdo_bus`	Input	Input from `o_tdo_bus` of another JTAP unit; used to chain units. Tie to `1'b0` if unused.
`o_tdo_bus`	Output	Output to `i_tdo_bus` of another JTAP unit (in a chain) or of the ACX_JTAP_INTERFACE.
User Design Interface		
`o_tck_core`	Output	JTAG clock. The frequency of this clock is typically <= 10 MHz and the clock may stop between transactions.
`o_jtag_reset_n`	Output	Active-low reset for user logic. This signal is asserted when the TAP controller enters the reset state, but only if the ACX_JTAP_REG_UNIT instance was selected at the time of reset. An effect of the reset is to deselect the unit (because the JTAG instruction register is reset to `JTAG_IDCODE`).
`o_unit_select`	Output	High when this unit has been selected and can receive transactions. This signal usually can be ignored because the other control signals (`o_jtag_reset_n`, `o_capture_dr`, and `o_update_dr`) are only asserted when this unit is selected.
`o_address [ADDR_WIDTH-1:0]`	Output	Address for the transaction (valid when `o_capture_dr` is asserted).
`o_write`	Output	High when the write bit is set (valid when `o_capture_dr` is asserted).
`o_capture_dr`	Output	Asserted high to indicate the start of a transaction.
`o_update_dr`	Output	Asserted high to indicate that `o_data` is valid. This signal is only asserted for write transactions. Transitions on the falling clock edge.
`i_data [DATA_WIDTH-1:0]`	Input	Input (read) value. This signal is registered `INPUT_DELAY` + 0.5 cycles after assertion of `o_capture_dr`.
`o_data [DATA_WIDTH-1:0]`	Output	Output (write) value. This signal is valid when `o_update_dr` is asserted and stable until assertion of `o_capture_dr`.

Each transaction has an address plus a write bit. Regardless of whether the write bit is set, all transactions return data to the off-chip environment. The environment can choose to ignore this data during write operations.

Every transaction starts with assertion of `o_capture_dr` ("capture data register"). By default `i_data` is sampled one half cycle after `o_capture_dr` is asserted (t = 0.5 in the diagram). To allow more time for data to appear, a non-zero `INPUT_DELAY` can be specified (data capture is delayed by `INPUT_DELAY` cycles). The following diagram illustrates `INPUT_DELAY` = S = 1.

The `o_update_dr` signal is only asserted for a write. Typically, the write data, `o_data` , is sampled while `o_update_dr` is high (at t=7 or t=7.5 in the diagram), though it remains valid until the next assertion of `o_capture_dr`.

The `o_write` signal is valid at the same time as `o_address`.

The following timing diagram illustrates the control and data signals, where D = `DATA_WIDTH`, S = `INPUT_DELAY` .

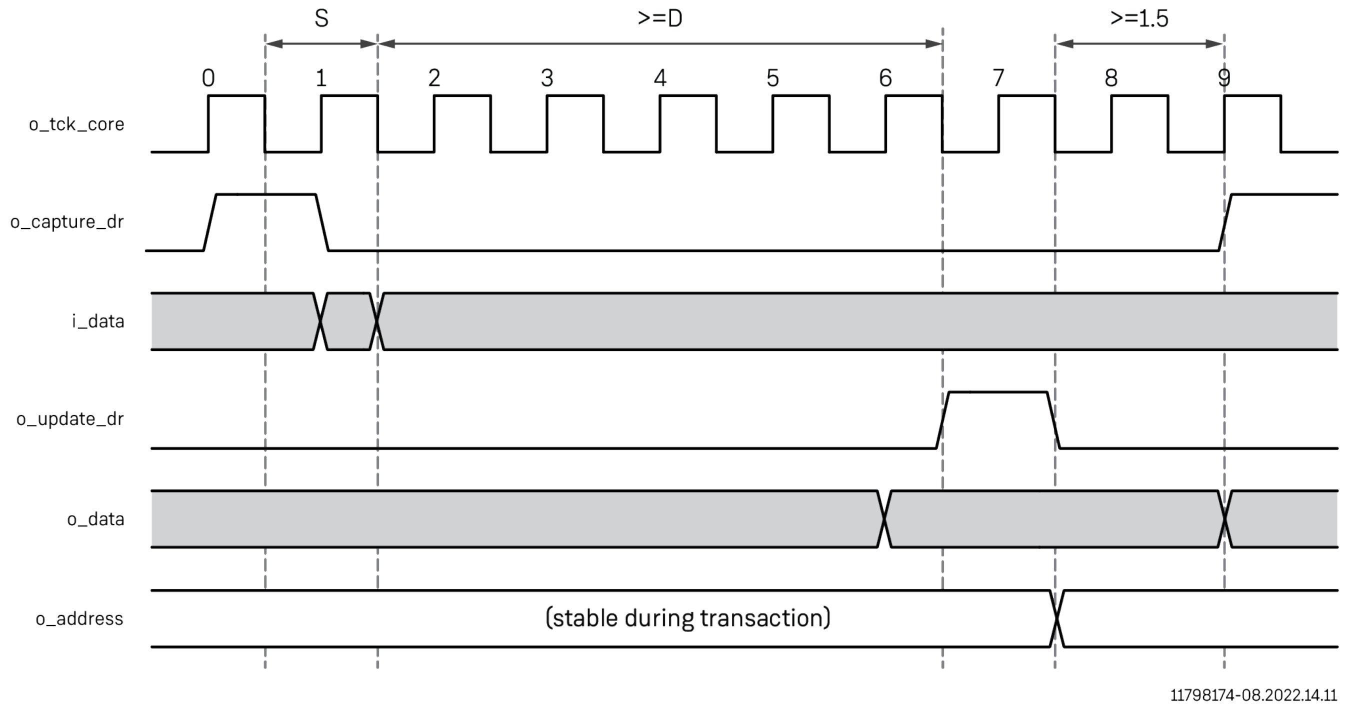


Figure 153: ***ACX_JTAP_REG_UNIT Signal Timing.***

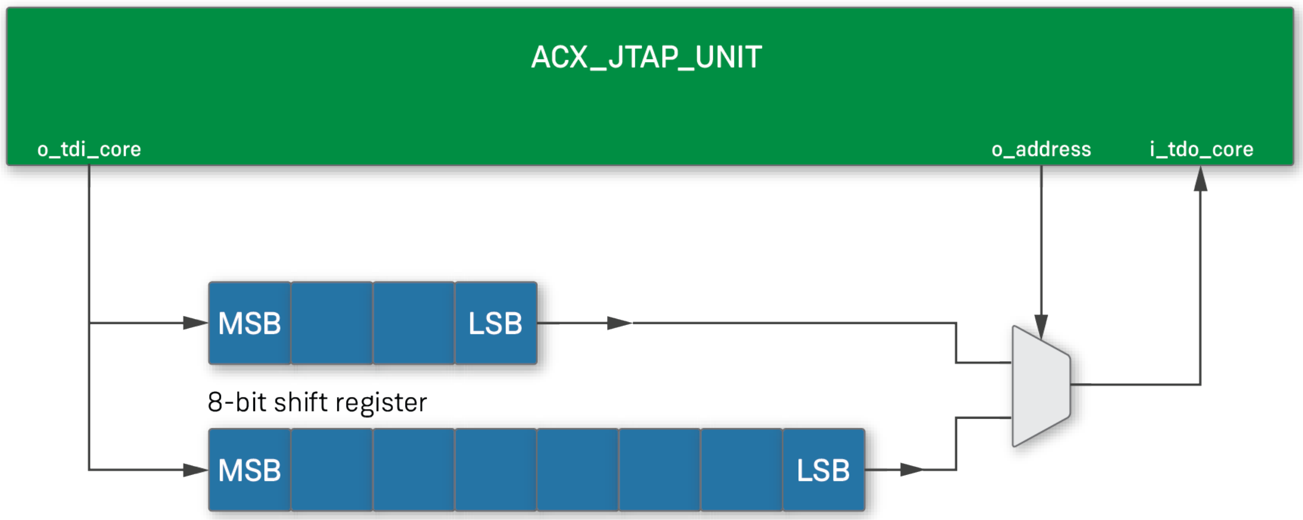
ACX_JTAP_UNIT

ACX_JTAP_UNIT connects to the JTAP bus and presents a serial interface to the user design. The serial interface closely matches a traditional JTAG interface — the user design must implement a shift register connected between the `o_tdi_core` and `i_tdo_core` pins of the ACX_JTAP_UNIT. During a transaction, read data is shifted from the shift register to the off-chip environment, while simultaneously, write data is shifted from the off-chip environment into the shift register. For convenience, the library provides a shift register module, ACX_JTAP_SHIFT_REG, but other shift register designs may be used as well.

While this shift register interface is slightly more complex than the parallel interface of the ACX_JTAP_REG_UNIT, the advantage of the serial interface is that it allows changing the data width on a per-transaction basis. For instance, the user design can have two shift registers of different sizes and use the address to select the appropriate register. The following figure provides an example of such an arrangement.

Note

The shift direction is from MSB to LSB, with the LSB of the shift register tied to the input `i_tdo_core` of the ACX_JTAP_UNIT.



11798174-11.2022.11.14

Figure 154: ***Example: ACX_JTAP_UNIT With Two Shift Registers of Different Width***

Each JTAP unit (ACX_JTAP_REG_UNIT or ACX_JTAP_UNIT) has a unique, user-selected unit ID. The off-chip environment specifies the ID of the unit to be selected. The unit control outputs are only asserted when the unit is selected.

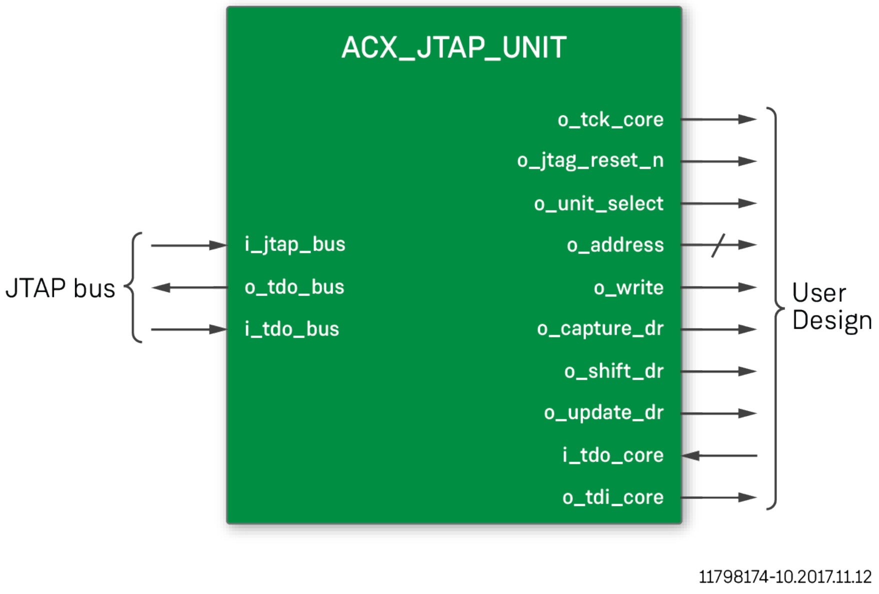


Figure 155: *ACX_JTAP_UNIT Pins*

Parameters

Table 293: *ACX_JTAP_UNIT Parameters*

Parameter	Default Value	Description
`UNIT_ID[5:0]`	1	Unique unit ID. ID 0 is reserved for Snapshot.
`ADDR_WIDTH`	0	Number of address bits, if any.
`ADDR_INC`	0	Address increment amount, if any.
`SHIFT_DELAY`	0	Extra delay between `o_capture_dr` and `o_shift_dr`.

Ports

Table 294: *ACX_JTAP_UNIT Pins*

Signal	Direction	Description
JTAP Bus		
`i_jtap_bus`	Input	JTAP bus input, driven by the ACX_JTAP_INTERFACE. Abstract type named `jtap_bus_tp`.
`i_tdo_bus`	Input	Input from `o_tdo_bus` of another JTAP unit; used to chain units. Tie to `1'b0` if unused.
`o_tdo_bus`	Output	Output to `i_tdo_bus` of another JTAP unit (in a chain) or of the ACX_JTAP_INTERFACE.
User Design Interface		
`o_tck_core`	Output	JTAG clock. The frequency of this clock is typically <= 10 MHz and the clock may stop between transactions.
`o_jtag_reset_n`	Output	Active-low reset for user logic. This signal is asserted when the TAP controller enters the reset state, but only if the ACX_JTAP_UNIT instance was selected at the time of reset. An effect of the reset is to deselect the unit (because the JTAG instruction register is reset to `JTAG_IDCODE`).
`o_unit_select`	Output	High when this unit has been selected and can receive transactions. This signal usually can be ignored because the other control signals (`o_jtag_reset_n`, `o_capture_dr`, `o_shift_dr`, and `o_update_dr`) are only asserted when this unit is selected.
`o_address [ADDR_WIDTH-1:0]`	Output	Address for the transaction (valid when `o_capture_dr` is asserted).
`o_write`	Output	High when the write bit is set (valid when `o_capture_dr` is asserted).
`o_capture_dr`	Output	Asserted high to indicate the start of a transaction.
`o_shift_dr`	Output	Asserted high when the user shift register must shift. Asserted `SHIFT_DELAY` + 1 cycles after assertion of `o_capture_dr`.
`o_update_dr`	Output	Asserted high to indicate that data was shifted into the user shift register, and is now valid. This signal is only asserted for write transactions with transitions occurring on falling clock edges.
`i_tdo_core`	Input	Serial data, tied to the LSB of the user shift register. Sampled at the falling clock edge when `o_shift_dr` is high.
`o_tdi_core`	Output	Serial data, input to the MSB of the user shift register.

Each transaction has an address plus a write bit. Regardless of whether the write bit is set, all transactions return data to the off-chip environment. The off-chip environment can choose to ignore this data during write operations.

The `o_capture_dr` signal indicates when the shift register should be initialized. The initial value of the register is returned to the off-chip environment as read data. The shift register must be initialized before the first shift. If `SHIFT_DELAY` = 0, initialization must take place on or before t = 1. Increasing `SHIFT_DELAY` postpones the first shift and, thus, provides more time to initialize the register. The following diagram shows `SHIFT_DELAY` = S = 1.

While `o_shift_dr` is high, the shift register must shift in the direction of the LSB. The LSB of the shift register must be connected to `i_tdo_core`. This input is sampled at the negative edge of the clock while `o_shift_dr` is high. In the diagram, with `SHIFT_DELAY` = 1, the LSB of the read data, `r0`, is sampled at t = 2.5. While `i_tdo_core` is always sampled on the negative edge of the clock, the register itself may be either a negative-edge or positive-edge shift register.

The shift register must have D bits, where D is the data width for this transaction. However, the actual number of shifts may well be larger than D (for example, multiple hardware devices may be combined in a single JTAG chain, which increases the number of shifts). Rather than counting shifts, the user design should rely on the `o_update_dr` signal to determine when the shift register contains valid data. `o_update_dr` is only asserted for a write. The `o_write` signal is valid at the same time as the address.

The timing following diagram illustrates the control and data signals, where S = `SHIFT_DELAY`, and D is the data width for this transaction.

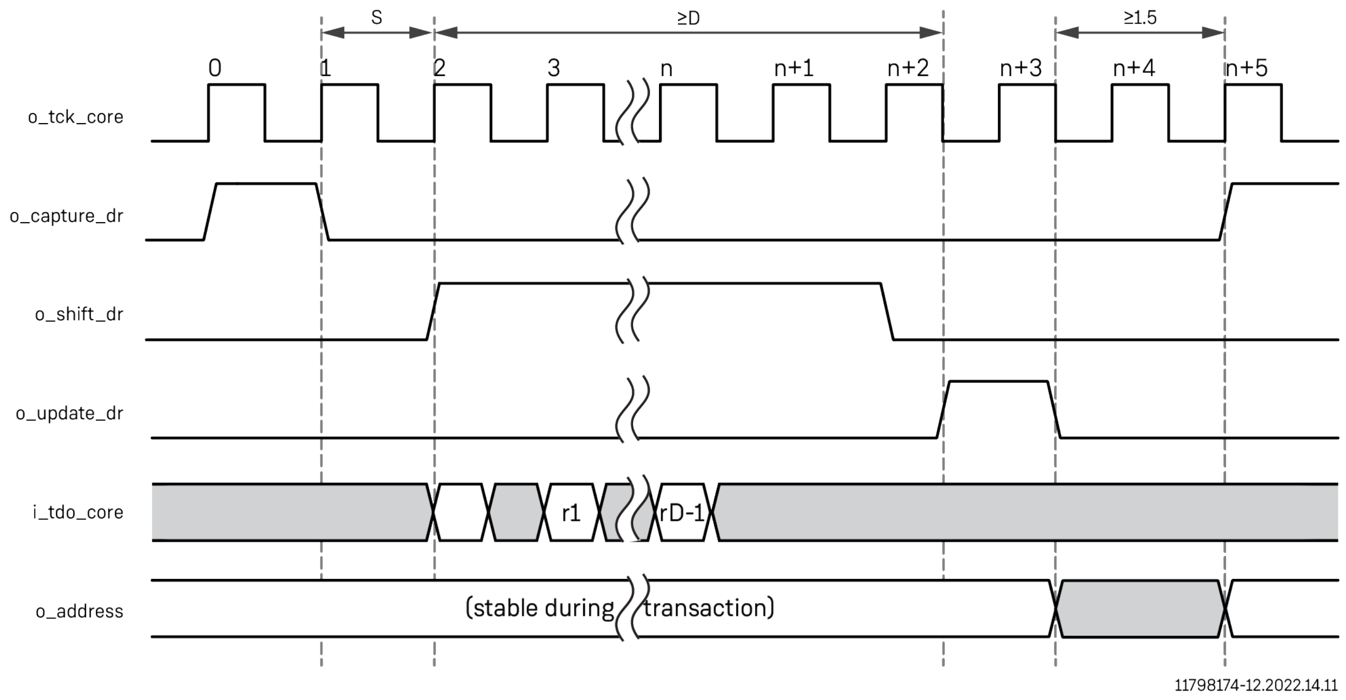


Figure 156: *ACX_JTAP_UNIT Signal Timing*

ACX_JTAP_SHIFT_REG

ACX_JTAP_SHIFT_REG is a negative-edge shift register suitable for use with ACX_JTAP_UNIT.

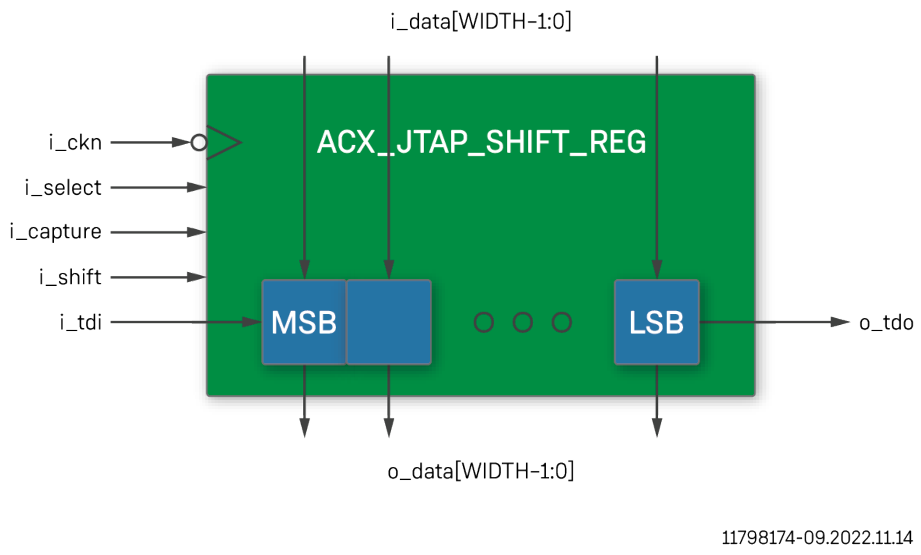


Figure 157: *ACX_JTAP_SHIFT_REG*

Parameters

Table 295: *ACX_JTAP_SHIFT_REG Parameters*

Parameter	Default Value	Description
`WIDTH`	32	Number of data bits
`INIT[WIDTH-1:0]`	`'x`	Startup register value, if specified.

Ports

Table 296: *ACX_JTAP_SHIFT_REG Pins*

Signal	Direction	Description
`i_ckn`	Input	Register clock (negative edge).
`i_data[WIDTH-1:0]`	Input	Data to be stored in shift register when `i_capture` is asserted.
`o_data[WIDTH-1:0]`	Output	Shift register value (changes on negative clock edges).
`i_select`	Input	Register select. The signals `i_capture` and `i_shift` are ignored if `i_select` is low.
`i_capture`(1)	Input	When asserted high during `i_select`, the `i_data` value is stored in the shift register.
`i_shift`(1)	Input	When asserted high during `i_select`, causes the register to shift.
`i_tdi`	Input	Serial data in, sampled at the negative edge of the clock when `i_select && i_shift`.
`o_tdi`	Output	Serial data out, i.e., the value of the lsb.

Table Notes

1. `i_capture` and `i_shift` are mutually exclusive

Communication

All interaction with the JTAP units is initiated by the off-chip environment by communicating through the JTAG interface. JTAG communication consists of two parts:

1. Setting the JTAG instruction register.
2. One or more data transactions.

All JTAG transactions are shifts, shifting the same number of bits in and out, logically corresponding to a read followed by a write. When only a read is needed, the input data consists of don't-care bits. When only a write is needed, the output data is simply ignored.

To communicate with a JTAP unit, it must be selected using its unit ID. Following selection, any number of data transactions can be performed. A data transaction has two parts:

1. An address action to specify the address.
2. A data action to transfer data.

Normally these actions alternate, but for efficiency, the address action can be skipped in some cases. The details are described as follows.

Achronix devices have a 23-bit JTAG instruction register. The following instructions are used in this section.

Table 297: *Achronix JTAG Instruction Codes*

Name	Value	Function
`JTAG_IDCODE`	`23'h7f_fffe`	Selects the device identification register.
`JTAG_JUSR1`	`23'h02_013a`	Sets the JTAP unit ID.
`JTAG_JUSR2`	`23'h02_003a`	For communication with JTAP units.

Selecting a JTAP Unit

A JTAP unit is selected when the following two conditions are both true:

1. The current unit ID matches the `UNIT_ID` parameter.
2. The JTAG instruction register is set to `JTAG_JUSR2`.

To write the unit ID:

1. Set the JTAG instruction register to `JTAG_JUSR1`.
2. Write 7 bits of data (a dummy LSB followed by the 6-bit unit ID).
3. The previous unit ID is returned.

After setting the unit ID, the instruction register must be set to `JTAG_JUSR2` to finish the selection. All communication with the JTAP unit uses the `JTAG_JUSR2` instruction.

Table 298: ***Setting the Unit ID (7 Bits)***

Number of Bits	6	1 (LSB)
Write	`unit_id`	X [(1)]
Read	prev `unit_id`	X [(1)]

Table Notes

1. "X" indicates don't care.

JTAG Reset

The JTAG interface supports two methods to apply reset:

1. A hard reset with a reset wire (the JTAG standard specifies that the reset wire is optional).
2. A soft reset where the TAP state machine is given a control sequence that puts it in a reset state.

When asserted, reset stays asserted until the TAP state machine is explicitly moved to a different state, typically by setting the instruction register.

Other than the method of application, both types of reset behave identically. If a JTAP unit was selected when reset is applied, the unit `o_jtag_reset_n` output is asserted. A reset always changes the JTAG instruction register to `JTAG_IDCODE`. As a result, a reset causes the unit to be de-selected (without affecting the duration of the `o_jtag_reset_n` signal). Before the unit can be accessed again, the instruction register must be set back to `JTAG_JUSR2`.

A reset does not change the current unit ID.

Address Action

An address action specifies an address, a write bit, and an inc bit. The returned data can be ignored. The address width, A, must match the JTAP unit `ADDR_WIDTH` parameter. The address and write values are passed to the design as `o_address` and `o_write`.

Table 299: ***Address action (A+2 bits)***

Number of Bits	1	1	A
Write	inc	write	address

When set, the inc bit causes the address to be incremented following a data action. The increment amount is the JTAP unit `ADDR_INC` parameter (even if `ADDR_INC` = 0, the inc bit must be specified).

When a unit has been selected, the first action must be an address action. Following an address action, the next action must be a data action (unless the unit is de-selected by changing the instruction register).

ADDR_WIDTH = 0

In the special case where `ADDR_WIDTH` = 0, both the address and the inc bit must be omitted. In that case, the address action consists of just one bit, write.

Data Action

A data action transfers D bits. For an ACX_JTAP_REG_UNIT, D must equal the `DATA_WIDTH` parameter. For an ACX_JTAP_UNIT, D must equal the width of the user shift register.

Note

Due to internal pipelining, the LSB is always a dummy bit. The user design only sees the D data bits.

The value S is the `INPUT_DELAY` or `SHIFT_DELAY` parameter of the unit; S may be 0.

Table 300: ***Data Action (D + S + 2 Bits)***

Number bits	1	D	S	1
Write	`skip_addr`	data	S × X	X
Read	X	data	S × X	X

If `skip_addr` is 0, the next action must be an address action. If `skip_addr` is 1, the next action must be a data action. In this case, the next action uses the existing address and write bit, except that the previously specified inc bit determines whether the address is incremented first.

If the JTAG instruction register is changed to something other than `JTAG_JUSR2` (possibly as the effect of a JTAG reset), the unit is de-selected. If it is selected again, the next action must be an address action, regardless of any previous `skip_addr`.

Revision History

Version	Date	Description
1.0	02 Aug 2016	• Initial Achronix release.
1.1	19 Aug 2016	• Added in sections for the DSP64 FIR filter implementation and the LRAMFIFO. • Corrected tables for the BRAMSDP macro.
1.2	13 Oct 2016	• Added in new clock enable and reset pins for IPIN/OPIN.
1.3	04 Nov 2016	• Updated the title of the document. • Updates to ACX_BRAMTDP (20-kb True Dual-Port Memory) (see page 319) , ACX_BRAMFIFO (20-kb FIFO Memory with Optional Error Correction) (see page 350) and ACX_LRAM (4096-bit (128x32) Simple-Dual-Port Memory) (see page 424).
1.4	04 Dec 2016	• Memories (see page 289): Cleaned up BRAMTDP, BRAMSDP, and BRAMFIFO timing diagrams, improved explanations of parameters and functionality. • Memories: (see page 289) Added read_peval parameter to the BRAMSDP macro. • Memories: (see page 289) Added ROM description to LRAM and BRAMSDP macros.
1.5	01 Feb 2017	• Logic Functions to be deleted: Added documentation for ACX_SYNCHRONIZER, ACX_SYNCHRONIZER_N, and ACX_SHIFTREG. • Memories (see page 289): Cleaned up LRAMFIFO parameters, timing diagrams, and improved explanations of parameters and functionality.
1.6	31 Mar 2017	• Speedcore Component Library User Guide (see page 9): Re-named the user guide and re-arranged sections to provide for a better organization. • Memories (see page 289): Updated figure, 20-kb True Dual-Port Memory.
1.7	16 Jul 2017	• Clock Functions TO BE DELETED: Added descriptions, waveforms and instantiation templates for the CLKSWITCH, CLKDIV, and CLKGATE macros.
1.8	19 Jul 2017	• Clock Functions TO BE DELETED: • Modified CLKDIV waveform to highlight that it's always 50% duty cycle. • Corrected some of the waveforms, figure titles and descriptions in the CLKSWITCH section.

Version	Date	Description
1.9	14 Nov 2017	• ACX_DSP_GEN (see page 102): Updated defaults value of addsub_bypass parameter in Table: DSP64 Parameters (see page 110). • JTAG TAP Controller Functions (see page 457): New chapter added. • ACX_BRAMSDP (20-kb Simple Dual-Port Memory with Error Correction) (see page 289): Changed tie-off requirements for we[3:0] port.
1.10	15 Nov 2017	• JTAG TAP Controller Functions (see page 457): Corrected timing shown in Figure: ACX_JTAP_UNIT Signal Timing (see page 468).
1.11	02 Jan 2018	• Clock Functions TO BE DELETED: Added CLKGATE timing diagram. • ACX_BRAMSDP (20-kb Simple Dual-Port Memory with Error Correction) (see page 289): Updated dbit_error to reflect that it's updated at the same time as the output data.
1.12	04 Apr 2018	• Clock Functions TO BE DELETED: Corrected CLKSWITCH SYNCHRONIZE_SEL description, added warning about simulating CLKSWITCH and CLKGATE.
1.13	17 May 2018	• Memories: (see page 289) Noted that BRAM GUI & Wrapper don't support multi-bit we.
1.14	19 Aug 2018	• LRAMFIFO: (see page 431) ◦ Added hold_output, rst_sync_mode and, prevent_overunderflow parameter descriptions. ◦ Updated FIFO reset description. • ACX_BRAMSDP (20-kb Simple Dual-Port Memory with Error Correction) (see page 289): ◦ Added special note for BRAMSDP ECC Mode's limitation on read_initval when the output register is disabled. • BRAMFIFO (see page 350) ◦ Updated Outregce Input signal description.
1.15	05 Sep 2018	• Logic Functions to be deleted: Highlighted that the use of ACX_SYNCHRONIZER is strongly recommended and added the advantages of using the macro versus constructing a synchronizer from two flops. • LRAMFIFO (see page 431): ◦ Added fwft_mode restriction. ◦ Added afull_offset and aempty_offset parameter value restriction.
1.16	05 Apr 2019	• ACX_DSP_GEN (see page 102): Updated description for `sel_addsub_a`, `sel_addsub_b`, `sel_48_dout`, `sat_mode` and `use_match_in` parameters to indicate restriction. • LRAMFIFO (see page 431): Added explanation of how fwft mode affects status signals.

Version	Date	Description
2.0	08 Aug 2023	• ACX_DSP_GEN (see page 102) Added inference templates, (moved from Synthesis UG). Added sel_48_dout to instantiation template • ACX_BRAMTDP (see page 319), ACX_BRAMSDP (see page 289) and ACX_LRAM (see page 424) Added inference templates (moved from Synthesis UG) • ACX_LRAM2K_SDP, ACX_LRAM2K_FIFO (see page 449), ACX_BRAM72K_SDP (see page 387) and ACX_BRAM72K_FIFO (see page 414) Added sections • Speedcore MLP72 (see page 159) Added sections • Clock Functions (see page 90) Updated constraint information • Speedcore Fabric Architecture (see page 11) Updated details for Gen4 and Gen5 Speedcore devices • Remove "IP" from document name • Add ACE Soft IP GUI flow details for generating DSP macros