Speedster7t Component Library User Guide (UG086)

Speedster FPGAs

Preliminary Data

Copyrights, Trademarks and Disclaimers

Copyright © 2021 Achronix Semiconductor Corporation. All rights reserved. Achronix, Speedcore, Speedster, and ACE are trademarks of Achronix Semiconductor Corporation in the U.S. and/or other countries All other trademarks are the property of their respective owners. All specifications subject to change without notice.

NOTICE of DISCLAIMER: The information given in this document is believed to be accurate and reliable. However, Achronix Semiconductor Corporation does not give any representations or warranties as to the completeness or accuracy of such information and shall have no liability for the use of the information contained herein. Achronix Semiconductor Corporation reserves the right to make changes to this document and the information contained herein at any time and without notice. All Achronix trademarks, registered trademarks, disclaimers and patents are listed at http://www.achronix.com/legal.

Preliminary Data

This document contains preliminary information and is subject to change without notice. Information provided herein is based on internal engineering specifications and/or initial characterization data.

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 9
ACX_ Prefix 9
Chapter - 2: Fabric Architecture 10
Introduction 10
RLB6 11
MLUT Mode 13
Routing Between RLB6s 14
RLB6 Detail 16
Lookup Table, (LUT), Functions 18
Six-Input Lookup Table (ACX_LUT6) 18
Dual Five-Input Lookup Table (ACX_LUT5x2) 21
Registers 24
Naming Convention 24
Register Primitives 25
Register Macros 55
Chapter - 3: Logic Functions 67
ACX_SYNCHRONIZER, ACX_SYNCHRONIZER_N 67
Using ACX_SYNCHRONIZER to Synchronize Reset 69
Instantiation Templates 70
ACX_SHIFTREG 71
Instantiation Templates 73
Chapter - 4: Clock Functions 75
ACX_CLKDIV (Clock Divider) 75
Constraints 77
Instantiation Templates 77
ACX_CLKGATE (Clock Gate) 78
Constraints 80
Instantiation Templates 80
ACX_CLKSWITCH (Clock Switch) 81
Constraints 84
Instantiation Templates 84

Chapter - 5: Arithmetic and DSP 85
Number Formats 85
Integer Formats 85
Integer Groups 85
Floating-Point Formats 88
Block Floating Point 90
ALU8 91
Description 91
Parameters 91
Ports 92
Functions 92
Instantiation Templates 94
ACX_MLP72 95
Numerical Formats 98
Parallel Multiplications 99
Memories 100
Instantiation 100
Common Stages 100
Integer Modes 110
Integrated LRAM 123
Block Floating-Point Modes 130
Floating-Point Modes 137
Verilog 147
ACX_MLP72_INT 150
Parameters 152
Ports 153
Input Data Mapping 154
Output Formatting and Error Conditions 156
Asynchronous Reset Rules 156
Inference 156
Instantiation Template 157
ACX_MLP72_INT8_MULT_4X 158
Parameters 159
Ports 161
Timing Diagrams 162
Inference 163
Instantiation Template 164

ACX_MLP72_INT16_MULT_2X 166
Parameters 167
Ports 169
Timing Diagrams 170
Inference 171
Instantiation Template 172
Integer Library 173
MLP Registers 173
Accumulation 173
ACX_INT_MULT 175
ACX_INT_MULT_N 181
ACX_INT_MULT_ADD 187
ACX_INT_RLB_MULT 193
ACX_INT_SQUARE_ADD 198
ACX_INT_COMPLEX_MULT 204
Floating-Point Library 209
Introduction 209
MLP Registers 209
Accumulation 209
Floating-Point Format 210
Output Status 211
ACX_FP_ADD 212
ACX_FP_MULT 217
ACX_FP_MULT_PLUS 222
ACX_FP_MULT_2X 227
ACX_FP_MULT_ADD 234
Chapter - 6: Memories 239
ACX_BRAM72K_FIFO 239
Parameters 240
Ports 242
Read and Write Operations 243
Inference 246
Instantiation Template 246
ACX_BRAM72K_SDP 248
Parameters 250
Ports 252
Memory Organization and Data Input/Output Pin Assignments 254

Read and Write Operations ... 257
Timing Diagrams ... 259
Memory Initialization ... 261
ECC Modes of Operation ... 262
Using ACX_BRAM72K_SDP as a Read-Only Memory (ROM) ... 263
Advanced Modes ... 263
Inference ... 265
Instantiation Template ... 269

ACX_LRAM2K_FIFO ... 272
Parameters ... 273
Ports ... 274
Read and Write Operations ... 275
Inference ... 277
Instantiation Templates ... 278

ACX_LRAM2K_SDP ... 280
Parameters ... 282
Ports ... 283
Memory Organization and Data Input/Output Pin Assignments ... 283
Read and Write Operations ... 284
Memory Initialization ... 285
Using ACX_LRAM2K_SDP as a Read-Only Memory (ROM) ... 285
Inference ... 286
Instantiation Templates ... 290

Chapter - 7: Network-on-Chip (NOC) Primitives ... 292

ACX_NAP_AXI_MASTER ... 292
Parameters ... 293
Ports ... 294
AXI-4 Specification ... 296
Inference ... 296
Instantiation Templates ... 297
Parameter Templates ... 299

ACX_NAP_AXI_SLAVE ... 300
Parameters ... 301
Ports ... 302
AXI-4 Specification ... 304
Inference ... 304
Instantiation Templates ... 305

Parameter Templates 308
ACX_NAP_HORIZONTAL 309
Parameters 310
Ports 311
Inference 311
Instantiation Templates 312
Parameter Templates 313
ACX_NAP_VERTICAL 314
Parameters 315
Ports 316
Inference 316
Instantiation Templates 317
Parameter Templates 318
ACX_NAP_ETHERNET 319
Parameters 320
Ports 323
Inference 324
Instantiation Templates 325
Parameter Templates 327
Revision History 328

Chapter - 1: Introduction

The Achronix Speedster7t component cell library provides the user with building blocks that may be instantiated into the user's design. These macros provide access to low-level fabric primitives, complex I/O block, and higher level design components. Each library element entry describes the operation of the macro as well as any parameters that must be initialized. Verilog and VHDL templates are also provided to aide in the implementation of the user's design.

This guide contains the following sections:

- Speedster7t Fabric Architecture (see page 10)
- Speedster7t Logic Functions (see page 67)
- Speedster7t GPIO Functions
- Speedster7t Clock Functions (see page 75)
- Speedster7t Arithmetic and DSP (see page 85)
- Speedster7t Memories (see page 239)
- Speedster7t Network on Chip Primitives (see page 292)
- Speedster7t Component Library User Guide Revision History (see page 328)

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 Speedster®7t fabric consists of 6-input LUTs, each with two flops, organized into logic groups. These logic groups are then organized into reconfigurable logic blocks (RLB6s), which are then arranged into 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. The fabric logic capabilities and functions are defined by the structure of the RLB6.

This floorplan of functional blocks and global interconnects is shown below.

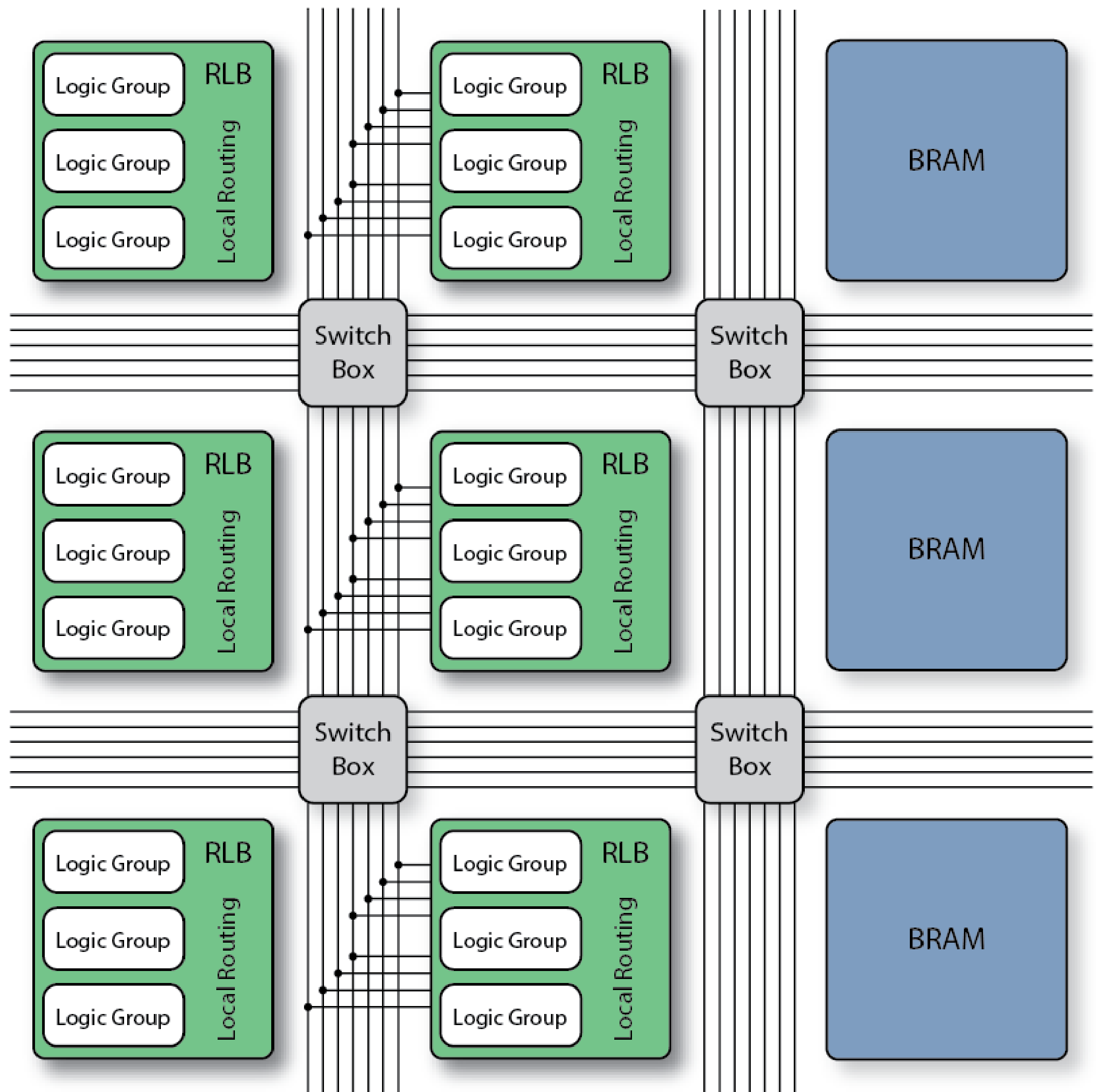


Figure 1: ***Speedster7t Fabric Floorplan***

RLB6

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

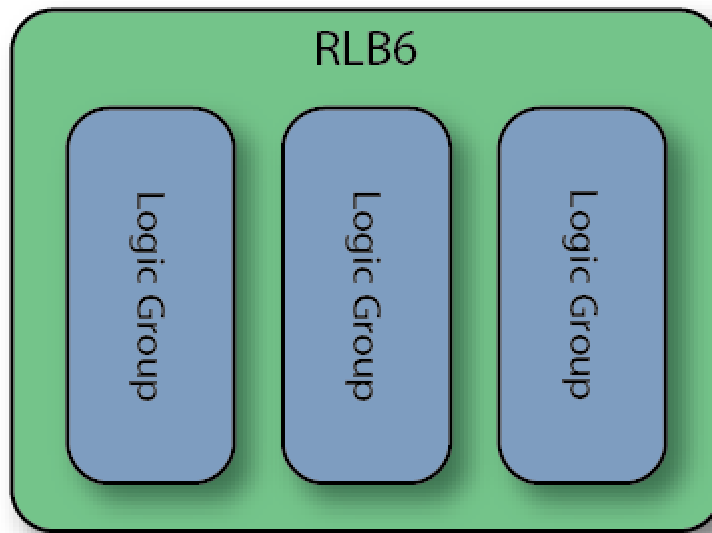


Figure 2: ***RLB6 Block Diagram***

Each logic group 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 table below provides information on the resource counts inside an RLB6.

Table 1: ***RLB6 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
- MAX function that efficiently compares two 8-bit numbers and chooses the maximum or minimum result
- 8-to-1 MUX with single-level delay
- 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
- 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 figure below provides details on the circuitry inside a single logic group.

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

MLUT Mode

The RLB includes an MLUT mode for an efficient LUT-based multiplication. MLUT mode results in 2 × 2 multiplier building blocks that can be stacked horizontally and vertically to generate any size signed multiplier. For example, a 2 × 4 multiplier building block can be generated with two LUT6s, and one RLB can perform a 6 × 8 multiply.

Note

MLUT mode is supported by the MLUT generator within ACE to help customers build the multiplier desired.

Routing Between RLB6s

There are special considerations when routing ALU carry chains and shift registers. Achronix's Gen4 fabric has hard-wired connections on the signals `carry_in/carry_out` of each ALU. As mentioned above, 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 figure below shows the `carry_in/carry_out` routing of an ALU.

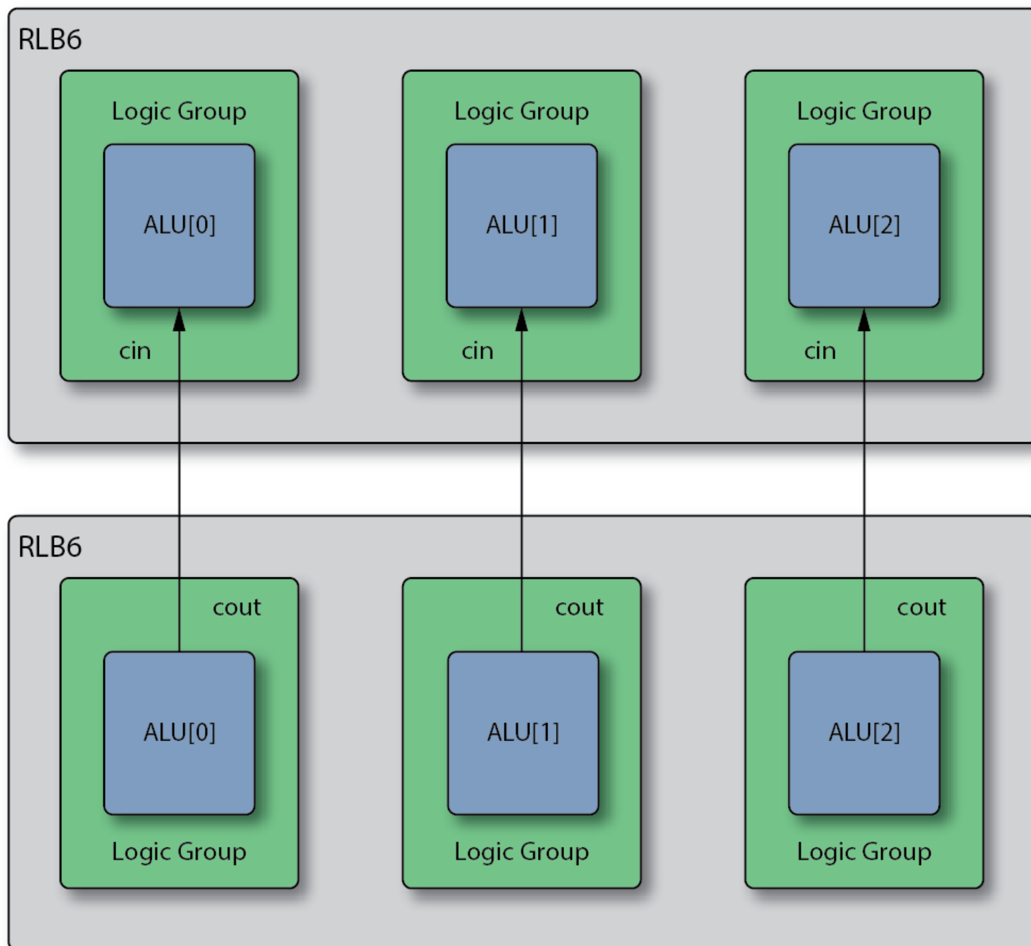


Figure 4: *ALU Carry Chain Routing*

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 figure below shows details of the routing in the Gen4 fabric.

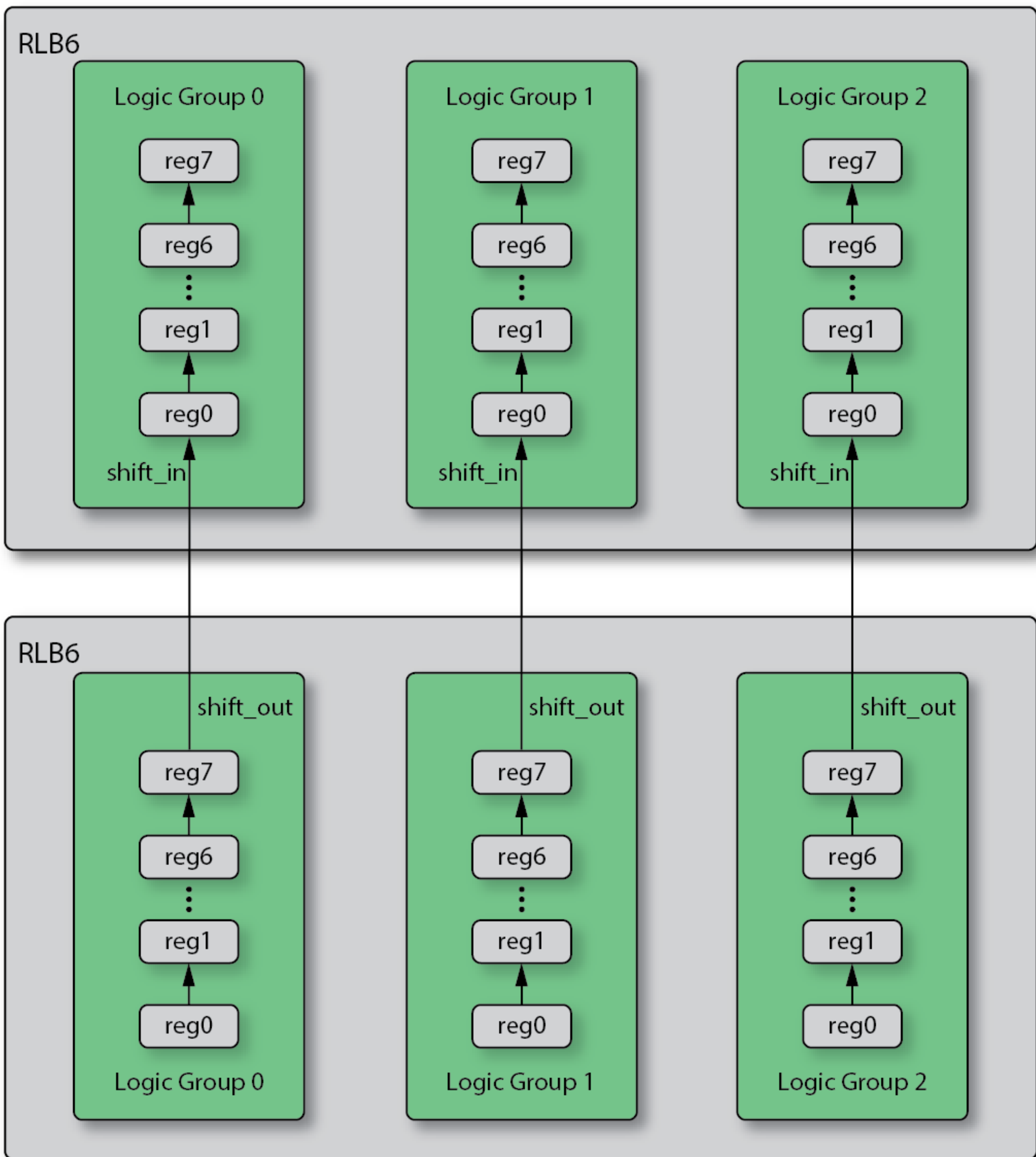


Figure 5: *Shift Register Routing*

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 routing detail of one fourth of a logic group (one LUT6 and two registers) is shown below.

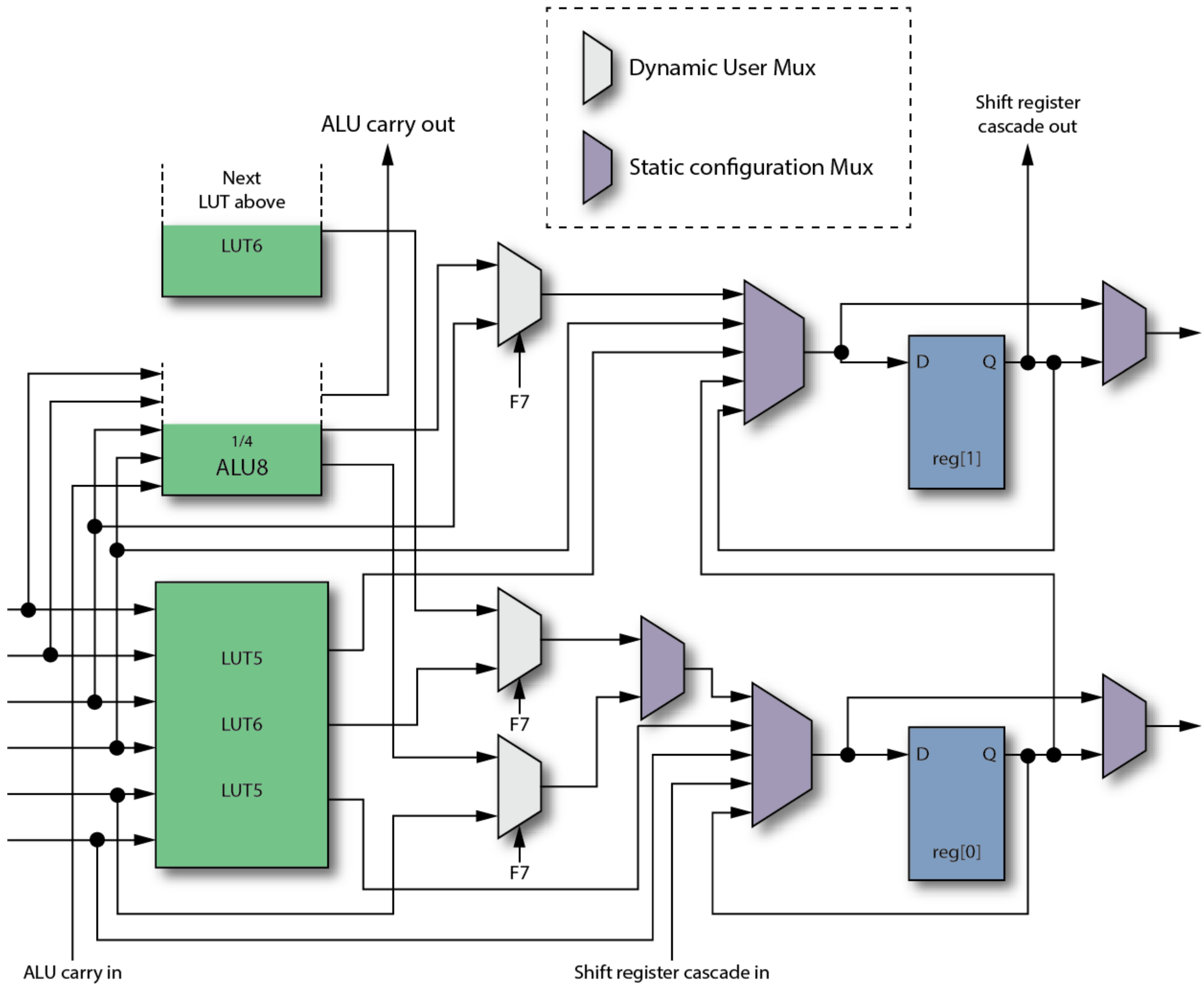


Figure 6: ***One-Fourth of a Logic Group (Connection Detail)***

The diagram above 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 then 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.

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`.

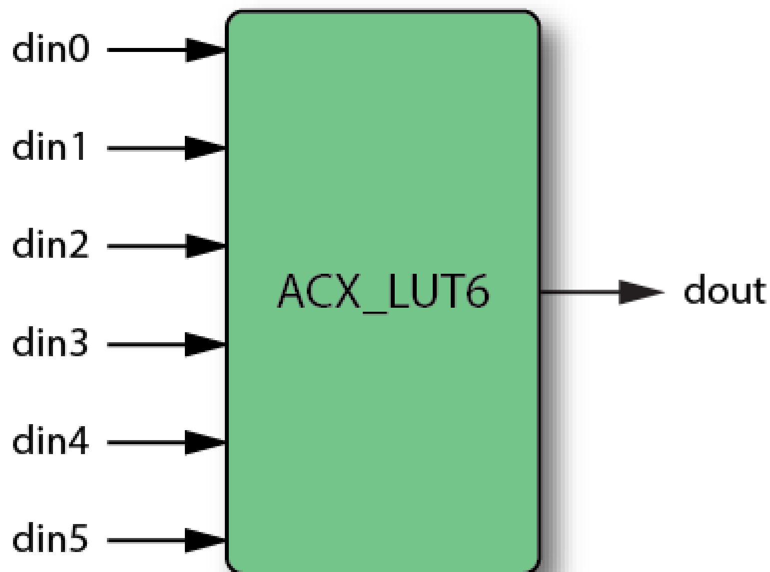


Figure 7: *Logic Symbol*

Parameters

Table 2: *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 function table (see page 19).

Ports

Table 3: ***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 4: ***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.

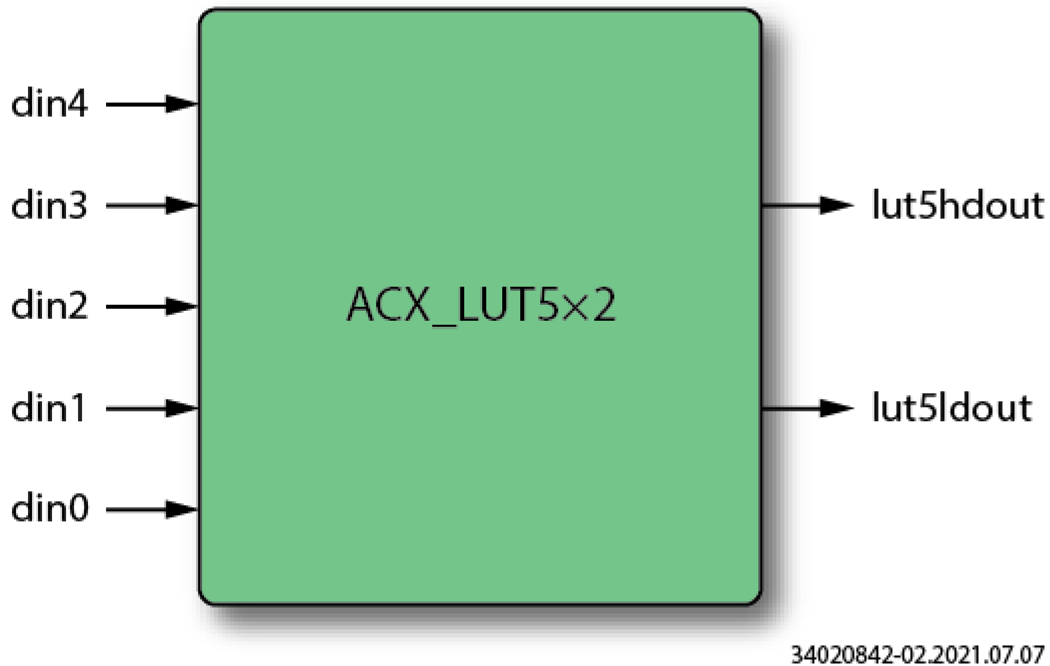


Figure 8: ***Dual LUT5 Lookup Tables***

Parameters

Table 5: ***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 22).

Ports

Table 6: ***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}.

Name	Type	Description
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 7: *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 8: *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'b1	din4	din3	din2	din1	din0	dout
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
);
```

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 indicates its unique properties.

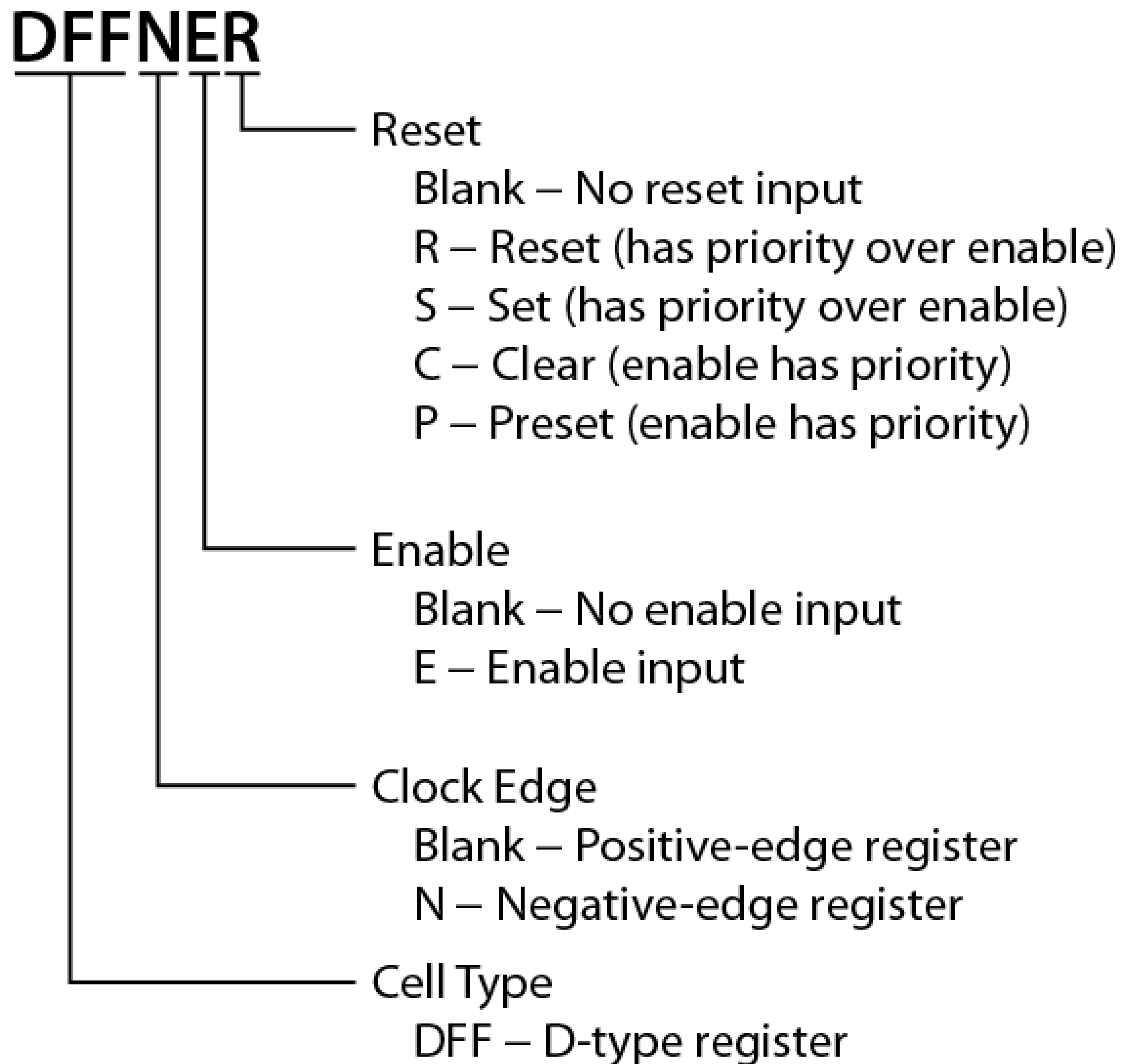


Figure 9: ***Register Naming Convention***

Register Primitives

ACX_DFF (Positive Clock Edge D-Type Register)

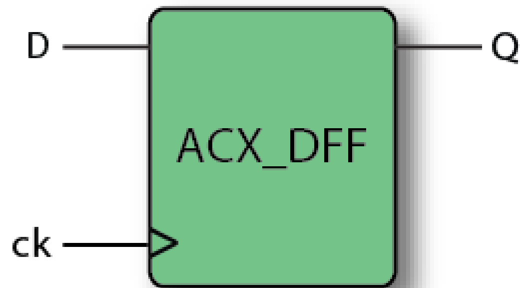


Figure 10: *Positive Clock Edge D-Type Register*

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 9: *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 10: *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 11: *Function Table*

Inputs		Output
d	**ck**	**q**
0	↑	0

Inputs		Output
1	↑	1

Instantiation Templates

Verilog

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

VHDL

```
-- For Speedcore7t
------------- ACHRONIX LIBRARY ------------
library speedcore7t;
use speedcore7t.components.all;
----------- DONE ACHRONIX LIBRARY ---------

-- For Speedster7t
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.components.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)

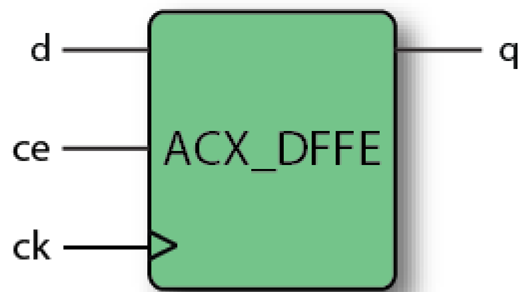


Figure 11: ***Positive Clock Edge D-Type Register with Clock Enable***

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 12: ***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 13: ***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 14: ***Function Table***

Inputs			Output
ce	**d**	**ck**	**q**
0	X	X	Hold

Inputs			Output
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

```
-- For Speedcore7t
------------- ACHRONIX LIBRARY ------------
library speedcore7t;
use speedcore7t.components.all;
----------- DONE ACHRONIX LIBRARY ---------


-- For Speedster7t
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.components.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)

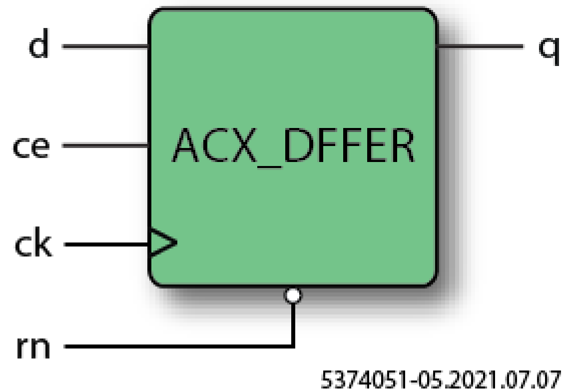


Figure 12: ***Positive Clock Edge D-Type Register with Clock Enable and Asynchronous/Synchronous Reset***

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 30) and See ACX_DFFER Function Table with sr_assertion = "clocked" (see page 30). 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 15: ***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 16: ***Pin Descriptions***

Name	Type	Description
d	Input	Data input.

Name	Type	Description
`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 17: ***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 18: ***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

```
-- For Speedcore7t
------------- ACHRONIX LIBRARY ------------
library speedcore7t;
use speedcore7t.components.all;
----------- DONE ACHRONIX LIBRARY ---------


-- For Speedster7t
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.components.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)

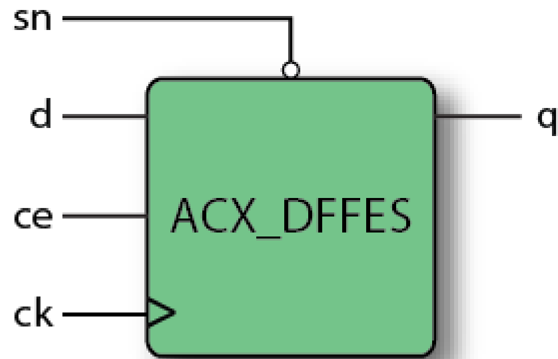


Figure 13: ***Positive Clock Edge D-Type Register with Clock Enable and Asynchronous/Synchronous Set***

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 30) and Table: ACX_DFFES Function Table with sr_assertion = "clocked" (see page 30). 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 19: ***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 20: ***Pin Descriptions***

Name	Type	Description
d	Input	Data input.

Name	Type	Description
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 21: ***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 22: ***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

```
-- For Speedcore7t
------------- ACHRONIX LIBRARY ------------
library speedcore7t;
use speedcore7t.components.all;
----------- DONE ACHRONIX LIBRARY ---------


-- For Speedster7t
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.components.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)

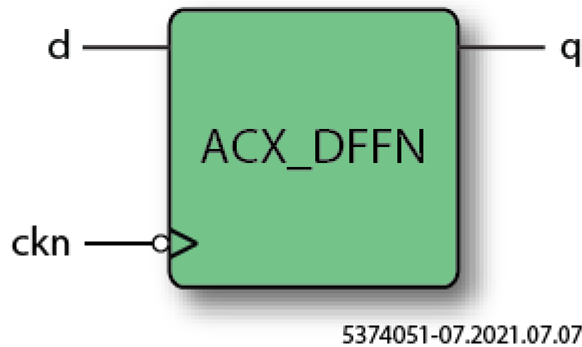


Figure 14: ***Negative Clock Edge D-Type Register***

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 23: ***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 24: ***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 25: ***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

```
-- For Speedcore7t
------------- ACHRONIX LIBRARY ------------
library speedcore7t;
use speedcore7t.components.all;
----------- DONE ACHRONIX LIBRARY ---------


-- For Speedster7t
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.components.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)

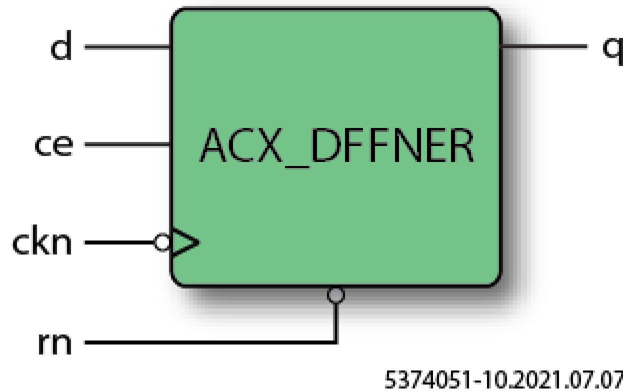


Figure 15: ***Negative Clock Edge D-Type Register with Clock Enable and Asynchronous/Synchronous Reset***

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 33) and Table: ACX_DFFNER Function Table with sr_assertion = "clocked" (see page 33). 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 26: ***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 27: ***Pin Descriptions***

Name	Type	Description
`d`	Input	Data input.

Name	Type	Description
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 28: ***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 29: ***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

```
-- For Speedcore7t
------------- ACHRONIX LIBRARY ------------
library speedcore7t;
use speedcore7t.components.all;
----------- DONE ACHRONIX LIBRARY ---------


-- For Speedster7t
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.components.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)

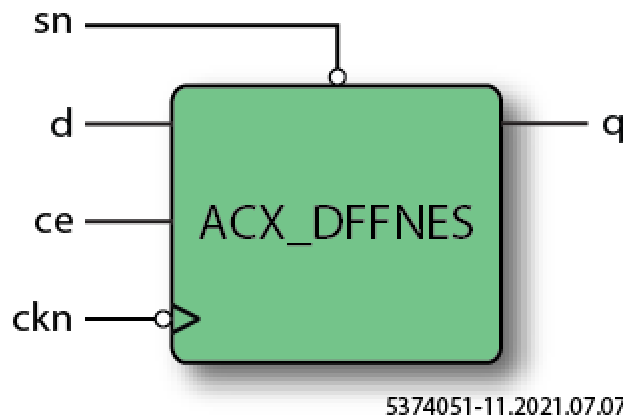


Figure 16: ***Negative Clock Edge D-Type Register with Clock Enable and Asynchronous/Synchronous Set***

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 37) and Table: ACX_DFFNES Function Table with sr_assertion = "clocked" (see page 38). 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 30: ***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 31: ***Pin Descriptions***

Name	Type	Description
`d`	Input	Data input.

Name	Type	Description
`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 32: ***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 33: ***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

```
-- For Speedcore7t
------------- ACHRONIX LIBRARY ------------
library speedcore7t;
use speedcore7t.components.all;
----------- DONE ACHRONIX LIBRARY ---------


-- For Speedster7t
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.components.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)

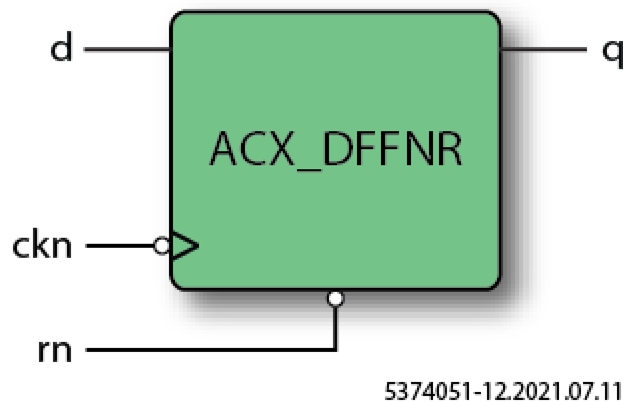


Figure 17: ***Negative Clock Edge D-Type Register with Asynchronous Reset***

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 34: ***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 35: ***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.

Name	Type	Description
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 36: ***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 37: ***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

```
-- For Speedcore7t
------------- ACHRONIX LIBRARY ------------
library speedcore7t;
use speedcore7t.components.all;
----------- DONE ACHRONIX LIBRARY ---------


-- For Speedster7t
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.components.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)

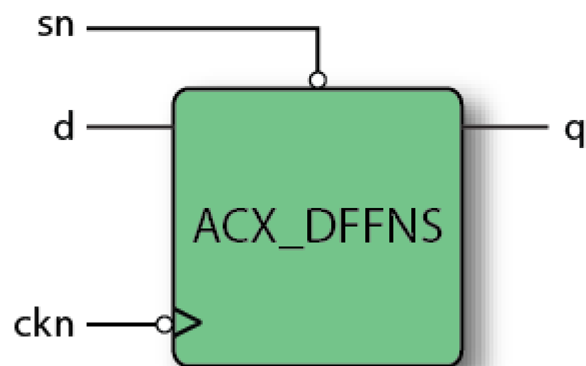


Figure 18: ***Negative Clock Edge D-Type Register with Asynchronous Set***

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 38: ***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 39: ***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.

Name	Type	Description
`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 40: ***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 41: ***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

```
-- For Speedcore7t
------------- ACHRONIX LIBRARY ------------
library speedcore7t;
use speedcore7t.components.all;
----------- DONE ACHRONIX LIBRARY ---------


-- For Speedster7t
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.components.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)

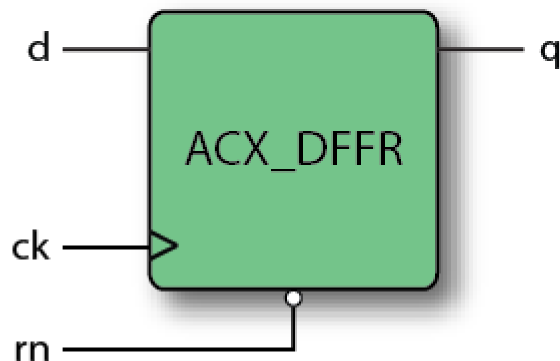


Figure 19: ***Positive Clock Edge D-Type Register with Asynchronous Reset***

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 42: ***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 43: ***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 44: ***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 45: ***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

```
-- For Speedcore7t
------------- ACHRONIX LIBRARY ------------
library speedcore7t;
use speedcore7t.components.all;
----------- DONE ACHRONIX LIBRARY ---------


-- For Speedster7t
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.components.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)

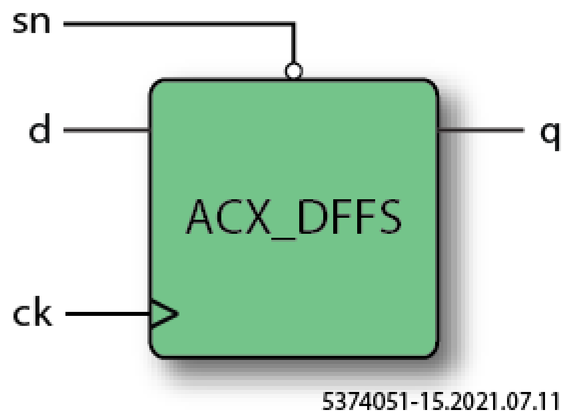


Figure 20: ***Positive Clock Edge D-Type Register with Asynchronous Set***

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 46: ***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 47: ***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 48: ***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 49: ***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

```
-- For Speedcore7t
------------- ACHRONIX LIBRARY ------------
library speedcore7t;
use speedcore7t.components.all;
----------- DONE ACHRONIX LIBRARY ---------


-- For Speedster7t
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.components.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)

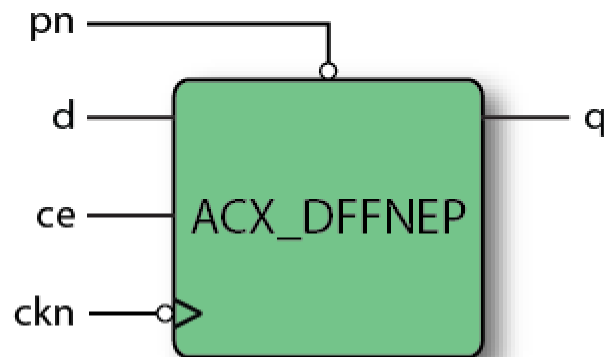


Figure 21: ***Negative Clock Edge D-Type Register with Clock Enable and Synchronous Preset***

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 50: ***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 51: ***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.

Name	Type	Description
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 52: *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

```
-- For Speedcore7t
------------- ACHRONIX LIBRARY ------------
library speedcore7t;
use speedcore7t.components.all;
----------- DONE ACHRONIX LIBRARY ---------


-- For Speedster7t
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.components.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)

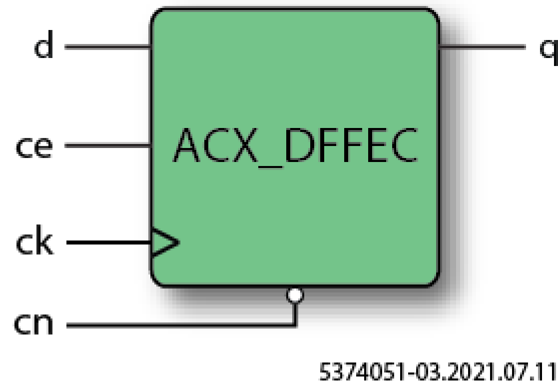


Figure 22: ***Positive Clock Edge D-Type Register with Clock Enable and Synchronous Clear***

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 53: ***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 54: ***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 55: *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

```
-- For Speedcore7t
------------- ACHRONIX LIBRARY ------------
library speedcore7t;
use speedcore7t.components.all;
----------- DONE ACHRONIX LIBRARY ---------


-- For Speedster7t
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.components.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)

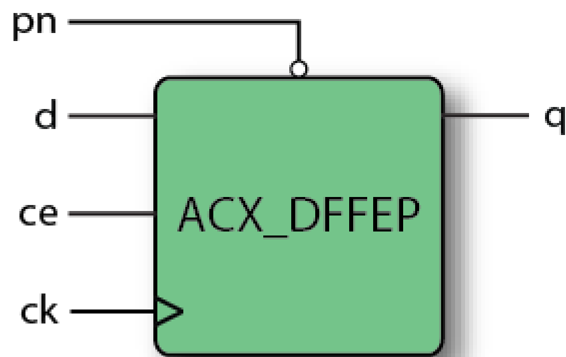


Figure 23: ***Positive Clock Edge D-Type Register with Clock Enable and Synchronous Preset***

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 56: ***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 57: ***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 58: *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

```
-- For Speedcore7t
------------- ACHRONIX LIBRARY ------------
library speedcore7t;
use speedcore7t.components.all;
----------- DONE ACHRONIX LIBRARY ---------


-- For Speedster7t
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.components.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)

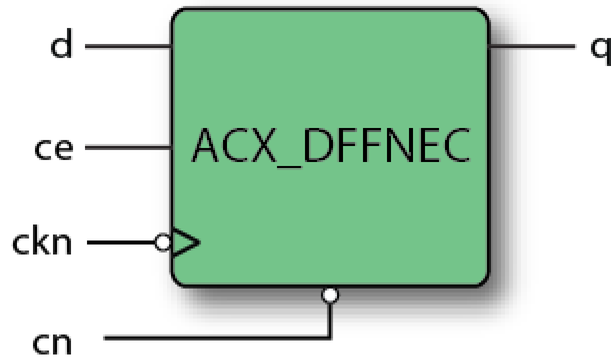


Figure 24: ***Negative Clock Edge D-Type Register with Clock Enable and Synchronous Clear***

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 59: ***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 60: ***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 61: ***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

```
-- For Speedcore7t
------------- ACHRONIX LIBRARY ------------
library speedcore7t;
use speedcore7t.components.all;
----------- DONE ACHRONIX LIBRARY ---------


-- For Speedster7t
------------- ACHRONIX LIBRARY ------------
library speedster7t;
use speedster7t.components.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
);
```

Chapter - 3: Logic Functions

ACX_SYNCHRONIZER, ACX_SYNCHRONIZER_N

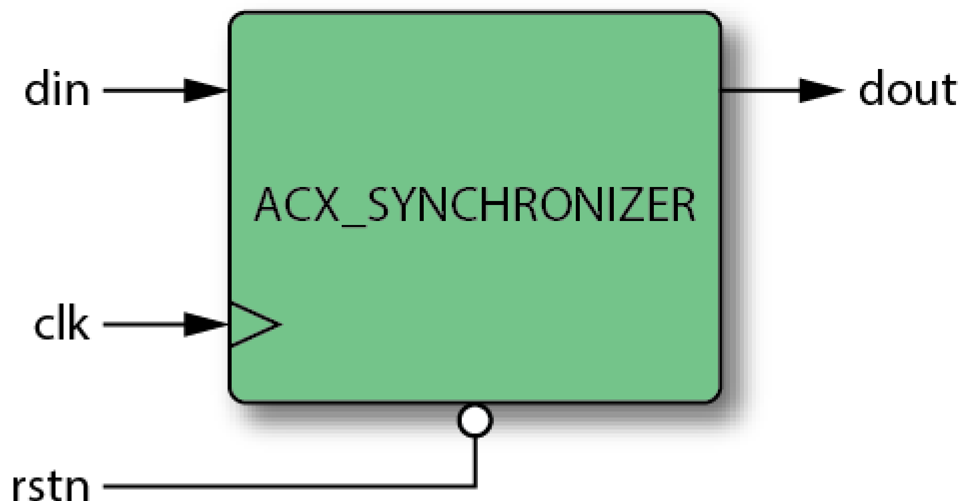


Figure 25: *ACX_SYNCHRONIZER Logic Symbol*

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 place the flip-flops far apart in 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 and result 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 62: *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 63: *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 64: *Function Table*

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

Using ACX_SYNCHRONIZER to Synchronize Reset

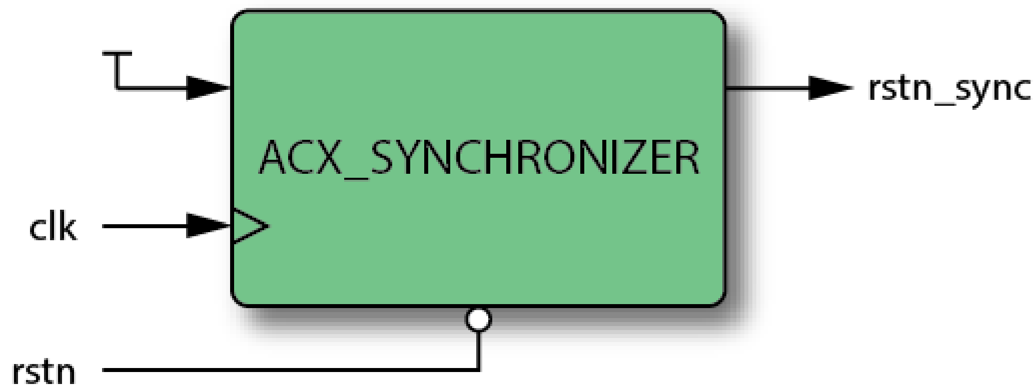


Figure 26: *ACX_SYNCHRONIZER Synchronizing Reset*

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 be 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)
);
```

VHDL

```
library speedster7t;
use speedster7t.components.all;


instance_name : ACX_SYNCHRONIZER
generic map (
    init     => 0
)
port map (
    clk      => user_output_clock,
    rstn     => user_reset_n,
    din      => user_din,
    dout     => user_out
);
```

ACX_SHIFTREG

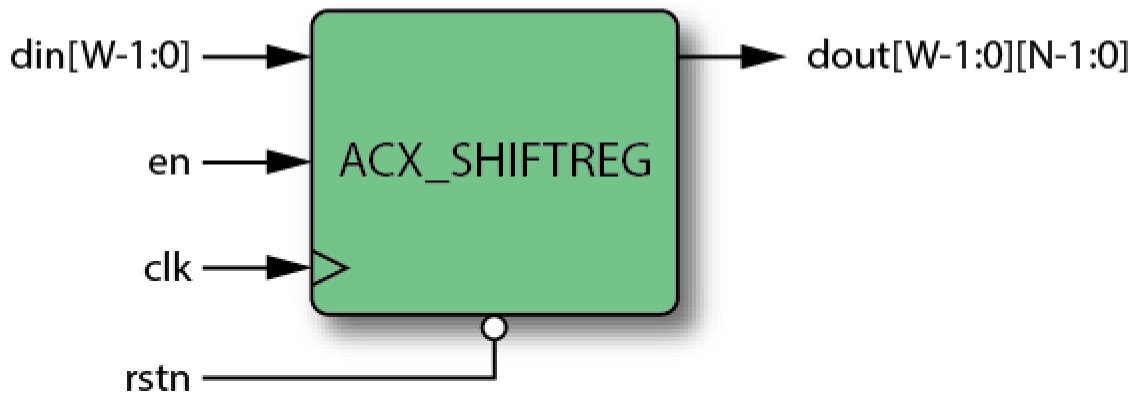


Figure 27: *ACX_SHIFTREG Logic Symbol*

ACX_SHIFTREG provides an efficient multi-tap shift register implementation using LRAMs, with a parameterizeable data width and a parameterizeable 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 parameterized 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.

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 65: *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. Each latency is measured from `din`, each value in the array must be larger than the previous value.

Parameter	Defined Values	Default Value	Description
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 the rising clock edge only.

Table 66: *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.
dout[(W-1):0]	Input	Data input.
dout[0][(W-1):0]	Output	An array of data outputs, where dout[0][(W-1):0] carries the data out from the first tap, and dout[N-1][(W-1):0] represents the data out from the last tap.

Table 67: *Function Table*

Inputs			Output
rstn	**en**	**clk**	**dout[n]**
0	X	X	0 (and resets all internal state)
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)
);
```

VHDL

```
-- Type definition

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


package types_pkg is
  type int_array_t is array (natural range <>) of integer;
  type slv_2d_array_t is array (natural range <>, natural range <>) of std_logic;
end package;

----------------------------------------
use work.types_pkg.all;
component ACX_SHIFTREG

    generic( W,    N       : integer;
             TAPS, MODE    : int_array_t );
    port(    rstn, clk, en : in bit;
             din           : in std_ulogic_vector (W-1 downto 0);
             dout          : out slv_2d_array_t (W-1 downto 0, N-1 downto 0) );

end component;
----------------------------------------

instance_name: ACX_SHIFTREG
generic map (
    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
)
port map (
    rstn => user_rstn,
    clk  => user_clk,
    en   => user_en,
```

```
    din  => user_din,
    dout => user_dout_array
);
```

Chapter - 4: Clock Functions

ACX_CLKDIV (Clock Divider)

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

Figure 28: ***ACX_CLKDIV Logic Symbol***

Table 68: ***Parameter Descriptions***

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

Table 69: ***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 device. They cannot connect directly with data tracks.

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

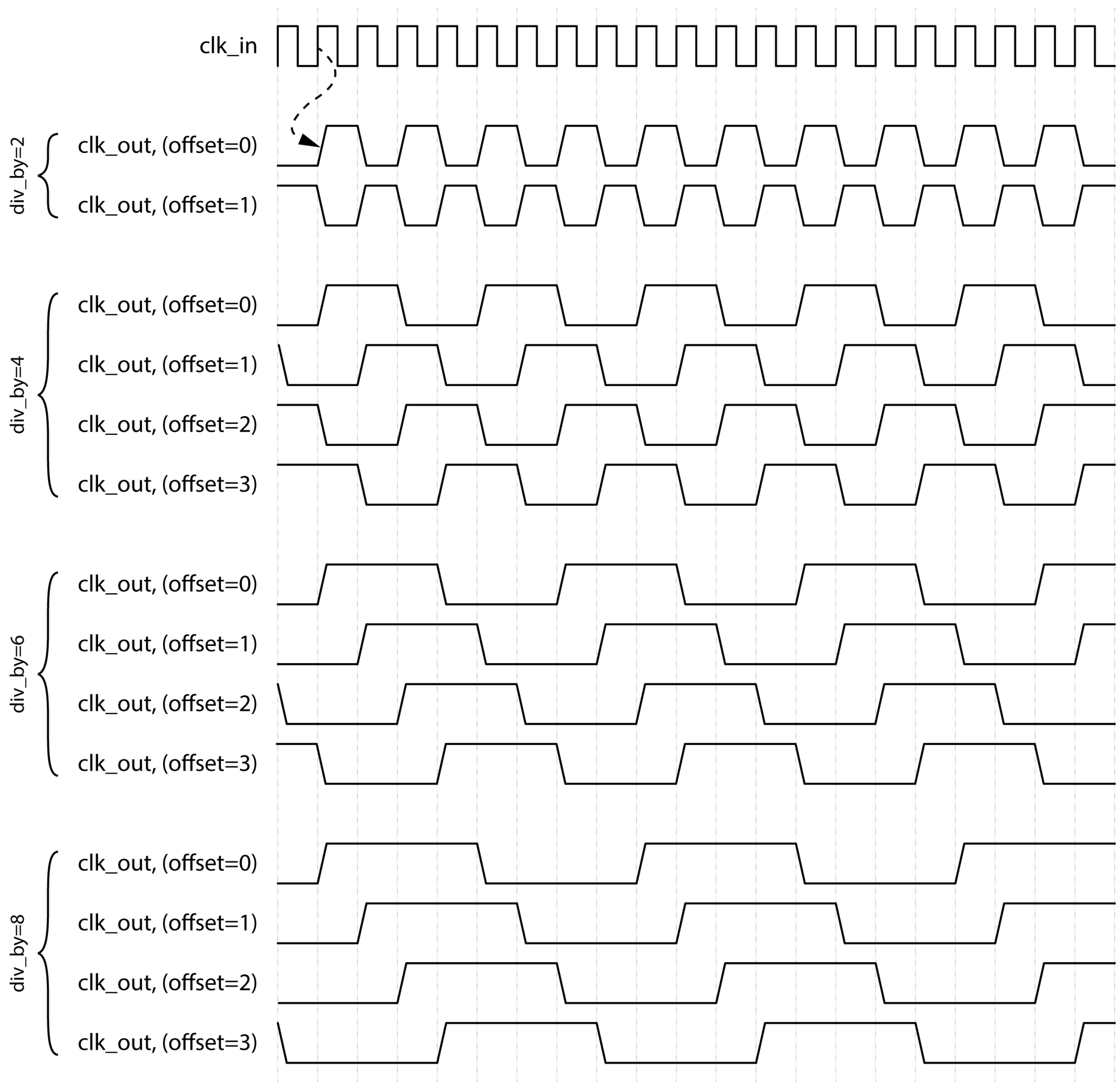


Figure 29: ***Output Clock Timing Diagram***

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 1. Internal net is "master_clk".
Output of divider connects to port named "o_clk_out_div_2"
create_generated_clock -name clk_div_2 -source [get_nets master_clk] -divide_by 2 -phase 180
[get_ports o_clk_out_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.components.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 30: *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. The ACX_CLKGATE 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 the simulator's scheduling of events and may vary with different simulators, different designs, and different simulation models.

Table 70: *Pin Descriptions*

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

Table Note

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

The following timing diagrams illustrate the behavior of the ACX_CLKGATE component.

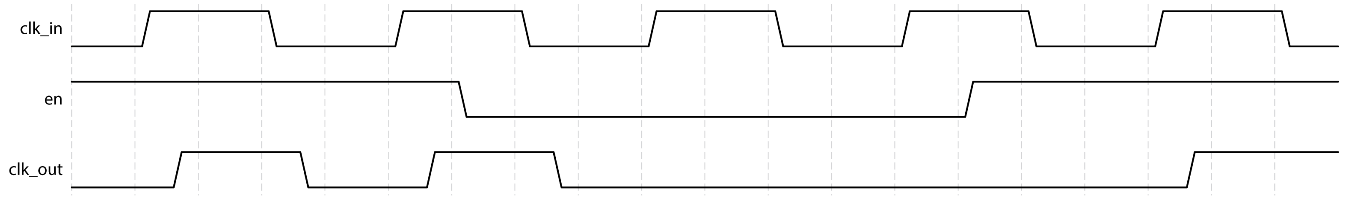


Figure 31: ***ACX_CLKGATE Timing Diagram***

Constraints

The ACX_CLKGATE component does propagate the input clock frequency from clk_in to clk_out. Therefore, it is not necessary to specify any additional constraints for clk_out. Synplify Pro and ACE correctly pass through the input clock domain to the output clock domain for static timing analysis purposes.

Note

In some circumstances, if re-using existing code and constraints, there can be an existing create_generated_clock constraint (using -divide_by 1) for the ACX_CLKGATE output. This in itself is not problematic, both Synplify and ACE still perform the correct static timing analysis. However, be aware that if the ACX_CLKGATE is cloned within ACE (to assist placement or routability) then the new cloned ACX_CLKGATEs retain the original input clock domain rather than the new clock domain specified by any create_generated_clock constraint that is applied to the original ACX_CLKGATE. Keep this behavior in mind when reviewing the ACE timing reports with regard to any cloned ACX_CLKGATE output.

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.components.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 32: *ACX_CLKSWITCH Logic Symbol*

The ACX_CLKSWITCH component implements clock switching functionality that allows the output clock to be glitchlessly switched between two different clock inputs. The CLKSWITCH component implements the glitchless behavior 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 of the `SYNCHRONIZE_SEL` parameter. For example, setting `SYNCHRONIZE_SEL` to a value of:

- 0 ensures the input `sel[]` for each clock is synchronized to the rising and then falling edge of the clock that it is selecting
- 1 synchronizes the input `sel[]` signal to the falling edge followed by the next falling edge of the clock that it is selecting
- 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 `sel[]` signal 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 the simulator's scheduling of events and may vary with different simulators, different designs, and different simulation models.
Using `desel[]` when the input clock is toggling might cause a glitch or runt pulse on the output.

Table 71: ***Parameter Descriptions***

Parameter	Defined Values	Default Value	Description
PRESEL	0, 1, 2	0	The PRESEL parameter is used to determine 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 of this parameter should match the startup value of the input sel[1:0].
SYNCHRONIZE_SEL	0, 1, 2	0	The SYNCHRONIZE_SEL parameter 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 the falling edge and then a second falling edge of the selected clock (two falling edges). 2 – synchronizes the input sel[1:0] to a single falling edge of the selected clock.

Table 72: ***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 device. They cannot connect directly with data tracks.

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

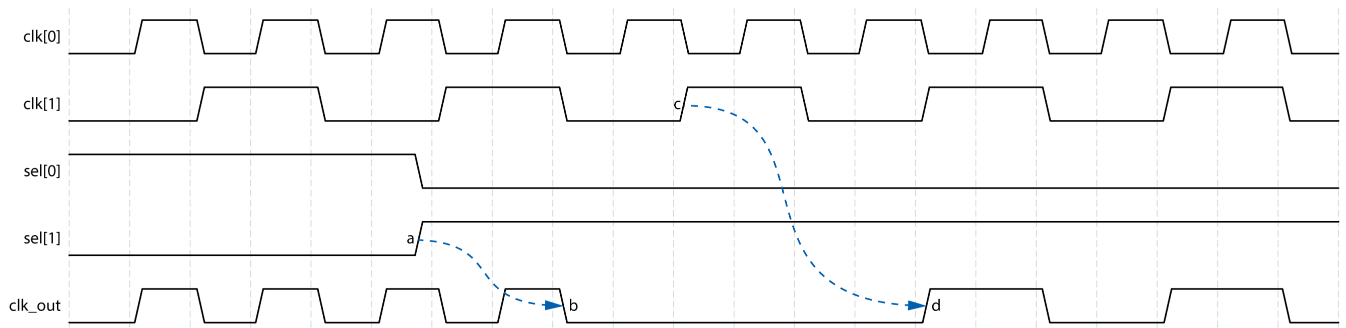


Figure 33: ***SYNCHRONIZE_SEL = 0 Timing Diagram***

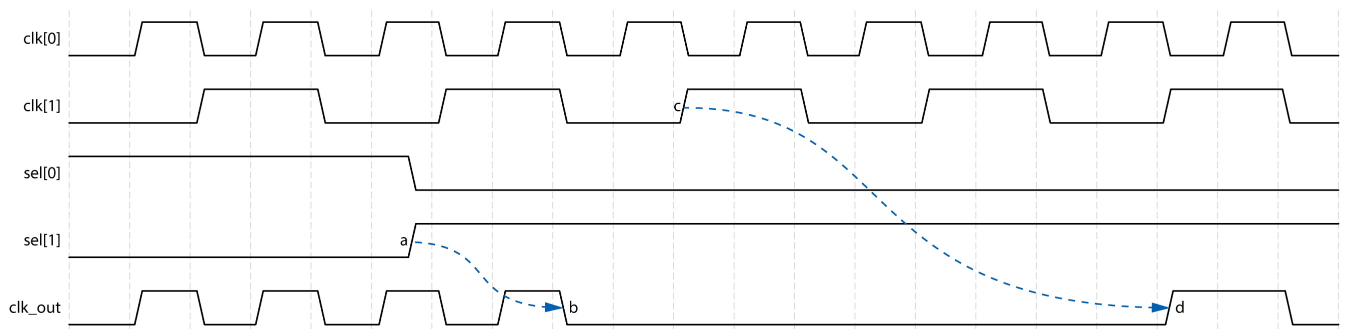


Figure 34: ***SYNCHRONIZE_SEL = 1 Timing Diagram***

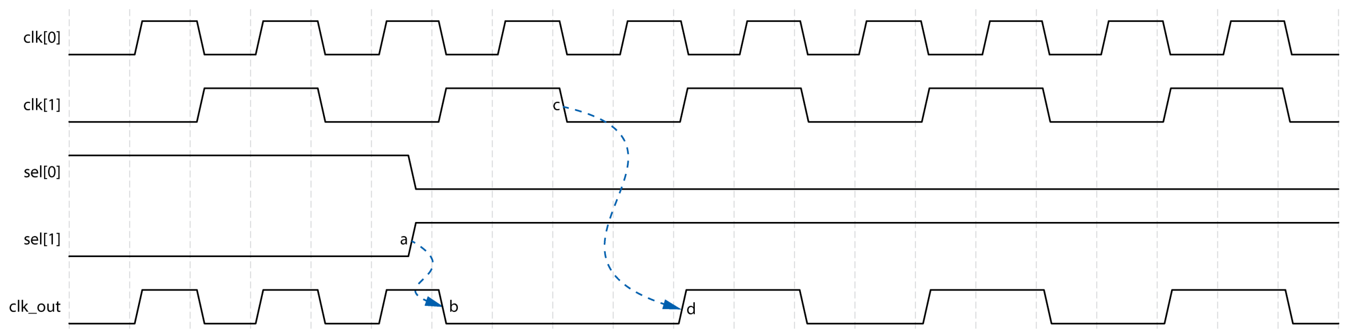


Figure 35: ***SYNCHRONIZE_SEL = 2 Timing Diagram***

Constraints

The ACX_CLKSWITCH component does propagate the input clock frequency from both `clk_in` ports to the `clk_out` port. Therefore, it is not necessary to specify any additional timing constraints for `clk_out`. Synplify Pro and ACE correctly pass the input clock domains through to the output clock domain for static timing analysis purposes.

Warning!

Because the ACX_CLKSWITCH component correctly propagates clock domains, do not apply constraints to the output of the ACX_CLKSWITCH. All clock switch constraints must be removed from any existing or re-used designs.

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.components.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

Number Formats

Within the machine learning processor (MLP72), a variety of different number formats are used. It is important to understand these formats, how they are represented and what their resolution is, so that the correct choices may be made with regard to the math to be performed, and subsequently how to configure correctly for the chosen number formats.

Integer Formats

Integer values can be represented in three possible formats:

- Signed – The highest order bit (MSB) represents the sign. The other bits equate to the distance from either 0 (for a positive number, sign bit = 1'b0) or the most negative possible value (sign bit = 1'b1). This format is commonly known as signed two's compliment.
- Unsigned – Only positive values are represented, with all bits representing the binary value, i.e., the distance from zero.
- Signed magnitude – The MSB is the sign bit. The remaining bits represent the value in the same form as unsigned, i.e., the distance from zero. Signed magnitude has a negative range one less than that of signed due to the fact that zero can be represented by 1000_0000 or 0000_0000 (using 8-bit values as the example).

The following examples shown how the same values can be represented by each method.

Table 73: *Integer Format Examples*

Decimal Value	Format	Bit 7	Bits [6:0]
24	Unsigned	0	`001_1000`
24	Signed	0	`001_1000`
24	Signed magnitude	0	`001_1000`
–24	Unsigned	Cannot be represented	
–24	Signed	1	`110_1000`[1]
–24	Signed magnitude	1	`001_1000`

Table Notes

1. For an 8-bit signed integer the most negative value is −128. The distance from the most negative value is 128 – 24 = 104. The binary representation of 104 is `110_1000`.

Integer Groups

The formats are listed in their groups, which represent their size in bits. These group names are used in the MLP parameters to represent the size of the integers directed to each respective multiplier. Within a group, each named format is followed by its token name. These token names are used by the MLP parameters to configure the modes of each multiplier.

INT16

Table 74: ***Signed 16 Bit (Signed16)***

Bit Position	15	[14:0]	Range
Function	Sign bit	Value	−32768 to +32767

Table 75: ***Unsigned 16 Bit (Unsigned16)***

Bit Position	[15:0]	Range
Function	Value	0 to +65535

Table 76: ***Signed Magnitude 16 Bit (SMAG16)***

Bit Position	15	[14:0]	Range
Function	Sign bit	Value	−32767 to +32767

INT8

Table 77: ***Signed 8 Bit (Signed8)***

Bit Position	7	[6:0]	Range
Function	Sign bit	Value	−128 to +127

Table 78: ***Unsigned 8 Bit (Unsigned8)***

Bit Position	[7:0]	Range
Function	Value	0 to +255

Table 79: ***Signed Magnitude 8 Bit (SMAG8)***

Bit Position	7	[6:0]	Range
Function	Sign bit	Value	−127 to +127

INT7

Table 80: *Signed 7 Bit (Signed7)*

Bit Position	6	[5:0]	Range
Function	Sign bit	Value	−64 to +63

Table 81: *Unsigned 7 Bit (Unsigned7)*

Bit Position	[6:0]	Range
Function	Value	0 to +127

Table 82: *Signed Magnitude 7 Bit (SMAG7)*

Bit Position	6	[5:0]	Range
Function	Sign bit	Value	−63 to +63

INT6

Table 83: *Signed 6 Bit (Signed6)*

Bit Position	5	[4:0]	Range
Function	Sign bit	Value	–32 to +31

Table 84: *Unsigned 6 Bit (Unsigned6)*

Bit Position	[5:0]	Range
Function	Value	0 to +63

Table 85: *Signed Magnitude 6 Bit (SMAG6)*

Bit Position	5	[4:0]	Range
Function	Sign bit	Value	−31 to +31

INT4

Table 86: *Signed 4 Bit (Signed4)*

Bit Position	3	[2:0]	Range
Function	Sign bit	Value	−8 to +7

Table 87: ***Unsigned 4 Bit (Unsigned4)***

Bit Position	[3:0]	Range
Function	Value	0 to +15

Table 88: ***Signed Magnitude 4 Bit (SMAG4)***

Bit Position	3	[2:0]	Range
Function	Sign bit	Value	−7 to +7

INT3

Table 89: ***Signed 3 Bit (Signed3)***

Bit Position	2	[1:0]	Range
Function	Sign bit	Value	−4 to +3

Table 90: ***Unsigned 3 Bit (Unsigned3)***

Bit Position	[2:0]	Range
Function	Value	0 to +7

Table 91: ***Signed Magnitude 3 Bit (SMAG3)***

Bit Position	2	[1:0]	Range
Function	Sign bit	Value	−3 to +3

Floating-Point Formats

A key feature of the MLP72 is the ability to process and manipulate floating-point (FP) numbers, which have a wider range of methods for representing values. Floating-point representations divide the bits into a mantissa and an exponent. The mantissa represents the value, and the exponent represents the equivalent of a bit shift left or right of this value. This shift is equivalent to multiplying the value by 2^n, where *n* represents the exponent value.

Formats

MLP72 supports three floating-point formats, characterized by their total size (`fp_size`) and by the number of exponent bits (`fp_exp_size`). The difference, `fp_size` − `fp_exp_size`, is the size of the mantissa bits and represents the precision. The three supported formats are listed below:

Table 92: ***Supported Floating-Point Formats***

Format	FP Size	FP Exponent Size	Precision	Alternative Names
fp24	24	8	16	
fp16	16	5	11	binary16, half precision
fp16e8	16	8	8	bfloat16 (brain float). Not to be confused with block floating point.

Table Note

- The fp16 format is defined in the IEEE-754 standard as binary16. The other formats follow the IEEE-754 standard's rules for representation and rounding, but are not specifically defined in the standard. The bfloat16 format is supported by TensorFlow.

The MLP72 supports all three formats for both input and output, but internally, all operations are performed with fp24. For example, when the internal accumulator is used, the accumulation is performed with the extra precision of fp24 even if the output is one of the 16-bit formats.

Representation

The binary representation of a floating-point number has the form:

Table 93: ***Floating-Point Number Representation***

	Sign	Exponent	Mantissa
Bits	1	fp_exp_size	(precision−1)

Positive numbers have sign = 0; negative numbers have sign = 1. The special cases of 0.0 and infinity can both have a sign.

The mantissa is normalized to have MSB = 1. Since the MSB is always 1, it is not stored. This "hidden 1" is why the precision is one higher than the number of bits in the mantissa. An exponent *e* is stored as *e* + bias, where the bias is defined in the table below. This table also lists the limits for the (absolute) value that can be represented.

There are two special exponents:

- If the exponent field is 0, the value of the floating-point number is +0.0 or −0.0.
- If the exponent field is all 1s (255 or 31, depending on fp_exp_size), the value is ±infinity.

If a number is not 0.0 and not infinity, its value is 1.mantissa × $2^{(exp\text{-}bias)}$ with the appropriate sign. Here, 1. mantissa starts with the hidden 1 and has the mantissa as a binary fraction.

Table 94: ***Floating-Point Number Range***

Format	Bias	Exp for inf	Minimum Positive	Maximum Positive
fp24	127	255	$2^{(-126)}$	$2^{128} - 2^{112}$
fp16	15	31	$2^{(-14)}$	$2^{16} - 2^{5} = 65504$
fp16e8	127	255	$2^{(-126)}$	$2^{128} - 2^{120}$

Subnormal numbers (numbers too close to 0 to be normalized) are not supported; they are changed to 0.0. Likewise, NaN ("not a number", for invalid operations), is not supported; it is changed to infinity.

Rounding

When the result of an operation does not fit exactly within the precision, it is rounded to the nearest value that can be represented. If the true result is equally close to two values, it is rounded to the even one (the one that has LSB = 0).

If the absolute value of a result is too large to be represented, the result is changed to infinity. Similarly, if the absolute value of a result is too small to be represented, it is changed to 0. The latter case is called underflow, describing the situation where the floating-point result is 0.0 but mathematically the result should not be 0 (for example, subtracting two not quite equal numbers might result in underflow if the numbers are small).

Internally, all operations are performed with fp24, including the rounding of multiplications and additions. If a 16-bit output format is selected, the fp24 result is then rounded a second time to 16 bits. In rare cases, this double rounding can give a slightly different result than if the operation had been performed with only 16 bits.

Block Floating Point

Block floating point (block fp) is not a number format *per se*, rather it is a method of processing a collection of floating-point numbers efficiently. The principle is that each floating-point number consists of a mantissa and an exponent. If the exponents are made the same (normalized), then the mantissas can be arithmetically combined in the same way as integer numbers. The result of the mantissa calculation can be recombined with the normalized exponent, resulting in a full floating-point result. During this process there are two semi-independent input formats involved:

- The original floating-point format
- The integer format used for the multiplication

There are a number of considerations when normalizing the exponent among a collection of floating-point numbers. The value of a floating-point number is

`mantissa × 2exponent` where `mantissa` is of the form 1.`fraction`.

However, in order to increase accuracy for the same number of bits, the '1' is not stored, it is implicit. A floating-point number is stored as {`sign_bit, exponent, fraction`}.

When converting from a floating-point value to a block floating-point value, firstly the '1' needs to be added to the fraction in order to give the full mantissa. The mantissa is then right-shifted, while incrementing the exponent, until the exponent has the desired value equivalent to the maximum exponent in the block. It is this shift process that requires the implicit '1' to firstly be re-inserted with the `fraction`; after shifting, the implicit '1' is no longer in the MSB position.

The shifted mantissa plus sign bit must fit within the integer format that is used. If it is necessary to right-shift a significant amount, then precision is lost. It is possible to end up with 0 if the original exponent was small enough.

The choice of integer size, therefore, depends on the required precision and the range of exponent values that are to be processed together.

ALU8

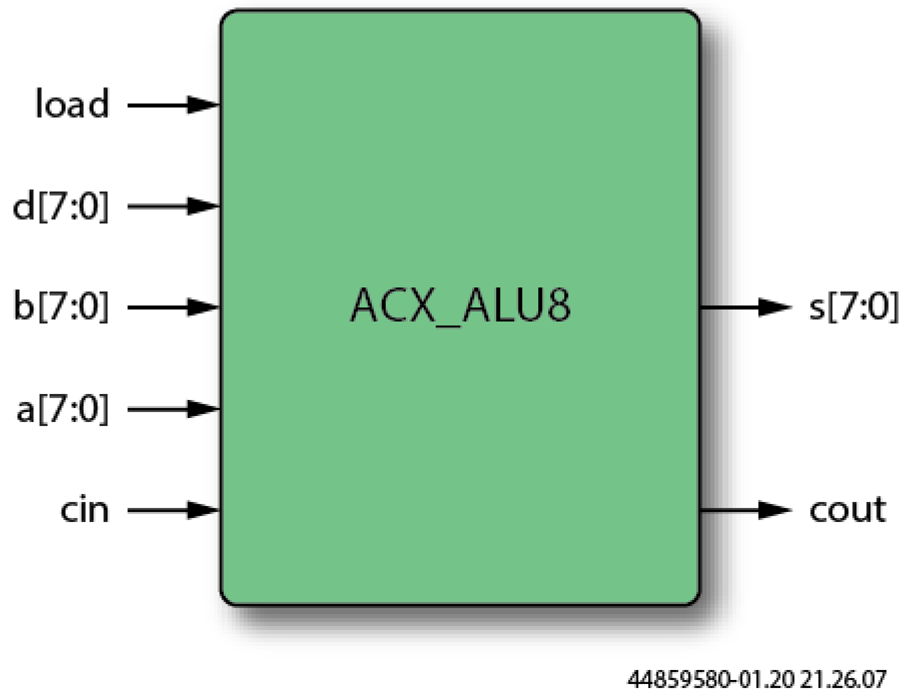


Figure 36: ***Eight-Input Adder/Subtractor with Programmable Load***

Description

The ACX_ALU8 implements either an 8-bit adder or 8-bit subtractor with the following inputs:

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

It generates the following outputs:

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

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.

Parameters

Table 95: *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_ALU8s 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 96: *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 97: ***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 98: ***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 Templates

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)
);
```

VHDL

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

-- Component Instantiation
instance_name : ACX_ALU8
generic map (
    invert_b => '0'
)
port map (
    s          => user_sum,
    cout      => user_carry_out,
    a          => user_a,
    b          => user_b,
    d          => user_d,
    load      => user_load,
    cin      => user_carry_in
);
```

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 amount of data inputs to the MLP block increases along with the number of multipliers available.

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.

Below are block diagrams showing the MLP using the fixed or floating-point formats.

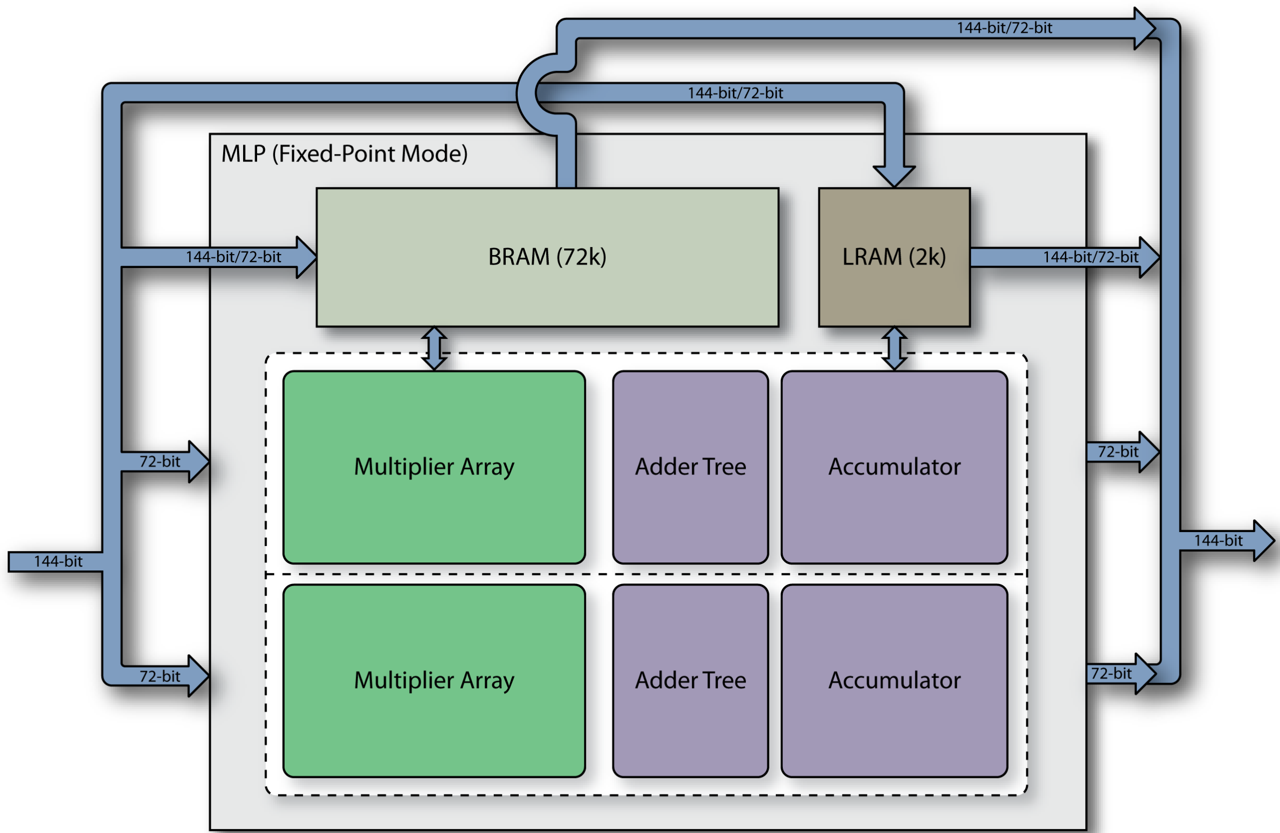


Figure 37: ***MLP Using Fixed-Point Mode***

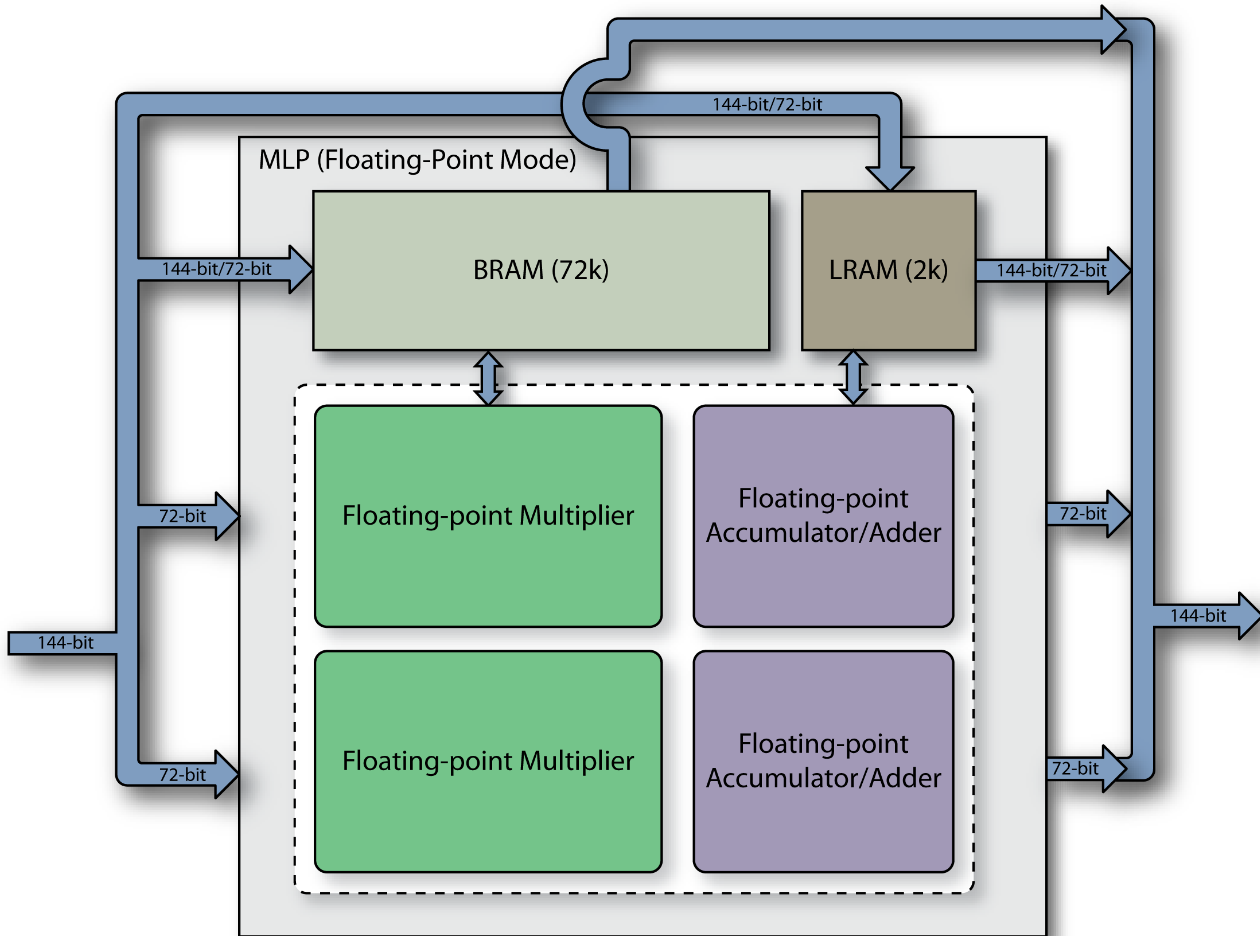


Figure 38: ***MLP Using Floating-Point Mode***

A powerful feature available in Achronix's 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. Below is a diagram showing the cascade paths across MLPs.

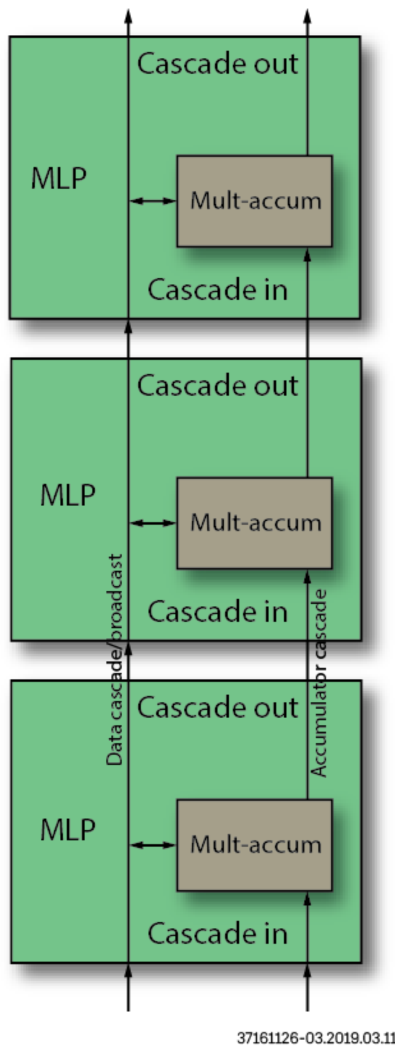


Figure 39: ***MLP Cascade Path***

> **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 99: ***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 where the data input to the MLP is sourced from and are described in the section Modes. (see page 101)

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

Table 100: ***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
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)

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
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 (see page 248) and a ACX_LRAM2K (see page 280) 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 (see page 173) and floating-point (see page 209) 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 101)
- Input Selection (see page 104)

- Integer Byte Selection (see page 110)
- Integer Multiplier Stage (see page 116)
- Integer Output Stage (see page 120)
- LRAM (see page 123)

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

- Block Floating Point (see page 130)
- Floating Point (see page 137)

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 figure, Delay Stage Structure (see page 105).

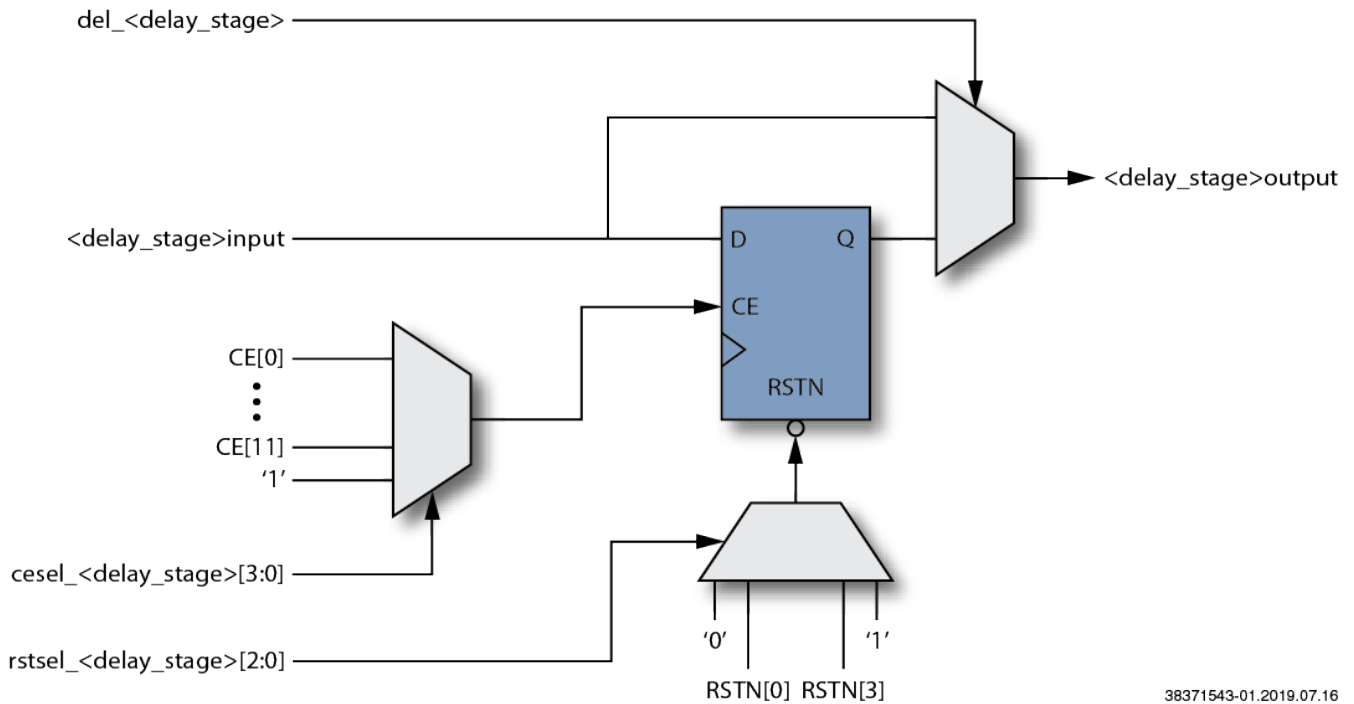


Figure 40: ***Delay Stage Structure***

In the diagrams which follow, showing the various stages of the MLP72, the delay stages are shown as a register with dotted outline, to indicate 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 Delay stage symbol (see page 106)

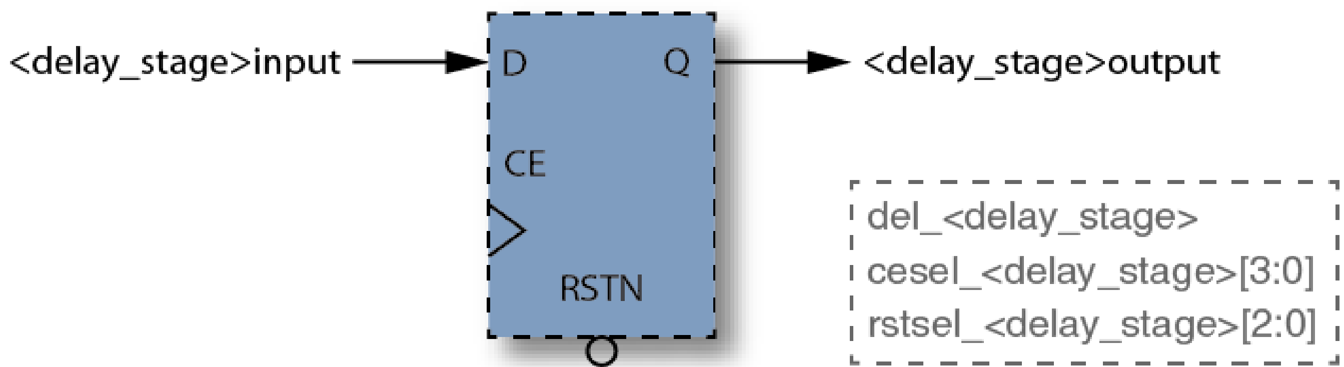


Figure 41: ***Delay Stage Symbol***

Parameters

Table 101: ***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.
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 102: ***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 diagrams below.

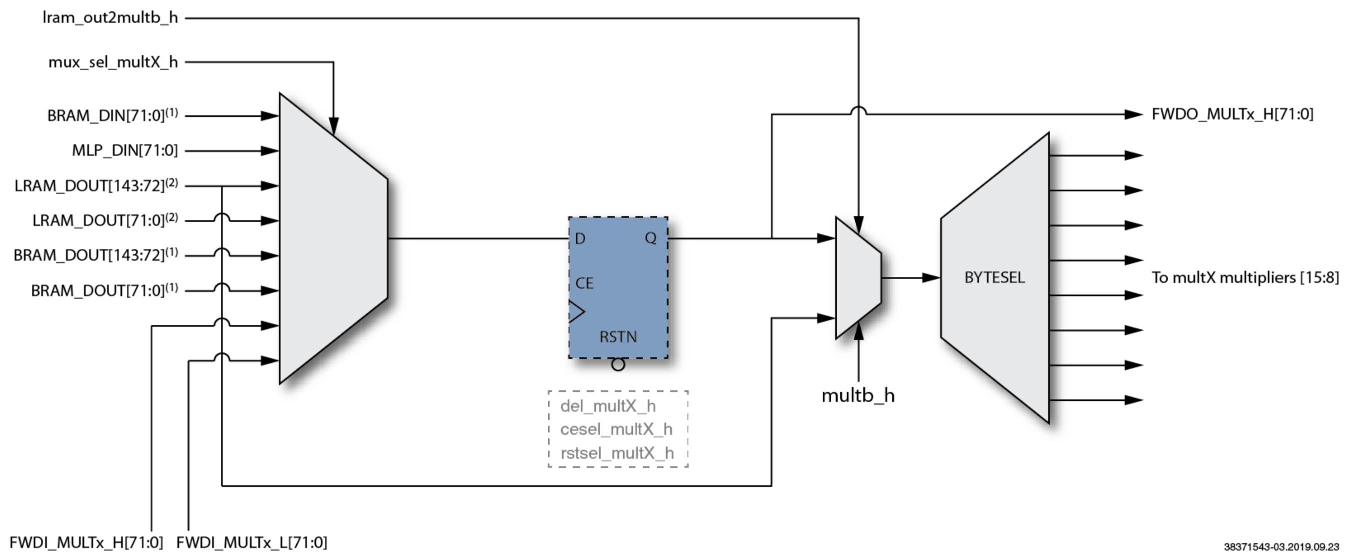


Figure Notes

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 Ports table below

Figure 42: ***High Bank Multipliers Input Selection***

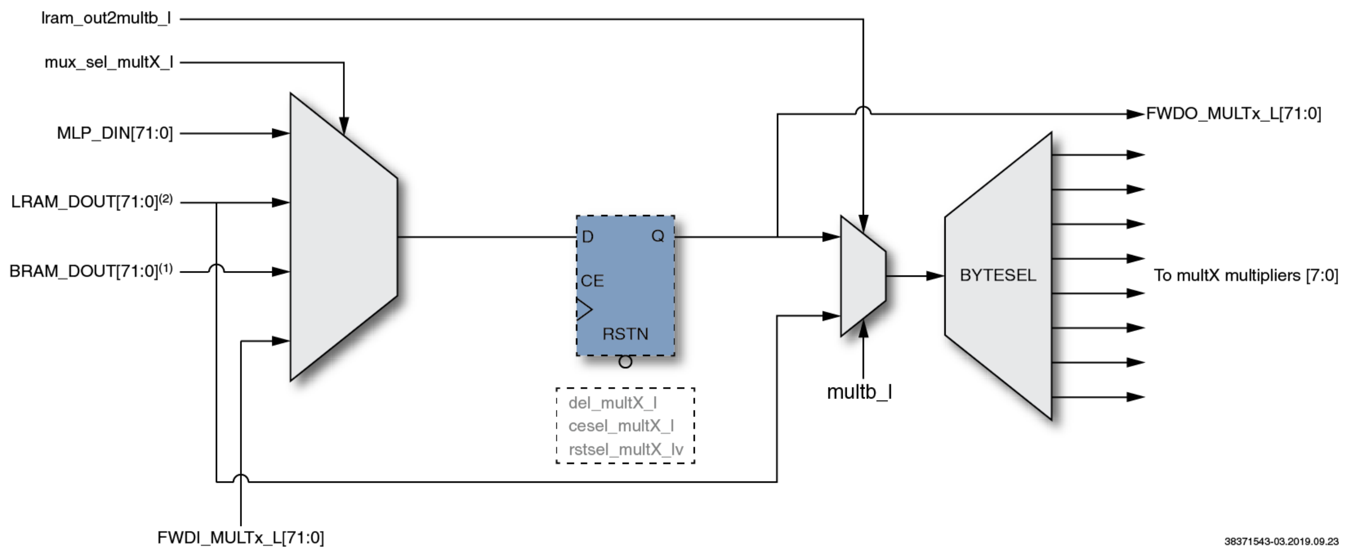


Figure Notes

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 Ports table below.

Figure 43: ***Low Bank Multipliers Input Selection***

Parameters

Table 103: ***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 Ports table below.

Ports

Table 104: ***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 tables below 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 105: ***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 106: ***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 107: ***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 108: ***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 109: ***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 110: ***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 111: ***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 112: ***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 113: ***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 114: ***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 115: ***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 116: ***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 117: ***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 118: ***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 119: ***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 120: ***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 121: ***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 figure below. The parameters which control signal selection, delay stage selection, add or subtract are shown as text only alongside the component they apply to.

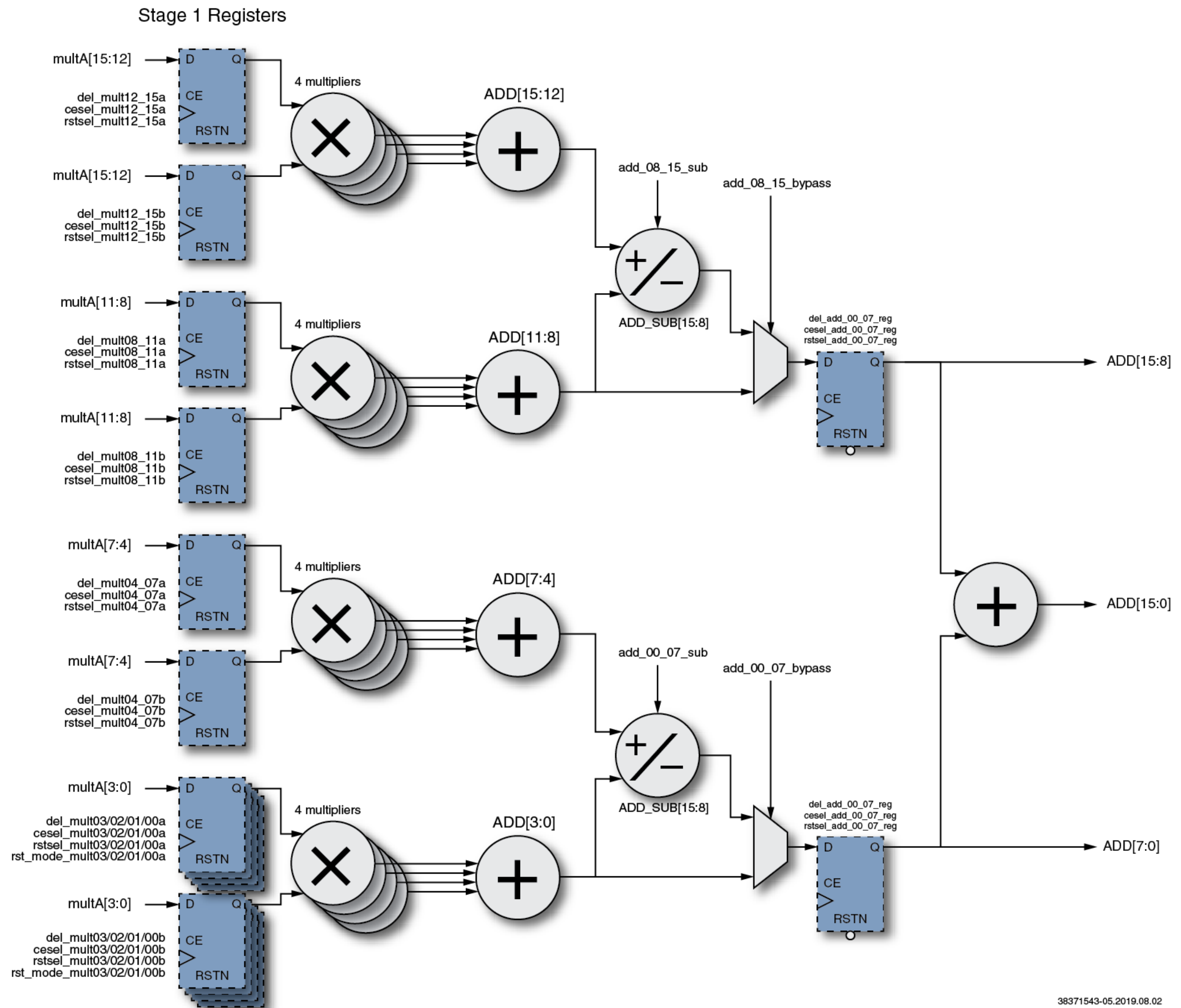


Figure 44: ***Multiplication Stage Structure (Integer)***

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 table below.

Table 122: ***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 (see page 85) . 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 116) 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 110). 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 table below. For the purposes of clarity, the delay stage parameters are not shown in this table, they are shown in the previous Figure (see page 117).

Table 123: *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.

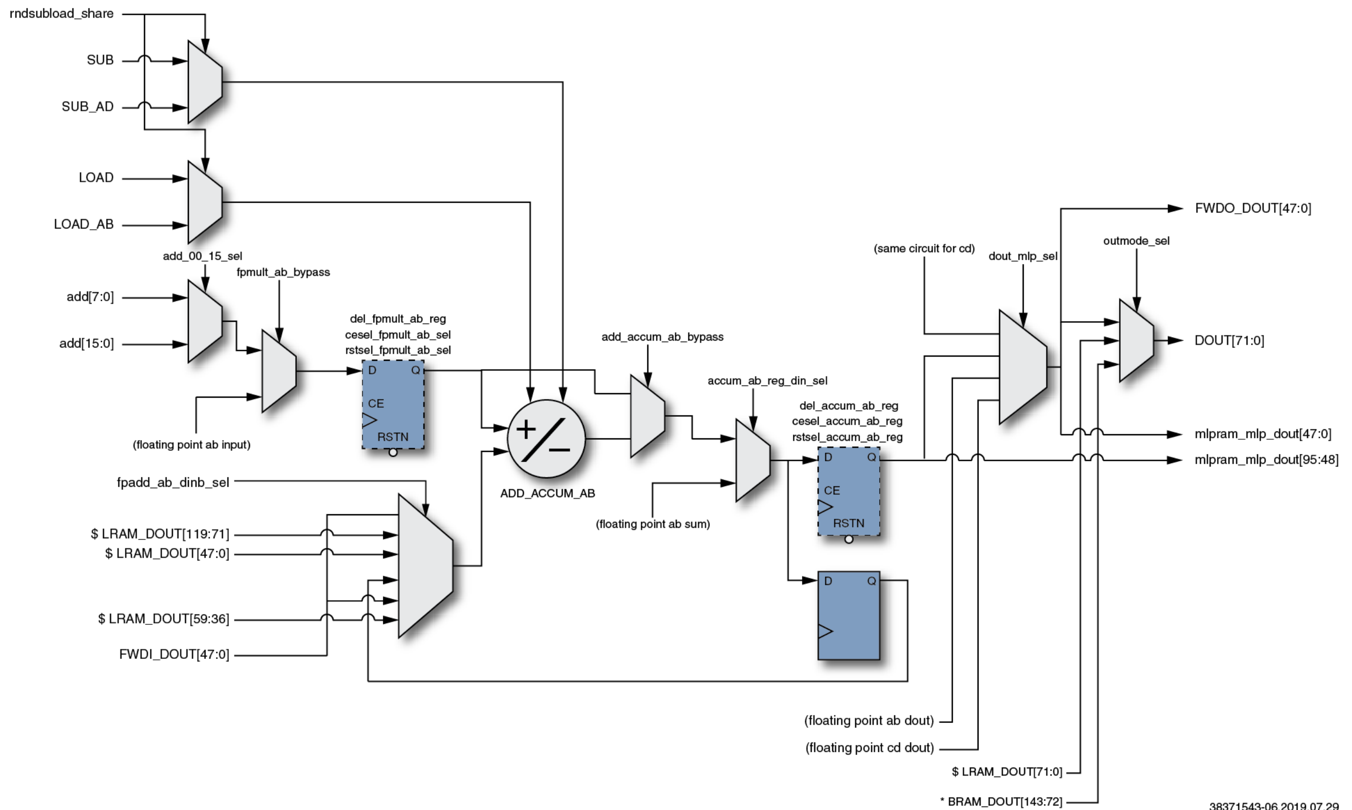


Figure 45: ***Output Stage***

Parameters

Table 124: *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 125: ***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 ACX_MLP72 below.
fwdo_dout[47:0]	Output	MLP72 internally calculated results, cascaded up to ACX_MLP72 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 (see page 280) or ACX_LRAM2K_FIFO (see page 272) 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 123)
- 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 125)). 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, they are recommended to discuss their 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 below.

Table 126: ***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 below.

lram_write_data_mode
lram_accum_data_input_sel
{84'h0, accum_ab_reg[23:0], 12'h0, accum_cd_reg[23:0]}
{24'h0, accum_ab_reg[47:0], 24'h0, accum_cd_reg[47:0]}
multb_h[71:0o], multb_l[71:0]
{MLPRAM_DIN2MLPDOUT[71:0], MLP_DIN[71:0]
MLPRAM_DIN2MLPDOUT[143:0]
CE[7]
MLPRAM_WREN
lram_wren
LRAM
DIN DOUT
WREN RDEN
WRADDR RDADDR
RSTN
lram_reg_dout
lram_sr_assertion
D Q
CE
RSTN
LRAM_DOUT[143:0]
CE[6]
MLPRAM_RDEN
lram_rden
RSTN[0]
1'b1
lram_regrstn
EXPB[7:2]
MLPRAM_WRADDR[5:0]
lram_wraddr
CE[9]
1'b0
lram_fsm_wrrst
lram_input_control_mode
FIFO Controller
{EXPB[1:0],CE[11:8]}
MLPRAM_RDADDR[5:0]
lram_rdaddr
CE[8]
1'b0
lram_fsm_rdrst
lram_output_control_mode
lram_fifo_enable
Text Legends
CAPITAL[71:0] = Inputs to MLP
lower case[71:0] = Internal MLP signals
Italics = parameters
38371543-06.2019.09.23

Figure Notes

1. LRAM_DOUT[143:0] is an internal connection only from the coupled LRAM. This is not available as an output port from the MLP.
2. BRAM_DIN[143:0] is a logical name for the respective signal path. The physical port names vary, and are listed in the Inputs selection ports table.

Figure 46: *LRAM connectivity*

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 127: *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.
			Enable LRAM FIFO mode:

Parameter	Supported Values	Default Value	Description
lram_fifo_enable	1'b0–1'b1	1'b0	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_write_data_mode[1:0]	2'b00–2'b11	2'b00	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. 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.
			Enable LRAM FIFO address pointers to ignore empty/full status

Parameter	Supported Values	Default Value	Description
lram_fifo_ignore_flags	1'b0–1'b1	1'b0	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 128: *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.
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.

Name	Direction	Description
`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 130) 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 byte selection tables below are listed by the mantissa size, which have the same conventions and names as their integer equivalents.

BFP Int8

Table 129: ***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 130: *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 131: *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 132: *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 Note

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

Table 133: *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 134: *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 135: *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 136: *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 137: ***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 138: ***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 139: ***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 140: ***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 141: *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 142: *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 Note

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

Table 143: *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 144: *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 145: ***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 146: ***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 147: ***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 148: *Block Floating-Point Inputs*

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

Parameters

Table 149: *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 (see page 209). 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 (see page 85).

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 150: ***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 151: *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 152: *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 153: *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 154: *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 155: ***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 156: ***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 157: ***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 158: ***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 159: ***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 160: ***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 161: ***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 162: ***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 163: ***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 164: ***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 diagram below 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.

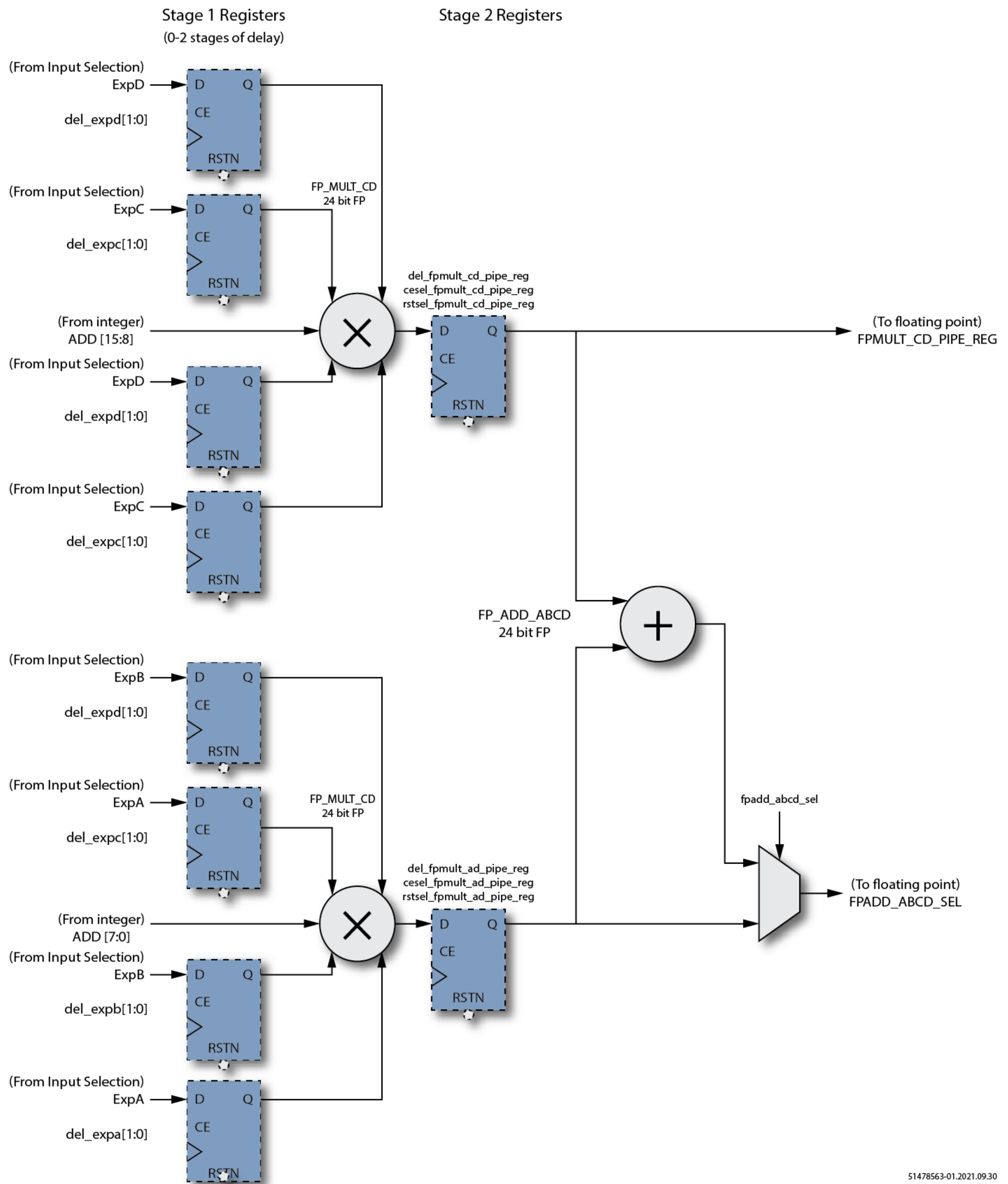


Figure 47: ***Floating-Point Multiplier Stage***

Parameters

Table 165: ***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.

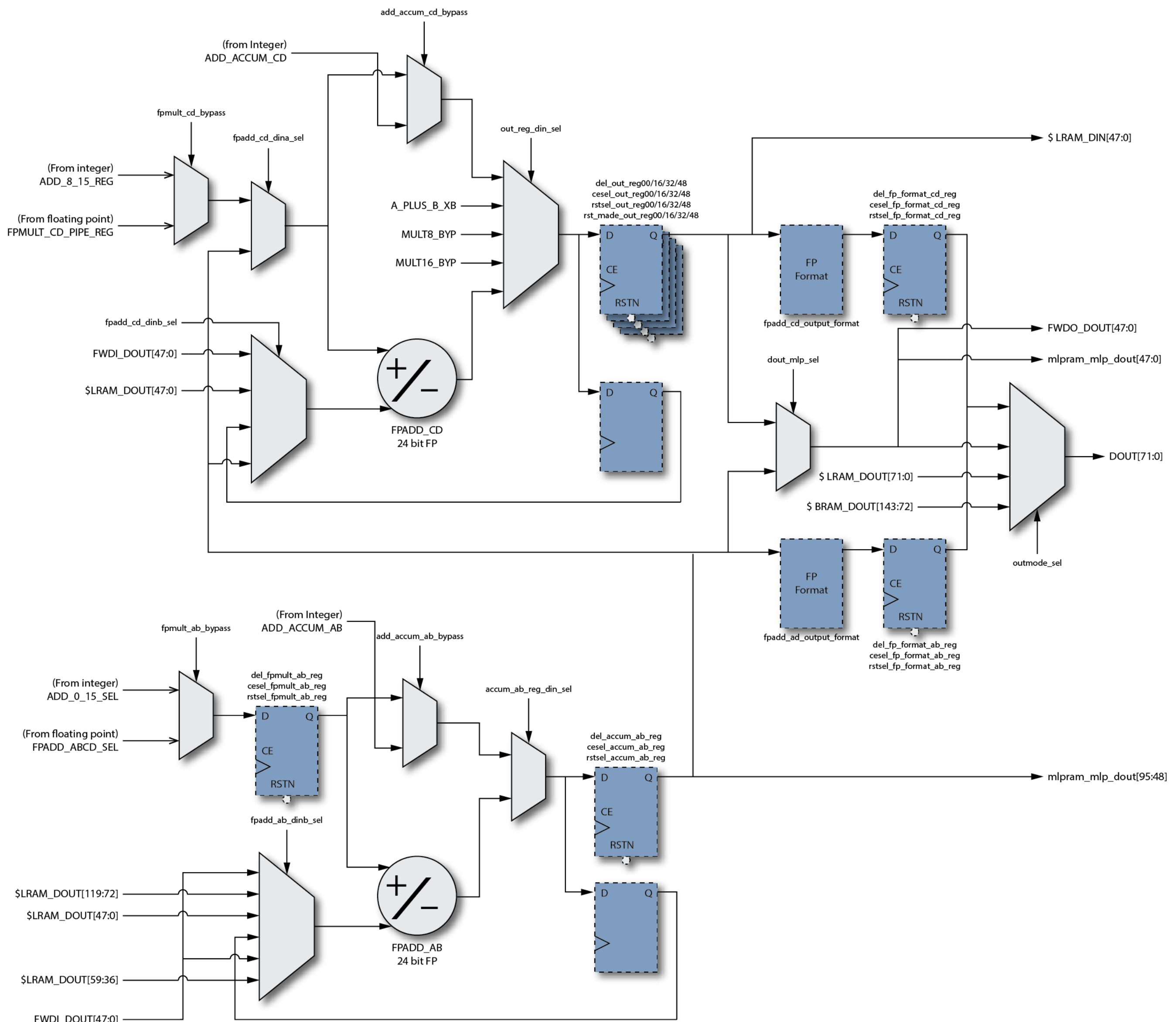


Figure 48: *Floating-Point Output Stage*

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 166: ***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).
			Selection of floating-point output format from FPADD_CD floating-point adder /accumulator:

Parameter	Supported Values	Default Value	Description
fpadd_cd_output_format[1:0]	2'b00–2'b10	2'b00	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}.

Verilog

code

```
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)
    );
```

ACX_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.

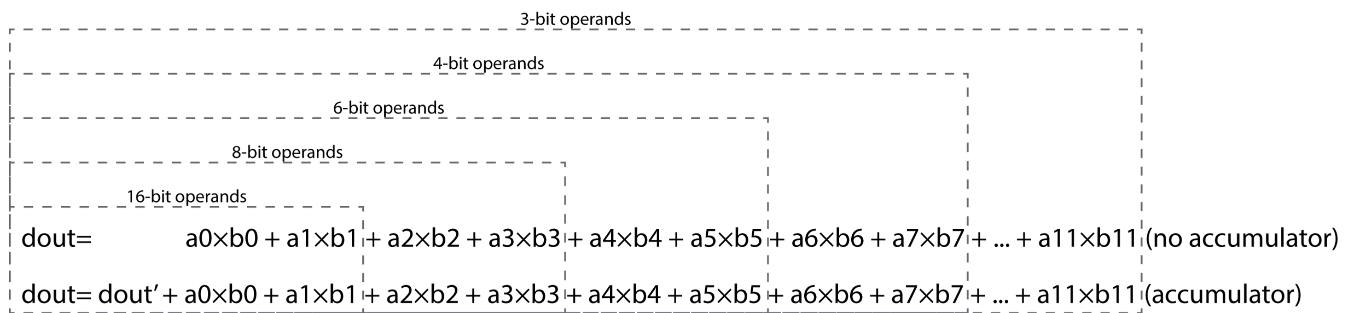


Figure 49: *ACX_MLP72_INT Arithmetic Expressions*

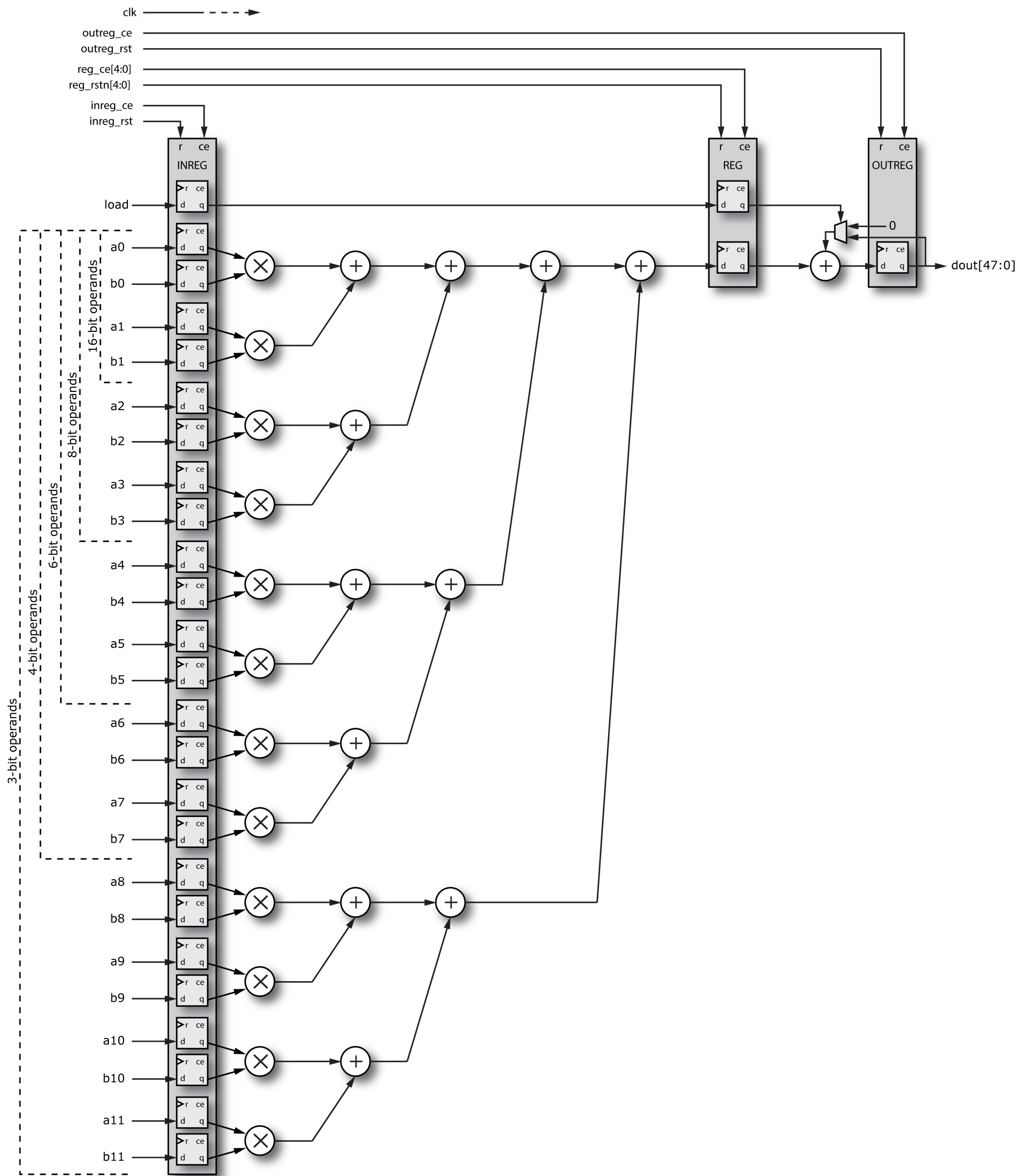


Figure 50: ***ACX_MLP72_INT Block Diagram***

Parameters

Table 167: *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 156) 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 168: *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 169: ***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 170: ***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]

B Operands	Input Widths					
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 table below describes valid scenarios when `inreg_sr_assertion` can be set to `"unclocked"` when the input register is enabled.

Table 171: *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        )
   );
```

ACX_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.

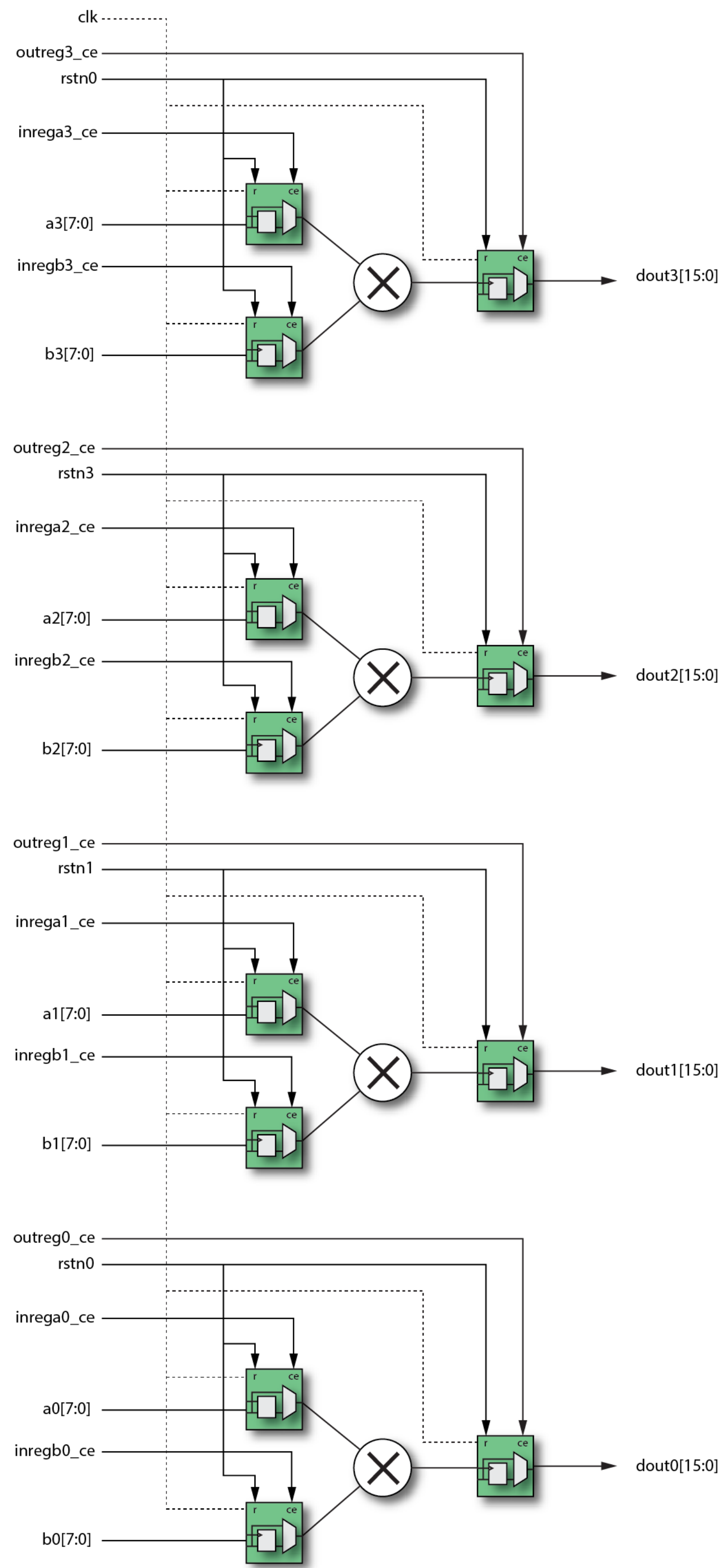


Figure 51: ***ACX_MLP72_INT8_MULT_4X Block Diagram***

Parameters

Table 172: ***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 173: *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.

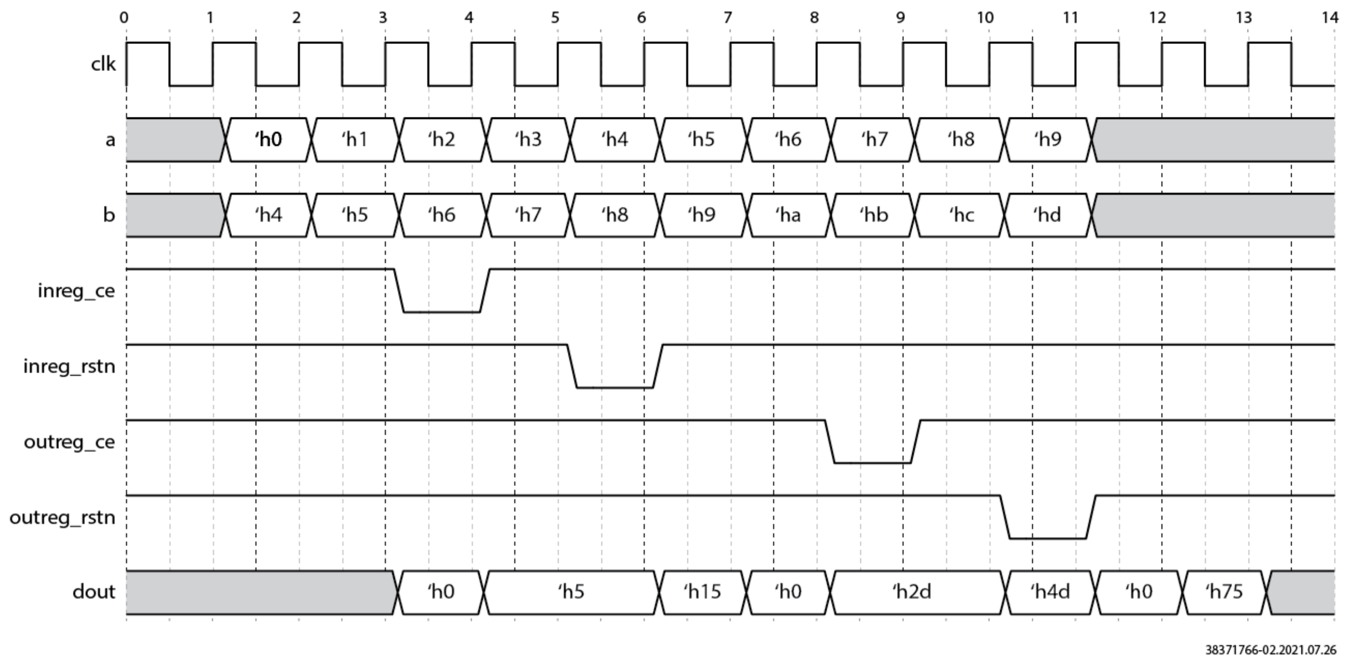


Figure 52: ***Timing Diagram for Single Multiplier Channel***

Inference

The ACX_MLP72_INT8_MULT_4X is inferrable using RTL constructs commonly used to infer multiplication operations, such as those shown below.

Note

- This component is appropriate for integer data widths of 8 bits and below.
- For widths larger than 8 bits, use ACX_MLP72_INT16_MULT_2X (see page 166).
- 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      )
    );
```

ACX_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.

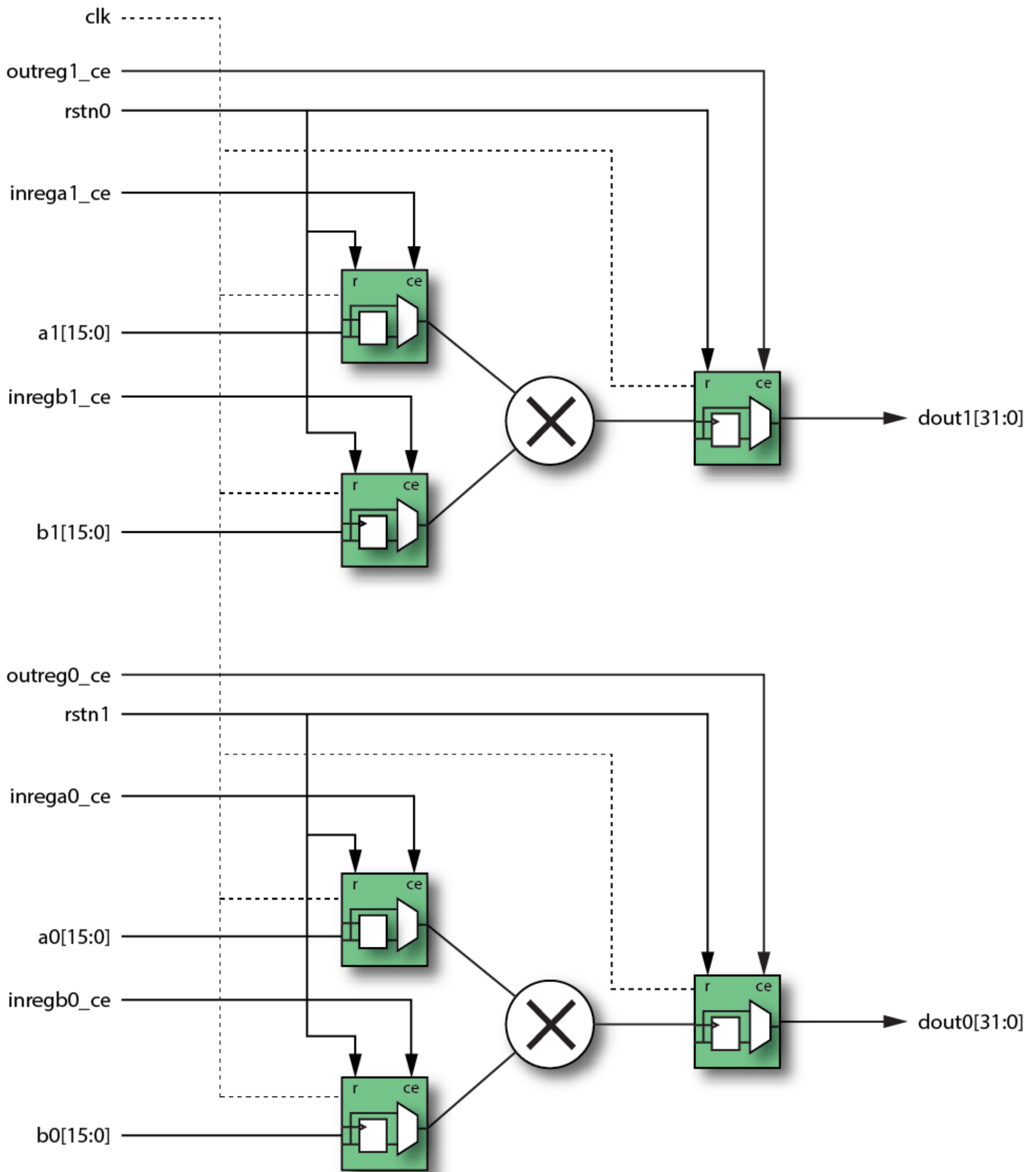


Figure 53: ***ACX_MLP72_INT16_MULT_2X Block Diagram***

Parameters

Table 174: *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 175: *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.

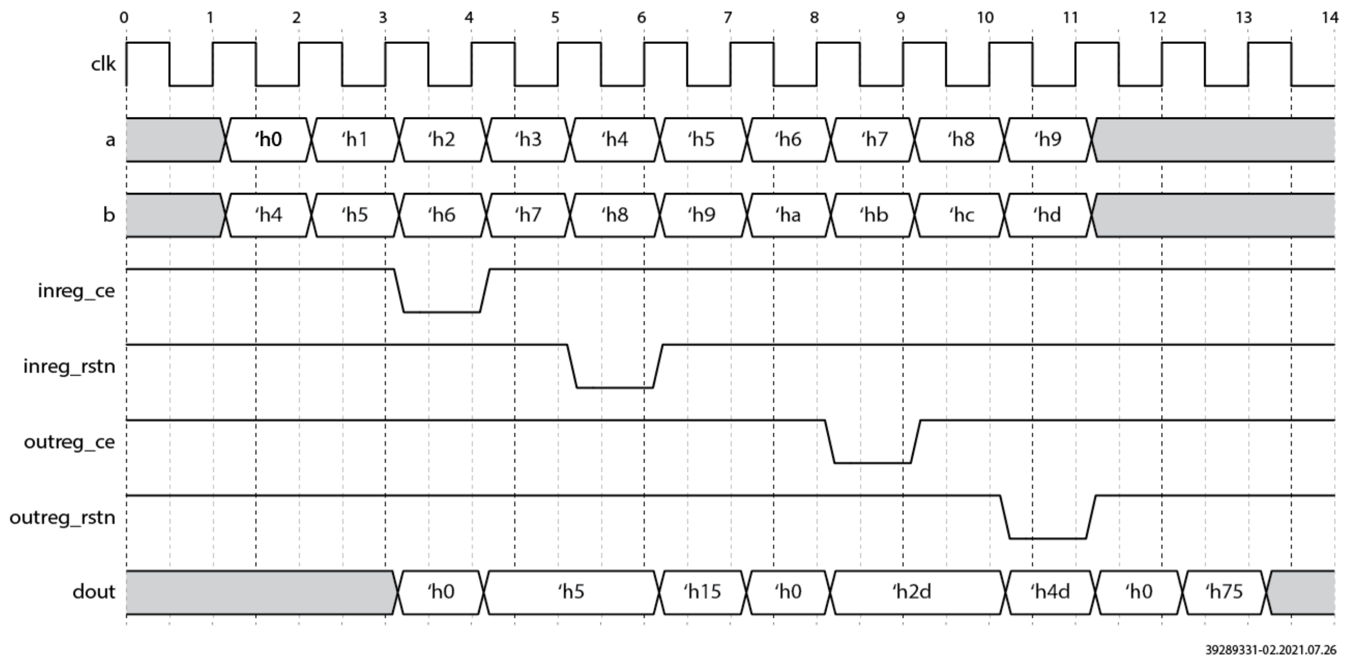


Figure 54: ***Timing Diagram for a Single Multiplier Channel***

Inference

The ACX_MLP72_INT16_MULT_2X is inferrable using RTL constructs commonly used to infer multiplication operations, such as those shown below.

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 (see page 158).

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.

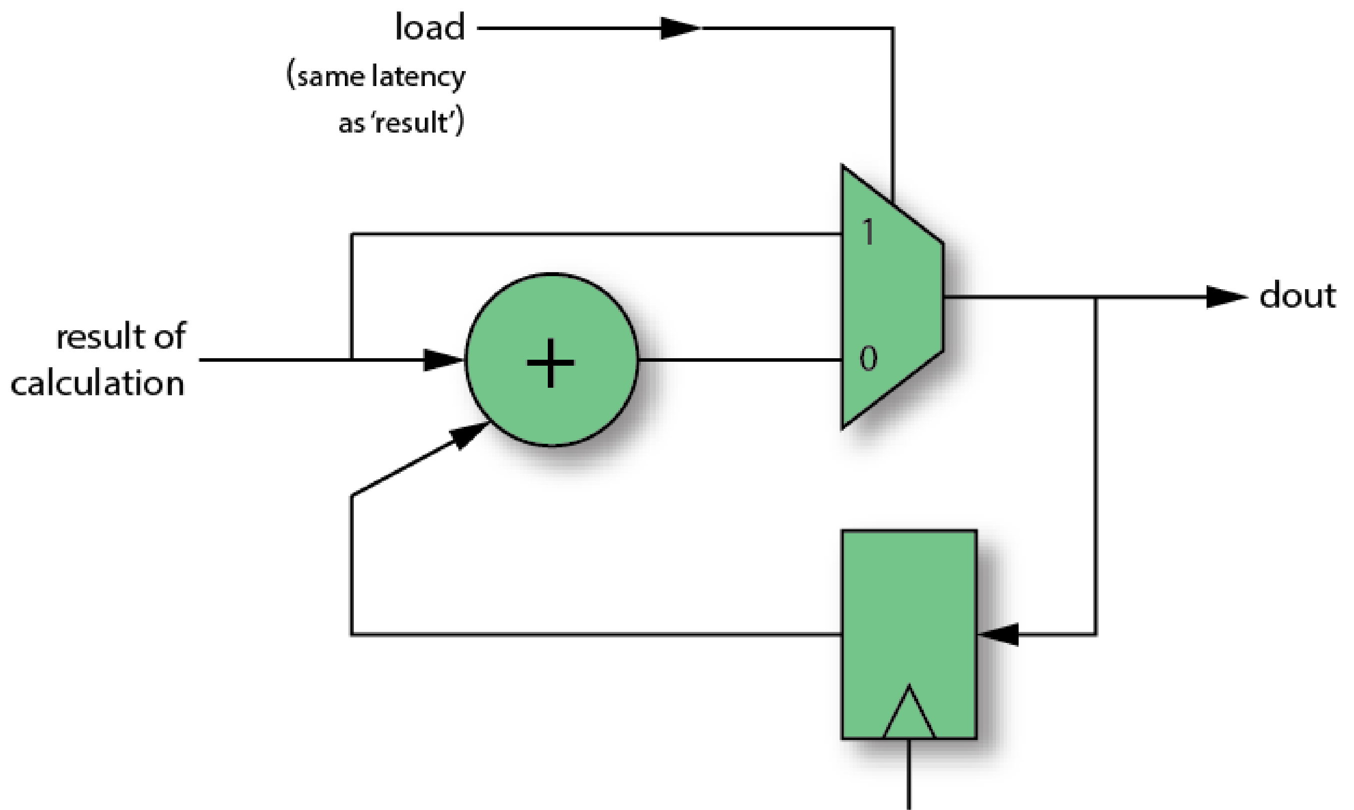


Figure 55: ***Accumulator with Load Signal***

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

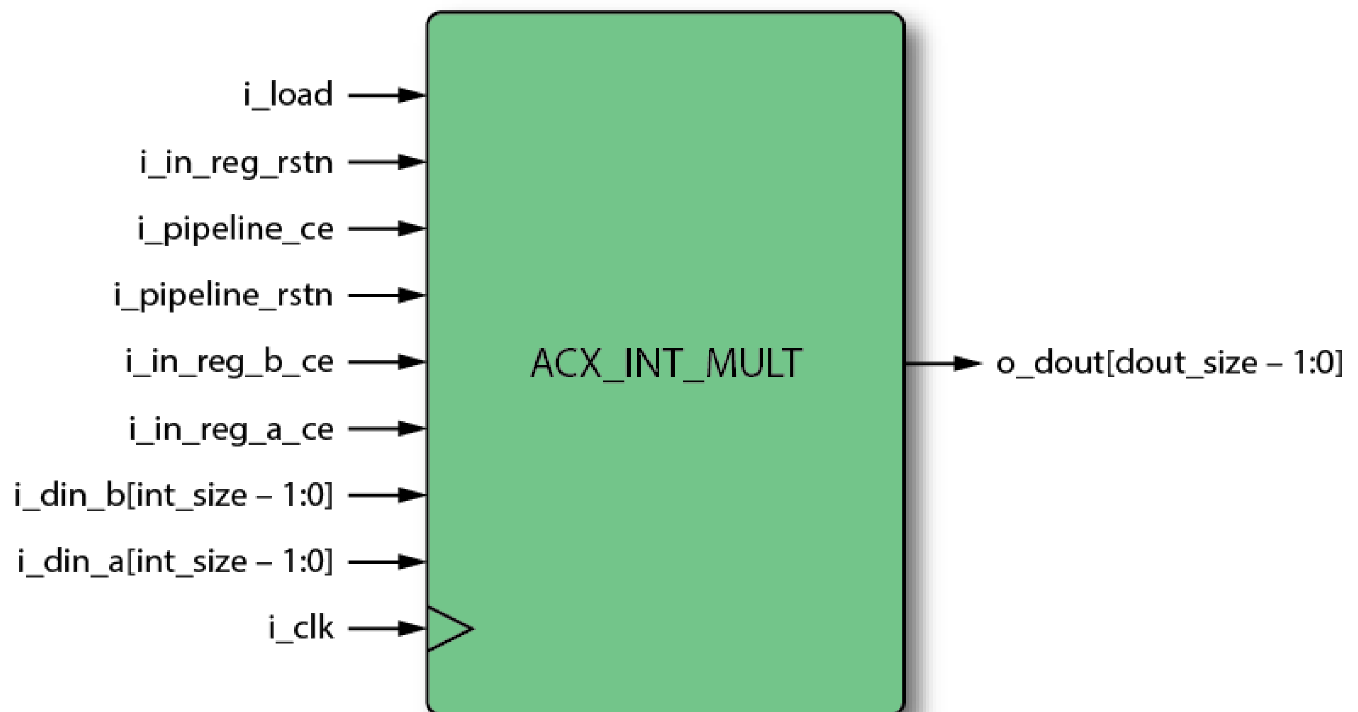


Figure 56: ***Integer Multiplier with Optional Accumulate***

Parameters

Table 176: *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 178)		Width of the `o_dout` output. The default and range are determined by several other parameters, as explained in the Output (see page 178) 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 178) for more information. "rlb" – implementation is with reconfigurable logic, including MLUT, ALU8 (see page 91), and DFF's (see page 24). "mlp" – implementation uses a single MLP72 (see page 95). "auto" – equivalent to "rlb" if `int_size <= 8`; equivalent to "mlp" if `int_size > 8`.

Ports

Table 177: ***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 (see page 18) count compared to other FPGAs.

For `int_size` ≤ 8, the `architecture` parameter can be used to select an implementation with an ACX_MLP72 (see page 95) (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 (see page 95) 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 178: *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

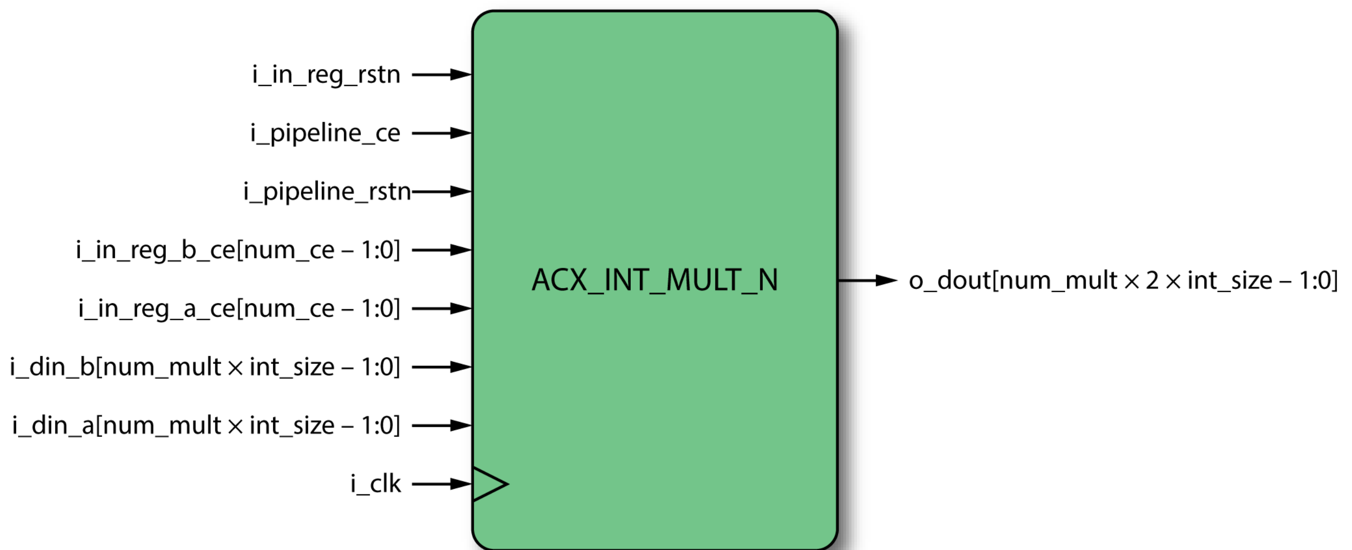


Figure 57: ***N Integer Parallel Multiplications***

Parameters

Table 179: *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 183) 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 below.

```
localparam integer num_ce = (int_size <= 4)? (num_mult + 1)/2 : num_mult
```

Ports

Table 180: *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 183) vector of A data input to multipliers.
`i_din_b[(num_mult*int_size-1):0]`	Input	Packed (see page 183) 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 183).
`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 183).
`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 183) 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 181: ***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 (see page 95). 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

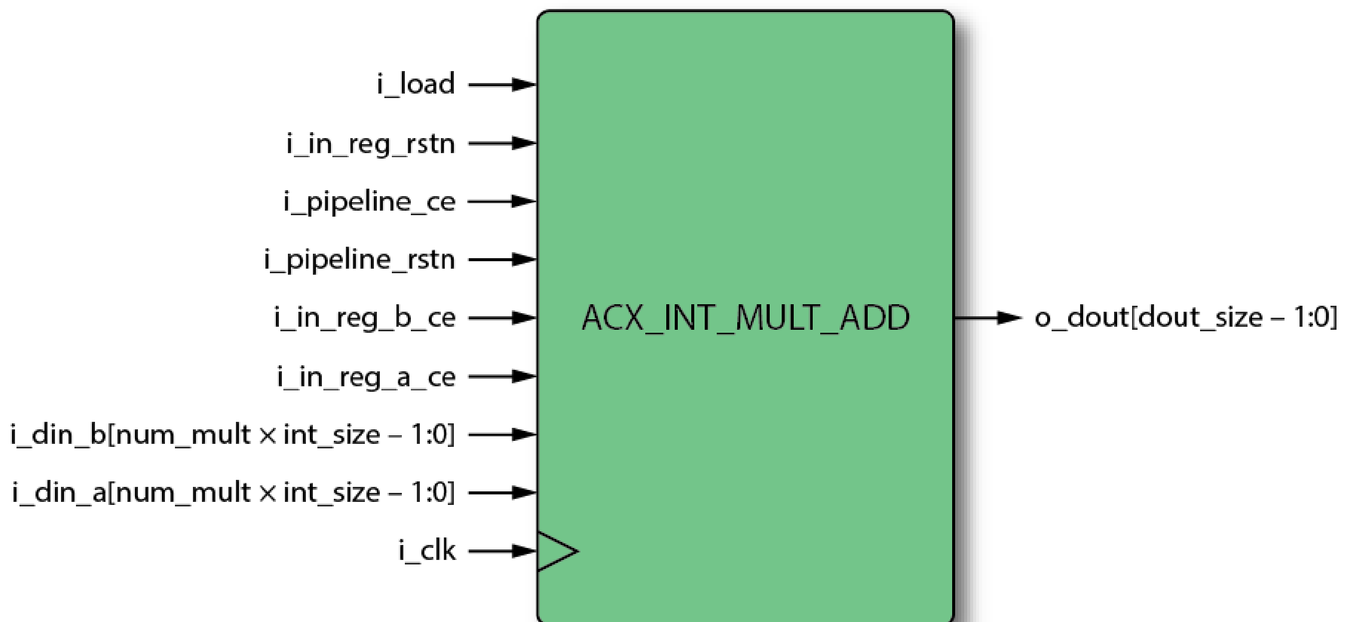


Figure 58: ***N Integer Sum of Products with Optional Accumulation***

Parameters

Table 182: *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 189) 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 183: ***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 189) vector of A data input to multipliers.
i_din_b[num_mult*int_size-1 : 0]	Input	Packed (see page 189) 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 (see page 95) 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 (see page 248) 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 table below 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 184: *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. (see page 95) 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
);
```

ACX_INT_RLB_MULT

The ACX_INT_RLB_MULT module implements integer multiplication with fabric logic. Both inputs must be signed, but may have different sizes. This macro features:

- N × M multiplication, for N, M = 3–9
- Optional accumulator
- Optional registers to enable higher frequency
- Implemented with RLB fabric logic only

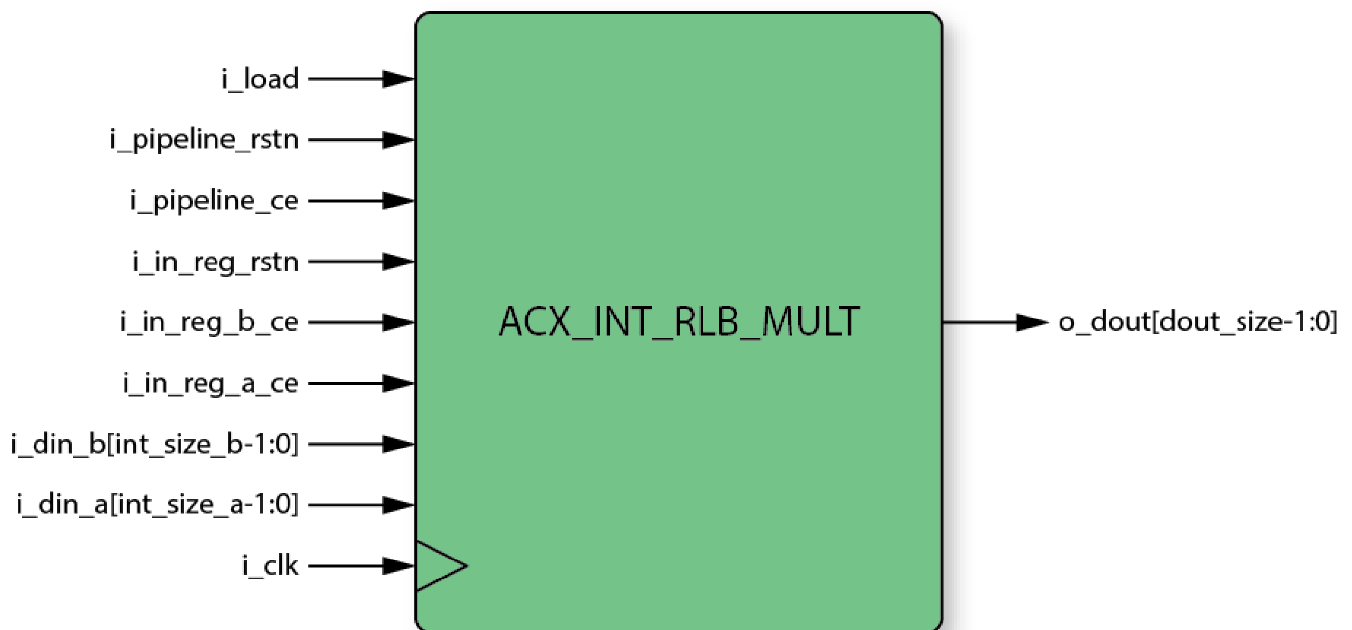


Figure 59: ***RLB-based Integer Multiplier with Optional Accumulate***

Parameters

Table 185: *ACX_INT_RLB_MULT Parameters*

Parameter	Supported Values	Default	Description
int_size_a	3, 4, 5, 6, 7, 8, 9	8	Number of bits for i_din_a.
int_size_b	3, 4, 5, 6, 7, 8, 9	8	Number of bits for i_din_b.
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	0	The number of pipeline registers, not counting the input register. The total latency is pipeline_regs + in_reg_enable.
dout_size	≥ 2	int_size_a + int_size_b	Width of the o_dout output. If accumulation is enabled, this value also represents the size of the accumulator. Signed results are sign-extended as necessary. Values that do not fit are truncated at the high-order bits.

Ports

Table 186: ***ACX_INT_RLB_MULT Pin Descriptions***

Name	Direction	Description
i_clk	Input	Clock input, used for the (optional) registers and accumulator.
i_din_a[int_size_a-1:0]	Input	A data input to multiplier (signed 2's complement).
i_din_b[int_size_b-1:0]	Input	B data input to multiplier (signed 2's complement).
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_RLB_MULT implements multiplication with reconfigurable logic (ACX_MLUT, ACX_ALU8 (see page 91), and ACX_DFF's (see page 24)), without using ACX_MLP72 (see page 95). This macro is similar to ACX_INT_MULT (see page 175) with the parameter `architecture = "rlb"`.

There are two reasons to use ACX_INT_RLB_MULT:

- it supports distinct sizes for 'a' and 'b' inputs, such as 5 × 8 multiplication
- it supports 9-bit integers

However, ACX_INT_RLB_MULT only supports signed integers and does not support 16-bit and 32-bit multiplication.

In addition to direct instantiation in Verilog or VHDL, an instance of ACX_INT_RLB_MULT can also be created in the ACE IP Configuration Perspective. See *Speedster7t Soft IP User Guide* (UG103) for details.

Note

It is not currently possible to infer MLUT-based multipliers. ACX_INT_MULT (see page 175) or ACX_INT_RLB_MULT must be instantiated directly.

Instantiation Templates

Verilog

```
// Verilog template for ACX_INT_RLB_MULT
ACX_INT_RLB_MULT #(
    .int_size_a    (int_size_a    ),
    .int_size_b    (int_size_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[int_size_a-1 : 0] ),
    .i_din_b           (user_i_din_b[int_size_b-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_RLB_MULT
component ACX_INT_RLB_MULT is
generic (
    int_size_a           : integer := 8;
    int_size_b           : integer := 8;
    accumulate           : integer := 0;
```

```
    in_reg_enable        : integer := 0;
    pipeline_regs        : integer := 0;
    dout_size            : integer := int_size_a + int_size_b
);
port (
    i_clk                : in  std_logic;
    i_din_a              : in  std_logic_vector( int_size_a-1 downto 0 );
    i_din_b              : in  std_logic_vector( int_size_b-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_RLB_MULT

-- VHDL Instantiation template for ACX_INT_RLB_MULT
instance_name : ACX_INT_RLB_MULT
generic map (
    int_size_a           => int_size_a,
    int_size_b           => int_size_b,
    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
);
```

ACX_INT_SQUARE_ADD

The ACX_INT_SQUARE_ADD module computes a parallel sum of squares, SUM a(i) × a(i), with optional accumulation. This macro features:

- integer size K = 3–8, or 16
- Inputs can be signed or unsigned
- Sum of N parallel multiplications
- Optional accumulator
- Optional registers to enable higher frequency

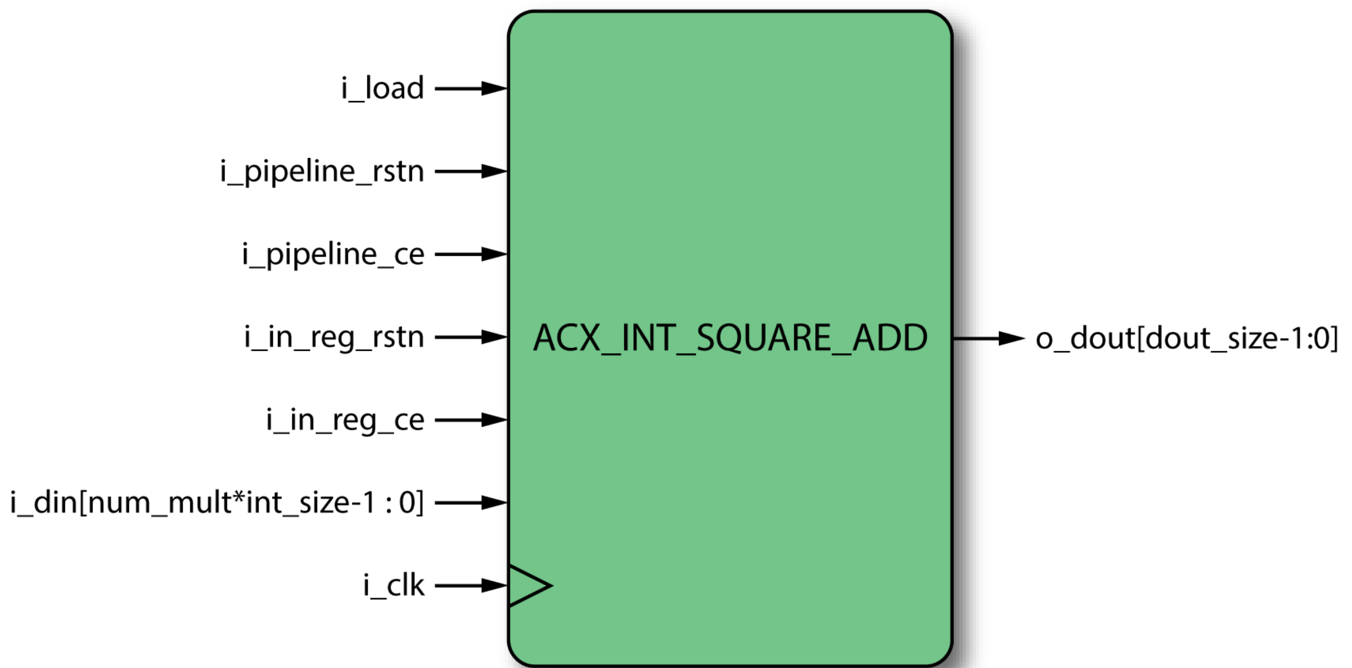


Figure 60: ***N Integer Sum of Squares with Optional Accumulation***

Parameters

Table 187: *ACX_INT_SQUARE_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	1–32	1	Number of parallel multiplications. Refer to Maximum Parallel Multiplications (see page 200) for the limits per number format.
int_unsigned	0, 1	0	0 – i_din is signed (two's complement). 1 – i_din is unsigned.
accumulate	0, 1	0	0 – No accumulation: dout = SUM(i_din(i) * i_din(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 is registered. The input registers are controlled by the i_in_reg_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 188: *ACX_INT_SQUARE_ADD Pin Descriptions*

Name	Direction	Description
`i_clk`	Input	Clock input, used for the (optional) registers and accumulator.
`i_din[(num_mult*int_size-1):0]`	Input	Packed (see page 200) vector of data input to multipliers.
`i_in_reg_ce`	Input	if `in_reg_enable=0` – ignored. if `in_reg_enable=1` – clock enable for `i_din`.
`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*i_din)`, ignoring the previous value. This signal is internally pipelined to have the same latency as `i_din`.
`o_dout[(dout_size-1):0]`	Output	Sum of squares, or result of accumulation.

Input Packing

Inputs are packed in a single input vector:

```
din(i) = i_din[i*int_size +: int_size];
```

Maximum Parallel Multiplications

Parameter `num_mult` specifies the number of parallel multiplications.

The ACX_MLP72 (see page 95) 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 closely-coupled ACX_BRAM72K (see page 248) site is used as route-through, meaning it is no longer available for BRAM placement.

The table below lists the maximum number of parallel multiplications for each format, for normal and wide modes. Wide mode is only used if `num_mult` is larger than the maximum for normal mode. For example, for `int_size`=8, if `num_mult` ≤ 8, the cost of ACX_INT_SQUARE_ADD is one ACX_MLP72, but if `num_mult` > 8 the cost is one ACX_MLP72 and one ACX_BRAM72K.

Table 189: *Maximum Number of Parallel Multiplications*

	Normal Input		Wide Input	
int_size	Max Signed Multiplications	Max Unsigned Multiplications	Max Signed Multiplications	Max Unsigned Multiplications
3	24	16	32	32
4	16	12	32	16
5	12	12	16	16
6	12	10	16	16
7	10	8	16	16
8	8	8	16	16
16	4	4	N/A	N/A

Usage and Inference

The ACX_INT_SQUARE_ADD module has the same functionality as ACX_INT_MULT_ADD. However, because each multiplication requires only one input, more parallel multiplications are enabled. While a sum of squares written in RTL, such as `x=a0*a0 + a1*a1` is recognized and inferred, it is treated as a normal sum of products without the benefit of a larger number of parallel multiplications.

In addition to direct instantiation in Verilog or VHDL, an instance of ACX_INT_SQUARE_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_SQUARE_ADD
ACX_INT_SQUARE_ADD #(
    .int_size      (int_size      ),
    .num_mult      (num_mult      ),
    .int_unsigned  (int_unsigned  ),
    .accumulate    (accumulate    ),
    .in_reg_enable (in_reg_enable ),
    .pipeline_regs (pipeline_regs ),
    .dout_size     (dout_size     ),
    .location      (location      )
) instance_name (
    .i_clk           (user_i_clk                          ),
    .i_din           (user_i_din[num_mult*int_size-1 : 0] ),
    .i_in_reg_ce     (user_i_in_reg_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_SQUARE_ADD
component ACX_INT_SQUARE_ADD is
generic (
    int_size             : integer := 8;
    num_mult             : integer := 1;
    int_unsigned         : 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                : in  std_logic_vector( num_mult*int_size-1 downto 0 );
    i_in_reg_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_SQUARE_ADD

-- VHDL Instantiation template for ACX_INT_SQUARE_ADD
instance_name : ACX_INT_SQUARE_ADD
generic map (
    int_size              => int_size,
    num_mult              => num_mult,
    int_unsigned          => int_unsigned,
    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                 => user_i_din,
    i_in_reg_ce           => user_i_in_reg_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_COMPLEX_MULT

The ACX_INT_COMPLEX_MULT module implements a complex multiplication with a single ACX_MLP72. This module features:

- Complex multiplication using 16-bit integers as coefficients
- Coefficients can be signed or unsigned
- Coefficients are internally duplicated, to reduce fabric routing
- Optional accumulator
- Optional registers to enable higher frequency operation

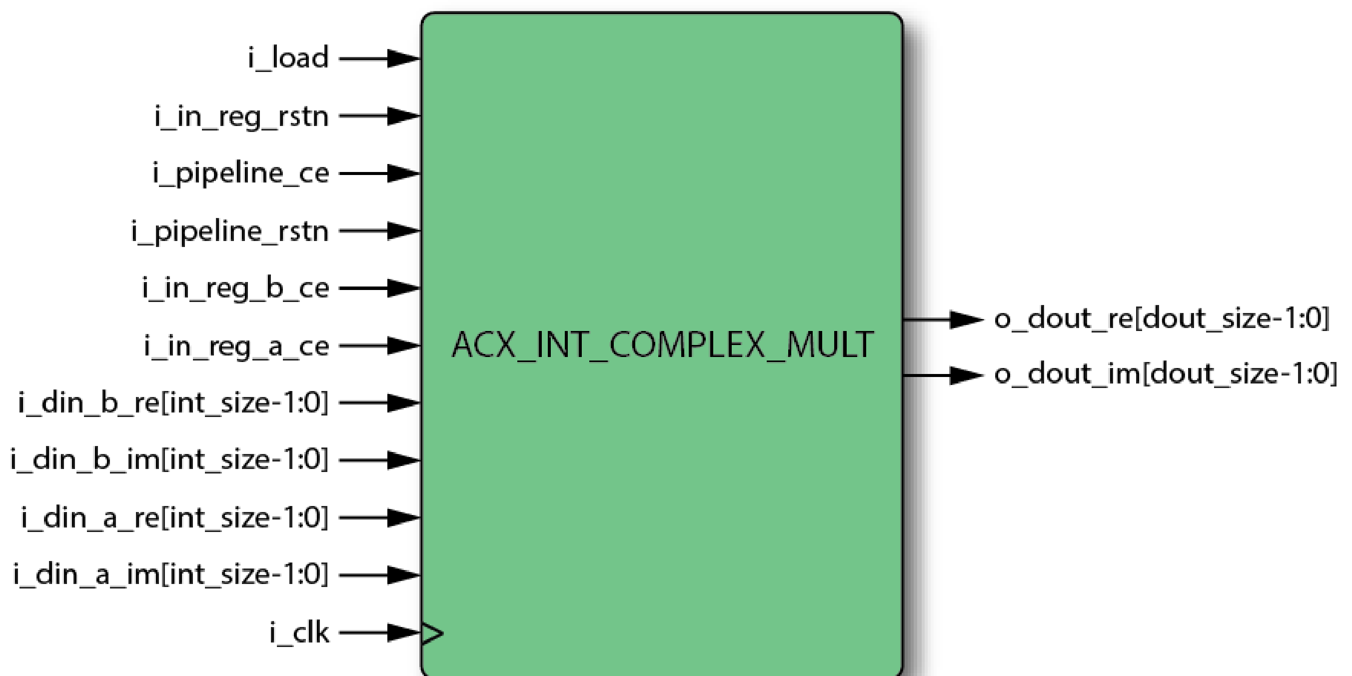


Figure 61: ***Complex Multiplier with Integer Coefficients and Optional Accumulate***

Parameters

Table 190: *ACX_INT_COMPLEX_MULT Parameters*

Parameter	Supported Values	Default	Description
`int_size`	16	16	Number of bits of each input coefficient.
`int_unsigned_a`	0, 1	0	0 – `i_din_a_re` and `i_din_a_im` are signed (two's complement). 1 – `i_din_a_re` and `i_din_a_im` are unsigned.
`int_unsigned_b`	0, 1	0	0 – `i_din_b_re` and `i_din_b_im` are signed (two's complement). 1 – `i_din_b_re` and `i_din_b_im` are unsigned.
`conjugate_b`	0, 1	0	Compute A × conjugate(B) instead of A × B.
`accumulate`	0, 1	0	0 – No accumulation: `(dout_re + dout_im·j) = (i_din_a_re + i_din_a_im·j) × (i_din_b_re + i_din_b_im·j)`. 1 – Accumulation: `dout_re + dout_im·j` 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_re`, `i_din_a_im`, `i_din_b_re`, and `i_din_b_im` 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`	≤36	–	Width of the `o_dout_re` and `o_dout_im` outputs. These output coefficients are always signed (2's complement) integers. Values that do not fit are truncated at the high-order bits.

Ports

Table 191: *ACX_INT_COMPLEX_MULT Pin Descriptions*

Name	Direction	Description
`i_clk`	Input	Clock input, used for the (optional) registers and accumulator.
`i_din_a_re[(int_size-1):0]`	Input	Real coefficient of A data input to multiplier.
`i_din_a_im[(int_size-1):0]`	Input	Imaginary coefficient of A data input to multiplier.
`i_din_b_re[(int_size-1):0]`	Input	Real coefficient of B data input to multiplier.
`i_din_b_im[(int_size-1):0]`	Input	Imaginary coefficient of 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_re` and `i_din_a_im`.
`i_in_reg_b_ce`	Input	If `in_reg_enable=0` – ignored. If `in_reg_enable=1` – clock enable for `i_din_b_re` and `i_din_b_im`.
`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 A × B, ignoring the previous value. This signal is internally pipelined to have the same latency as A × B.
`o_dout_re[(dout_size-1):0]`	Input	Real coefficient of the result of multiplication and accumulation. Always signed (2's complement).
`o_dout_im[(dout_size-1):0]`	Output	Imaginary coefficient of the result of multiplication and accumulation. Always signed (2's complement).

Usage and Inference

ACX_INT_COMPLEX_MULT is more efficient than simply writing the complex product in terms of the coefficients, because it maps the operation to a single ACX_MLP72, and each coefficient must be input only once.

When accumulation is enabled, the internal accumulators are 48 bits wide. However, each output coefficient is limited to the 36 least significant bits of the accumulator result.

Instantiation Templates

Verilog

```
// Verilog template for ACX_INT_COMPLEX_MULT
ACX_INT_COMPLEX_MULT #(
    .int_size        (int_size        ),
    .int_unsigned_a  (int_unsigned_a  ),
    .int_unsigned_b  (int_unsigned_b  ),
    .conjugate_b     (conjugate_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_re       (user_i_din_a_re       ),
    .i_din_a_im       (user_i_din_a_im       ),
    .i_din_b_re       (user_i_din_b_re       ),
    .i_din_b_im       (user_i_din_b_im       ),
    .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_re        (user_o_dout_re        ),
    .o_dout_im        (user_o_dout_im        )
);
```

VHDL

```
-- VHDL Component template for ACX_INT_COMPLEX_MULT
component ACX_INT_COMPLEX_MULT is
generic (
    int_size                : integer := 16;
    int_unsigned_a          : integer := 0;
    int_unsigned_b          : integer := 0;
    conjugate_b             : integer := 0;
    accumulate              : integer := 0;
    in_reg_enable           : integer := 0;
    pipeline_regs           : integer := 0;
    dout_size               : integer := 36;
 );
port (
    i_clk                   : in  std_logic;
```

```
    i_din_a_re          : in  std_logic_vector( int_size-1 downto 0 );
    i_din_a_im          : in  std_logic_vector( int_size-1 downto 0 );
    i_din_b_re          : in  std_logic_vector( int_size-1 downto 0 );
    i_din_b_im          : 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_re           : out std_logic_vector( dout_size-1 downto 0 );
    o_dout_im           : out std_logic_vector( dout_size-1 downto 0 )
);
end component ACX_INT_COMPLEX_MULT

-- VHDL Instantiation template for ACX_INT_COMPLEX_MULT
instance_name : ACX_INT_COMPLEX_MULT
generic map (
    int_size            => int_size,
    int_unsigned_a      => int_unsigned_a,
    int_unsigned_b      => int_unsigned_b,
    conjugate_b         => conjugate_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_re          => user_i_din_a_re,
    i_din_a_im          => user_i_din_a_im,
    i_din_b_re          => user_i_din_b_re,
    i_din_b_im          => user_i_din_b_im,
    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_re           => user_o_dout_re,
    o_dout_im           => user_o_dout_im
);
```

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.

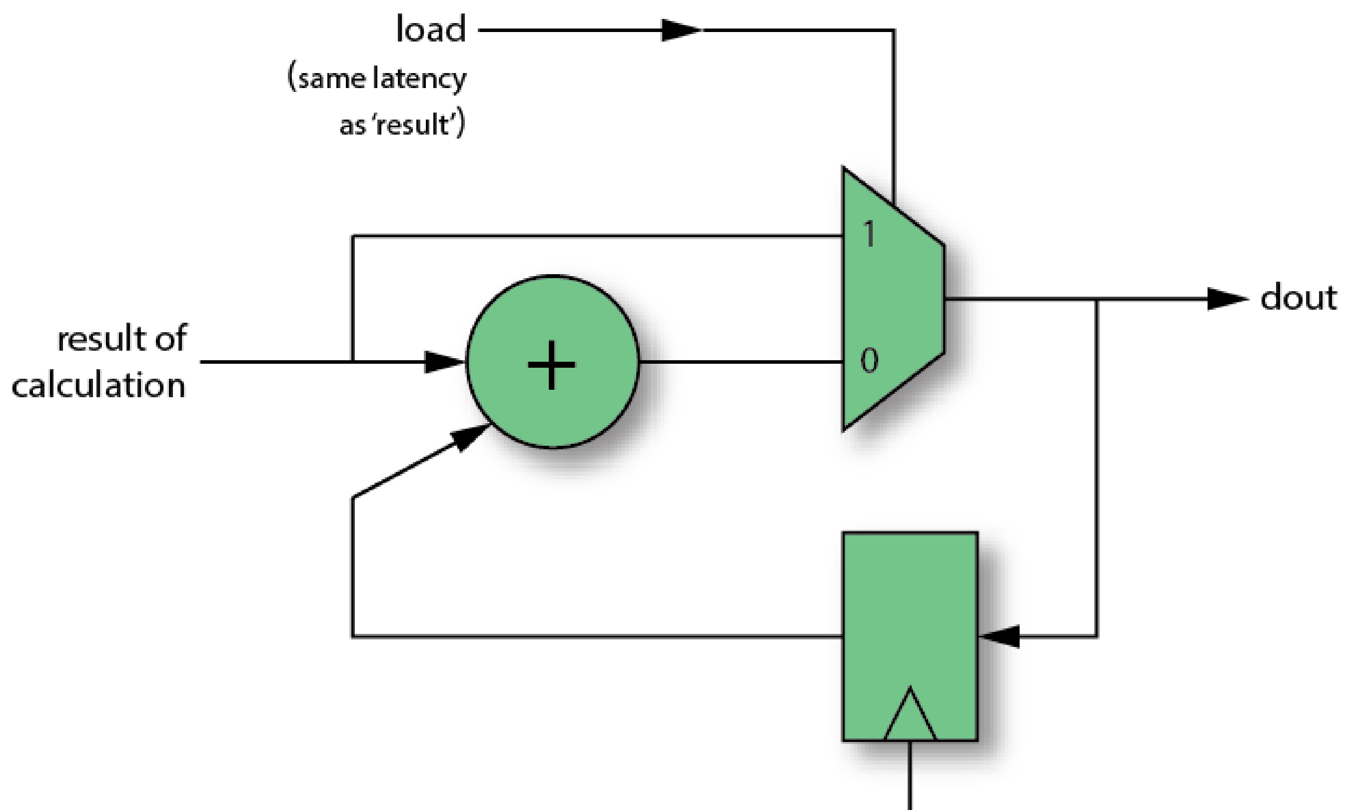


Figure 62: *Accumulator with Load Signal*

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 (see page 85) 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 192: *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.

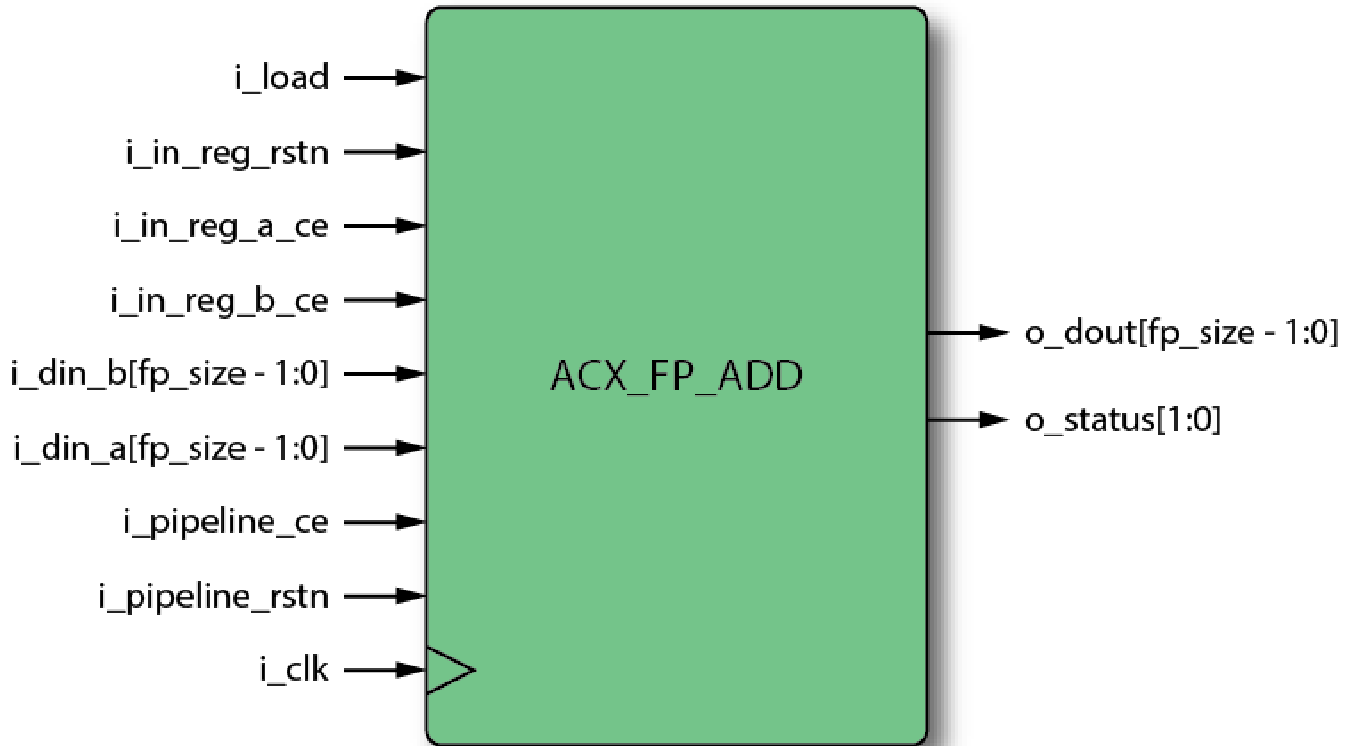


Figure 63: ***Floating-Point Adder with Optional Accumulate***

Parameters

Table 193: *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 194: ***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 (see page 211) 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.

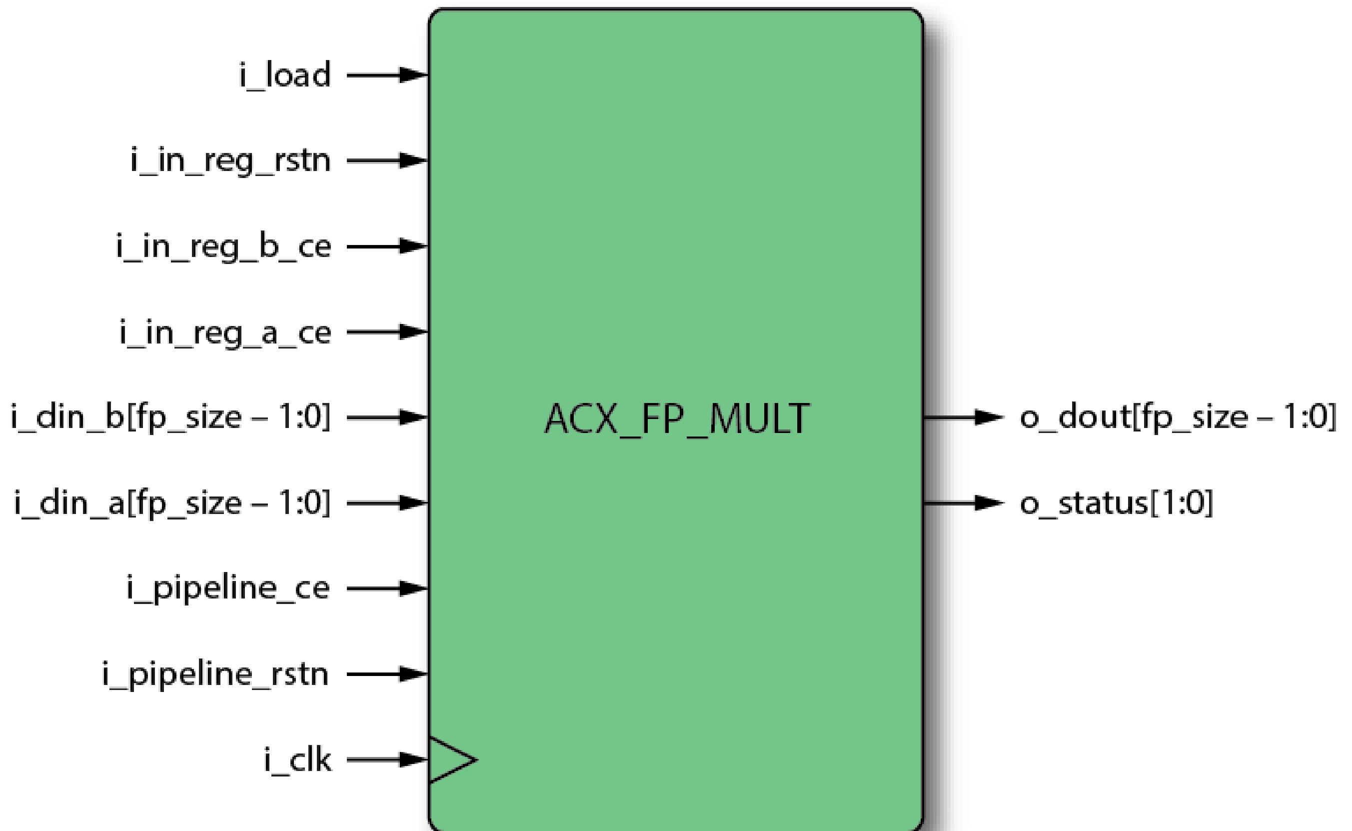


Figure 64: ***Floating-Point Multiplier with Optional Accumulate***

Parameters

Table 195: *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 196: *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 (see page 211) 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.

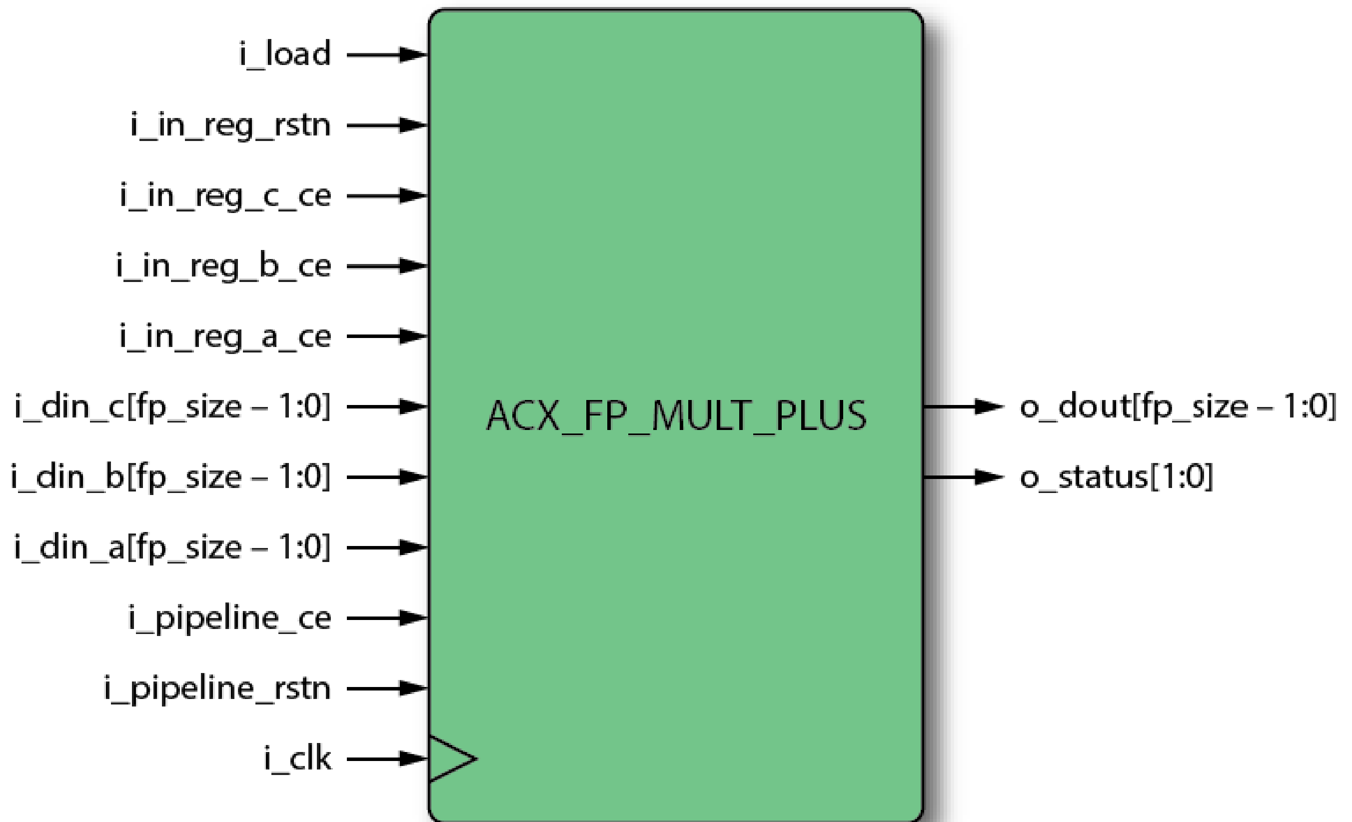


Figure 65: ***Floating-Point Multiplier Plus Adder with Optional Accumulate***

Parameters

Table 197: *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 198: ***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 (see page 211) 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 (see page 217), but uses a single ACX_MLP72 (see page 95) 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`

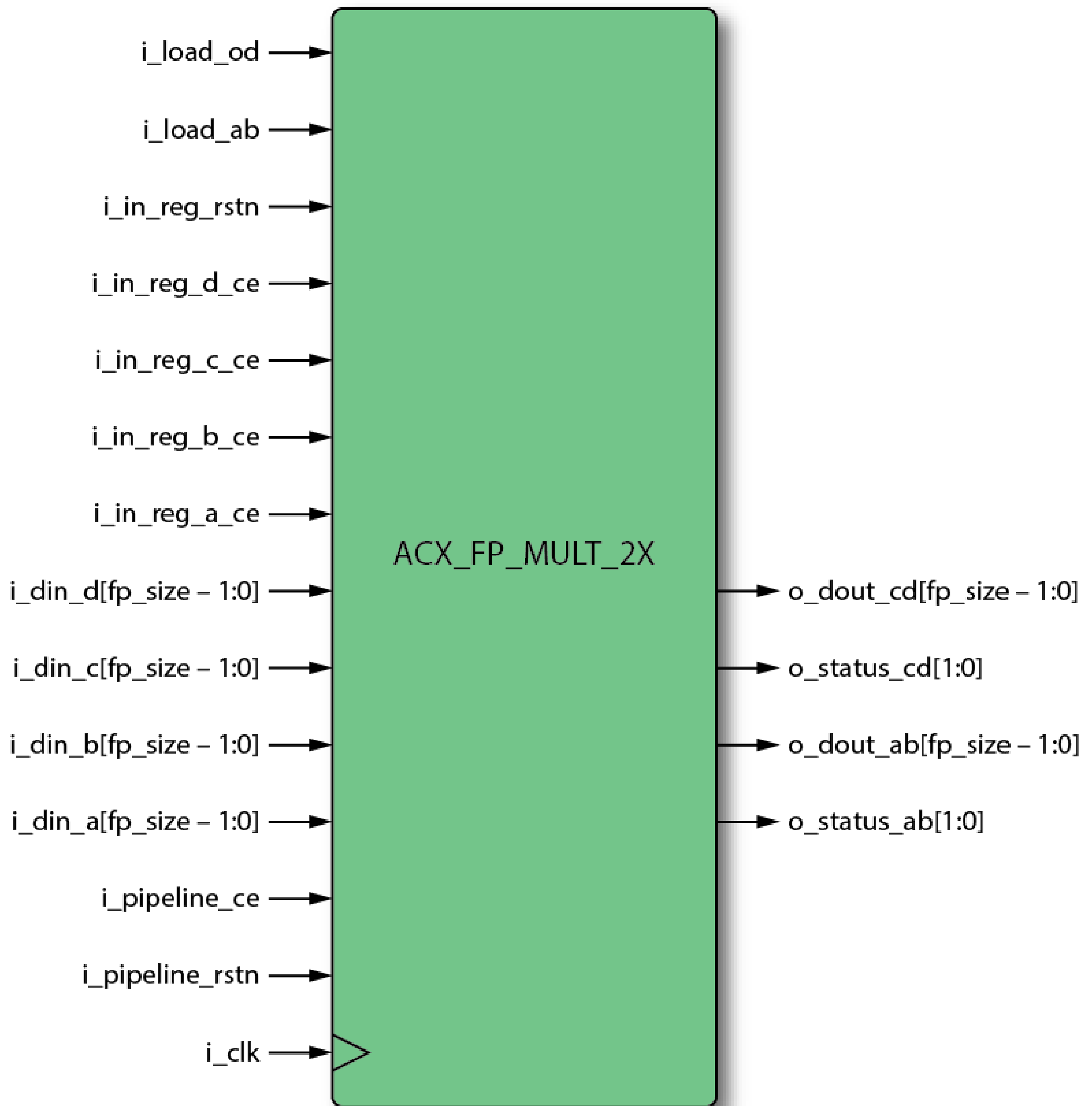


Figure 66: ***Twin Floating-Point Multipliers with Optional Accumulate***

Parameters

Table 199: *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 200: *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 (see page 211) 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 (see page 95) that performs the operation is used in wide input mode. In this mode, the adjacent ACX_BRAM72K (see page 248) 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.

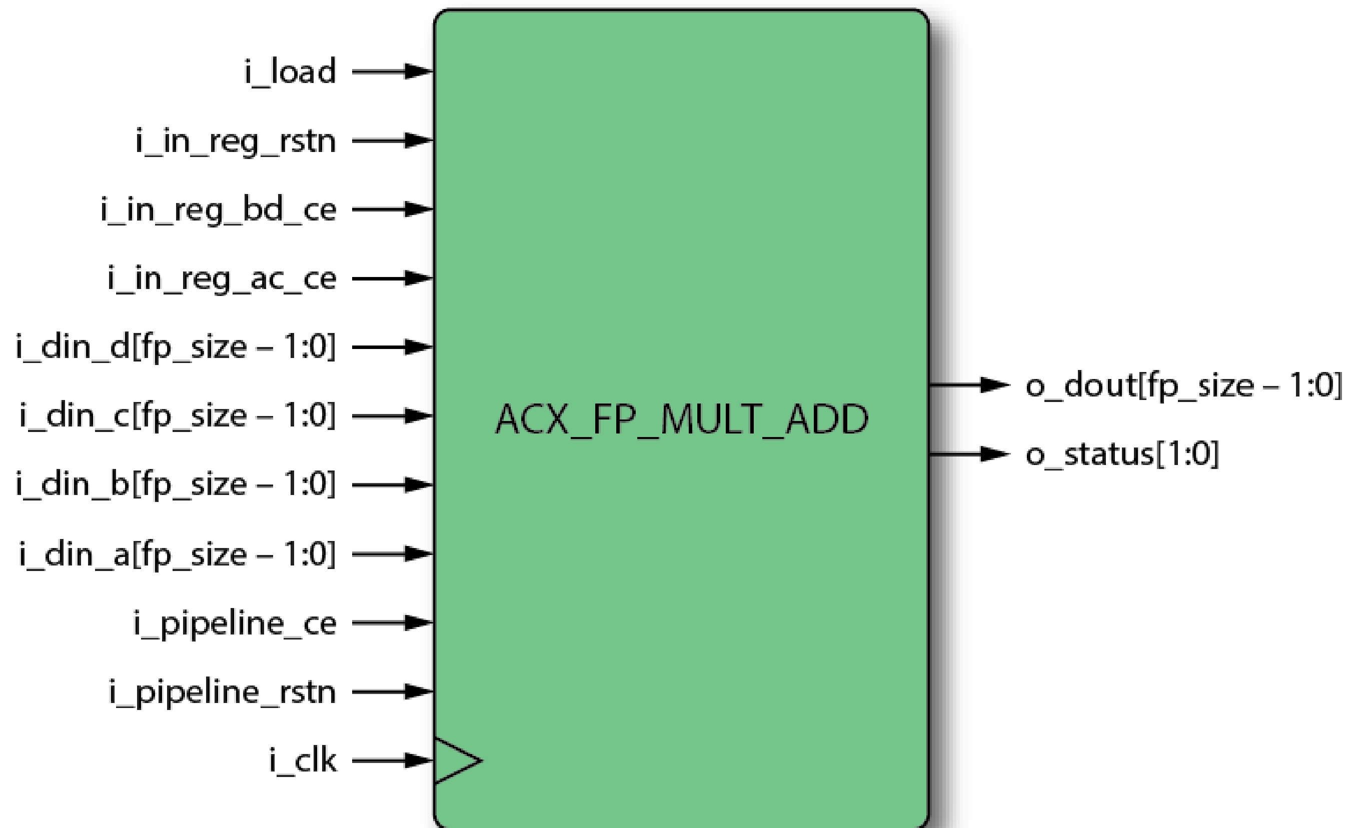


Figure 67: ***Twin Floating-Point Multiplies with Addition and optional Accumulation***

Parameters

Table 201: *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 202: ***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 (see page 211) 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 (see page 95) that performs the operation is used in wide input mode. In this mode, the adjacent ACX_BRAM72K (see page 248) 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_BRAM72K_FIFO

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.

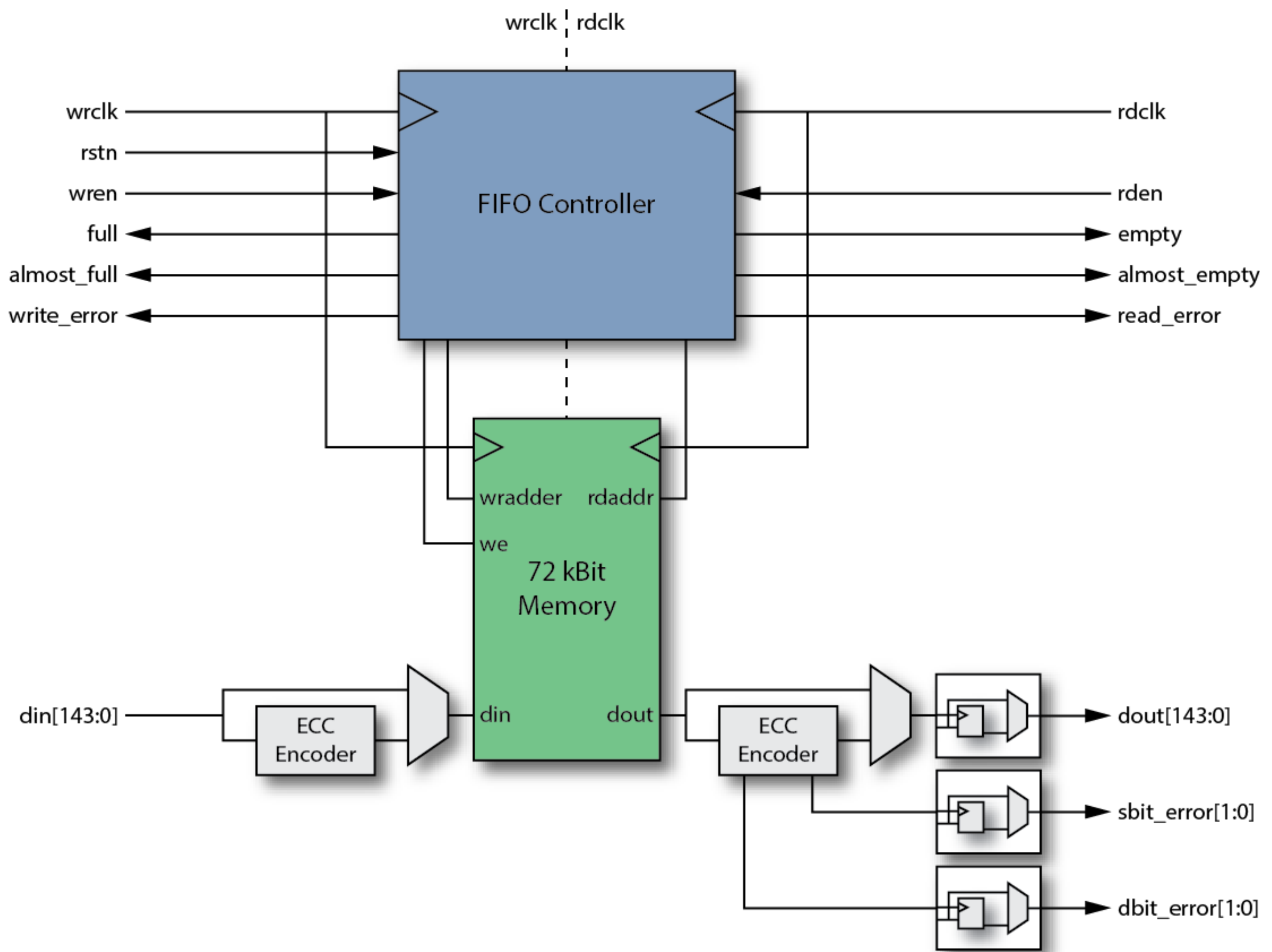


Figure 68: ***ACX_BRAM72K_FIFO Block Diagram***

Parameters

Table 203: *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`	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 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, 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`(2)	`0-14'h3FFF`	`14'h10`	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 `14'h0004` 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`(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 (or delayed by two `rdclk` cycles if `outreg_enable=1`) coincident with the de-assertion of `empty`. The data is held until `rden` is asserted.
`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 204: *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` – Full data width. `din=data_in`. `write_width=128` – `din={8'h0, data_in[127:64], 8'h0, data_in[63:0]}`. `write_width<128` – `data_in` should start from index 0 (right justified): `din[0]=data_in[0]`. 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]`	Output	Read port data output. When `read_width144`, the data output is aligned similarly to `din`.

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.

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` ia asserted and `empty` is deasserted. 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 205: ***ACX_BRAM72K_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 value.

Table Notes

Function assumes rising-edge clock and active-high port enable.

Table 206: ***ACX_BRAM72K_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

Operation [1]	`rdclk`	`outreg_rstn`	`outregce`	`dout`
Hold	↑	1	0	Previous `dout`.
Update Output	↑	1	1	Registered from latch output.

Table Notes

Function 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 waveform below (see page 245)

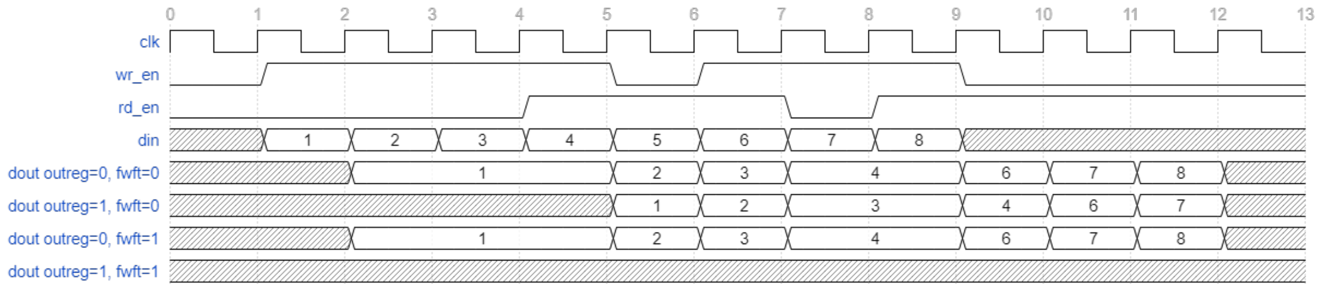


Figure 69: ***Output Timing with*** *`sync_mode = 1`*

Asynchronous Mode

Data output, `dout`, timing for all combinations of `outreg_enable` and `fwft_mode` is shown in the waveform below (see page 245)

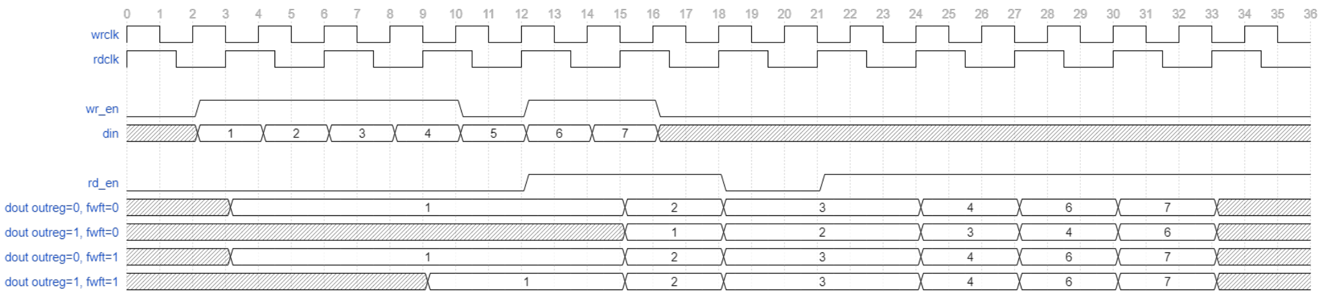


Figure 70: ***Output Timing with*** *`sync_mode = 0`*

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_BRAM72K_SDP

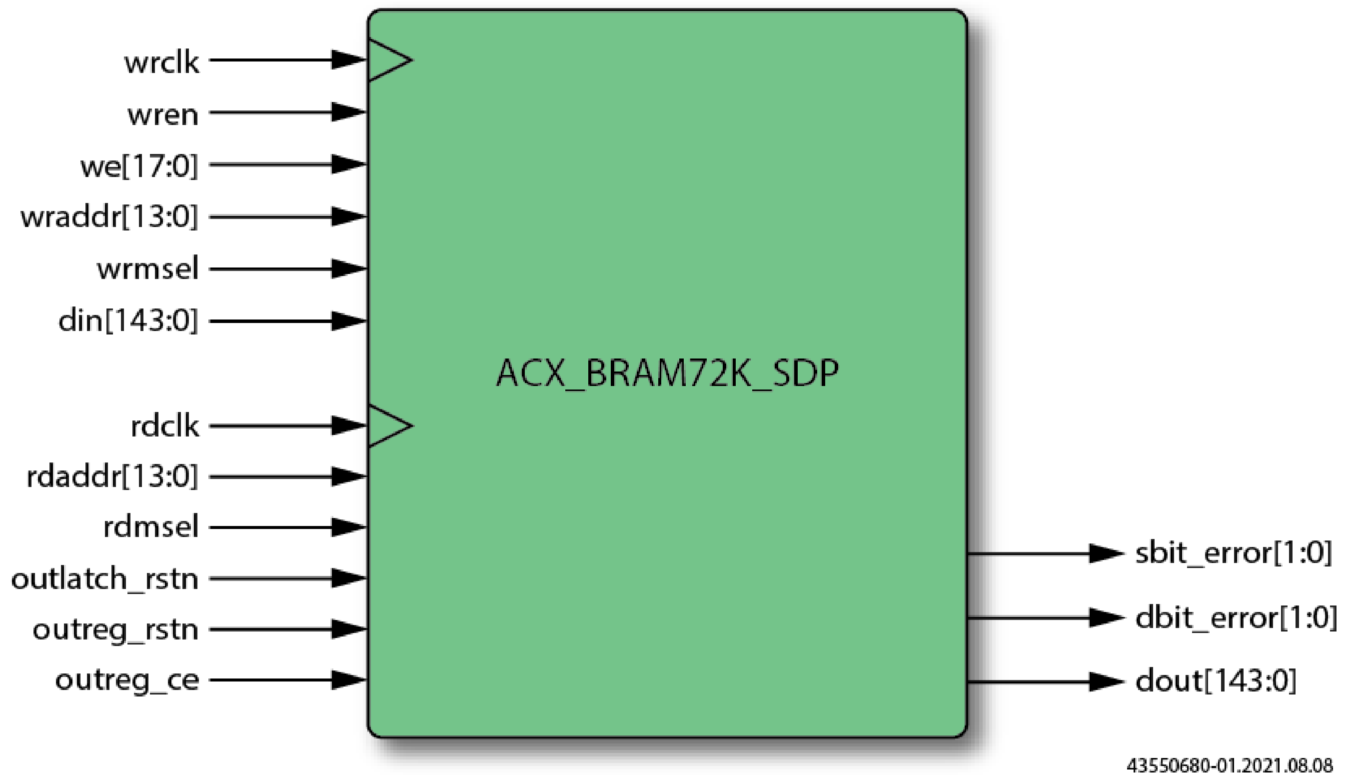


Figure 71: *ACX_BRAM72K_SDP Logic Symbol*

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.

A block diagram showing the data flow through the ECC modules, memories, and optional output registers is shown below.

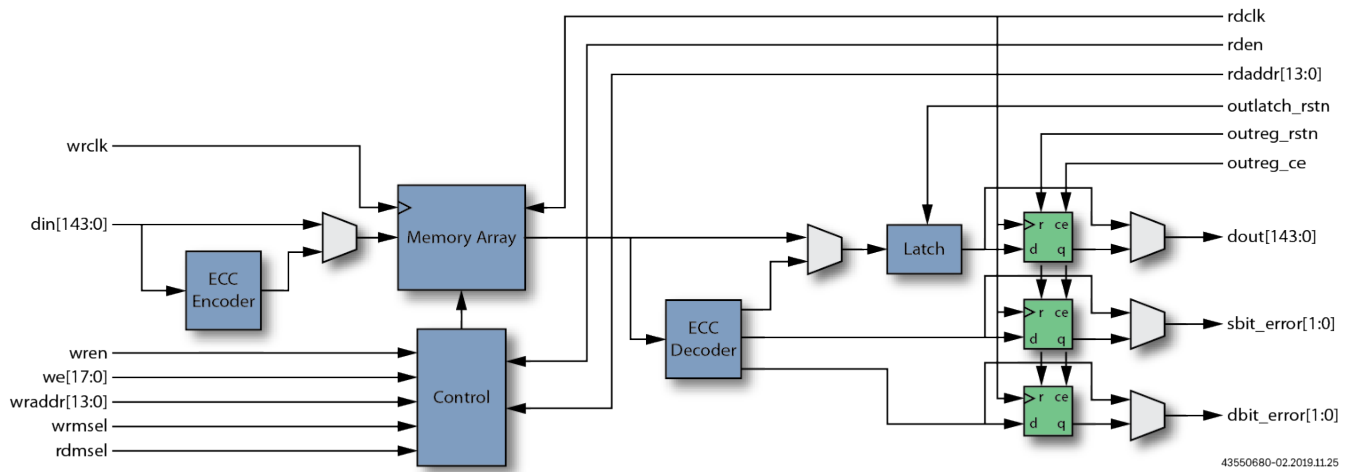


Figure 72: ***ACX_BRAM72K_SDP Block Diagram***

Parameters

Table 207: *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 261). • 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 261).
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. See Advanced Modes (see page 263) 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. See Advanced Modes (see page 263) 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 208: *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 write port address and data bus mapping tables below 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. See Advanced Modes (see page 263) 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 write port address and data bus mapping tables below 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 read port address and data bus mapping tables below 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. See Advanced Modes (see page 263) 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 read port address and data bus mapping tables below 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. Please see ECC Modes of Operation (see page 262) 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 209: ***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 210: *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 211: *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`

Read Port Configuration	Data Output Assignment	Read Word Address Assignment
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.

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 212: ***ACX_BRAM72K_SDP Output Function Table for Latched Mode***

Operation	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	`mem[rdaddr]`

Table Note

- Operation assumes rising-edge clock and active-high port enable.

Table 213: ***ACX_BRAM72K_SDP Output Function Table for Registered Mode***

Operation	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

Operation	rdclk	outreg_rstn	outregce	dout[]

Table Note

- Operation assumes active-high clock, output register clock enable, and output register reset.

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 timing diagrams for both values of the `outreg_enable` parameter are shown below. The first timing diagram illustrates the behavior of a ACX_BRAM72K_SDP instance with the output register disabled. The following describes the behavior of the ACX_BRAM72K_SDP on each clock cycle of the diagram, where each line represents a transaction that spans the clock cycles indicated:

Write clock

- 1 – no-op. `wren` is asserted but `we` is not asserted. Nothing is written to the memory array.
- 2, 3, 4 – write. `wren` and `we` are both asserted. Data on `din[]` is committed to the `wraddr[]` location in the memory array.

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.

Figure 73: ***ACX_BRAM72K_SDP Timing Diagram***

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 214: *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 215: *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 216: ***8-bit Byte Mode (byte_width == 8) when write_width is 72 or 144***

		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	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 217: ***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. Read operations place the 8-bit error correction syndrome on `dout [71:64]`. 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. Decode-only mode can be used as a building block to provide error correction for off-chip memories.

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 261)) 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, (see ACX_MLP72 LRAM).

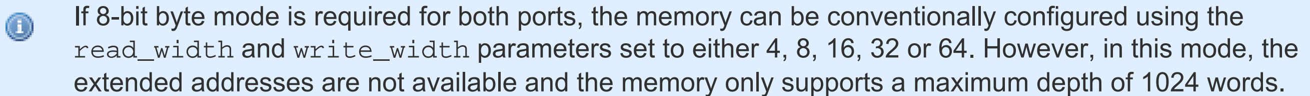
The advanced modes are enabled in the read and write sides by asserting the `wrmsel` and `rdmsel` inputs respectively. When asserted, `wrmsel` and `rdmsel` are combined with `wraddr[11]` and `rdaddr[11]` respectively to configure the write and read side advanced mode.

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

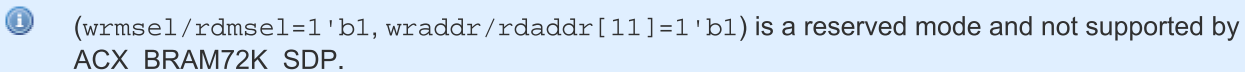


If 8-bit byte mode is required for both ports, the memory can be conventionally configured using the `read_width` and `write_width` parameters set to either 4, 8, 16, 32 or 64. However, in this mode, the extended addresses are not available and the memory only supports a maximum depth of 1024 words.

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_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.

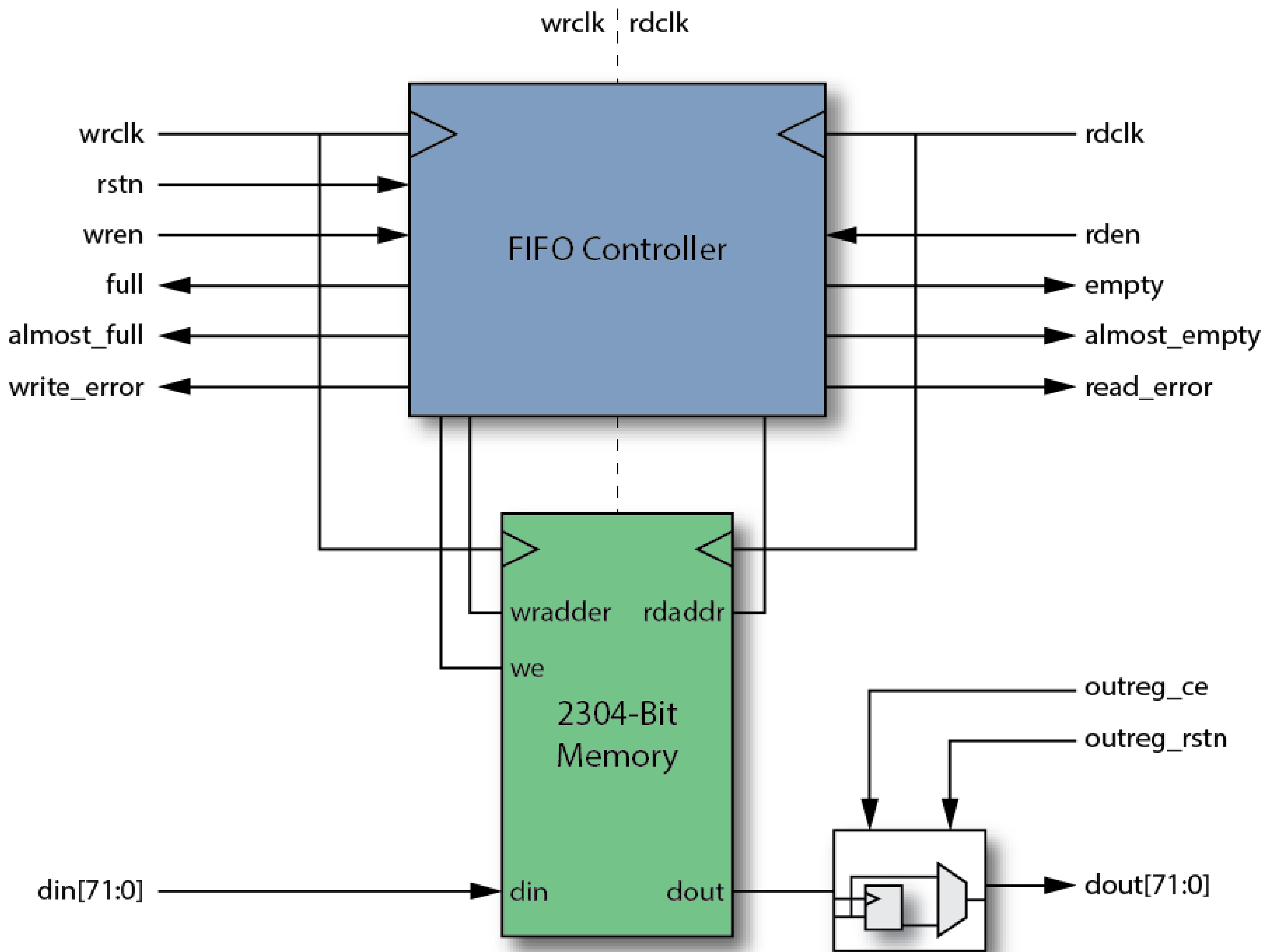


Figure 74: ***ACX_LRAM2K_FIFO Block Diagram***

Parameters

Table 218: *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 219: *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 220: ***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 221: ***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 waveform below (see page 277).

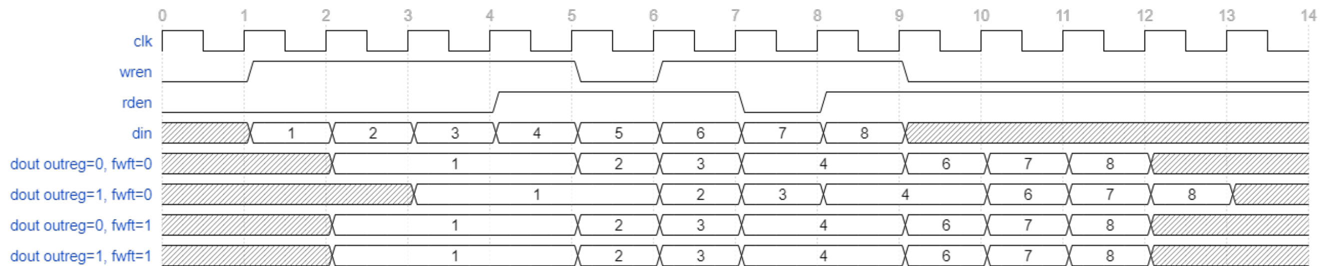


Figure 75: *Output Timing with `sync_mode = 1`*

Asynchronous Mode

Data output, `dout`, timing for all combinations of `outreg_enable` and `fwft_mode` is shown in the waveform below (see page 277).

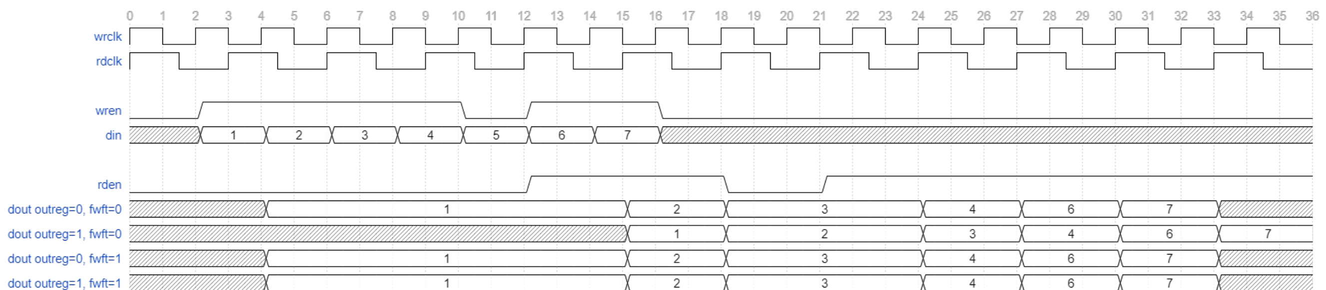


Figure 76: *Output Timing with `sync_mode = 0`*

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
    );
```

ACX_LRAM2K_SDP

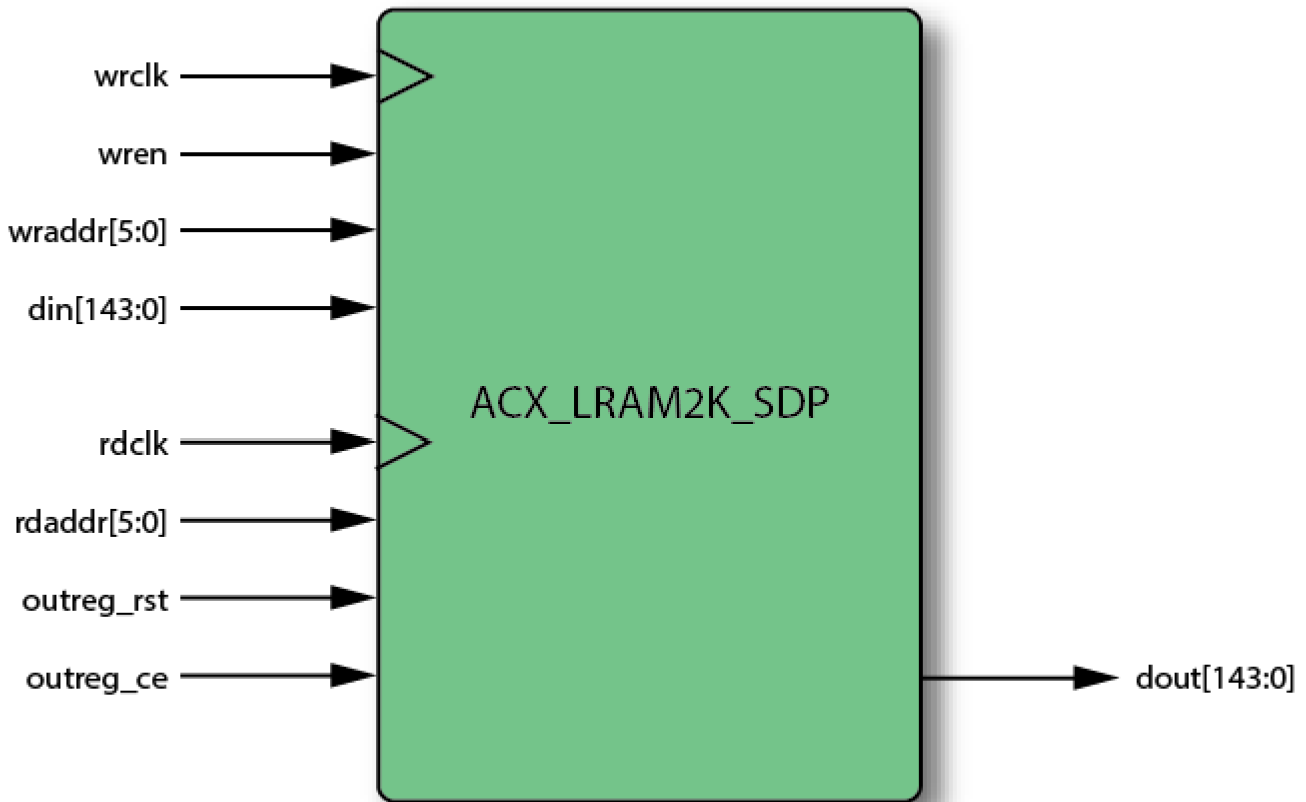


Figure 77: *ACX_LRAM2K_SDP Logic Symbol*

ACX_LRAM2K_SDP implements a 2Kb, simple dual-port (SDP) memory block with one write port and one read port. Each port can be configured as either a 16 × 144, 32 × 72 or 64 × 36 memory array. The write operation is synchronous, while the read operation is combinatorial. For higher performance operation, an additional output register can be enabled. Enabling the output register causes an additional cycle of read latency. There are no per-byte write enables, the entire word is written on each cycle that `wren` is asserted.

The initial value of the memory contents may be specified either with parameters or with a memory initialization file.

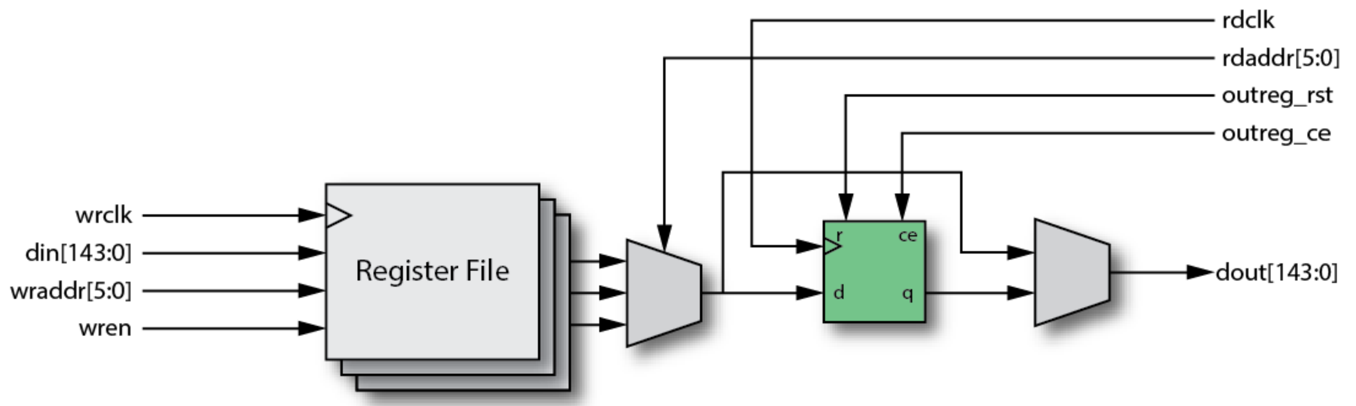


Figure 78: ***ACX_LRAM2K_SDP Block Diagram***

Parameters

Table 222: *ACX_LRAM2K_SDP Parameters*

Parameter	Supported Values	Default Value	Description
`read_width`	36, 72, 144	72	Read port data width.
`write_width`	36, 72, 144	72	Write port data width.
`wrclk_polarity`	"rise", "fall"	"rise"	Determines whether the `wrclk` signal uses the falling edge or the rising edge: "rise" – rising edge. "fall" – falling edge.
`rdclk_polarity`	"rise", "fall"	"rise"	Determines whether the `rdclk` signal uses the falling edge or the 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 zero cycles (i.e., combinatorial read, `dout[]` changes immediately corresponding to changes in `rdaddr[]`). 1 – enables the output register and results in a read latency of one cycle.
`clear_enable`	0, 1	0	Determines if the contents of the memory array are reset by `outreg_rstn`: 0 – memory contents are not changed when `outreg_rstn` is asserted. 1 – memory contents are reset to 0 when `outreg_rstn` is asserted in addition to any output register values also being reset to 0.
`outreg_sr_assertion`	"clocked", "unclocked"	"clocked"	Determines whether the assertion of the reset of the output register 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.
`mem_init_file`	Path to HEX file	""	Provides a mechanism to set the initial contents of the memory: • If the `mem_init_file` parameter is defined, the memory 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 285). • If the `mem_init_file` parameter is left at the default value of "", the initial contents are defined by the values of the `initd_0` through`initd_31`parameters. • If the memory initialization parameters and the `mem_init_file` parameters are not defined, the contents of the memory remain uninitialized.
`initd_0-initd_31`	72-bit hex number	X	The `initd_0` through `initd_31` parameters define the initial contents of the memory associated with `dout[71:0]` as defined in Memory Initialization (see page 285).

Ports

Table 223: *ACX_LRAM2K_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` 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.
`wraddr[5:0]`	Input	The `wraddr[]` signal determines which memory location is being written to. When `write_width` is 72 bits, the address must be top-justified, meaning that the low-order address bits must be 0.
`din[143:0]`	Input	The `din[]` signal determines the data to write to the memory array during a write operation.
`rdclk`	Input	Read clock input. When `outreg_enable=1`, read operations are fully synchronous and occur upon the active edge of the `rdclk` clock input. The active edge of `rdclk` is determined by the `rdclk_polarity` parameter.
`rdaddr[5:0]`	Input	The `rdaddr[]` signal determines which memory location is being read from. When `read_width` is 72 bits, the address must be top-justified, meaning that the low-order address bits must be 0.
`outreg_rstn`	Input	Output register and memory array reset. When `outreg_rstn` is asserted low, the value of the output registers is reset to 0. In addition, if `clear_enable` is set, the contents of the memory array are also reset to 0. The parameter `outreg_sr_assertion` controls whether this output register reset is synchronous or asynchronous. The memory array is always reset synchronous to `wrclk`.
`outreg_ce`	Input	Output register clock enable (active high). When `outreg_enable=1`, de-asserting `outreg_ce` causes the ACX_LRAM2K_SDP to hold the `dout[]` signal unchanged, independent of a read operation. When `outreg_enable=0`, the `outreg_ce` input is ignored.
`dout[143:0]`	Output	Read port data output: `outreg_enable=0` – the `dout[]` output is updated with the memory contents addressed by `rdaddr[]` immediately (asynchronously). `outreg_enable=1` – the `dout[]` output is updated with the memory contents addressed by `rdaddr[]` on the next active edge of `rdclk`.

Memory Organization and Data Input/Output Pin Assignments

The ACX_LRAM2K_SDP block supports up to a 144-bit wide memory, with each port arranged as a 16 × 144, 32 × 72 or 64 × 36 memory array.

Read and Write Operations

Timing Options

The ACX_LRAM2K_SDP has two options for interface timing, controlled by the `outreg_enable` parameter:

- Combinatorial mode – when `outreg_enable=0`, the port is in combinatorial mode. In combinatorial mode, the read address is used to select the data to be read, which is driven directly to the output pins, providing a read operation with zero cycles of latency.
- Registered mode – when `outreg_enable=1`, the port is in registered mode. In registered mode, there is an additional register supporting higher-frequency designs while having a read operation with one cycle of latency.

Read Operation

Read operations are signaled by driving the `rdaddr[]` signal with the address to be read. The requested read data arrives on the `dout[]` signal immediately, or on the following clock cycle, depending on the `outreg_enable` parameter value.

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 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 or if `outreg_en=0` and the memory is operating as a combinatoral output). If one of the ports is writing to an address while the other port is reading from the same address, the write operation occurs, and the read data is invalid. The data may be reliably read on the next cycle if there is no longer a write collision.

Note

Clearing the memory array (`clear_enable=1` and `outreg_rstn=0`) should be considered the same as a write for the purposes of simultaneous memory operations. As a result, a read of any memory location can return invalid values if it occurs within a single `wrclk` period from the end of the clear operation.

Memory Initialization

Initializing with Parameters

The initial memory contents may be defined by setting the 72-bit parameters `initd_0` through `initd_31`. The data memory is organized as little-endian where bit 0 of the memory corresponds to bit zero of parameter `initd_0` and bit 2303 of the memory corresponds to bit 71 of parameter `initd_31`.

Initializing with Memory Initialization File

An ACX_LRAM2K_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_LRAM2K_SDP memory organization is always configured in 9-bit byte mode (equivalent to the ACX_BRAM72K_SDP when `byte_width == 9`). For all modes, ×144, ×72 and ×36, the `mem_init_file` contains 32 lines with 72-bits of initialization data per line, organized as detailed in the following table.

Table 224: *Initialization File Organization*

		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
...	...	...	...	...	...	...	...	...	...
32nd line	`initd_31`	9byte255	9byte254	9byte253	9byte252	9byte251	9byte250	9byte249	9byte248

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 the parameter `mem_init_file` is defined, the memory is initialized with the values in the file referenced by the parameter. If the `mem_init_file` parameter is left at the default value of "", the initial contents are defined by the values of the `initd_0` through `initd_31` parameters. If neither the memory initialization parameters nor the `mem_init_file` parameters are defined, the contents of the memory remain uninitialized, and the contents are unknown until the memory locations have been written to.

Using ACX_LRAM2K_SDP as a Read-Only Memory (ROM)

The ACX_LRAM2K_SDP 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 285)) and never asserting the `wren` signal. All signals on the read side of the ACX_LRAM2K_SDP operate as described above. This configuration allows reading from the memory, but not writing to it.

Inference

The ACX_LRAM2K_SDP is inferrable using RTL constructs commonly used to infer synchronous and combinatorial RAMs and ROMs with a variety of clock enable and reset schemes and polarities.

```
//---------------------------------------------------------------------------------
//
// 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 Templates

Verilog

```
ACX_LRAM2K_SDP#(
  .wrclk_polarity            (   "rise"),
  .rdclk_polarity            (   "rise"),
  .wren_polarity             (        1),
  .outreg_enable             (        1),
  .outreg_sr_asssertion      ("clocked"),
  .mem_init_file             (       ""),
  .initd_0                   (        0),
 <...>
  .initd_31                  (        0)
) instance_name (
  .wraddr                    (user_wraddr      ),
  .din                       (user_din         ),
  .wren                      (user_wren        ),
  .outreg_rstn               (user_outreg_rstn ),
  .outregce                  (user_outregce    ),
  .wrclk                     (user_wrclk       ),
  .dout                      (user_dout        ),
  .rdaddr                    (user_rdaddr      ),
  .rdclk                     (user_rdclk       )
);
```

VHDL

```
-- VHDL Component template for ACX_LRAM2K_SDP
component ACX_LRAM2K_SDP is
generic (
    clear_enable             : integer := 0;
    initd_0                  : std_logic_vector( 143 downto 0 ) := X"x";
    <...>
    initd_31                 : std_logic_vector( 143 downto 0 ) := X"x";
    mem_init_file            : string := "";
    outreg_enable            : integer := 0;
    outreg_sr_assertion      : string := "clocked";
    rdclk_polarity           : string := "rise";
    read_width               : integer := 72;
    wrclk_polarity           : string := "rise";
    write_width              : integer := 72
);
port (
    rdaddr                   : in  std_logic_vector( 5 downto 0 );
    outreg_ce                : in  std_logic;
    outreg_rstn              : in  std_logic;
    rdclk                    : in  std_logic;
    dout                     : out std_logic_vector( 143 downto 0 );
    wraddr                   : in  std_logic_vector( 5 downto 0 );
    din                      : in  std_logic_vector( 143 downto 0 );
    wren                     : in  std_logic;
    wrclk                    : in  std_logic
);
end component ACX_LRAM2K_SDP
```

```
-- VHDL Instantiation template for ACX_LRAM2K_SDP
instance_name : ACX_LRAM2K_SDP
generic map (
    clear_enable            => clear_enable,
    initd_0                 => initd_0,
    <...>
    initd_31                => initd_31,
    mem_init_file           => mem_init_file,
    outreg_enable           => outreg_enable,
    outreg_sr_assertion     => outreg_sr_assertion,
    rdclk_polarity          => rdclk_polarity,
    read_width              => read_width,
    wrclk_polarity          => wrclk_polarity,
    write_width             => write_width
)
port map (
    rdaddr                  => user_rdaddr,
    outreg_ce               => user_outreg_ce,
    outreg_rstn             => user_outreg_rstn,
    rdclk                   => user_rdclk,
    dout                    => user_dout,
    wraddr                  => user_wraddr,
    din                     => user_din,
    wren                    => user_wren,
    wrclk                   => user_wrclk
);
```

Chapter - 7: Network-on-Chip (NOC) Primitives

ACX_NAP_AXI_MASTER

The ACX_NAP_AXI_MASTER component presents a 256-bit AXI master to user logic hosted in the FPGA, providing the user logic with the ability to accept and respond to read and write commands from other masters in the system, such as FPGA logic and PCIe.

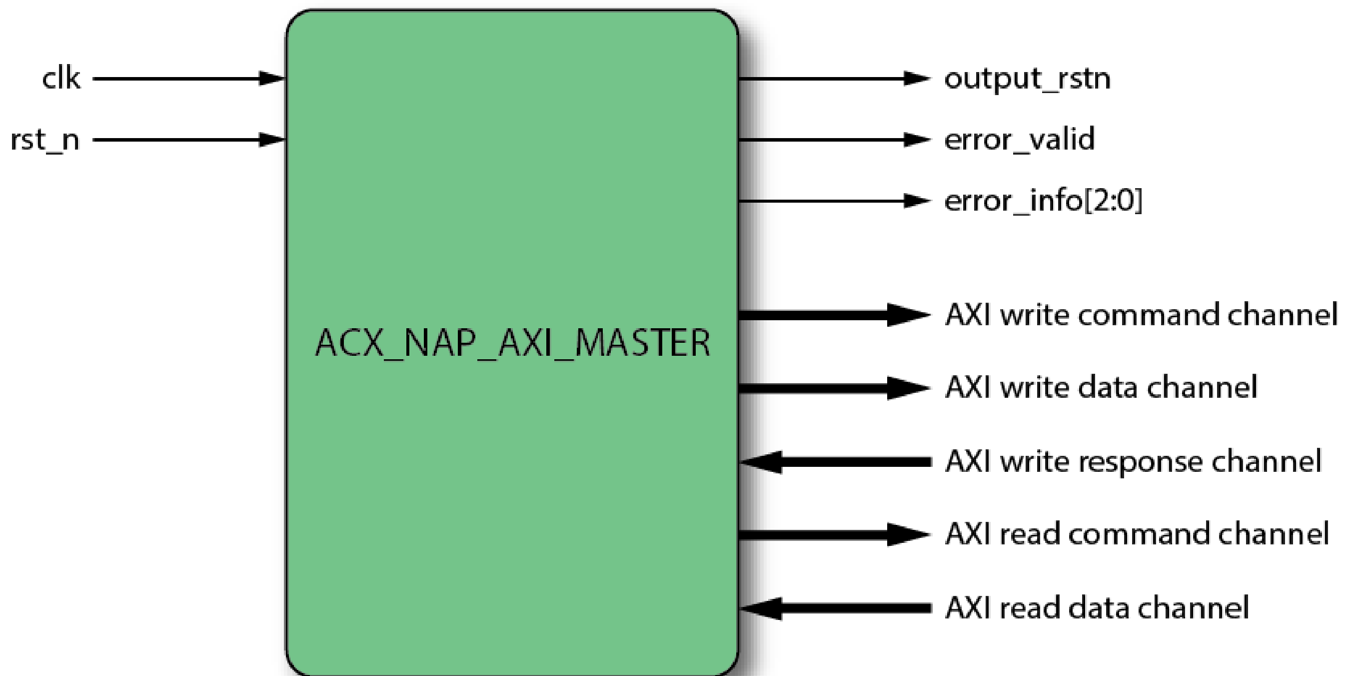


Figure 79: ***ACX_NAP_AXI_MASTER Logic Symbol***

Parameters

Table 225: *Parameters*

Parameter	Supported Values	Default Value	Description
n2s_arbitration_schedule s2n_arbitration_schedule	32'h0-32'hffff_fffe (1)	32'haaaa_aaaa	A 32-bit value used to initialize the arbitration schedule mechanism. Bit 0 of the arbitration schedule vector is used to determine if the local node wins arbitration when there is competing traffic from the upstream node. If bit 0 has a value of '1', the local traffic wins, while if bit 0 has a value of '0', the upstream node wins. After each cycle where both the local node and the upstream node are competing for access, the value in the schedule register rotates to the left. (2)(4) A value of 32'h8888_8888 means that the local node has high priority on every fourth cycle. A value of 32'haaaa_aaaa means that the local node has high priority on every second cycle.
row[3:0](3)(4)	4'd1-4'd8	4'hx	Fixes the NAP row location in the NoC.
col[3:0](3)(4)	4'd1-4'd10	4'hx	Fixes the NAP column location in the NoC.

Table Notes

1. A value of 32'hffff_ffff is not legal and is ignored. This would result in the upstream node never being serviced.
2. Although it is possible to directly assign the NAP arbitration using these parameters, it is recommended that the arbitration is specified in the relevant placement design constraints (.pdc) file. The arbitration value is dependent upon both the physical location of the NAP in the column and on the number and type of other NAPs on the same column. These physical attributes can all be managed and controlled within the single .pdc file.
3. Although it is possible to directly assign the NAP location using these parameters; it is recommended that locations are specified in the relevant placement design constraints (.pdc) file. Using the PDC to assign placements removes physical constraints from the functional RTL, thus allowing the same source code to be reused and placed in different locations on the die.
4. If these parameters are defined in both RTL and a .pdc file. The PDC assignments take priority.

Examples of setting the location and arbitration parameters are shown in Parameter Templates (see page 299)

Ports

Table 226: *Port Descriptions*

Name	Direction	Description
rstn	Input	Asynchronous reset input. This signal resets the NAP interface but does not affect the NoC.
output_rstn	Output	Reset output from the NAP to the fabric logic. **Do not use.** Intended for use with partial reconfiguration. Signal is controlled by a write to the configuration space. Currently, the signal is fixed to 1'b0.
clk	Input	All operations are fully synchronous and occur upon the active edge of the clk input.
awid[7:0]	Output	AXI write command channel.
awaddr[27:0]	Output	
awlen[7:0](1)	Output	
awsize[2:0]	Output	
awburst[1:0]	Output	
awlock	Output	
awqos[3:0]	Output	
awvalid	Output	
awready	input	
wdata[255:0]	Output	AXI write data channel.
wstrb[31:0]	Output	
wlast	Output	
wvalid	Output	
wready	Input	

Name	Direction	Description
bid[7:0]	Input	AXI write response channel.
bresp[1:0]	Input	
bvalid	Input	
bready	Output	
arid[7:0]	Output	AXI read command channel.
araddr[27:0]	Output	
arlen[7:0][1]	Output	
arsize[2:0]	Output	
arburst[1:0]	Output	
arlock	Output	
arqos[3:0]	Output	
arvalid	Output	
arready	Input	
rid[7:0]	Input	AXI read response channel.
rdata[255:0]	Input	
rresp[1:0]	Input	
rlast	Input	
rvalid	Input	
rready	Output	

Name	Direction	Description
`error_valid`	Output	Asserted high for one cycle to indicate that an error has been detected.
`error_info[2:0]`	Output	Code indicating cause of error. Validated by `error_valid == 1'b1`: `3'b000` – received transaction type does not match node configuration. `3'b001` – received transaction destination ID does not match node ID. Others – reserved.

Table Notes

1. Although the AXI-4 specification supports burst lengths of 256 beats, the NoC only supports transfers of up to 16 beats. Therefore, the user logic only needs to support `awlen` and `arlen` values up to 15.

AXI-4 Specification

The ACX_NAP_AXI_MASTER interface is fully compliant with the AXI-4 specification, *AMBA AXI and ACE Protocol Specification* (IHI0022). Familiarity with the AXI-4 protocol is recommended in order to make effective use of the ACX_NAP_AXI_MASTER.

Inference

It is not possible to infer the ACX_NAP_AXI_MASTER. It must be directly instantiated.

Instantiation Templates

Verilog

```
ACX_NAP_AXI_MASTER #(
    .column                         (column),
    .n2s_arbitration_schedule       (n2s_arbitration_schedule),
    .row                            (row),
    .s2n_arbitration_schedule       (s2n_arbitration_schedule)
) instance_name (
    .clk                            (user_clk),
    .rstn                           (user_rstn),
    .output_rstn                    (user_output_rstn),
    .arready                        (user_arready),
    .arvalid                        (user_arvalid),
    .arqos                          (user_arqos),
    .arburst                        (user_arburst),
    .arlock                         (user_arlock),
    .arsize                         (user_arsize),
    .arlen                          (user_arlen),
    .arid                           (user_arid),
    .araddr                         (user_araddr),
    .awready                        (user_awready),
    .awvalid                        (user_awvalid),
    .awqos                          (user_awqos),
    .awburst                        (user_awburst),
    .awlock                         (user_awlock),
    .awsize                         (user_awsize),
    .awlen                          (user_awlen),
    .awid                           (user_awid),
    .awaddr                         (user_awaddr),
    .wready                         (user_wready),
    .wvalid                         (user_wvalid),
    .wlast                          (user_wlast),
    .wstrb                          (user_wstrb),
    .wdata                          (user_wdata),
    .rready                         (user_rready),
    .rvalid                         (user_rvalid),
    .rresp                          (user_rresp),
    .rid                            (user_rid),
    .rlast                          (user_rlast),
    .rdata                          (user_rdata),
    .bready                         (user_bready),
    .bvalid                         (user_bvalid),
    .bid                            (user_bid),
    .bresp                          (user_bresp),
    .error_valid                    (user_error_valid),
    .error_info                     (user_error_info)
);
```

VHDL

```
component ACX_NAP_AXI_MASTER is
generic (
    column                          : integer := X"x";
```

```
    n2s_arbitration_schedule         : integer := X"AAAAAAAA";
    row                              : integer := X"x";
    s2n_arbitration_schedule         : integer := X"AAAAAAAA"
);
port (
    clk                              : in  std_logic;
    rstn                             : in  std_logic;
    output_rstn                      : out std_logic;
    arready                          : in  std_logic;
    arvalid                          : out std_logic;
    arqos                            : out std_logic_vector( 3 downto 0 );
    arburst                          : out std_logic_vector( 1 downto 0 );
    arlock                           : out std_logic;
    arsize                           : out std_logic_vector( 2 downto 0 );
    arlen                            : out std_logic_vector( 7 downto 0 );
    arid                             : out std_logic_vector( 7 downto 0 );
    araddr                           : out std_logic_vector( 31 downto 0 );
    awready                          : in  std_logic;
    awvalid                          : out std_logic;
    awqos                            : out std_logic_vector( 3 downto 0 );
    awburst                          : out std_logic_vector( 1 downto 0 );
    awlock                           : out std_logic;
    awsize                           : out std_logic_vector( 2 downto 0 );
    awlen                            : out std_logic_vector( 7 downto 0 );
    awid                             : out std_logic_vector( 7 downto 0 );
    awaddr                           : out std_logic_vector( 31 downto 0 );
    wready                           : in  std_logic;
    wvalid                           : out std_logic;
    wlast                            : out std_logic;
    wstrb                            : out std_logic_vector( 31 downto 0 );
    wdata                            : out std_logic_vector( 255 downto 0 );
    rready                           : out std_logic;
    rvalid                           : in  std_logic;
    rresp                            : in  std_logic_vector( 1 downto 0 );
    rid                              : in  std_logic_vector( 7 downto 0 );
    rlast                            : in  std_logic;
    rdata                            : in  std_logic_vector( 255 downto 0 );
    bready                           : out std_logic;
    bvalid                           : in  std_logic;
    bid                              : in  std_logic_vector( 7 downto 0 );
    bresp                            : in  std_logic_vector( 1 downto 0 );
    error_valid                      : out std_logic;
    error_info                       : out std_logic_vector( 2 downto 0 )
);
end component ACX_NAP_AXI_MASTER

-- VHDL Instantiation template for ACX_NAP_AXI_MASTER
instance_name : ACX_NAP_AXI_MASTER
generic map (
    column                           => column,
    n2s_arbitration_schedule         => n2s_arbitration_schedule,
    row                              => row,
    s2n_arbitration_schedule         => s2n_arbitration_schedule
)
port map (
    clk                              => user_clk,
    rstn                             => user_rstn,
    output_rstn                      => user_output_rstn,
    arready                          => user_arready,
```

```
    arvalid                          => user_arvalid,
    arqos                            => user_arqos,
    arburst                          => user_arburst,
    arlock                           => user_arlock,
    arsize                           => user_arsize,
    arlen                            => user_arlen,
    arid                             => user_arid,
    araddr                           => user_araddr,
    awready                          => user_awready,
    awvalid                          => user_awvalid,
    awqos                            => user_awqos,
    awburst                          => user_awburst,
    awlock                           => user_awlock,
    awsize                           => user_awsize,
    awlen                            => user_awlen,
    awid                             => user_awid,
    awaddr                           => user_awaddr,
    wready                           => user_wready,
    wvalid                           => user_wvalid,
    wlast                            => user_wlast,
    wstrb                            => user_wstrb,
    wdata                            => user_wdata,
    rready                           => user_rready,
    rvalid                           => user_rvalid,
    rresp                            => user_rresp,
    rid                              => user_rid,
    rlast                            => user_rlast,
    rdata                            => user_rdata,
    bready                           => user_bready,
    bvalid                           => user_bvalid,
    bid                              => user_bid,
    bresp                            => user_bresp,
    error_valid                      => user_error_valid,
    error_info                       => user_error_info
);
```

Parameter Templates

An example of a `.pdc` file setting both the location and arbitration parameters of the ACX_NAP_AXI_MASTER is shown below.

```
# Example NAP hierarchical name, (using nap_master_wrapper.sv), is my_block.i_nap_master_wrapper.
i_axi_master

# Fix location to column 1, row 2
# Note use of "i:" at start of instance name
set_placement -fixed i:my_block.i_nap_master_wrapper.i_axi_master s:x_core.NOC[1][2].logic.noc.
nap_m


# Set north to south arbitration to 0x1234_5678
# Note no "i:" at start of instance name
set_property n2s_arbitration_schedule 32'h1234_5678 my_block.i_nap_master_wrapper.i_axi_master

# Set south to north arbitration to 0x8888_8888
# Note no "i:" at start of instance name
set_property s2n_arbitration_schedule 32'h8888_8888 my_block.i_nap_master_wrapper.i_axi_master
```

ACX_NAP_AXI_SLAVE

The ACX_NAP_AXI_SLAVE component presents a 256-bit AXI slave to user logic hosted in the FPGA, providing the user logic with the ability to access all of the peripherals on the device. To achieve this functionality, the user design implements an AXI master that issues read and write commands

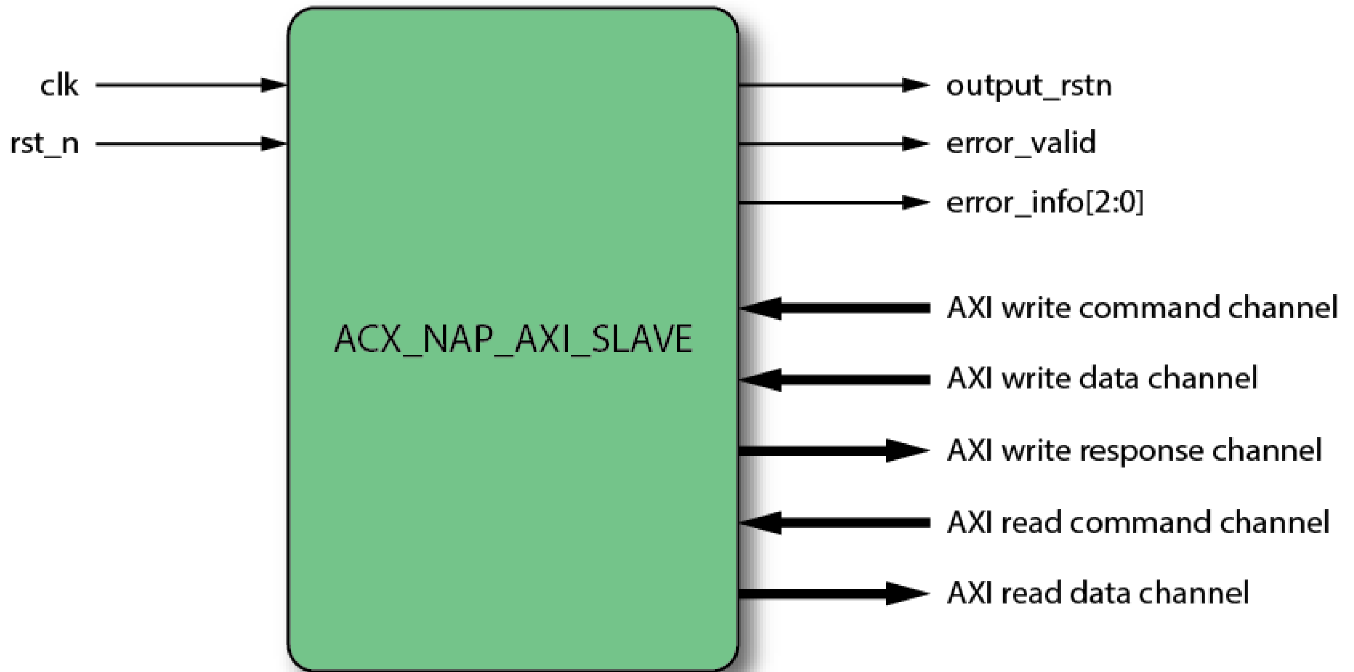


Figure 80: ***ACX_NAP_AXI_SLAVE Logic Symbol***

Parameters

Table 227: *Parameters*

Parameter	Supported Values	Default Value	Description
`e2w_arbitration_schedule` `w2e_arbitration_schedule`	`32'h–32'hffff_fffe`(1)	`32'haaaa_aaaa`	A 32-bit value used to initialize the arbitration schedule mechanism. Bit 0 of the arbitration schedule vector is used to determine if the local node wins arbitration when there is competing traffic from the upstream node. If bit 0 has a value of '1', the local traffic wins. While if bit 0 has a value of '0', the upstream node wins. After each cycle where both the local node and the upstream node are competing for access, the value in the schedule register rotates to the left. A value of `32'h8888_8888` means that the local node has priority on every fourth cycle. A value of `32'haaaa_aaaa` means that the local node has priority on every second cycle. (2)(4)
`csr_access_enable`	`1'b0`, `1'b1`	`1'b0`	Enables the NAP to access `CSR_SPACE` in the 2D NoC global address map. Refer to the *Speedster7t Network on Chip User Guide* (UG089) for details.

`att_ddr_0[6:0]`
...
`att_ddr_127[6:0]`
	`7'h0`
	...
	`7'h7f`
		`7'h0`
		...
		`7'h7f`
			DDR address translation table. Default value matches the entry number (`att_ddr_0=7'h0`, `att_ddr_1=7'h1`, etc.).

`att_gddr_0[6:0]`
...
`att_gddr_127[6:0]`
	`7'h0`
	...
	`7'h7f`
		`7'h0`
		...
		`7'h7f`
			GDDR address translation table. Default value matches the entry number (`att_gddr_0=7'h0`, `att_gddr_1=7'h1`, etc.).

`att_nap_0[6:0]`
...
`att_nap_79[6:0]`
	`7'h0`
	...
	`7'h4f`
		`7'h0`
		...
		`7'h4f`
			NAP address translation table. Default value matches the entry number (`att_nap_0=7'h0`, `att_nap_1=7'h1`, etc.).

`row[3:0]`	`4'd1-4'd10`	`4'hx`	Fixes the NAP row location in the NoC. (3)(4)
`col[3:0]`	`4'd1-4'd8`	`4'hx`	Fixes the NAP column location in the NoC (3)(4)

Table Notes

1. A value of `32'hffff_ffff` is not legal and is ignored. This would result in the upstream node never being serviced.
2. Although it is possible to directly assign the NAP arbitration using these parameters, it is recommended that the arbitration is specified in the relevant placement design constraints (`.pdc`) file. The arbitration value is dependent on both the physical location of the NAP in the row and on the number and type of other NAPs on the same row. These physical attributes can all be managed and controlled within the single `.pdc` file.
3. Although it is possible to directly assign the NAP location using these parameters, it is recommended that locations are specified in the relevant placement design constraints (`.pdc`) file. Using the PDC to assign placements removes physical constraints from the functional RTL, thus allowing the same source code to be reused and placed in different locations on the die.
4. If these parameters are defined in both RTL, and a `.pdc` file. The PDC assignments take priority.

Examples of setting the location and arbitration parameters are shown in Parameter Templates (see page 308).

Ports

Table 228: *Port Descriptions*

Name	Direction	Description
`rstn`	Input	Asynchronous reset input. This signal resets the NAP interface but does not affect the NoC. Any transactions for the NAP that are already in flight when `rstn` is asserted are held in the NoC buffers until the NAP is released from reset. It is strongly recommended to only assert `rstn` when it is known that there are no outstanding transactions in flight to the NAP.
`output_rstn`	Output	Reset output from the NAP to the fabric logic. **Do not use**. Intended for use with partial reconfiguration. Signal controlled by a write to the configuration space. Currently, the signal is fixed to `1'b0`.
`clk`	Input	All operations are fully synchronous and occur upon the active edge of the clk input.
`awid[7:0]`	Input	AXI write command channel.
`awaddr[41:0]`	Input	
`awlen[7:0]`(1)	Input	
`awsize[2:0]`	Input	
`awburst[1:0]`	Input	
`awlock`	Input	
`awqos[3:0]`	Input	
`awvalid`	Input	
`awready`	Output	
`wdata[255:0]`	Input	AXI write data channel.
`wstrb[31:0]`	Input	
`wlast`	Input	
`wvalid`	Input	
`wready`	Output	

Name	Direction	Description
bid[7:0]	Output	AXI write response channel.
bresp[1:0]	Output	
bvalid	Output	
bready	Input	
arid[7:0]	Input	AXI read command channel.
araddr[41:0]	Input	
arlen[7:0][1]	Input	
arsize[2:0]	Input	
arburst[1:0]	Input	
arlock	Input	
arqos[3:0]	Input	
arvalid	Input	
arready	Output	
rid[7:0]	Output	AXI read response channel.
rdata[255:0]	Output	
rresp[1:0]	Output	
rlast	Output	
rvalid	Output	
rready	Input	
error_valid	Output	Asserted high for one cycle to indicate that an error has been detected.
error_info[2:0]	Output	Code indicating the cause of error. Validated by error_valid == 1'b1: 3'b000 – received transaction type does not match node configuration. 3'b001 – received transaction destination ID does not match node ID. 3'b010 – transaction timeout. Others – reserved.

Name	Direction	Description

Table Notes

1. Although the AXI-4 specification supports transfers of up to 256 beats, the NoC only supports AXI-4 transfers of 16 beats. Therefore, `awlen` and `arlen` may not be set to a value greater than 15.

AXI-4 Specification

The ACX_NAP_AXI_SLAVE interface is fully compliant with the AXI-4 specification, *AMBA AXI and ACE Protocol Specification* (IHI0022). Familiarity with the AXI-4 protocol is recommended in order to make effective use of the ACX_NAP_AXI_SLAVE.

Inference

It is not possible to infer the ACX_NAP_AXI_SLAVE. It must be directly instantiated.

Instantiation Templates

Verilog

```
ACX_NAP_AXI_SLAVE #(
    .att_ddr_0                  (att_ddr_0),
    ....
    .att_ddr_127                (att_ddr_127),
    .att_gddr_0                 (att_gddr_0),
    ....
    .att_gddr_127               (att_gddr_127),
    .att_nap_0                  (att_nap_0),
    ....
    .att_nap_79                 (att_nap_79),
    .column                     (column),
    .e2w_arbitration_schedule   (e2w_arbitration_schedule),
    .row                        (row),
    .w2e_arbitration_schedule   (w2e_arbitration_schedule)
) instance_name (
    .clk                        (clk),
    .rstn                        (rstn),
    .output_rstn                (output_rstn),
    .arready                    (arready),
    .arvalid                    (arvalid),
    .arqos                        (arqos),
    .arburst                    (arburst),
    .arlock                        (arlock),
    .arsize                        (arsize),
    .arlen                        (arlen),
    .arid                        (arid),
    .araddr                        (araddr),
    .awready                    (awready),
    .awvalid                    (awvalid),
    .awqos                        (awqos),
    .awburst                    (awburst),
    .awlock                        (awlock),
    .awsize                        (awsize),
    .awlen                        (awlen),
    .awid                        (awid),
    .awaddr                        (awaddr),
    .wready                        (wready),
    .wvalid                        (wvalid),
    .wdata                        (wdata),
    .wstrb                        (wstrb),
    .wlast                        (wlast),
    .rready                        (rready),
    .rvalid                        (rvalid),
    .rresp                        (rresp),
    .rid                        (rid),
    .rdata                        (rdata),
    .rlast                        (rlast),
    .bready                        (bready),
    .bvalid                        (bvalid),
    .bid                        (bid),
    .bresp                        (bresp),
    .error_valid                (error_valid),
    .error_info                 (error_info)
```

```
    ) /* synthesis syn_noprune=1 */;

// Note : It is sometimes necessary to add the syn_noprune to the NAP instantiation, as shown
here.
//        This is because the synthesis tool is not aware of the NOC behind the NAP and so may
remove the NAP in some instances.
```

VHDL

```
-- VHDL Component template for ACX_NAP_AXI_SLAVE
component ACX_NAP_AXI_SLAVE is
generic (
    att_ddr_0                        : integer := X"0";
    ...
    att_ddr_127                      : integer := X"7f";
    att_gddr_0                       : integer := X"0";
    ...
    att_gddr_127                     : integer := X"7f";
    att_nap_0                        : integer := X"0";
    ...
    att_nap_79                       : integer := X"4f";
    column                           : integer := X"x";
    e2w_arbitration_schedule         : integer := X"AAAAAAAA";
    row                              : integer := X"x";
    w2e_arbitration_schedule         : integer := X"AAAAAAAA"
);
port (
    clk                              : in  std_logic;
    rstn                             : in  std_logic;
    output_rstn                      : out std_logic;
    arready                          : out std_logic;
    arvalid                          : in  std_logic;
    arqos                            : in  std_logic_vector( 3 downto 0 );
    arburst                          : in  std_logic_vector( 1 downto 0 );
    arlock                           : in  std_logic;
    arsize                           : in  std_logic_vector( 2 downto 0 );
    arlen                            : in  std_logic_vector( 7 downto 0 );
    arid                             : in  std_logic_vector( 7 downto 0 );
    araddr                           : in  std_logic_vector( 41 downto 0 );
    awready                          : out std_logic;
    awvalid                          : in  std_logic;
    awqos                            : in  std_logic_vector( 3 downto 0 );
    awburst                          : in  std_logic_vector( 1 downto 0 );
    awlock                           : in  std_logic;
    awsize                           : in  std_logic_vector( 2 downto 0 );
    awlen                            : in  std_logic_vector( 7 downto 0 );
    awid                             : in  std_logic_vector( 7 downto 0 );
    awaddr                           : in  std_logic_vector( 41 downto 0 );
    wready                           : out std_logic;
    wvalid                           : in  std_logic;
    wdata                            : in  std_logic_vector( 255 downto 0 );
    wstrb                            : in  std_logic_vector( 31 downto 0 );
    wlast                            : in  std_logic;
    rready                           : in  std_logic;
    rvalid                           : out std_logic;
    rresp                            : out std_logic_vector( 1 downto 0 );
    rid                              : out std_logic_vector( 7 downto 0 );
```

```
    rdata                           : out std_logic_vector( 255 downto 0 );
    rlast                           : out std_logic;
    bready                          : in  std_logic;
    bvalid                          : out std_logic;
    bid                             : out std_logic_vector( 7 downto 0 );
    bresp                           : out std_logic_vector( 1 downto 0 );
    error_valid                     : out std_logic;
    error_info                      : out std_logic_vector( 2 downto 0 )
);
end component ACX_NAP_AXI_SLAVE

-- VHDL Instantiation template for ACX_NAP_AXI_SLAVE
instance_name : ACX_NAP_AXI_SLAVE
generic map (
    att_ddr_0                       => att_ddr_0,
    ...
    att_ddr_127                     => att_ddr_127,
    att_gddr_0                      => att_gddr_0,
    ...
    att_gddr_127                    => att_gddr_127,
    att_nap_0                       => att_nap_0,
    ...
    att_nap_79                      => att_nap_79,
    column                          => column,
    e2w_arbitration_schedule        => e2w_arbitration_schedule,
    row                             => row,
    w2e_arbitration_schedule        => w2e_arbitration_schedule
)
port map (
    clk                             => user_clk,
    rstn                            => user_rstn,
    output_rstn                     => user_output_rstn,
    arready                         => user_arready,
    arvalid                         => user_arvalid,
    arqos                           => user_arqos,
    arburst                         => user_arburst,
    arlock                          => user_arlock,
    arsize                          => user_arsize,
    arlen                           => user_arlen,
    arid                            => user_arid,
    araddr                          => user_araddr,
    awready                         => user_awready,
    awvalid                         => user_awvalid,
    awqos                           => user_awqos,
    awburst                         => user_awburst,
    awlock                          => user_awlock,
    awsize                          => user_awsize,
    awlen                           => user_awlen,
    awid                            => user_awid,
    awaddr                          => user_awaddr,
    wready                          => user_wready,
    wvalid                          => user_wvalid,
    wdata                           => user_wdata,
    wstrb                           => user_wstrb,
    wlast                           => user_wlast,
    rready                          => user_rready,
    rvalid                          => user_rvalid,
    rresp                           => user_rresp,
    rid                             => user_rid,
```

```
    rdata                            => user_rdata,
    rlast                            => user_rlast,
    bready                           => user_bready,
    bvalid                           => user_bvalid,
    bid                              => user_bid,
    bresp                            => user_bresp,
    error_valid                      => user_error_valid,
    error_info                       => user_error_info
);
```

Parameter Templates

An example of a `.pdc` file setting both the location and arbitration parameters of the ACX_NAP_AXI_SLAVE is shown below.

```
# Example NAP hierarchical name, (using axi_slave_wrapper.sv), is my_block.i_axi_slave_wrapper.
i_axi_slave

# Fix location to column 1, row 2
# Note use of "i:" at start of instance name
set_placement -fixed i:my_block.i_axi_slave_wrapper.i_axi_slave s:x_core.NOC[1][2].logic.noc.nap_s

# Set north to south arbitration to 0x1234_5678
# Note no "i:" at start of instance name
set_property n2s_arbitration_schedule 32'h1234_5678 my_block.i_axi_slave_wrapper.i_axi_slave

# Set south to north arbitration to 0x8888_8888
# Note no "i:" at start of instance name
set_property s2n_arbitration_schedule 32'h8888_8888 my_block.i_axi_slave_wrapper.i_axi_slave
```

ACX_NAP_HORIZONTAL

The ACX_NAP_HORIZONTAL component provides for 288-bit bidirectional data transport. When a word of data is presented to the transmission interface along with a destination ID, the data and all of the fields are captured and are sent to the destination node at the column indicated by the `tx_dest[3:0]` signal, which then presents it to the FPGA logic using the destination NAP's receiver interface.

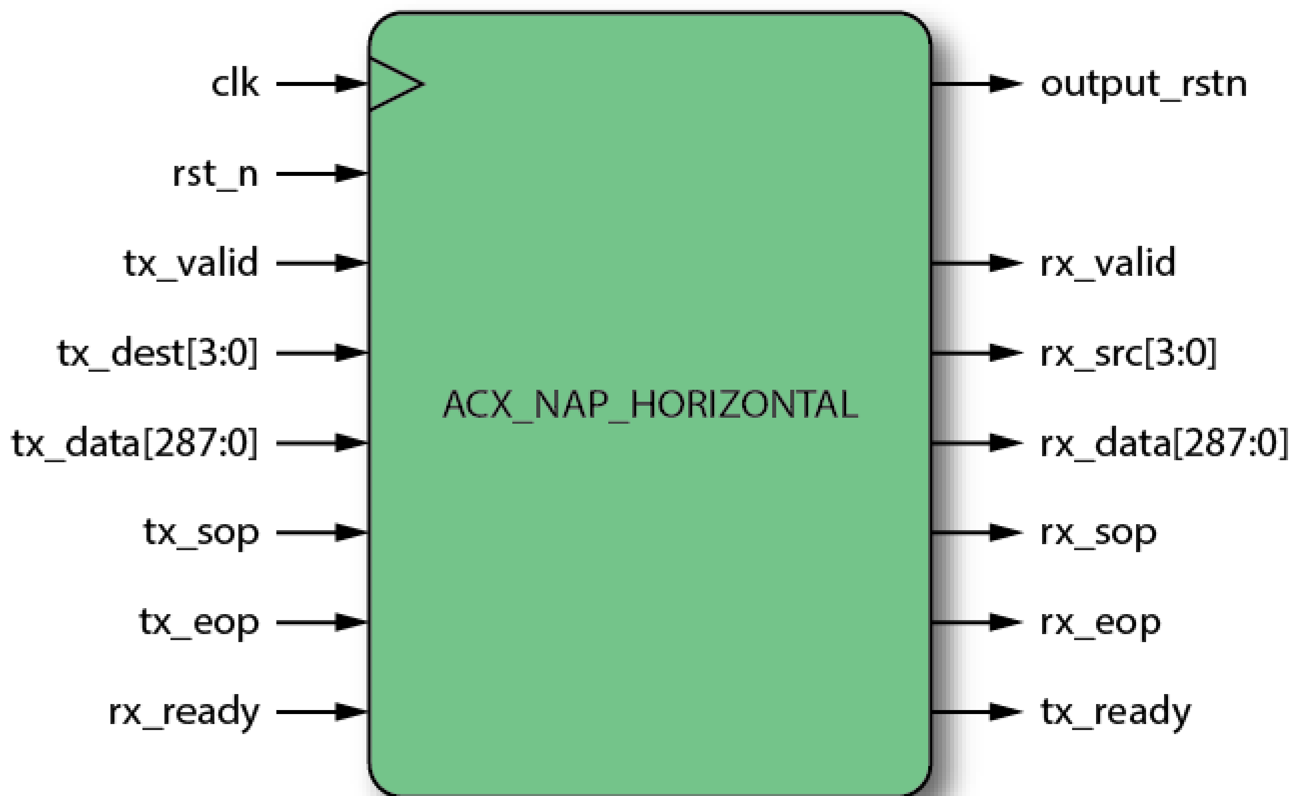


Figure 81: ***ACX_NAP_HORIZONTAL Logic Symbol***

Parameters

Table 229: *Parameters*

Parameter	Supported Values	Default Value	Description
`e2w_arbitration_schedule` `w2e_arbitration_schedule`	`32'h0–32'hffff_fffe`(1)	`32'haaaa_aaaa`	A 32-bit value used to initialize the arbitration schedule mechanism. Bit 0 of the arbitration schedule vector is used to determine if the local node wins arbitration when there is competing traffic from the upstream node. If bit 0 has a value of '1', the local traffic wins, while if bit 0 has a value of '0', the upstream node wins. After each cycle where both the local node and the upstream node are competing for access, the value in the schedule register rotates to the left. (2)(4) A value of `32'h8888_8888` means that the local node has priority on every fourth cycle. A value of `32'haaaa_aaaa` means that the local node has priority on every second cycle.
`row[3:0]`(3)(4)	`4'd1–4'd8`	`4'hx`	Fixes the NAP row location in the NoC.
`col[3:0]`(3)(4)	`4'd1–4'd10`	`4'hx`	Fixes the NAP column location in the NoC.

Table Notes

1. A value of 32'hffff_ffff is not legal and is ignored. This would result in the upstream node never being serviced.
2. Although it is possible to directly assign the NAP arbitration using these parameters, it is recommended that the arbitration is specified in the relevant placement design constraints (`.pdc`) file. The arbitration value is dependent upon both the physical location of the NAP in the row and on the number and type of other NAPs on the same row. These physical attributes can all be managed and controlled within the single `.pdc` file.
3. Although it is possible to directly assign the NAP location using these parameters, it is recommended that locations are specified in the relevant placement design constraints (`.pdc`) file. Using the PDC to assign placements removes physical constraints from the functional RTL, thus allowing the same source code to be reused and placed in different locations on the die.
4. If these parameters are defined in both the RTL, and a `.pdc` file, the PDC assignments take priority.

Examples of setting the location and arbitration parameters are shown in Parameter Templates (see page 313).

Ports

Table 230: *Port Descriptions*

Name	Direction	Description
`rstn`	Input	Asynchronous reset input. This signal resets the NAP interface and does not affect the NoC.
`output_rstn`	Output	Reset output from NAP to fabric logic. **Do not use**. Intended for use with partial reconfiguration. Signal is controlled by a write to the configuration space. Currently, the signal is fixed to `1'b0`.
`clk`	Input	All operations are fully synchronous and occur upon the active edge of the clk input.
`tx_ready`	Output	Asserted high when the NAP can accept data.
`tx_valid`	Input	Assert high to issue a word of data to the NAP.
`tx_dest[3:0]`	Input	The 4-bit destination ID of the column to which data is to be transmitted.
`tx_sop` `tx_eop`	Input	Transmitted data start and end of packet indicators. Use of these signals is optional. The purpose is to indicate to the receiving node the start and end of any packet transmission. The NAP (and NoC), do not require or make use of these signals. The `tx_data` word, `tx_sop` and `tx_eop` are transmitted on each `tx_valid` assertion.
`tx_data[287:0]`	Input	Data transmitted to the destination node.
`rx_ready`	Input	Asserted high to indicate that user logic is ready to receive.
`rx_valid`	Output	Asserted high with each valid `rx_data` word.
`rx_src[3:0]`	Output	The 4-bit source ID indicating the column that originated the data.
`rx_sop` `rx_eop`	Output	Received data start and end of packet indicators. Received unmodified from the source.
`rx_data[287:0]`	Output	Received data.

Inference

It is not possible to infer the ACX_NAP_HORIZONTAL. It must be directly instantiated.

Instantiation Templates

Verilog

```
ACX_NAP_HORIZONTAL #(
    .column                         (column),
    .e2w_arbitration_schedule       (e2w_arbitration_schedule),
    .row                            (row),
    .w2e_arbitration_schedule       (w2e_arbitration_schedule)
) instance_name (
    .clk                            (user_clk),
    .rstn                           (user_rstn),
    .output_rstn                    (user_output_rstn),
    .rx_ready                       (user_rx_ready),
    .rx_valid                       (user_rx_valid),
    .rx_data                        (user_rx_data),
    .rx_src                         (user_rx_src),
    .rx_eop                         (user_rx_eop),
    .rx_sop                         (user_rx_sop),
    .tx_ready                       (user_tx_ready),
    .tx_valid                       (user_tx_valid),
    .tx_dest                        (user_tx_dest),
    .tx_eop                         (user_tx_eop),
    .tx_sop                         (user_tx_sop),
    .tx_data                        (user_tx_data)
);
```

VHDL

```
-- VHDL Component template for ACX_NAP_HORIZONTAL
component ACX_NAP_HORIZONTAL is
generic (
    column                          : integer := X"x";
    e2w_arbitration_schedule        : integer := X"AAAAAAAA";
    row                             : integer := X"x";
    w2e_arbitration_schedule        : integer := X"AAAAAAAA"
);
port (
    clk                             : in  std_logic;
    rstn                            : in  std_logic;
    output_rstn                     : out std_logic;
    rx_ready                        : in  std_logic;
    rx_valid                        : out std_logic;
    rx_data                         : out std_logic_vector( 287 downto 0 );
    rx_src                          : out std_logic_vector( 3 downto 0 );
    rx_eop                          : out std_logic;
    rx_sop                          : out std_logic;
    tx_ready                        : out std_logic;
    tx_valid                        : in  std_logic;
    tx_dest                         : in  std_logic_vector( 3 downto 0 );
    tx_eop                          : in  std_logic;
    tx_sop                          : in  std_logic;
    tx_data                         : in  std_logic_vector( 287 downto 0 )
);
end component ACX_NAP_HORIZONTAL
```

```
-- VHDL Instantiation template for ACX_NAP_HORIZONTAL
instance_name : ACX_NAP_HORIZONTAL
generic map (
    column                          => column,
    e2w_arbitration_schedule        => e2w_arbitration_schedule,
    row                             => row,
    w2e_arbitration_schedule        => w2e_arbitration_schedule
)
port map (
    clk                             => user_clk,
    rstn                            => user_rstn,
    output_rstn                     => user_output_rstn,
    rx_ready                        => user_rx_ready,
    rx_valid                        => user_rx_valid,
    rx_data                         => user_rx_data,
    rx_src                          => user_rx_src,
    rx_eop                          => user_rx_eop,
    rx_sop                          => user_rx_sop,
    tx_ready                        => user_tx_ready,
    tx_valid                        => user_tx_valid,
    tx_dest                         => user_tx_dest,
    tx_eop                          => user_tx_eop,
    tx_sop                          => user_tx_sop,
    tx_data                         => user_tx_data
);
```

Parameter Templates

An example of a `.pdc` file setting both the location and arbitration parameters of the ACX_NAP_HORIZONTAL is shown below.

```
# Example NAP hierarchical name, (using axi_horizontal_wrapper.sv), is my_block.
i_axi_horizontal_wrapper.i_nap_horizontal

# Fix location to column 1, row 2
# Note use of "i:" at start of instance name
set_placement -fixed i:my_block.i_axi_horizontal_wrapper.i_nap_horizontal s:x_core.NOC[1][2].
logic.noc.nap_s

# Set north to south arbitration to 0x1234_5678
# Note no "i:" at start of instance name
set_property n2s_arbitration_schedule 32'h1234_5678 my_block.i_axi_horizontal_wrapper.
i_nap_horizontal

# Set south to north arbitration to 0x8888_8888
# Note no "i:" at start of instance name
set_property s2n_arbitration_schedule 32'h8888_8888 my_block.i_axi_horizontal_wrapper.
i_nap_horizontal
```

ACX_NAP_VERTICAL

The ACX_NAP_VERTICAL component provides for 293-bit vertical bidirectional data transport. When the user logic presents a word of data to the transmit interface along with a destination ID, the data and all of the fields are captured and sent to the destination node at the row indicated by the `tx_dest[3:0]` signal, which then presents it to the FPGA logic using the destination NAP's receive interface.

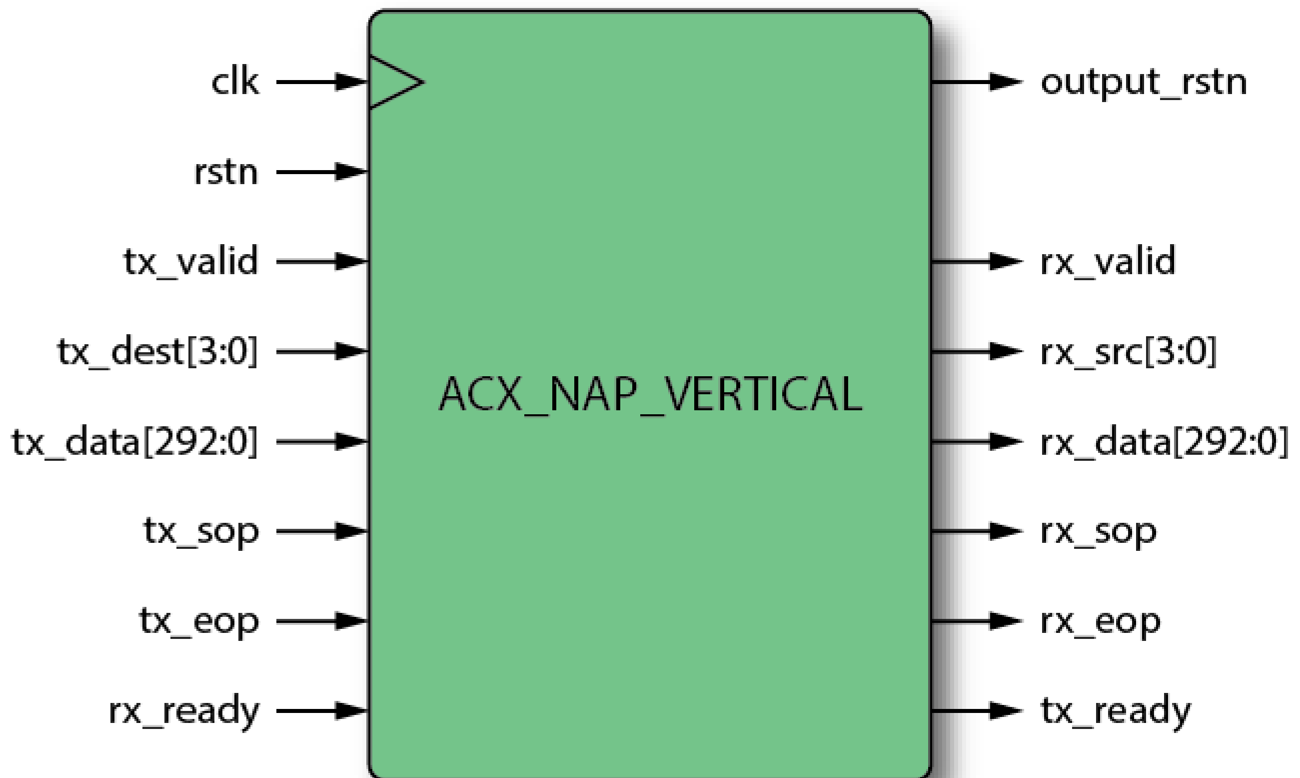


Figure 82: ***ACX_NAP_VERTICAL Logic Symbol***

Parameters

Table 231: *Parameters*

Parameter	Supported Values	Default Value	Description
n2s_arbitration_schedule s2n_arbitration_schedule	32'h0-32'hffff_fffe (1)	32'haaaa_aaaa	A 32-bit value used to initialize the arbitration schedule mechanism. Bit 0 of the arbitration schedule vector is used to determine if the local node wins arbitration when there is competing traffic from the upstream node. If bit 0 has a value of '1', the local traffic wins, while if bit 0 has a value of '0', the upstream node wins. After each cycle where both the local node and the upstream node are competing for access, the value in the schedule register rotates to the left. (2)(4) A value of 32'h8888_8888 means that the local node has high priority on every fourth cycle. A value of 32'haaaa_aaaa means that the local node has high priority on every second cycle.
row[3:0](3)(4)	4'd1-4'd8	4'hx	Fixes the NAP row location in the NoC.
col[3:0](3)(4)	4'd1-4'd10	4'hx	Fixes the NAP column location in the NoC.

Table Notes

1. A value of 32'hffff_ffff is not legal and is ignored. This would result in the upstream node never being serviced.
2. Although it is possible to directly assign the NAP arbitration using these parameters, it is recommended that the arbitration is specified in the relevant placement design constraints (.pdc) file. The arbitration value is dependent upon both the physical location of the NAP in the column and on the number and type of other NAPs on the same column. These physical attributes can all be managed and controlled within the single .pdc file.
3. Although it is possible to directly assign the NAP location using these parameters; it is recommended that locations are specified in the relevant placement design constraints (.pdc) file. Using the PDC to assign placements removes physical constraints from the functional RTL, thus allowing the same source code to be reused and placed in different locations on the die.
4. If these parameters are defined in both RTL and a .pdc file. The PDC assignments take priority.

Examples of setting the location and arbitration parameters are shown in Parameter Templates (see page 318)

Ports

Table 232: *Port Descriptions*

Name	Direction	Description
rstn	Input	Asynchronous reset input. Resets the NAP interface but does not affect the NoC.
output_rstn	Output	Reset output from the NAP to the fabric logic. **Do not use**. Intended for use with partial reconfiguration. Signal is controlled by a write to the configuration space. Currently, the signal is fixed to 1'b0.
clk	Input	All operations are fully synchronous and occur upon the active edge of the clk input.
tx_ready	Output	Asserted high when the NAP can accept data.
tx_valid	Input	Assert high to issue a word of data to the NAP.
tx_dest[3:0]	Input	The 4-bit destination ID of the row to which this word of data should be sent.
tx_sop tx_eop	Input	Start of packet and end of packet indicators. These values are sent unmodified to the destination. Use of these signals is optional. The purpose is to indicate to the receiving node the start and end of any packet transmission. The NAP (and NoC) do not require or make use of these signals. The tx_data word, tx_sop and tx_eop are transmitted on each tx_valid assertion.
tx_data[292:0]	Input	Data transmitted to the destination node.
rx_ready	Input	Asserted high by the user logic to indicate that it is ready to receive data.
rx_valid	Output	Asserted high with each valid rx_data word.
rx_src[3:0]	Output	The 4-bit transmission source ID indicating the row that originated the data.
rx_sop rx_eop	Output	Start of packet and end of packet indicators. These values are received unmodified from the source.
rx_data[292:0]	Output	Received data.

Inference

It is not possible to infer the ACX_NAP_VERTICAL. It must be directly instantiated.

Instantiation Templates

Verilog

```
ACX_NAP_VERTICAL #(
    .column                         (column),
    .n2s_arbitration_schedule       (n2s_arbitration_schedule),
    .row                            (row),
    .s2n_arbitration_schedule       (s2n_arbitration_schedule)
) instance_name (
    .clk                            (user_clk),
    .rstn                           (user_rstn),
    .output_rstn                    (user_output_rstn),
    .rx_ready                       (user_rx_ready),
    .rx_valid                       (user_rx_valid),
    .rx_sop                         (user_rx_sop),
    .rx_eop                         (user_rx_eop),
    .rx_data                        (user_rx_data),
    .rx_src                         (user_rx_src),
    .tx_ready                       (user_tx_ready),
    .tx_valid                       (user_tx_valid),
    .tx_sop                         (user_tx_sop),
    .tx_eop                         (user_tx_eop),
    .tx_dest                        (user_tx_dest),
    .tx_data                        (user_tx_data)
);
```

VHDL

```
-- VHDL Component template for ACX_NAP_VERTICAL
component ACX_NAP_VERTICAL is
generic (
    column                          : integer := X"x";
    n2s_arbitration_schedule        : integer := X"AAAAAAAA";
    row                             : integer := X"x";
    s2n_arbitration_schedule        : integer := X"AAAAAAAA"
);
port (
    clk                             : in  std_logic;
    rstn                            : in  std_logic;
    output_rstn                     : out std_logic;
    rx_ready                        : in  std_logic;
    rx_valid                        : out std_logic;
    rx_sop                          : out std_logic;
    rx_eop                          : out std_logic;
    rx_data                         : out std_logic_vector( 292 downto 0 );
    rx_src                          : out std_logic_vector( 3 downto 0 );
    tx_ready                        : out std_logic;
    tx_valid                        : in  std_logic;
    tx_sop                          : in  std_logic;
    tx_eop                          : in  std_logic;
    tx_dest                         : in  std_logic_vector( 3 downto 0 );
    tx_data                         : in  std_logic_vector( 292 downto 0 )
);
end component ACX_NAP_VERTICAL
```

```
-- VHDL Instantiation template for ACX_NAP_VERTICAL
instance_name : ACX_NAP_VERTICAL
generic map (
    column                          => column,
    n2s_arbitration_schedule        => n2s_arbitration_schedule,
    row                             => row,
    s2n_arbitration_schedule        => s2n_arbitration_schedule
)
port map (
    clk                             => user_clk,
    rstn                            => user_rstn,
    output_rstn                     => user_output_rstn,
    rx_ready                        => user_rx_ready,
    rx_valid                        => user_rx_valid,
    rx_sop                          => user_rx_sop,
    rx_eop                          => user_rx_eop,
    rx_data                         => user_rx_data,
    rx_src                          => user_rx_src,
    tx_ready                        => user_tx_ready,
    tx_valid                        => user_tx_valid,
    tx_sop                          => user_tx_sop,
    tx_eop                          => user_tx_eop,
    tx_dest                         => user_tx_dest,
    tx_data                         => user_tx_data
);
```

Parameter Templates

An example of a `.pdc` file setting both the location and arbitration parameters of the ACX_NAP_VERTICAL is shown below.

```
# Example NAP hierarchical name, (using nap_vertical_wrapper.sv), is my_block.
i_axi_vertical_wrapper.i_nap_vertical

# Fix location to column 1, row 2
# Note use of "i:" at start of instance name
set_placement -fixed i:my_block.i_axi_vertical_wrapper.i_nap_vertical s:x_core.NOC[1][2].logic.
noc.nap_m

# Set north to south arbitration to 0x1234_5678
# Note no "i:" at start of instance name
set_property n2s_arbitration_schedule 32'h1234_5678 my_block.i_axi_vertical_wrapper.i_nap_vertical

# Set south to north arbitration to 0x8888_8888
# Note no "i:" at start of instance name
set_property s2n_arbitration_schedule 32'h8888_8888 my_block.i_axi_vertical_wrapper.i_nap_vertical
```

ACX_NAP_ETHERNET

The ACX_NAP_ETHERNET component is used to provide Ethernet packet transmission and reception. The NAP is placed on one of the specific Ethernet columns (see the "Speedster7t Ethernet System Architecture" chapter in the *Speedster7t Ethernet User Guide* (UG097) for details). The NAP then transmits and receives packets from the Ethernet Interface Unit (EIU) within the Ethernet subsystem.

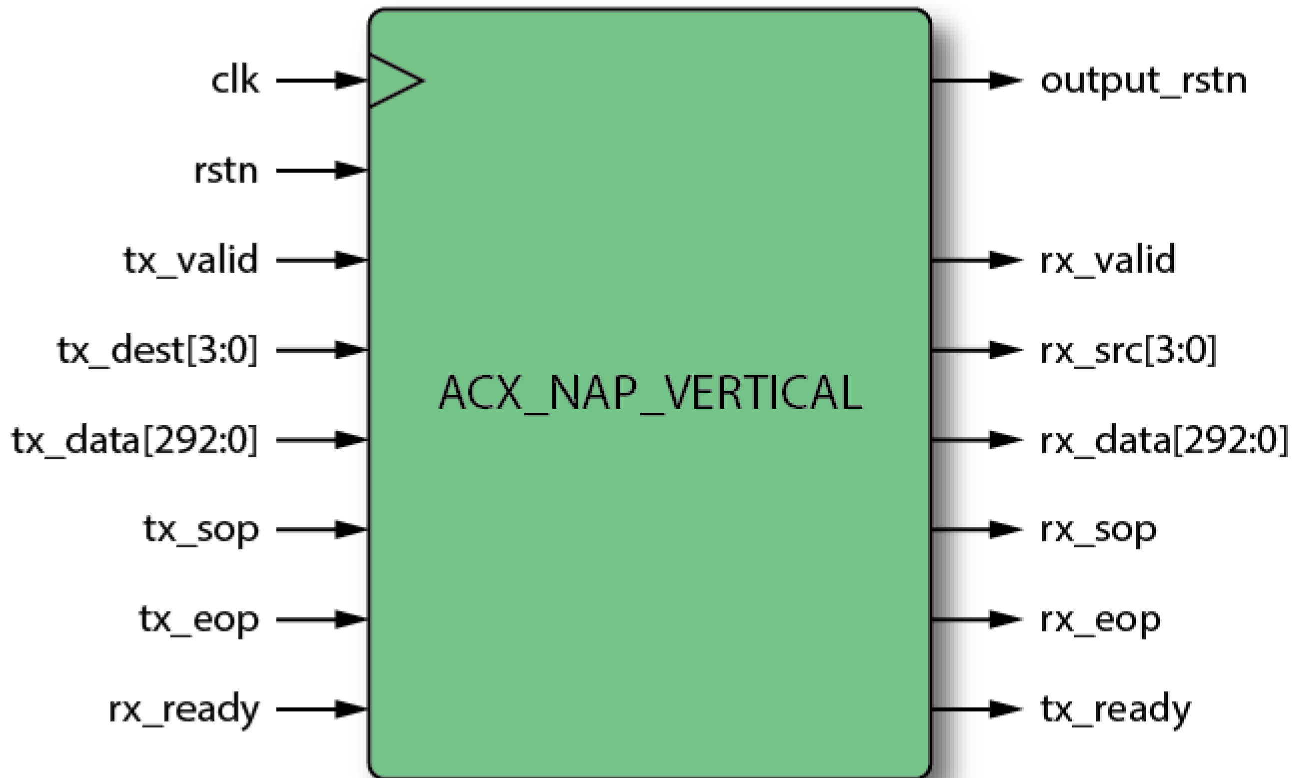


Figure 83: ***ACX_NAP_ETHERNET Logic Symbol***

Parameters

Table 233: *Parameters*

Parameter	Supported Values	Default Value	Description
row[3:0](1)(4)	4'b0001–4'b1000	4'b1000	Fixes the NAP row location in the NoC.
column[3:0](1)(4)	4'b0001, 4'b0010, 4'b0100, 4'b0101	4'b0100	Fixes the NAP column location in the NoC.
tx_enable rx_enable	1'b0–1'b1	1'b1	Enables/disables the interface: 1'b1 – enable interface. 1'b0 – disable interface.
tx_mode[3:0] rx_mode[3:0]	4'b0000–4'b1000	4'b0100	Modes supported by the Ethernet IP can be selected as follows: 4'b0000 – 10G. 4'b0001 – 25G. 4'b0010 – 40G. 4'b0011 – 50G. 4'b0100 – 100G. 4'b0101 – 200G_PKT. 4'b0110 – 200G_QSI. 4'b0111 – 400G_PKT. 4'b1000 – 400G_QSI.
tx_mac_id[1:0](2) rx_mac_id[1:0]	2'b00–2'b11	2'b10	Select the appropriate MAC within the Ethernet Subsystem: 2'b00 – selects 400G_MAC0. 2'b01 – selects 400G_MAC1. 2'b10 – selects QUAD_MAC0. 2'b11 – selects QUAD_MAC1.
tx_eiu_channel[4:0] rx_eiu_channel[4:0]	5'b0000–5'b11111	5'b00000	Select the EIU channels (see page 322) to which the NAP transmits, or from which the NAP receives.
tx_threshold[31:0]	32'h0–32'hffff_ffff	32'haaaa_aaaa	Threshold values for the EIU buffer channel FIFOs. When the NAP is assigned to a buffering subsystem within the EIU, traffic from the NAP is transmitted via the EIU TX FIFO. This FIFO has four threshold indicators. The levels at which each of these indicators are set are determined by the corresponding byte within the four-byte tx_threshold. If the EIU TX FIFO fill level exceeds a threshold value, the corresponding <ethernet name>_<mac name>_tx_buffer<num>_at_threshold signal is asserted. For example, for a tx_threshold value of 32'he0c0_8040, the four threshold indicators are asserted if the TX FIFO level exceeds 0x40, 0x80, 0xc0 and 0xe0 words, respectively. If the NAP is assigned to a pass-through EIU channel, the tx_threshold value is unused.
rx_threshold[31:0]	32'h0–32'hffff_ffff	32'haaaa_aaaa	Threshold values for the EIU buffer channel FIFOs. When the NAP is assigned to a buffering subsystem within the EIU, traffic from the NAP is received via the EIU RX FIFO. This FIFO has four threshold indicators. The levels at which each of these indicators are set are determined by the corresponding byte within the four-byte rx_threshold. If the EIU RX FIFO fill level exceeds a threshold value, then the corresponding <ethernet name>_<mac name>_rx_buffer<num>_at_threshold signal is asserted. For example, for a rx_threshold value of 32'he0c0_8040, the four threshold indicators are asserted if the RX FIFO level exceeds 0x40, 0x80, 0xc0 and 0xe0 words, respectively. If the NAP is assigned to a pass-through EIU channel, the rx_threshold value is unused.

Parameter	Supported Values	Default Value	Description
`n2s_arbitration_schedule`[3] `s2n_arbitration_schedule`	`32'h0–32'hffff_fffe`[5]	`32'haaaa_aaaa`	A 32-bit value used to initialize the arbitration schedule mechanism. Bit 0 of the arbitration schedule vector is used to determine if the local node wins arbitration when there is competing traffic from the upstream node. If bit 0 has a value of '1', the local traffic wins, while if bit 0 has a value of '0', the upstream node wins. After each cycle where both the local node and the upstream node are competing for access, the value in the schedule register rotates to the left. A value of `32'h8888_8888` means that the local node has priority on every fourth cycle. A value of `32'haaaa_aaaa` means that the local node has priority on every second cycle.

Table Notes

1. Although it is possible to directly assign the NAP location using these parameters; it is recommended that locations are specified in the relevant placement design constraints (`.pdc`) file. Using the PDC to assign placements removes physical constraints from the functional RTL, thus allowing the same source code to be reused and placed in different locations on the die. If these parameters are defined in both the RTL, and a `.pdc` file, the PDC assignments take priority.
2. The `mac_id` must align with `tx_mode`/`rx_mode`. Therefore, if the mode is 100G or below, the `mac_id` must be set to one of the Quad MACs. Equally, if the mode is 200G or 400G, then the `mac_id` must be set to one of the 400G_MACs.
3. Although it is possible to directly assign the NAP arbitration using these parameters, it is recommended that the arbitration is specified in the relevant placement design constraints (`.pdc`) file. The arbitration value is dependent on both the physical location of the NAP in the column and on the number and type of other NAPs on the same column. These physical attributes can all be managed and controlled within the single `.pdc` file.
4. If these parameters are defined in both the RTL, and a `.pdc` file, the PDC assignments take priority.
5. A value of `32'hffff_ffff` is not legal and is ignored. This would result in the upstream node never being serviced.

Examples of setting the location and arbitration parameters are shown in Parameter Templates (see page 327).

EIU Channels

Each Ethernet subsystem contains an EIU which links the traffic from the NAPs to the Ethernet MACs. Each EIU consists of 32 channels. Each channel must be exclusively assigned to a single NAP. The channels are detailed below.

Table 234: ***EIU Channel Functions***

<table>
<tr><th>Channel Number</th><th>Name</th><th>Function</th><th>Compatible Modes</th><th>Comments</th></tr>
<tr><td>0 to 7</td><td>Buffer SubSystem0</td><td>400G/200G buffering and packet structuring</td><td>200G_PKT, 200G_QSI, 400G_PKT, 400G_QSI</td><td rowspan="2">The four NAPs assigned to either QSI or PKT mode must each be assigned to an individual EIU channel.</td></tr>
<tr><td>8 to 15</td><td>Buffer SubSystem1</td><td>400G/200G buffering and packet structuring</td><td>200G_PKT, 200G_QSI, 400G_PKT, 400G_QSI</td></tr>
<tr><td>16 to 19</td><td>Quad MAC 0, Pre-emptive Lanes</td><td>10G to 100G pass-through</td><td>10G, 25G, 40G, 50G, 100G</td><td rowspan="4">Each individual Ethernet channel has a single NAP and must be assigned to a single EIU channel.</td></tr>
<tr><td>20 to 23</td><td>Quad MAC 0, Express Lanes</td><td>10G to 100G pass-through</td><td>10G, 25G, 40G, 50G, 100G</td></tr>
<tr><td>24 to 27</td><td>Quad MAC 1, Pre-emptive Lanes</td><td>10G to 100G pass-through</td><td>10G, 25G, 40G, 50G, 100G</td></tr>
<tr><td>28 to 31</td><td>Quad MAC 1, Express Lanes</td><td>10G to 100G pass-through</td><td>10G, 25G, 40G, 50G, 100G</td></tr>
</table>

Ports

Table 235: *Port Descriptions*

Name	Direction	Description
rstn	Input	Asynchronous reset input. This signal resets the NAP interface but does not affect the NoC.
output_rstn	Output	Reset output from the NAP to the fabric logic. **Do not use**. Intended for use with partial reconfiguration. This signal is controlled by a write to the configuration space. Currently, the signal is fixed to `1'b0`.
clk	Input	All operations are fully synchronous and occur upon the active edge of the `clk` input.
tx_ready	Output	Asserted high when the NAP can accept data.
tx_valid	Input	Assert high to issue a word of data to the NAP.
tx_dest[3:0]	Input	The 4-bit destination ID for the Ethernet packets. This value must be fixed to `4'hf` to indicate that packets must be sent to the EIU reserved address.
tx_sop tx_eop	Input	Start of packet and end of packet indicators. These signals are required to be set for each packet transmitted to the Ethernet subsystem.
tx_data[292:0]	Input	Ethernet packet to be transmitted to the EIU: `tx_data[255:0]` – packet data. `tx_data[260:256]` – mod data. Valid when `tx_eop` is set. `tx_data[290:261]` – transmit flags or timestamp. Timestamp is valid when `tx_sop` is set. Transmit flags are valid on other packet cycles. `tx_data[292:291]` – unused. Please refer to the "Speedster7t Ethernet System Architecture" chapter in the *Speedster7t Ethernet User Guide* (UG097) for full details on the transmit flags.
rx_ready	Input	Asserted high by user logic to indicate it is ready to receive data.
rx_valid	Output	Asserted high with each valid `rx_data` word.
rx_src[3:0]	Output	The 4-bit transmission source ID indicating the row that originated the data. This value is always `4'hf` to indicate that the packet was received from the EIU. This output is generally unused.
rx_sop rx_eop	Output	Start and end of Ethernet packet indicators.
rx_data[292:0]	Output	Received Ethernet Packet: `rx_data[255:0]` – packet data. `rx_data[260:256]` – mod data. Valid when rx_eop is asserted. `rx_data[290:261]` – receive flags or timestamp. Timestamp is valid when `rx_sop` is set. Receive flags are valid on other packet cycles. `rx_data[292:291]` – unused. Please refer to the "Speedster7t Ethernet System Architecture" chapter in the *Speedster7t Ethernet User Guide* (UG097) for full details of the receive flags.

Mod Values

Ethernet packet interfaces at the NAP are 256-bits wide, equal to 32-bytes. The mod value indicated on the last word of a packet is a 5-bit value. This mod value indicates numerically how many bytes are valid in the last word of a packet. If the mod value equals 0, then all bytes are valid. Any bytes that are in byte lanes greater than the mod value are not used and can be set to any value.

Inference

It is not possible to infer the ACX_NAP_ETHERNET. It must be directly instantiated.

Instantiation Templates

Verilog

```
ACX_NAP_ETHERNET #(
    .column          (COLUMN),
    .row             (ROW),
    .tx_enable       (TX_ENABLE),
    .tx_mode         (TX_MODE),
    .tx_mac_id       (TX_MAC_ID),
    .tx_eiu_channel  (TX_EIU_CHANNEL),
    .tx_threshold    (TX_THRESHOLD),
    .rx_enable       (RX_ENABLE),
    .rx_mode         (RX_MODE),
    .rx_mac_id       (RX_MAC_ID),
    .rx_eiu_channel  (RX_EIU_CHANNEL),
    .rx_threshold    (RX_THRESHOLD)
) instance_name (
    .clk             (user_clk),
    .rstn            (user_rstn),
    .output_rstn     (user_output_rstn),
    .rx_ready        (user_rx_ready),
    .rx_valid        (user_rx_valid),
    .rx_sop          (user_rx_sop),
    .rx_eop          (user_rx_eop),
    .rx_data         (user_rx_data),
    .rx_src          (user_rx_src),
    .tx_ready        (user_tx_ready),
    .tx_valid        (user_tx_valid),
    .tx_sop          (user_tx_sop),
    .tx_eop          (user_tx_eop),
    .tx_dest         (user_tx_dest),
    .tx_data         (user_tx_data)
);
```

VHDL

```
-- VHDL Component template for ACX_NAP_ETHERNET
component ACX_NAP_ETHERNET is
generic (
    column                              : integer := X"x";
    row                                 : integer := X"x";
    tx_enable                           : integer := X"x";
    tx_mode                             : integer := X"x";
    tx_mac_id                           : integer := X"x";
    tx_eiu_channel                      : integer := X"x";
    tx_threshold                        : integer := X"xxxxxxxx";
    rx_enable                           : integer := X"x";
    rx_mode                             : integer := X"x";
    rx_mac_id                           : integer := X"x";
    rx_eiu_channel                      : integer := X"x";
    rx_threshold                        : integer := X"xxxxxxxx"
);
port (
    clk                                 : in  std_logic;
```

```
    rstn                         : in  std_logic;
    output_rstn                  : out std_logic;
    rx_ready                     : in  std_logic;
    rx_valid                     : out std_logic;
    rx_sop                       : out std_logic;
    rx_eop                       : out std_logic;
    rx_data                      : out std_logic_vector( 292 downto 0 );
    rx_src                       : out std_logic_vector( 3 downto 0 );
    tx_ready                     : out std_logic;
    tx_valid                     : in  std_logic;
    tx_sop                       : in  std_logic;
    tx_eop                       : in  std_logic;
    tx_dest                      : in  std_logic_vector( 3 downto 0 );
    tx_data                      : in  std_logic_vector( 292 downto 0 )
);
end component ACX_NAP_ETHERNET

-- VHDL Instantiation template for ACX_NAP_ETHERNET
instance_name : ACX_NAP_ETHERNET
generic map (
    column                       => COLUMN,
    row                          => ROW,
    tx_enable                    => TX_ENABLE,
    tx_mode                      => TX_MODE,
    tx_mac_id                    => TX_MAC_ID,
    tx_eiu_channel               => TX_EIU_CHANNEL,
    tx_threshold                 => TX_THRESHOLD,
    rx_enable                    => RX_ENABLE,
    rx_mode                      => RX_MODE,
    rx_mac_id                    => RX_MAC_ID,
    rx_eiu_channel               => RX_EIU_CHANNEL,
    rx_threshold                 => RX_THRESHOLD
)
port map (
    clk                          => user_clk,
    rstn                         => user_rstn,
    output_rstn                  => user_output_rstn,
    rx_ready                     => user_rx_ready,
    rx_valid                     => user_rx_valid,
    rx_sop                       => user_rx_sop,
    rx_eop                       => user_rx_eop,
    rx_data                      => user_rx_data,
    rx_src                       => user_rx_src,
    tx_ready                     => user_tx_ready,
    tx_valid                     => user_tx_valid,
    tx_sop                       => user_tx_sop,
    tx_eop                       => user_tx_eop,
    tx_dest                      => user_tx_dest,
    tx_data                      => user_tx_data
);
```

Parameter Templates

An example of a `.pdc` file setting both the location and arbitration parameters of the ACX_NAP_ETHERNET is shown below.

```
# Example NAP hierarchical name, (using ACX_ETHERNET_NODE.sv), is my_block.i_nap_eth.
i_ethernet_node.u_nap_ethernet

# Fix location to column 1, row 2
# Note use of "i:" at start of instance name
set_placement -fixed i:my_block.i_nap_eth.i_ethernet_node.u_nap_ethernet s:x_core.NOC[1][2].logic.
noc.nap_m

# Set north to south arbitration to 0x1234_5678
# Note no "i:" at start of instance name
set_property n2s_arbitration_schedule 32'h1234_5678 my_block.i_nap_eth.i_ethernet_node.
u_nap_ethernet

# Set south to north arbitration to 0x8888_8888
# Note no "i:" at start of instance name
set_property s2n_arbitration_schedule 32'h8888_8888 my_block.i_nap_eth.i_ethernet_node.
u_nap_ethernet
```

Revision History

Version	Date	Description
1.0	25 Mar 2020	• Initial release.
1.1	18 Aug 2020	**Updates**: • Corrected description of afull_threshold for all FIFOs. Updated description of aempty_threshold. • Corrected address widths of NAP_AXI_MASTER. Included AXI-4 specification details. Added 16 beat burst length restriction details. • Updated Integer library components. **Additions:** • Added inference code for MLP primitives, BRAM72K_SDP and LRAM2K_SDP. • Added VHDL templates for NAP components. • Additional details on LRAM2K_SDP simultaneous memory accesses.
2.0	21 Dec 2021	• Add ACX_ prefix to all instances • Added NAP_ETHERNET