Speedster7t IP Component Library User Guide (UG086)

Speedster FPGAs

Data Acceleration

Copyrights, Trademarks and Disclaimers

Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

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

Table of Contents

Chapter - 1: Introduction 9
ACX_ Prefix 9
Chapter - 2: Fabric Architecture 10
Introduction 10
RLB6 11
MLUT Mode 13
Routing Between RLB6s 14
RLB6 Detail 15
Lookup Table, (LUT), Functions 17
Six-Input Lookup Table (LUT6) 17
Dual Five-Input Lookup Table (LUT5x2) 20
Registers 23
Naming Convention 23
Register Primitives 25
Register Macros 54
Chapter - 3: Logic Functions 64
ACX_SYNCHRONIZER, ACX_SYNCHRONIZER_N 64
Using ACX_SYNCHRONIZER to Synchronize Reset 65
Instantiation Templates 66
ACX_SHIFTREG 66
Instantiation Templates 68
Chapter - 4: Clock Functions 70
CLKDIV (Clock Divider) 70
Constraints 72
Instantiation Templates 72
CLKGATE (Clock Gate) 73
Constraints 74
Instantiation Templates 74
CLKSWITCH (Clock Switch) 75
Constraints 77
Instantiation Templates 78

Chapter - 5: Arithmetic and DSP 79

Number Formats 79

Integer Formats 79

Integer Groups 80

Floating-Point Formats 82

Block Floating Point 84

ALU8 85

Description 85

Ports 85

Parameters 86

Functions 86

Instantiation Templates 87

MLP72 88

Numerical Formats 91

Parallel Multiplications 92

Memories 93

Instantiation 93

Common Stages 93

Integer Modes 102

Integrated LRAM 117

Block Floating-Point Modes 125

Floating-Point Modes 133

Verilog 145

MLP72_INT 148

Parameters 150

Ports 151

Input Data Mapping 152

Output Formatting and Error Conditions 153

Asynchronous Reset Rules 153

Inference 154

Instantiation Template 154

MLP72_INT8_MULT_4X 155

Parameters 156

Ports 158

Timing Diagrams 159

Inference 159

Instantiation Template 159

MLP72_INT16_MULT_2X . . . 161
Parameters . . . 161
Ports . . . 163
Timing Diagrams . . . 163
Inference . . . 164
Instantiation Template . . . 164
Floating-Point Library . . . 165
Introduction . . . 165
MLP Registers . . . 165
Accumulation . . . 166
Floating-Point Format . . . 166
Output Status . . . 166
FP_ADD . . . 167
FP_MULT . . . 171
FP_MULT_PLUS . . . 174
FP_MULT_2X . . . 178
FP_MULT_ADD . . . 183
Integer Library . . . 187
MLP Registers . . . 187
Accumulation . . . 187
Integer Format . . . 188
INT_MULT . . . 188
INT_MULT_N . . . 192
INT_MULT_ADD . . . 195
Chapter - 6: Memories . . . 200
BRAM72K_FIFO . . . 200
Parameters . . . 201
Ports . . . 203
Memory Organization and Data Input/Output Pin Assignments . . . 204
Read and Write Operations . . . 204
Inference . . . 205
Instantiation Template . . . 205
BRAM72K_SDP . . . 206
Parameters . . . 207
Ports . . . 210
Memory Organization and Data Input/Output Pin Assignments . . . 212
Read and Write Operations . . . 214

Timing Diagrams . . . 216
Memory Initialization . . . 217
ECC Modes of Operation . . . 219
Using BRAM72K_SDP as a Read-Only Memory (ROM) . . . 220
Advanced Modes . . . 220
Inference . . . 221
Instantiation Template . . . 221

LRAM2K_FIFO . . . 222
Parameters . . . 223
Ports . . . 224
Memory Organization and Data Input/Output Pin Assignments . . . 225
Read and Write Operations . . . 225
Inference . . . 226
Instantiation Template . . . 226

LRAM2K_SDP . . . 227
Parameters . . . 228
Ports . . . 229
Memory Organization and Data Input/Output Pin Assignments . . . 230
Read and Write Operations . . . 230
Memory Initialization . . . 231
Using LRAM2K_SDP as a Read-Only Memory (ROM) . . . 231
Inference . . . 231
Instantiation Template . . . 232

Chapter - 7: Network-on-Chip (NOC) Primitives . . . 233

NAP_AXI_MASTER . . . 233
Parameters . . . 233
Ports . . . 234

NAP_AXI_SLAVE . . . 236
Parameters . . . 236
Ports . . . 237
Instantiation Template . . . 239

NAP_HORIZONTAL . . . 240
Parameters . . . 240
Ports . . . 240

NAP_VERTICAL . . . 241
Parameters . . . 241
Ports . . . 242

Revision History 243

Chapter - 1: Introduction

The Achronix Speedster7t macro 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 64)
- Speedster7t Clock Functions (see page 70)
- Speedster7t Arithmetic and DSP (see page 79)
- Speedster7t Memories (see page 200)
- Speedster7t Network on Chip Primitives (see page 233)
- Speedster7t IP Component Library User Guide Revision History (see page 243)

ACX_ Prefix

All Achronix silicon primitives start with `ACX_` as their formal name — when directly instantiating any primitive, 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 primitives 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, ports and parameters to be masked off, reducing the scope for error when directly instantiating.

For readability, components in this user guide are listed according to their root name with the `ACX_` prefix removed. However, all instantiation templates use the `ACX_` prefix to indicate the correct form of direct instantiation.

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 primitives required will be the sum of these instances.

Chapter - 2: Fabric Architecture

Introduction

The Speedster7t 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.

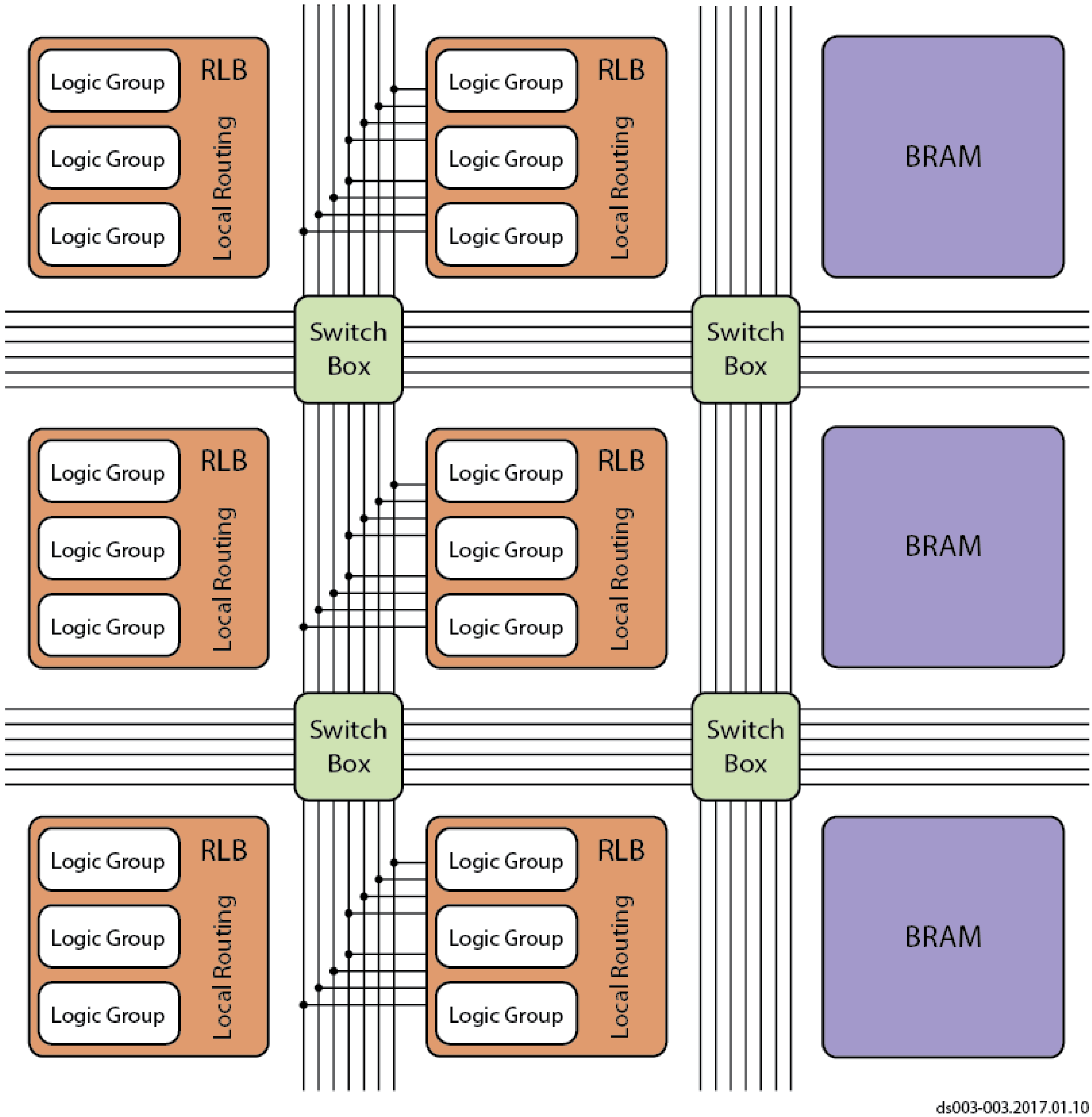

Figure 1: ***Speedster7t Fabric Floorplan***

RLB6

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

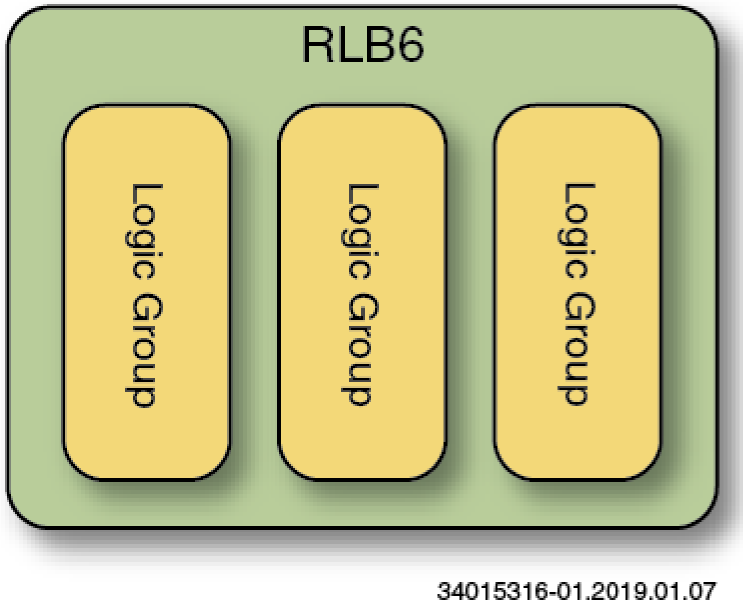

Figure 2: *RLB6 Block Diagram*

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 RLB6:

- 8-to-1 MUX with single-level delay (using LUT6s rather than dedicated MUX4 or MUX8 logic)
- 8-bit ALU for adders, counters, and comparators
- MAX function that efficiently compares two 8-bit numbers and chooses the max or min result
- Dedicated connections for high-efficiency shift registers
- Multiplier LUT (MLUT) mode for efficient multipliers

The figure below provides details on the circuitry inside a single logic group.

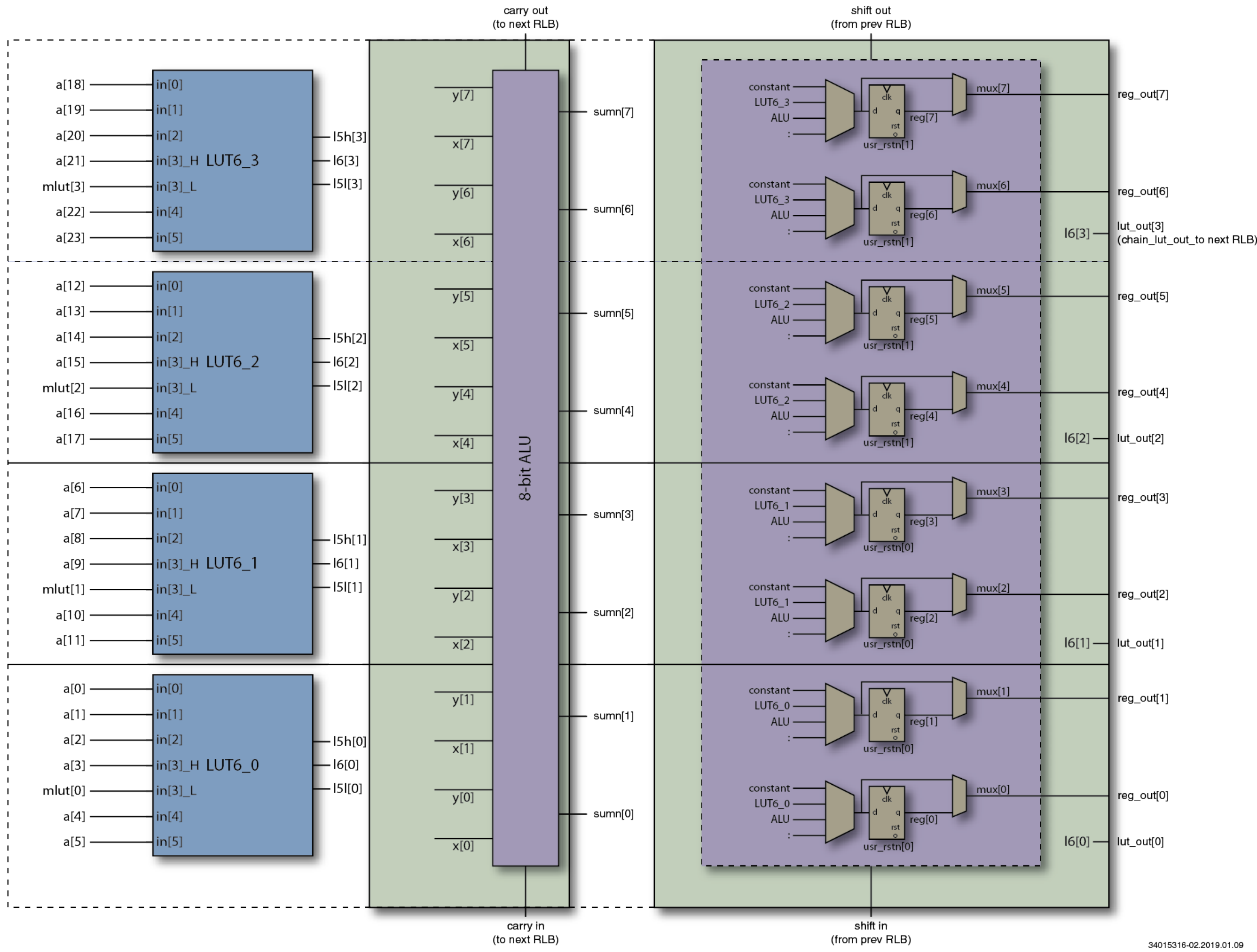

Figure 3: *Logic Group Details*

MLUT Mode

The RLB6 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 RLB6 can perform a 6 × 8 multiply.

> **Note**
>
> MLUT mode is supported by the MLUT generator 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.

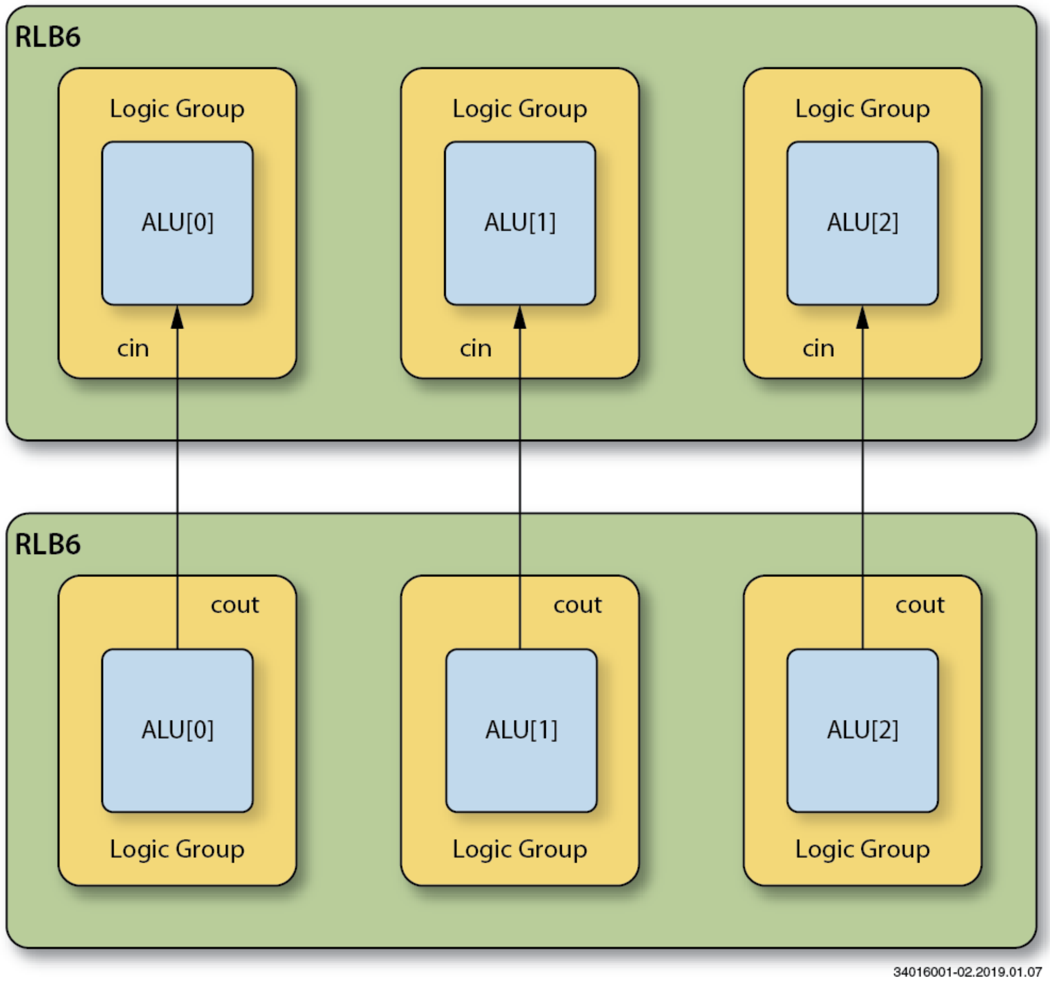

Figure 4: *ALU Carry Chain Routing*

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.

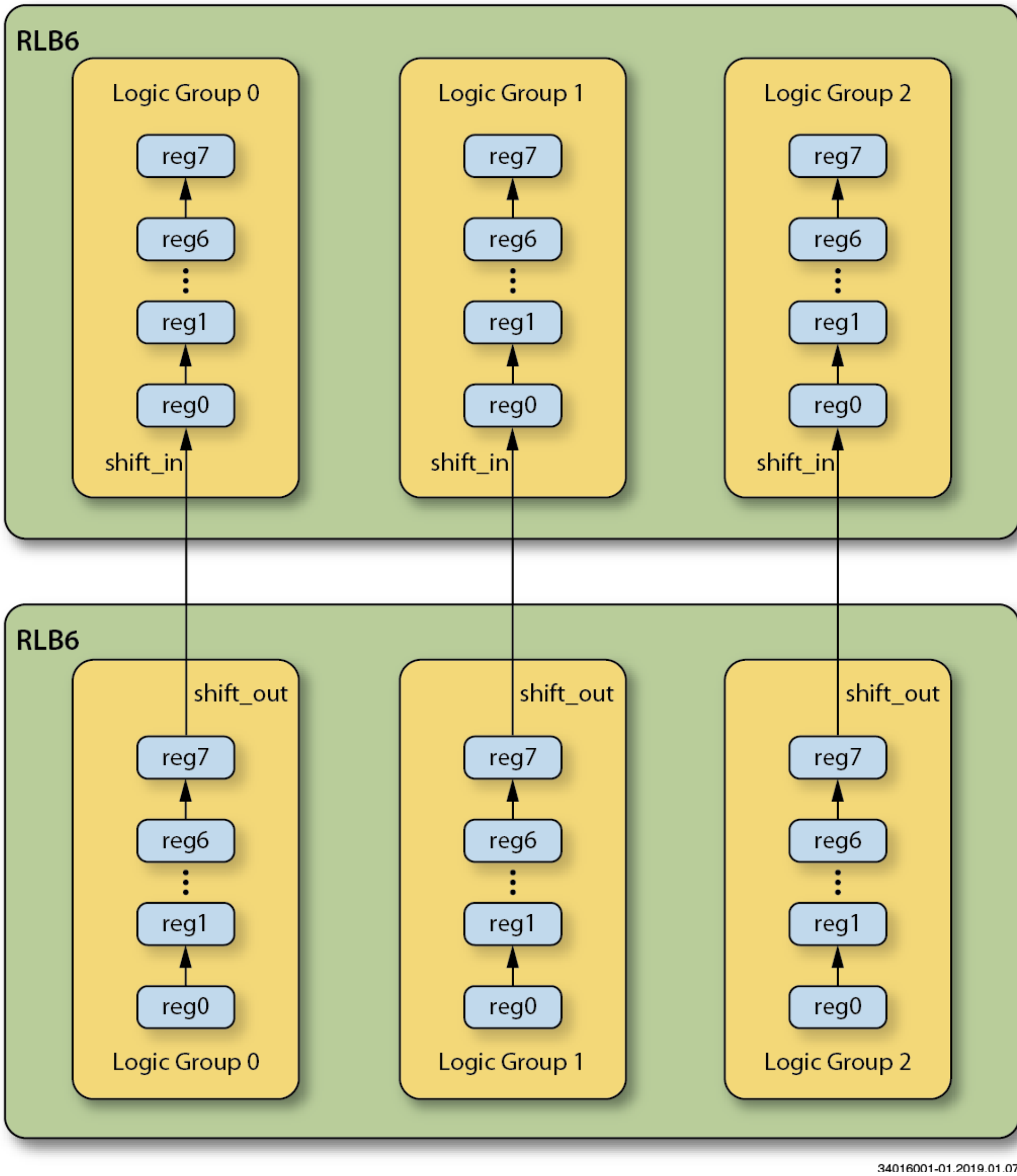

Figure 5: *Shift Register Routing*

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.

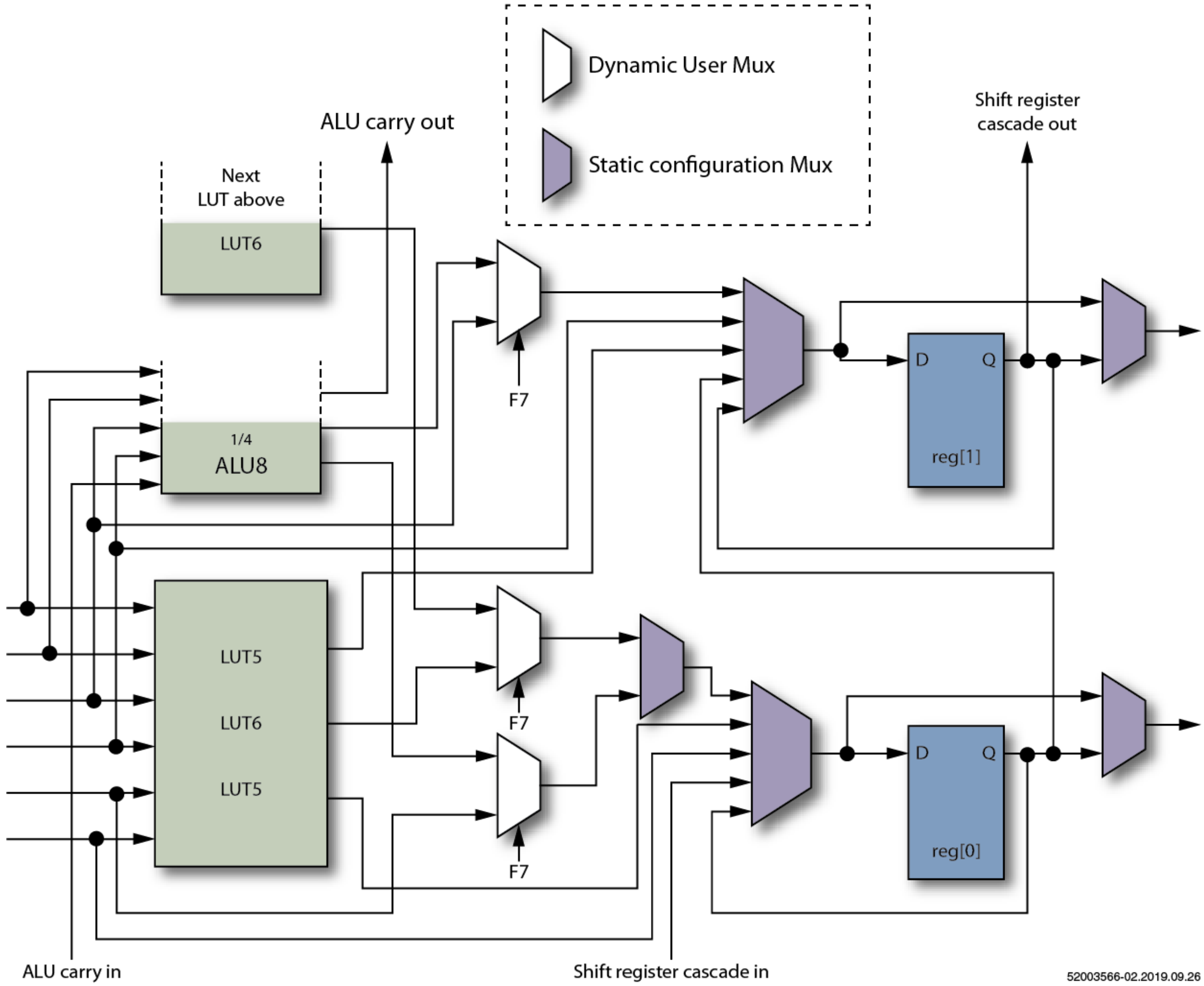

Figure 6: ***One-Fourth of a Logic Group (Connection Detail)***

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, users should try to 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 (LUT6)

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`.

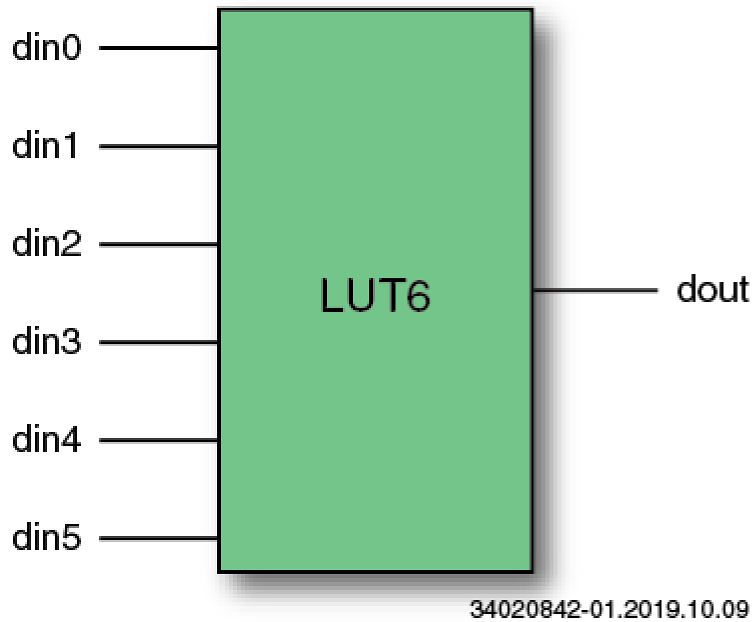

Figure 7: *Logic Symbol*

Ports

Table 2: *Port 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}.

Parameters

Table 3: *Parameters*

Parameter	Defined Values	Default Value	Description
lut_function	64-bit hexadecimal value	64’h0	The lut_function parameter defines the value on the dout output of the LUT6 as detailed in function table (see page 18). The default value of the lut_function parameter is 64’h0.

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 (LUT5x2)

LUT5x2 implements dual LUT5 lookup tables with data inputs (`din0` – `din5`) and data output (`lut5ldout and lut5hdout`). Each of the outputs is defined by a function which is defined by the 64-bit parameter `lut_function`.

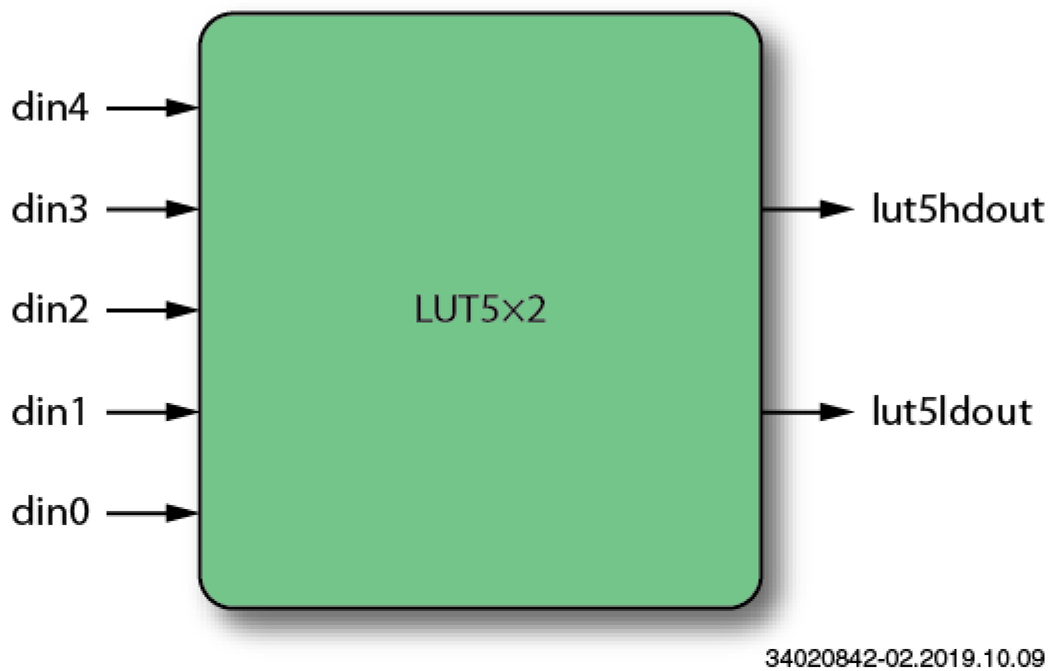

Figure 8: ***Dual LUT5 Lookup Tables***

Ports

Table 5: ***LUT5x2 Port Descriptions***

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

Parameters

Table 6: ***LUT5x2 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 21). The default value of the lut_function parameter is 64’h0.

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	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.

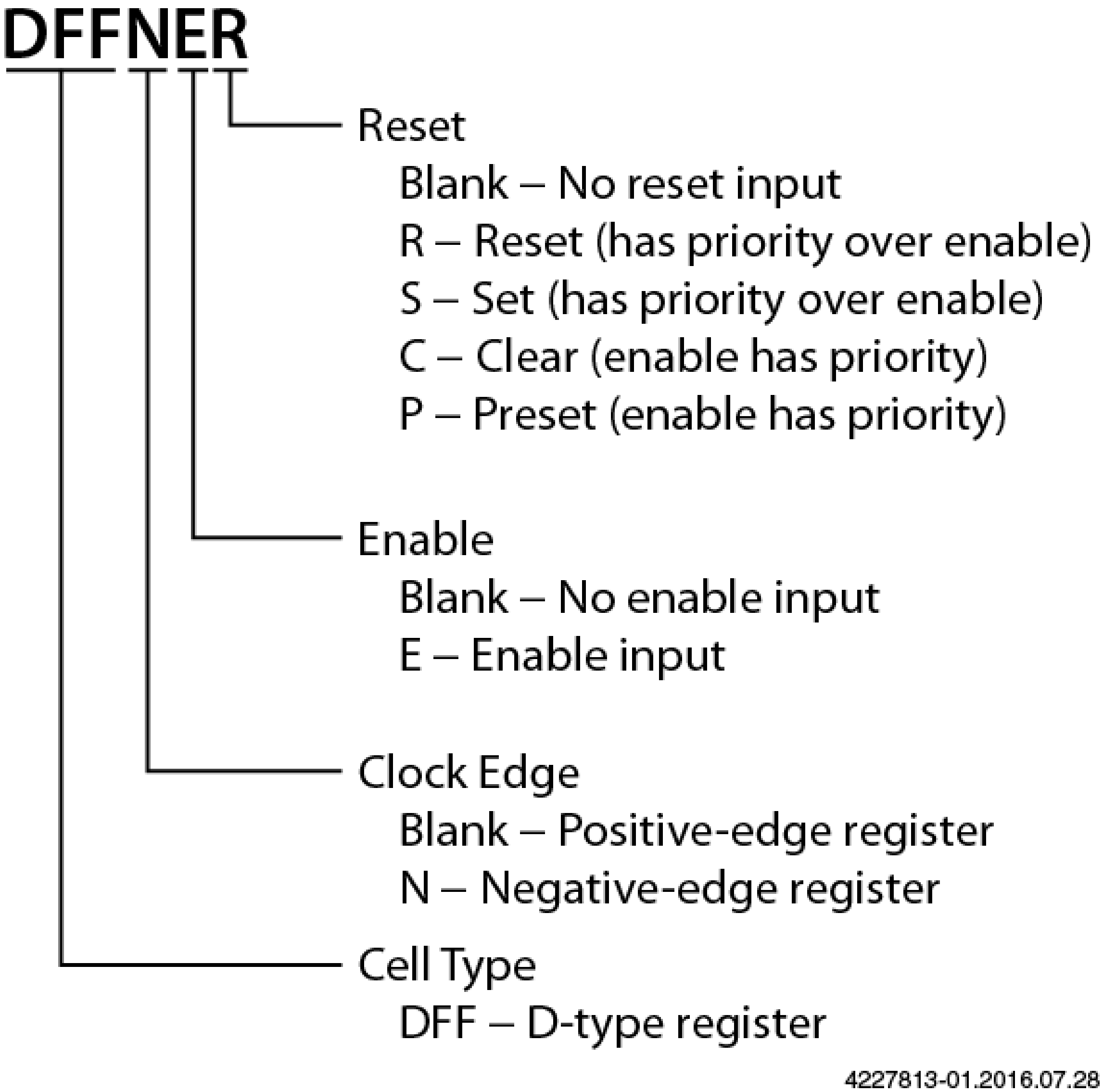

Figure 9: ***Register Naming Convention***

Register Primitives

DFF (Positive Clock Edge D-Type Register)

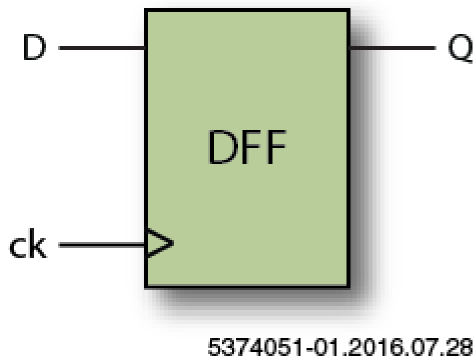

Figure 10: ***Positive Clock Edge D-Type Register***

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: ***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 10: ***Parameters***

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

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
);
```

DFFE (Positive Clock Edge D-Type Register with Clock Enable)

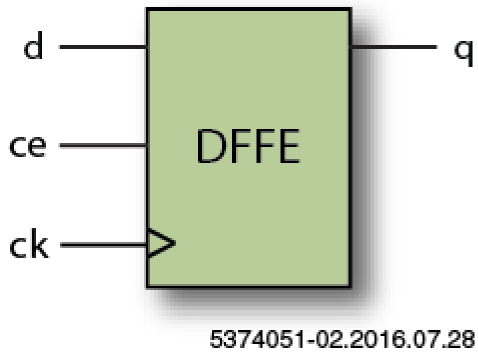

Figure 11: ***Positive Clock Edge D-Type Register with Clock Enable***

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: ***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 13: ***Parameters***

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

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
);
```

DFFER (Positive Clock Edge D-Type Register with Clock Enable and Asynchronous /Synchronous Reset)

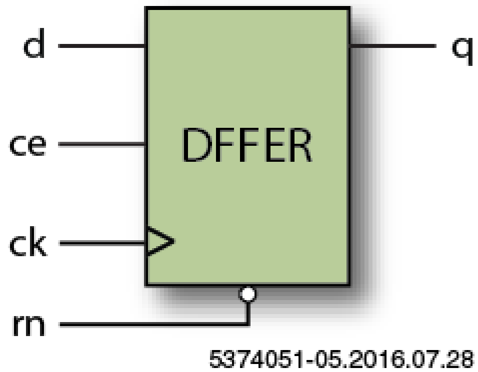

Figure 12: ***Positive Clock Edge D-Type Register with Clock Enable and Asynchronous/Synchronous Reset***

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 DFFER Function Table when sr_assertion = “unclocked” and See DFFER Function Table when sr_assertion = “clocked”. 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: ***Pin Descriptions***

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

Table 16: *Parameters*

Parameter	Defined Values	Default Value	Description
init	1'b0, 1'b1†	1'b0	The init parameter defines the initial value of the output of the DFFER register. This is the value the register takes upon the initial application of power to the FPGA. The default value of the init parameter is 1'b0.
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. The default value of the sr_assertion parameter is "unclocked".

Table Note

† The hardware does not natively support the non-default value of the init parameter when sr_assertion is set to "unclocked". When this combination of parameters is used, ACE transparently resolves the DFF into a logically equivalent macro.

Table 17: *DFFER Function Table when 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: *DFFER Function Table when sr_assertion = "clocked"*

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

Inputs				Output
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
);
```

DFFES (Positive Clock Edge D-Type Register with Clock Enable and Asynchronous /Synchronous Set)

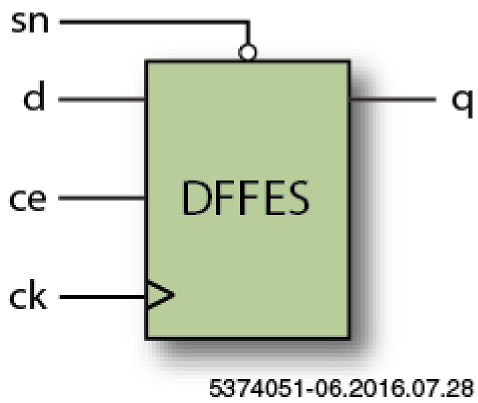

Figure 13: ***Positive Clock Edge D-Type Register with Clock Enable and Asynchronous/Synchronous Set***

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 resonse of the q output in response to the asserted set depends on the value of the sr_assertion parameter and is detailed in Table: DFFES Function Table when sr_assertion = "unclocked" (see page 30) and Table: DFFES Function Table when 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: ***Pin Descriptions***

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

Table 20: ***Parameters***

Parameter	Defined Values	Default Value	Description
init	1'b0†, 1'b1	1'b1	The init parameter defines the initial value of the output of the DFFES register. This is the value the register takes upon the initial application of power to the FPGA. The default value of the init parameter is 1'b1.
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. The default value of the sr_assertion parameter is "unclocked".

Table Note

† The hardware does not natively support the non-default value of the init parameter when sr_assertion is set to "unclocked". When this combination of parameters is used, ACE transparently resolves the DFF into a logically equivalent macro.

Table 21: ***DFFES Function Table when 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: ***DFFES Function Table when 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
);
```

DFFN (Negative Clock Edge D-Type Register)

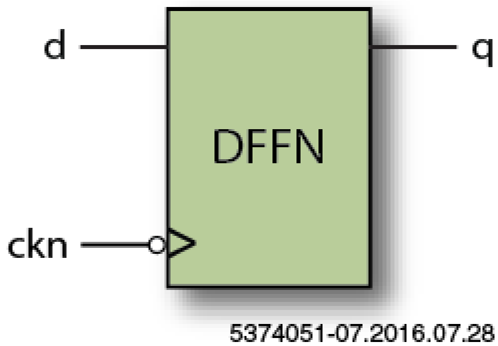

Figure 14: ***Negative Clock Edge D-Type Register***

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: ***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 24: ***Parameters***

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

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
);
```

DFFNER (Negative Clock Edge D-Type Register with Clock Enable and Asynchronous /Synchronous Reset)

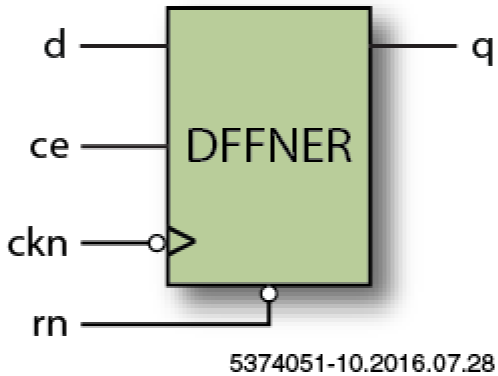

Figure 15: ***Negative Clock Edge D-Type Register with Clock Enable and Asynchronous/Synchronous Reset***

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 resonse of the q output in response to the asserted reset depends on the value of the sr_assertion parameter and is detailed in Table: DFFNER Function Table when sr_assertion = "unclocked" (see page 33) and Table: DFFNER Function Table when sr_assertion = "clocked" (see page 34). 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: ***Pin Descriptions***

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

Table 27: *Parameters*

Parameter	Defined Values	Default Value	Description
init	1’b0, 1’b1†	1’b0	The init parameter defines the initial value of the output of the DFFNER register. This is the value the register takes upon the initial application of power to the FPGA. The default value of the init parameter is 1’b0.
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. The default value of the sr_assertion parameter is “unclocked”.

Table Note

† The hardware does not natively support the non-default value of the init parameter when sr_assertion is set to "unclocked". When this combination of parameters is used, ACE transparently resolves the DFF into a logically equivalent macro.

Table 28: *DFFNER Function Table when 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: *DFFNER Function Table when sr_assertion = “clocked”*

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

Inputs				Output
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
);
```

DFFNES (Negative Clock Edge D-Type Register with Clock Enable and Asynchronous /Synchronous Set)

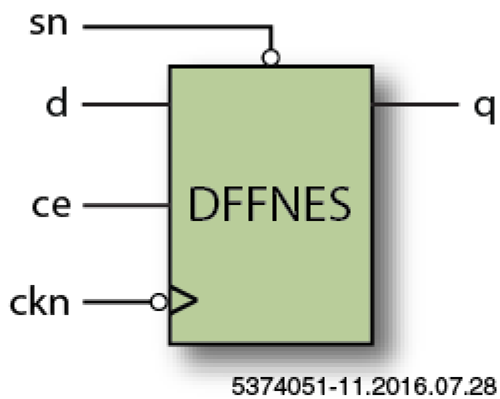

Figure 16: ***Negative Clock Edge D-Type Register with Clock Enable and Asynchronous/Synchronous Set***

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 resonse of the q output in response to the asserted set depends on the value of the sr_assertion parameter and is detailed in Table: DFFNES Function Table when sr_assertion = "unclocked" (see page 38) and Table: DFFNES Function Table when sr_assertion = "clocked" (see page 39). 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: ***Pin Descriptions***

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

Table 31: ***Parameters***

Parameter	Defined Values	Default Value	Description
init	1’b0†, 1’b1	1’b1	The init parameter defines the initial value of the output of the DFFNES register. This is the value the register takes upon the initial application of power to the FPGA. The default value of the init parameter is 1’b1.
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. The default value of the sr_assertion parameter is “unclocked”.

Table Note

† The hardware does not natively support the non-default value of the init parameter when sr_assertion is set to "unclocked". When this combination of parameters is used, ACE transparently resolves the DFF into a logically equivalent macro.

Table 32: ***DFFNES Function Table when 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: ***DFFNES Function Table when sr_assertion = “clocked”***

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

Inputs				Output
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
);
```

DFFNR (Negative Clock Edge D-Type Register with Asynchronous Reset)

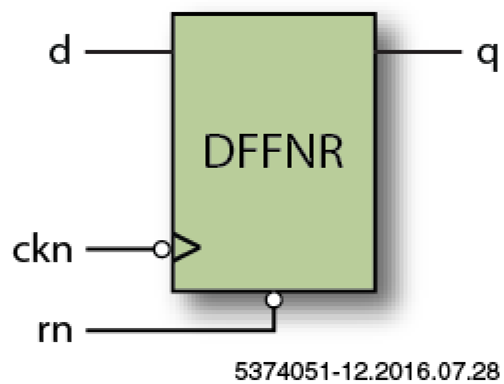

Figure 17: ***Negative Clock Edge D-Type Register with Asynchronous Reset***

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 all other inputs when it is asserted low and sets the data output low. If the asynchronous 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: ***Pin Descriptions***

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

Table 35: ***Parameters***

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

Parameter	Defined Values	Default Value	Description
sr_assertion	"unclocked", "clocked"	"unclocked"	signal, where the q output is set to one at the next rising edge of the clock. The default value of the sr_assertion parameter is "unclocked".

Table Note

† The hardware does not natively support the non-default value of the init parameter when sr_assertion is set to "unclocked". When this combination of parameters is used, ACE transparently resolves the DFF into a logically equivalent macro.

Function Table when sr_assertion = "unclocked'

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

Table 36: ***Function Table when 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
);
```

DFFNS (Negative Clock Edge D-Type Register with Asynchronous Set)

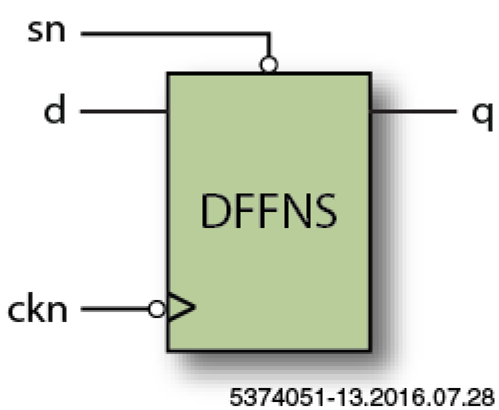

Figure 18: ***Negative Clock Edge D-Type Register with Asynchronous Set***

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 all other inputs when it is asserted low and sets the data output high. If the asynchronous 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 37: ***Pin Descriptions***

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

Table 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. The default value of the init parameter is 1'b1.
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. The default value of the sr_assertion parameter is "unclocked".

Table Note

† The hardware does not natively support the non-default value of the init parameter when sr_assertion is set to "unclocked". When this combination of parameters is used, ACE transparently resolves the DFF into a logically equivalent macro.

Table 39: ***Function Table when sr_assertion = "unclocked'***

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

Inputs			Output
1	1	↓	1

Table 40: ***Function Table when 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
);
```

DFFR (Positive Clock Edge D-Type Register with Asynchronous Reset)

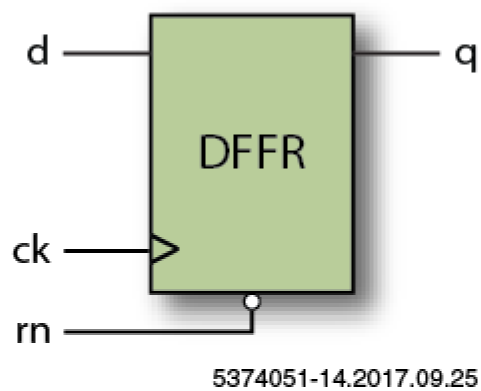

Figure 19: ***Positive Clock Edge D-Type Register with Asynchronous Reset***

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 all other inputs when it is asserted low and sets the data output low. If the asynchronous 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**
>
> Users may see references 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 41: ***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 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. The default value of the init parameter is 1’b0.
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. The default value of the sr_assertion parameter is “unclocked”.

Table Note

† The hardware does not natively support the non-default value of the init parameter when sr_assertion is set to "unclocked". When this combination of parameters is used, ACE transparently resolves the DFF into a logically equivalent macro.

Table 43: ***Function Table when sr_assertion = “unclocked’***

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

Table 44: ***Function Table when sr_assertion = “clocked’***

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

Inputs			Output
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
);
```

DFFS (Positive Clock Edge D-Type Register with Asynchronous Set)

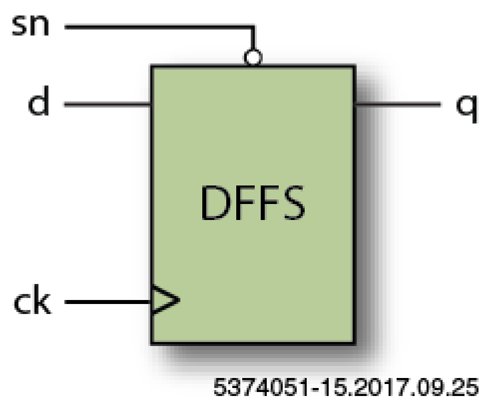

Figure 20: ***Positive Clock Edge D-Type Register with Asynchronous Set***

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 all other inputs when it is asserted low and sets the data output high. If the asynchronous 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**
>
> Users may see references 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 45: ***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 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. The default value of the init parameter is 1’b1.

Parameter	Defined Values	Default Value	Description
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. The default value of the sr_assertion parameter is “unclocked”.

Table Note

† The hardware does not natively support the non-default value of the init parameter when sr_assertion is set to "unclocked". When this combination of parameters is used, ACE transparently resolves the DFF into a logically equivalent macro.

Table 47: ***Function Table when sr_assertion = “unclocked’***

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

Table 48: ***Function Table when 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.

DFFNEP (Negative Clock Edge D-Type Register with Clock Enable and Synchronous Preset)

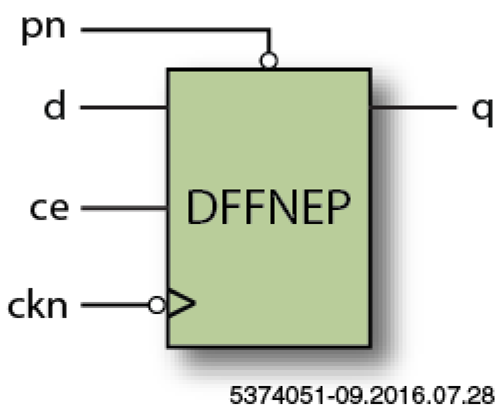

Figure 21: ***Negative Clock Edge D-Type Register with Clock Enable and Synchronous Preset***

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 49: ***Pin Descriptions***

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

Table 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. The default value of the init parameter is 1'b1.

Table 51: *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
);
```

DFFEC (Positive Clock Edge D-Type Register with Clock Enable and Synchronous Clear)

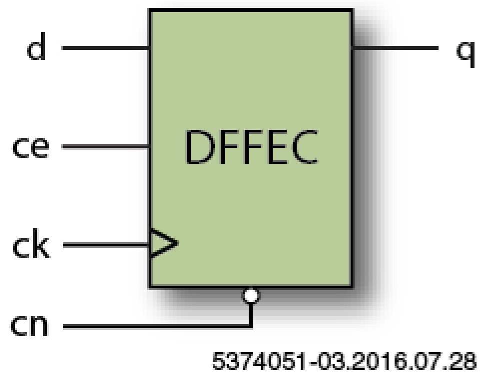

Figure 22: ***Positive Clock Edge D-Type Register with Clock Enable and Synchronous Clear***

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 synchronout 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 52: ***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 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. The default value of the init parameter is 1'b0.

Table 54: ***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
);
```

DFFEP (Positive Clock Edge D-Type Register with Clock Enable and Synchronous Preset)

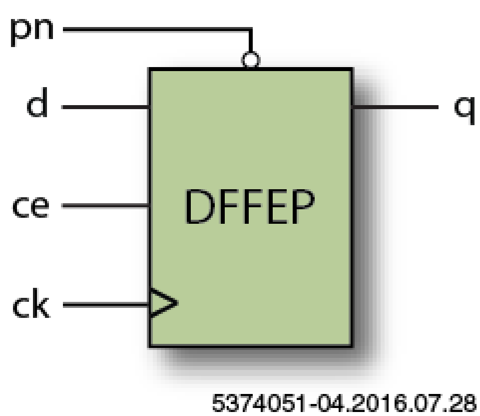

Figure 23: ***Positive Clock Edge D-Type Register with Clock Enable and Synchronous Preset***

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 55: ***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.

Name	Type	Description
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 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. The default value of the init parameter is 1'b1.

Table 57: *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
);
```

DFFNEC (Negative Clock Edge D-Type Register with Clock Enable and Synchronous Clear)

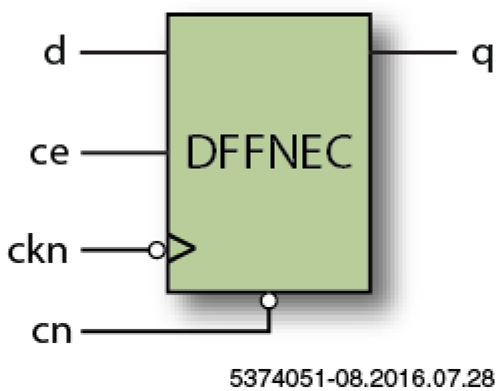

Figure 24: ***Negative Clock Edge D-Type Register with Clock Enable and Synchronous Clear***

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 58: ***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 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. The default value of the init parameter is 1'b0.

Table 60: ***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

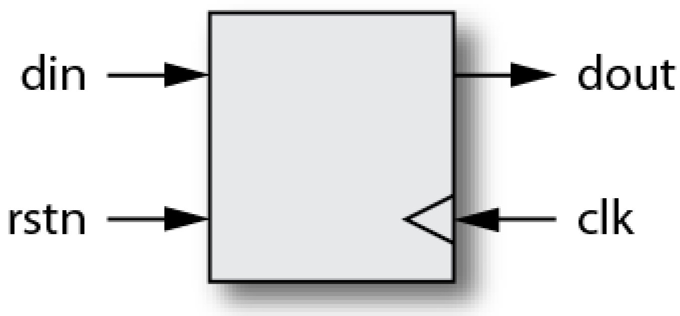

Figure 25: *ACX_SYNCHRONIZER Logic Symbol*

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 may 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), the user has to 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 61: *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.

Name	Type	Clock Domain	Description
clk	Input		Clock reference. The dout signal is by synchronized to the rising edge of this clock.
dout	Output	clk	Data output.

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 eFPGA.

Table 63: *Function Table*

Inputs		Output
din	clk	Dout
0	↑↑	0
1	↑↑	1

Using ACX_SYNCHRONIZER to Synchronize Reset

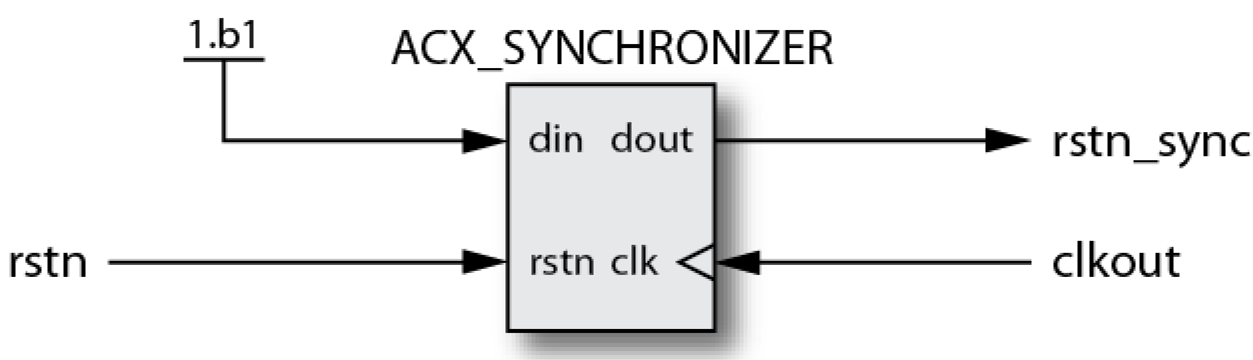

Figure 26: *ACX_SYNCHRONIZER Logic Symbol*

An instance of the ACX_SYNCHRONIZER module can also be used to synchronize reset signals. In this case, the active-low 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, at which point the output is de-asserted, synchronous to the destination clock.

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

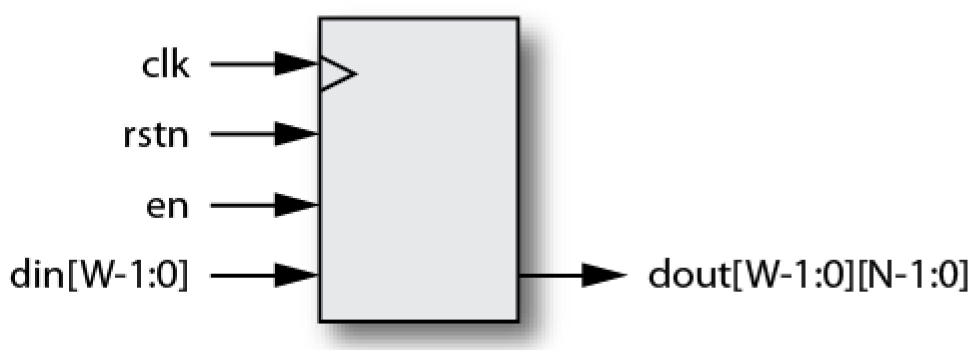

Figure 27: *ACX_SHIFTREG Logic Symbol*

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 64: *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 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 n^{th} entry in the TAPS array specified the latency of the n^{th} 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.
MODE	[0,1]	[0, 0,…]	Array of modes. Setting the n^{th} 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: *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

CLKDIV (Clock Divider)

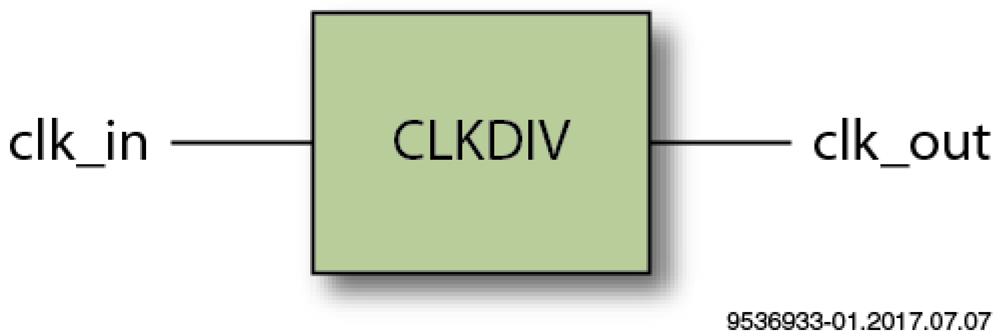

Figure 28: *CLKDIV Logic Symbol*

The 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.

Table 67: *Pin Descriptions*

Name	Type	Description
clk_in	Input	Input clock to be divided.
clk_out	Output	Divided clock output.

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 how many inputs clock cycles the output clock is delayed by.

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

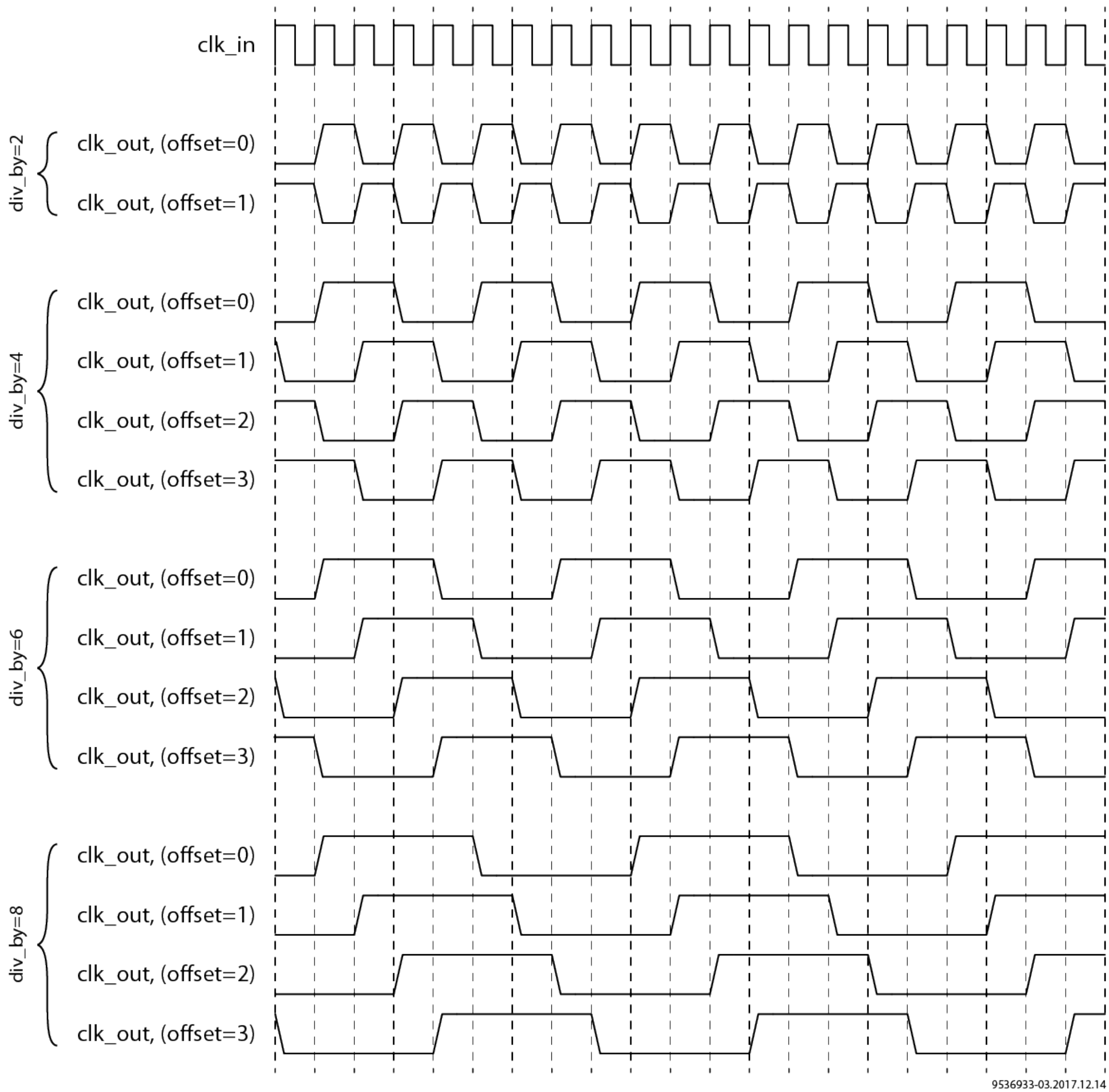

Figure 29: ***Timing Diagram***

Constraints

The CLKDIV component does *not* propagate the input clock frequency from `clk_in` to `clk_out`. Therefore, the user must specify suitable constraints for `clk_out` 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
);
```

CLKGATE (Clock Gate)

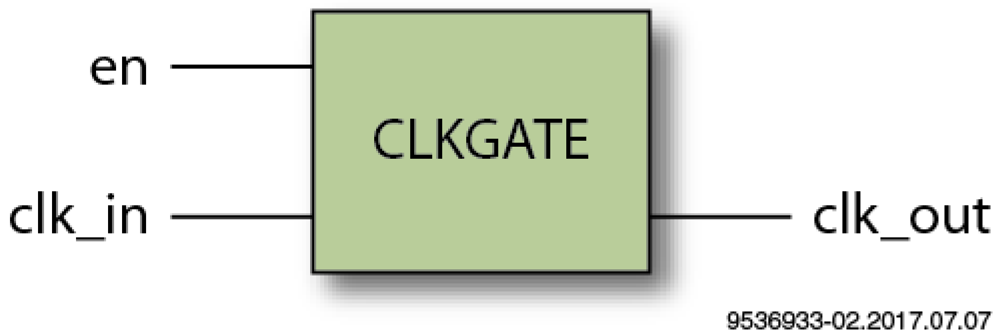

Figure 30: ***CLKGATE Logic Symbol***

The CLKGATE component implements a clock gate component that allows the output to toggle only when the input en is asserted high. The CLKGATE component turns off the clock only when the clock input is '0', guaranteeing that the output is glitchless. In other words, the output is guaranteed to never have a pulse width narrower in time than the input pulse width.

Note

When simulating the 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 69: ***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.

The following timing diagrams illustrates the behavior of the CLKGATE component.

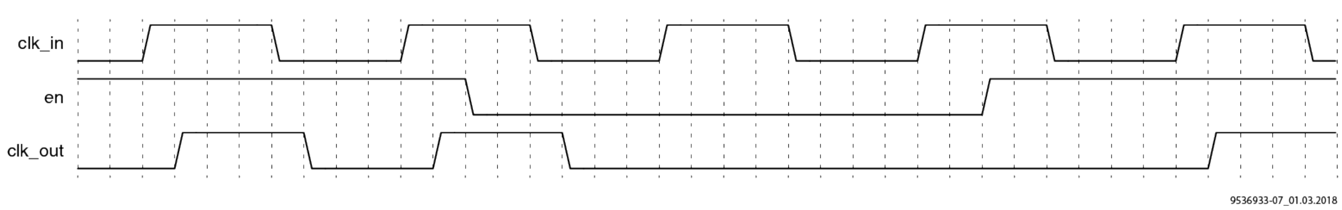

Figure 31: ***CLKGATE Timing Diagram***

Constraints

The CLKGATE component does *not* propagate the input clock frequency from `clk_in` to `clk_out`. Therefore, the user must specify suitable constraints for `clk_out` 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 for a clock gate with an input from a clock port "clk_in",
which generates an internal net named "clk_gated".
create_generated_clock -name gated_clk -source [get_ports clk_in] -divide_by 1 -phase 0 [get_nets
clk_gated]
```

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
);
```

CLKSWITCH (Clock Switch)

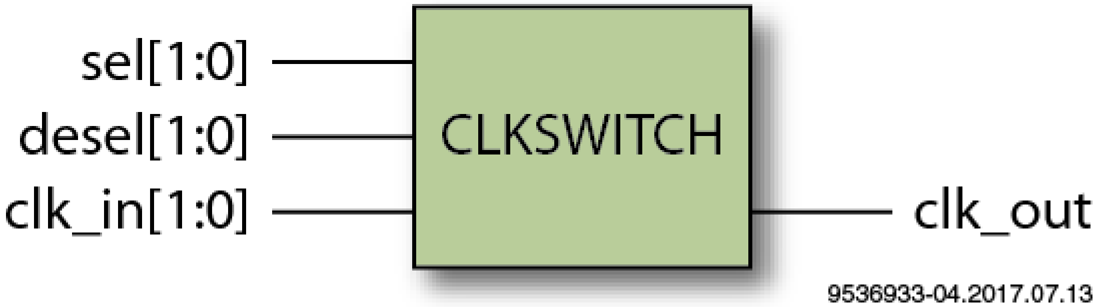

Figure 32: *CLKSWITCH Logic Symbol*

The 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.

True glitchless behavior is enabled when the SYNCHRONIZE_SEL parameter has a value of '0'. In this case, the input sel for each clock is synchronized to the rising and then falling edge of the clock that it is selecting. Setting SYNCHRONIZE_SEL to 1 (or 2) disables the synchronization, such that the input sel switches between clocks immediately. This mode does not guarantee that the output is is free of glitches and short pulses.

To ensure glitchless operation, set SYNCHRONIZE_SEL = 0 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 deasserting the `sel[]` input bit for that clock will 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 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.

> **Note**
>
> Using `desel[]` when the input clock is toggling may cause a glitch or runt pulse on the output.

Table 70: *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 will stop toggling. Asserting both bits of sel[] at the same time results in unpredictable output.

Name	Type	Description
desel[1:0]	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. **Note** Using desel[] to deselect a clock while it is toggling can cause a glitch on the output clock.
clk_in[1:0]	Input	Input clocks.
clk_out	Output	Output clock.

Table 71: *Parameter Descriptions*

Parameter	Defined Values	Default Value	Description
PRESEL	2'b00, 2'b01, 2'b10	2'b00	The PRESEL parameter is used to determine the operation of the startup 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 synchronization stages are used to synchronize the inputs sel[1:0]: • 1, 2 – Disables sel[] synchronization, such that changing the input sel causes the CLKSWITCH to switch between clocks immediately. This mode does not guarantee that the output is free of glitches and short pulses. • 0 – Synchronizes the input sel to the rising and then falling edge of the selected clock. Use this mode to ensure glitchless operation.

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

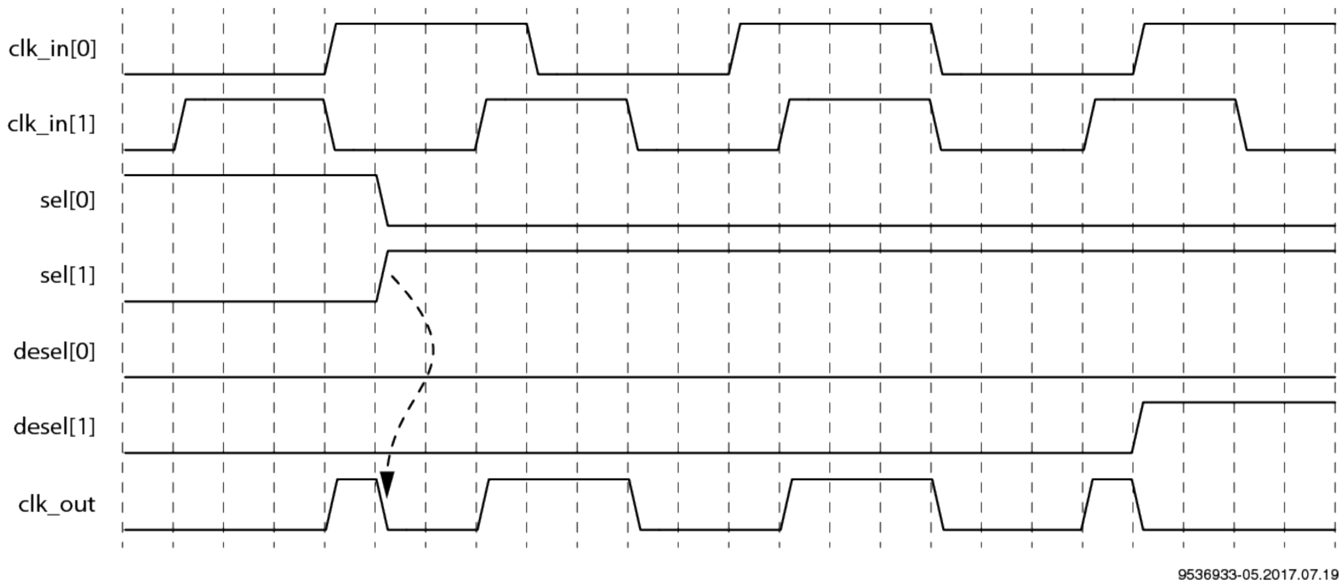

Figure 33: *SYNCHRONIZE_SEL=1,2 Timing Diagram*

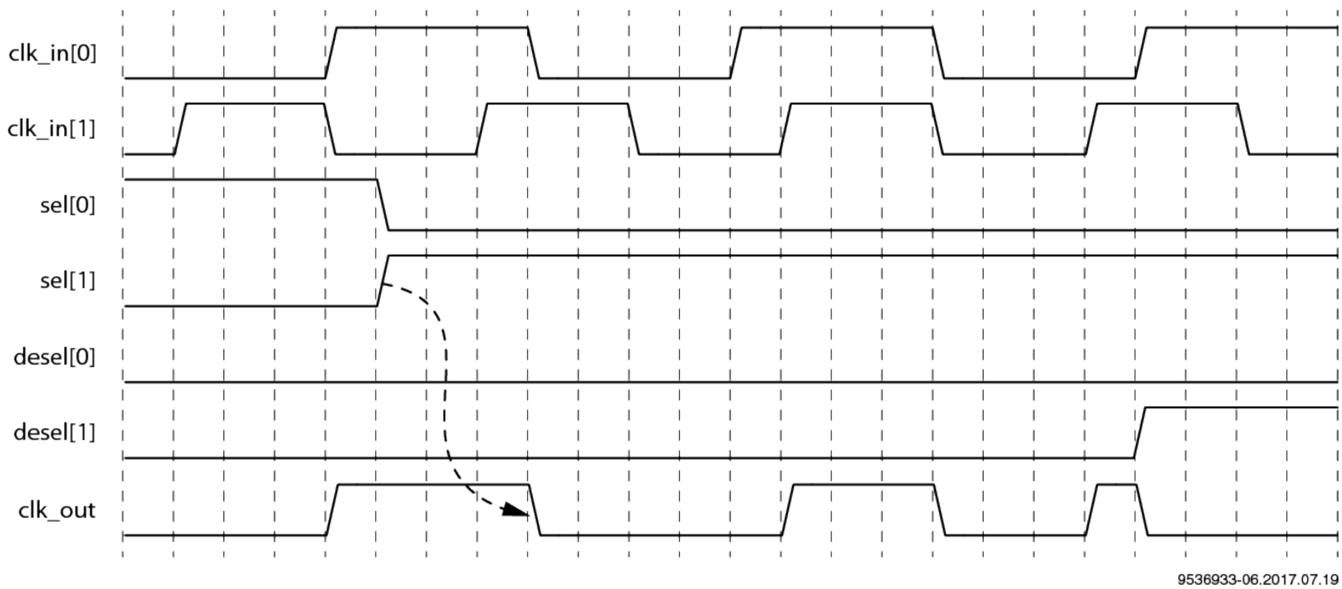

Figure 34: *SYNCHRONIZE_SEL= 0 Timing Diagram*

Constraints

The CLKSWITCH component does *not* propagate the input clock frequency from either `clk_in` to `clk_out`. Therefore, the user must specify suitable constraints for `clk_out` to ensure that correct timing is applied. It is recommended that the faster of clk_in[0] or clk_in[1] is applied to `clk_out`. These constraints should be present in both the Synplify Pro and ACE constraint files.

```
# Example of constraint required with for a clock switch with inputs from two clock ports "clk_slo
w" and "clk_fast", which generates an internal net named "clk_switched".
create_generated_clock -name switched_clk -source [get_ports clk_fast] -divide_by 1 -phase 0
[get_nets clk_switched]
```

Instantiation Templates

Verilog

```
ACX_CLKSWITCH #(
   .SYNCHRONIZE_SEL ( 2 ),
   .PRESEL             ( 2'b01 )
) 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 => 2 ,
    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 user can make the correct choices with regard to math they wish to perform, 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 72: *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(†)
–24	Signed magnitude	1	001_1000

Note

† 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 that will be directed to each respective multiplier. Within a group, each named format is followed by it's token name; these token names are used by the MLP parameters to configure the modes of each multiplier.

INT16

Table 73: ***Signed 16 Bit (Signed16)***

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

Table 74: ***Unsigned 16 Bit (Unsigned16)***

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

Table 75: ***Signed Magnitude 16 Bit (SMAG16)***

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

INT8

Table 76: ***Signed 8 Bit (Signed8)***

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

Table 77: ***Unsigned 8 Bit (Unsigned8)***

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

Table 78: ***Signed Magnitude 8 Bit (SMAG8)***

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

INT7

Table 79: *Signed 7 Bit (Signed7)*

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

Table 80: *Unsigned 7 Bit (Unsigned7)*

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

Table 81: *Signed Magnitude 7 Bit (SMAG7)*

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

INT6

Table 82: *Signed 6 Bit (Signed6)*

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

Table 83: *Unsigned 6 Bit (Unsigned6)*

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

Table 84: *Signed Magnitude 6 Bit (SMAG6)*

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

INT4

Table 85: *Signed 4 Bit (Signed4)*

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

Table 86: ***Unsigned 4 Bit (Unsigned4)***

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

Table 87: ***Signed Magnitude 4 Bit (SMAG4)***

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

INT3

Table 88: ***Signed 3 Bit (Signed3)***

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

Table 89: ***Unsigned 3 Bit (Unsigned3)***

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

Table 90: ***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 91: ***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 Notes**
>
> 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 done 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 92: ***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 × $2^{(exp-bias)}$ with the appropriate sign. Here, 1. mantissa starts with the hidden 1 and has the mantissa as binary fraction.

Table 93: *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 may 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 may give a slightly different result than if the operation had been done with 16 bits only.

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

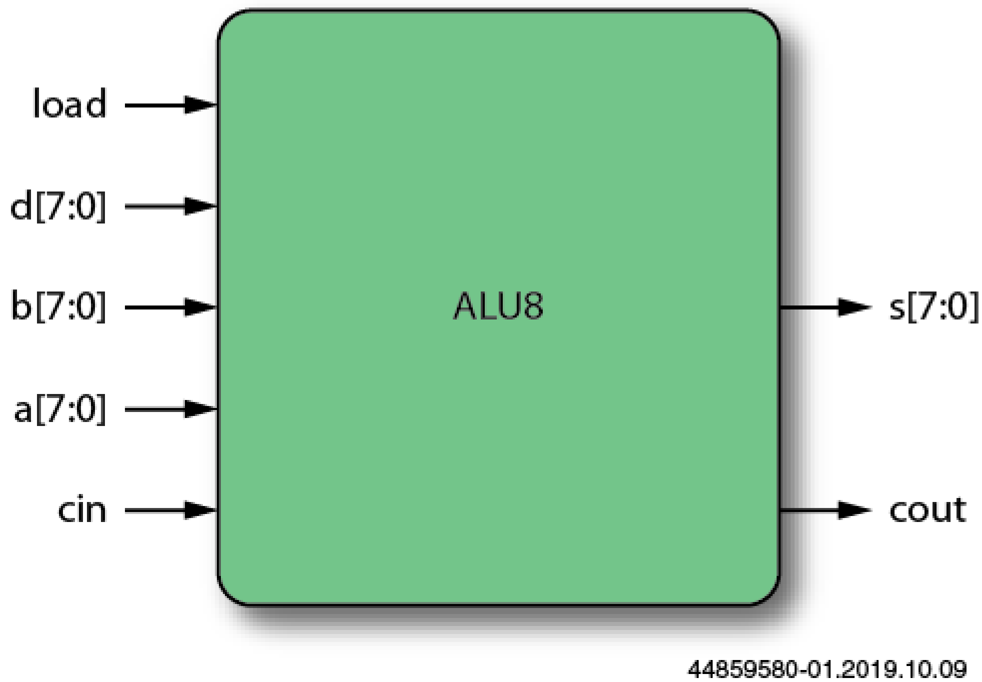

Figure 35: ***Eight-Input Adder/Subtractor with Programmable Load***

Description

The ALU8 implements either a 8-bit adder or 8-bit subtractor with adder/subtractor inputs (`a[7:0]`, `b[7:0]`), load value (`d[7:0]`), Load enable (`load`) and carry-in (`cin`) inputs. It generates the sum/difference (`s[7:0]`) and carry-out (`cout`) outputs.

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

Multiple 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 ALU8 is configured as an adder or subtractor is determined by the value of the invert_b parameter.

Ports

Table 94: ***ALU8 Pin Descriptions***

Name	Type	Description
a[7:0]	Input	Data input a. Data input a is 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.

Name	Type	Description
b[7:0]	Input	Data input b. Data inputs b is a 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 the value that 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 high set 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, the s[7:0] output will reflect the sum of the a, b, and cin inputs if the load input is low. If the invert_b parameter is set to 1'b1, the s[7:0] output will reflect the difference of the a, b, cin inputs if the load input is low.
cout	Output	Carry-out output. The cout is set high during an add when the s[7:0] output overflows.

Parameters

Table 95: *ALU8 Parameters*

Parameter	Defined Values	Default Value	Description
invert_b	1'b0, 1'b1	1'b0	The invert_b parameter defines if the ALU8 functions as an adder or a subtractor. Setting the invert_b parameter to 1'b0 configures the ALU8 to perform two's complement addition of a[7:0] + b[7:0] + cin. Setting the invert_b parameter to 1'b1 configures the ALU8 to invert the b[7:0] input so that the two's complement subtraction of a[7:0] - b[7:0] is performed. When subtractions is desired, the cin input must be tied high. When multiple ALU8s are connected to perform higher resolution subtractors, only the cin of the LSB of the subtractor is to be tied high.

Functions

Table 96: *Function Table When 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 97: *Function Table When 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
);
```

MLP72

Arithmetic within the Speedster7t architecture is primarily focused on the machine learning processing block (MLP72). This dedicated silicon block is optimized for artificial intelligence and machine learning (AI/ML) functions.

The machine learning processing block (MLP) is an array of up to 32 multipliers, followed by an adder tree, an accumulator, and a rounding/saturation/normalize block.The MLP also includes two memory blocks, a BRAM72k and LRAM2k, that 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 including integer multiply with optional accumulate, bfloat16 operations, floating point 16, block floating point, and floating point 24. Below is a list of features available with the MLP block:

- Configurable multiply precision and multiplier count
    - Up to 32 multiplies for 4-bit integers or block floating point 12 in a single MLP
    - Up to 16 multiplies for 8-bit integers or block floating point 16 in a single MLP
    - Support for 16-bit integer, Bfloat16, (Brain Float), floating point 16, and floating point 24 in a single MLP
- Multiple number formats (fixed and floating point)
- Deep adder tree and accumulator block
- Multiple rounding and saturation features
- Tightly coupled circular register file (LRAM) 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. There is a normalization block for floating-point and block-floating-point operations. 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.

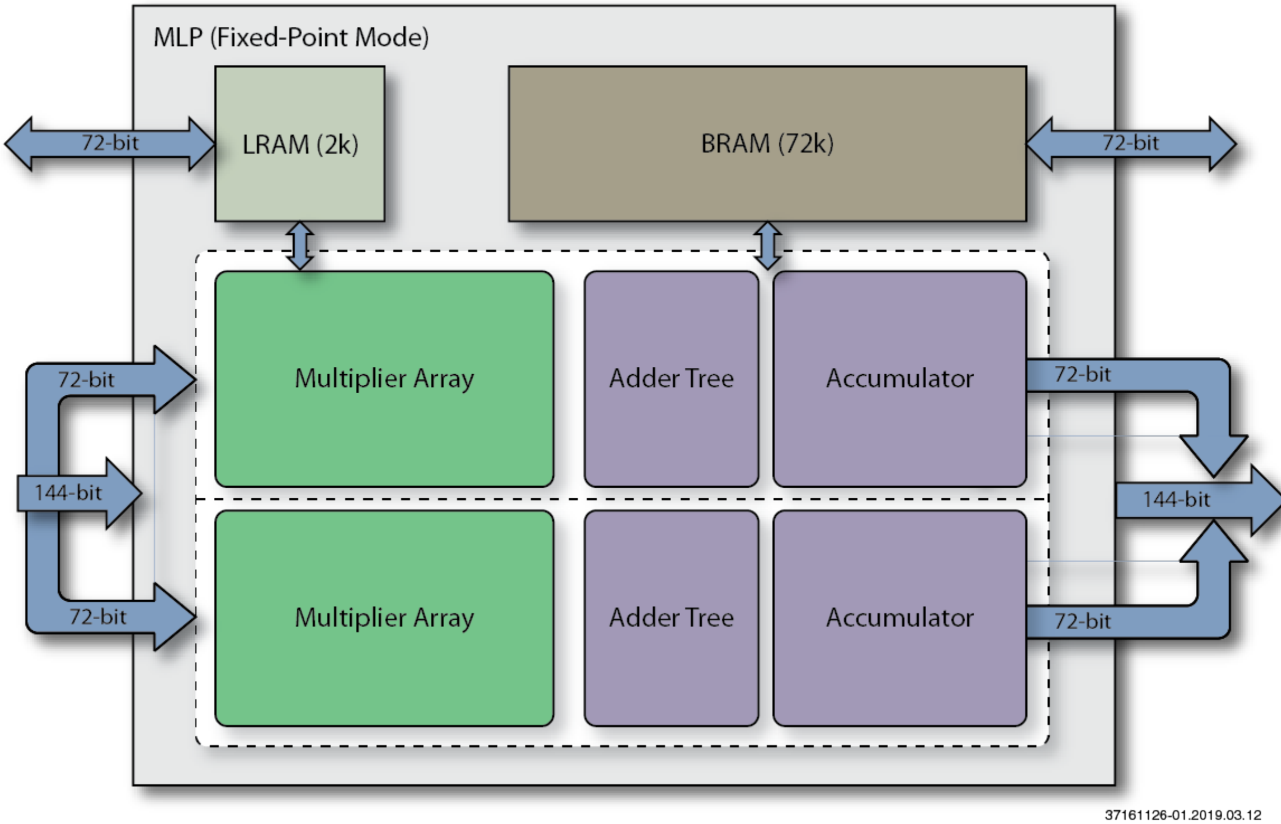

Figure 36: ***MLP Using Fixed-Point Mode***

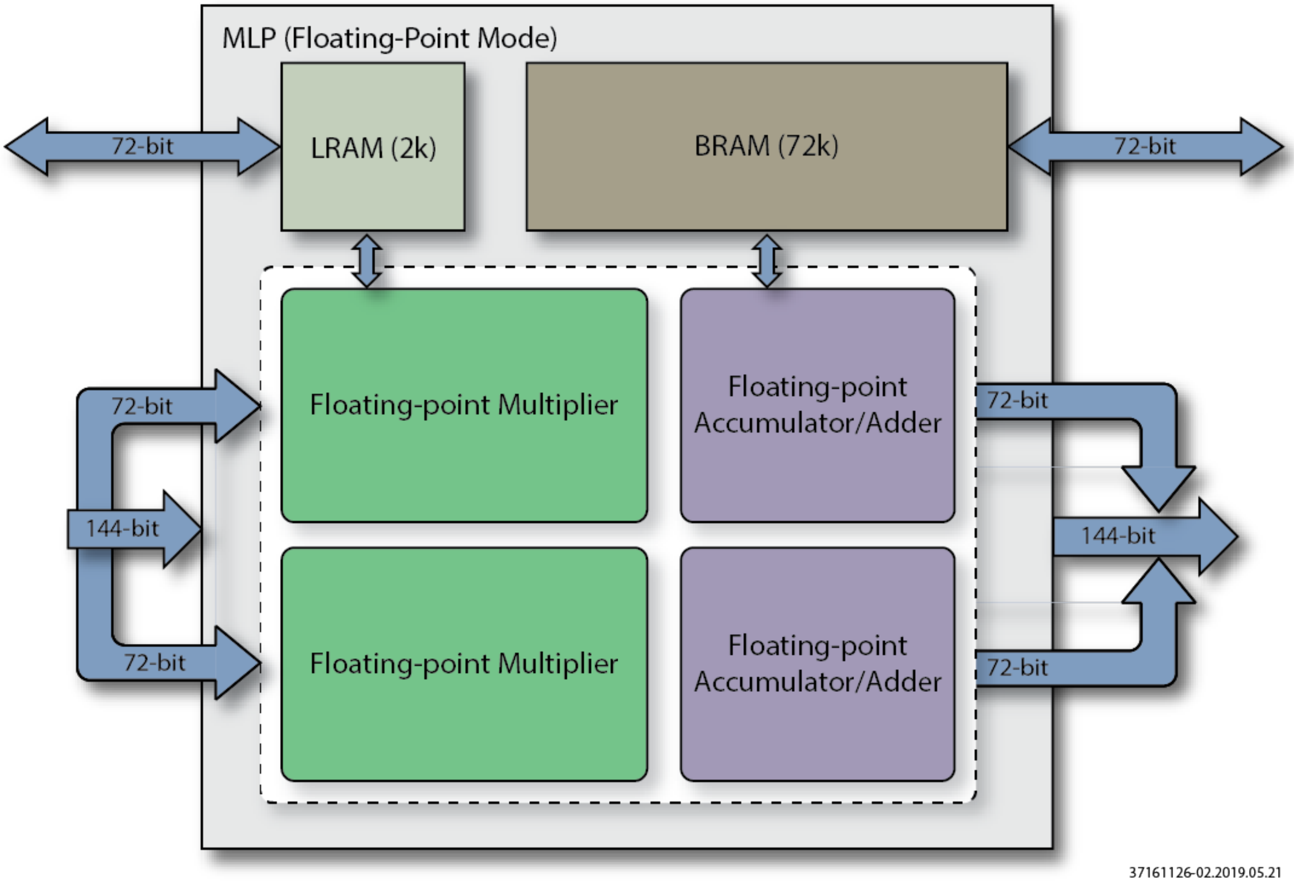

Figure 37: ***MLP Using Floating-Point Mode***

A cascade path allows for the adder tree to extend across multiple MLP blocks in a column without using extra fabric resources, and a data cascade/broadcast path is available to send operands across multiple MLP blocks. Below is a diagram showing the cascade paths across MLPs.

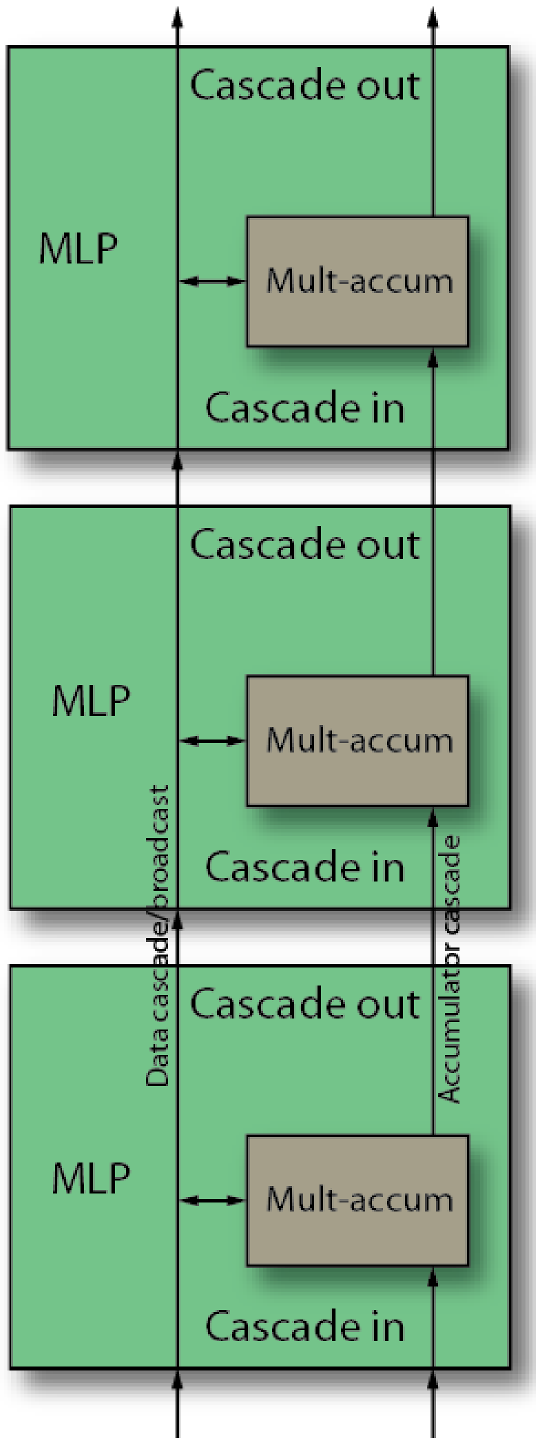

Figure 38: *MLP Cascade Path*

> **Note**
>
> Straight addition within the MLP72 (without a leading multiplication) is not supported.

Numerical Formats

The MLP72 can process the following numerical formats:

Table 98: ***MLP72 Supported Numerical Formats***

	Formats
Integer	int3, int4, int6, int8, int16
Floating Point	fp3, fp4, fp6, fp8, fp16, bfloat16, fp24.

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

Parallel Multiplications

The following table lists the number of multiplies that are supported in the MLP as a function of the data type. All of these numbers assume that one operand for each operation is coming from the BRAM. If both operations are coming from the FPGA fabric, the multiplier count is half of that shown, unless noted otherwise. For Block float operations, the bit width shown is the mantissa width.

Table 99: ***Parallel Multiplication Capabilities***

Data Type	Using only Inputs from FPGA Fabric	Inputs from FPGA Fabric and BRAM
3b Block FP	12	24
3b int	12	24
4b Block FP	8	16
4b int	8	16
6b Block FP	6	12
6b int	6	12
8b Block FP	4	8
8b int	4	8
16b Block FP	1	–
16b int	1	–
BFloat16	1	1
FP16	1	1

Data Type	Using only Inputs from FPGA Fabric	Inputs from FPGA Fabric and BRAM
FP24 (8-bit exponent); rounding on mantissa LSB only		–

Memories

A key feature of the MLP72 is its tight coupling with local memories. 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.

The MLP72 is combined with a Speedster7t BRAM72K_SDP and a Speedster7t LRAM2K_SDP (see page 227) to form a single silicon entity titled BMLP (BRAM and MLP). This tight coupling supports 144-bit paths between the BRAM and the MLP72, allowing for the full set of 16 multipliers to be used in parallel.

Instantiation

Currently it is not possible to infer the full MLP72. In addition, due to the complexity of the full MLP72, Achronix supports, and recommends, the use of the MLP72 via library of macros and primitive functions derived from the full MLP72. This library enables the user to implement complex mathematical functions, all within a single block, with a simplified interface.

If users have particular use cases not covered by the library of MLP72 macros and primitives, then details of the full MLP72 device are provided. Users are recommended to refer to Achronix reference designs for further examples of direct instantiation of the full MLP72 device.

Common Stages

Stages

Due to the complexity of the 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 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. The user is directed to understand each stage thoroughly before configuring it via the various parameters.

The initial overview of the full MLP72 structure focuses on the integer modes. This overview details

- Common Signals (see page 94)
- Input Selection (see page 97)
- Integer Byte Selection (see page 103)
- Integer Multiplier Stage (see page 108)
- Integer Output Stage (see page 112)
- LRAM (see page 117)

Once the user is familiar with the overall MLP72 integer structure and data flow, there are additional sections on floating-point support

- Block Floating Point (see page 125)
- Floating Point (see page 133)

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 MLP72, in these cases, unless indicated otherwise, the user can assume that the other half operates in an identical manner.

Modes

Operation of the 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 will halve or double respectively.

- **By-one mode (×1)** – This mode is when just the four multipliers in the lower half of the MLP72 are in use, mult[3:0]. 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)** – This mode is when all eight of the multipliers in the lower half of the MLP72 are in use, mult[7:0]. This requires each of the A and B input buses to have 64 bits of data, so at least two of the input sources will be 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 MLP72 are used.
- **By-four mode (×4)** – This mode is when all 16 multipliers in the MLP72 are in use; this requires that there are 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 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 Processor User Guide

Common Signals

There are a number of signals and parameters that are common to multiple sections of the MLP72. These common signals are primarily for controlling delay stages throughout the 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 98).

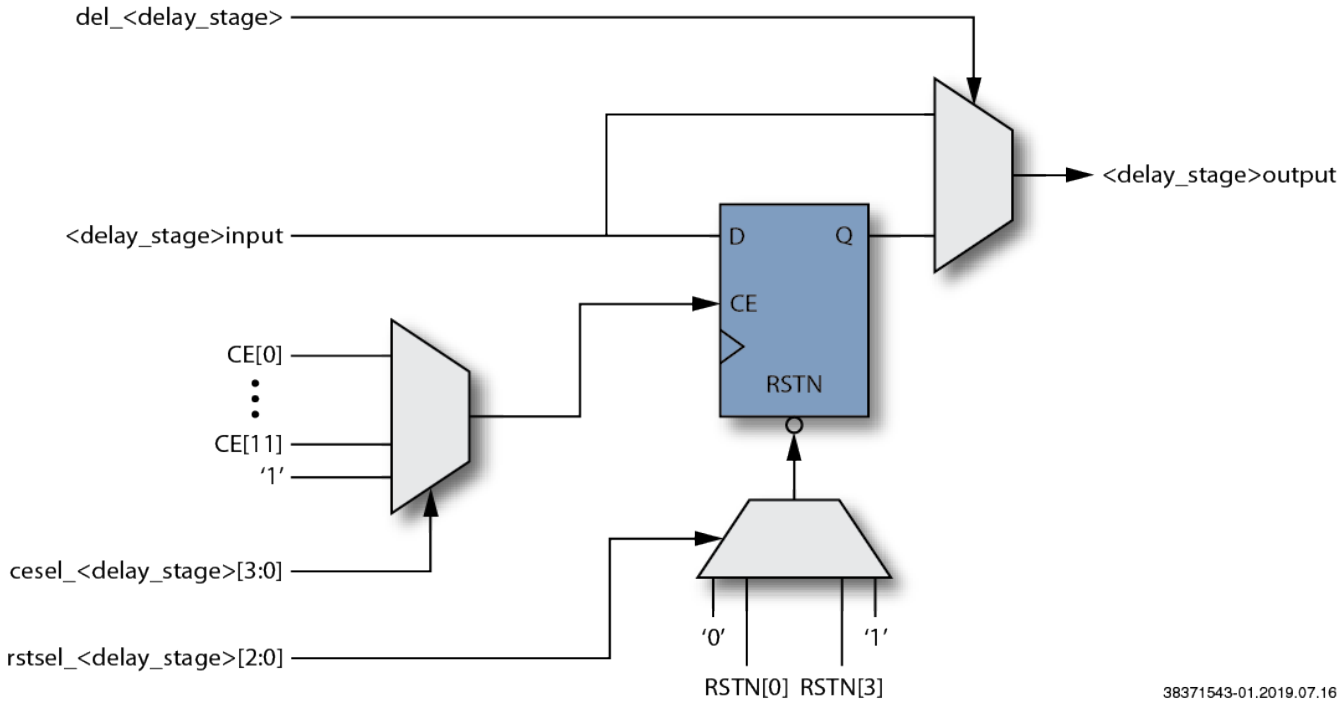

Figure 39: ***Delay Stage Structure***

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 99)

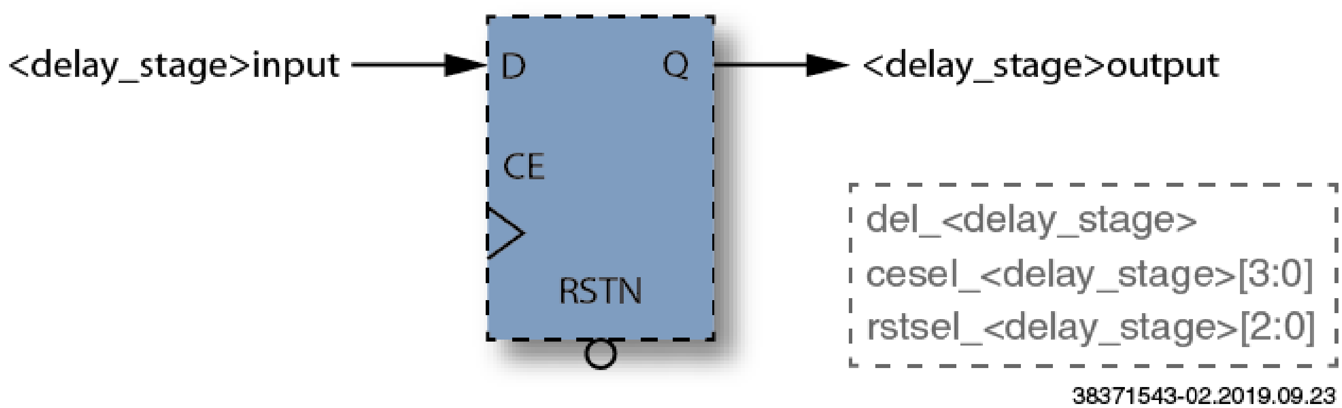

Figure 40: ***Delay Stage Symbol***

Parameters

Table 100: ***Common Parameters***

Parameter	Supported Values	Default Value	Description
cesel_*[3:0]	4'b0000 - 4'b1101	Need to set the value between 0 to 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	Need to set the value between 0 to 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 101: ***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 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 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

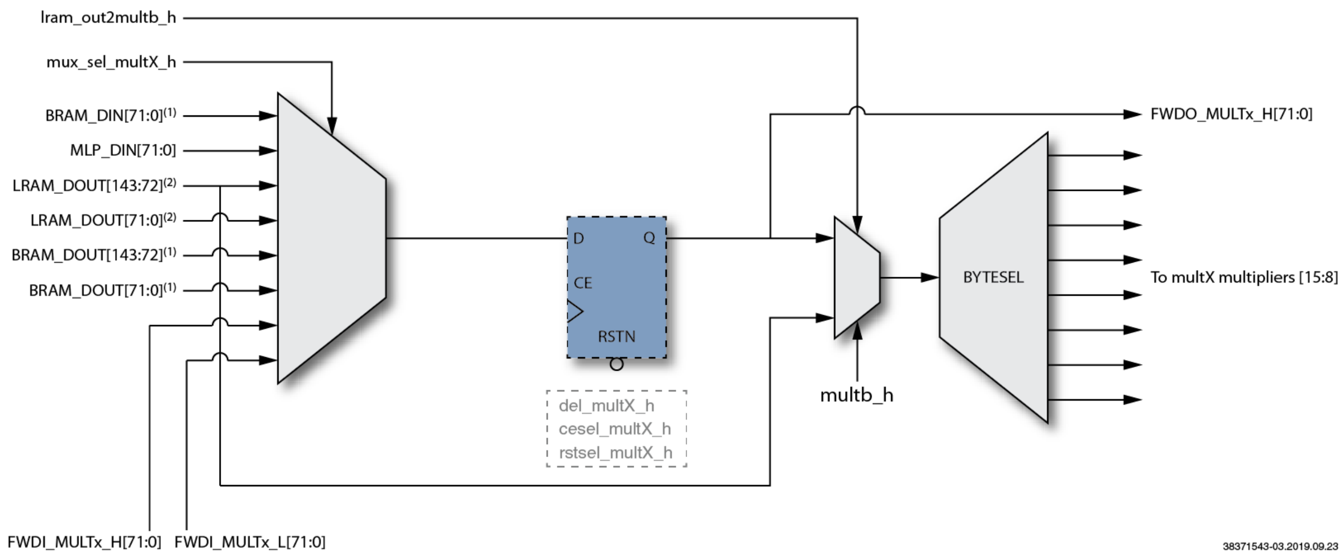

Figure Notes

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 41: ***High Bank Multipliers Input Selection***

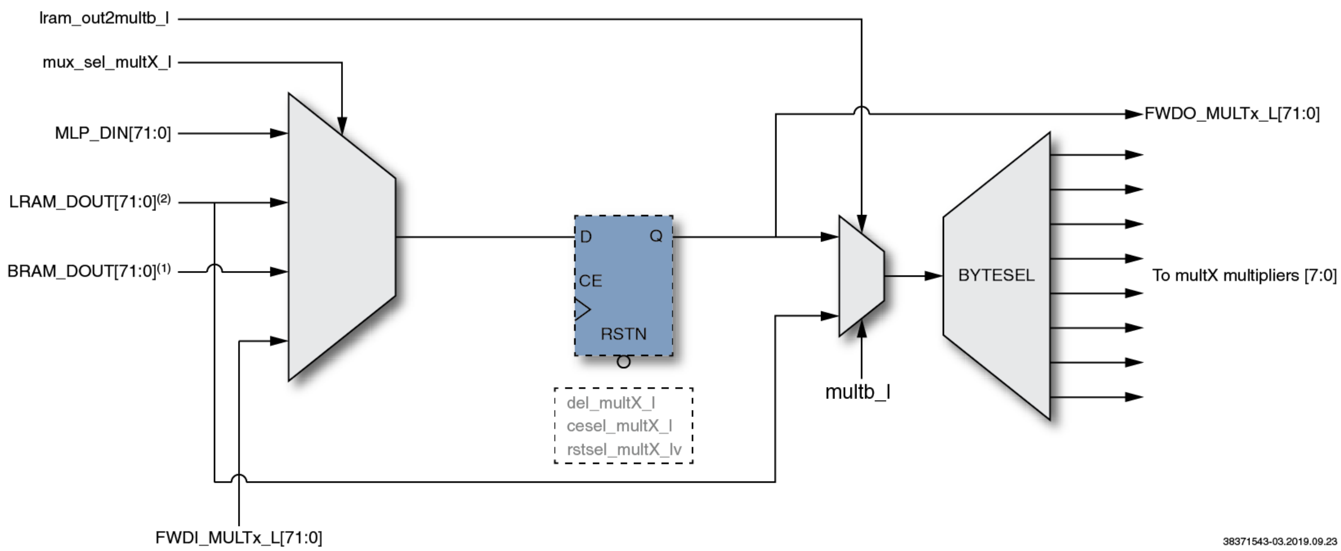

Figure Notes

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 42: ***Low Bank Multipliers Input Selection***

Parameters

Table 102: ***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]
			• 2'b00: MLP_DIN[71:0] • 2'b01: LRAM_DOUT[71:0] (1)

Parameter	Supported Values	Default Value	Description
mux_sel_multa_l[1:0]	2'b00 - 2'b11	2'b00	• 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)
			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]

Parameter	Supported Values	Default Value	Description
cesel_multX_Y[3:0]	4'b0000 - 4'b1101	Need to set the value between 0 to 13	• 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_multX_Y[2:0]	3'b000 - 3'b101	Need to set the value between 0 to 5	Selects the rstn input for each register group. • 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: No-Pipeline Register • 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 103: ***Input Selection Ports***

Name	Direction	Description
din[71:0]	Input	MLP_DIN[71:0] data inputs
mlpram_bramdin2mlpdin[71:0]	Input	Dedicated path from co-sited BRAM72K. Connects BRAM_DIN[71:0] port to MLP.(†)
mlpram_bramdout2mlp[143:0]	Input	Dedicated path from co-sited 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
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 Note

† This port can only be connected to equivalent, same-named output on a BRAM72K. This port cannot be driven directly by fabric logic. The user must instantiate a BRAM to use this connection.

Integer Modes

The most straightforward operation of the MLP72 is in integer mode, when up to 32 parallel multiplications can be performed, and combined with various adder and accumulation stages.

Byte Selection

Once 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 dependant 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 104: ***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 105: ***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 106: ***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 107: ***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 108: ***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 109: ***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 110: ***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 111: ***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 112: ***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 113: ***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 114: ***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 115: ***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 the user should 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 the user should 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 116: ***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 117: ***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 118: ***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 119: ***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 120: ***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 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 in red. Internal signals are shown in black.

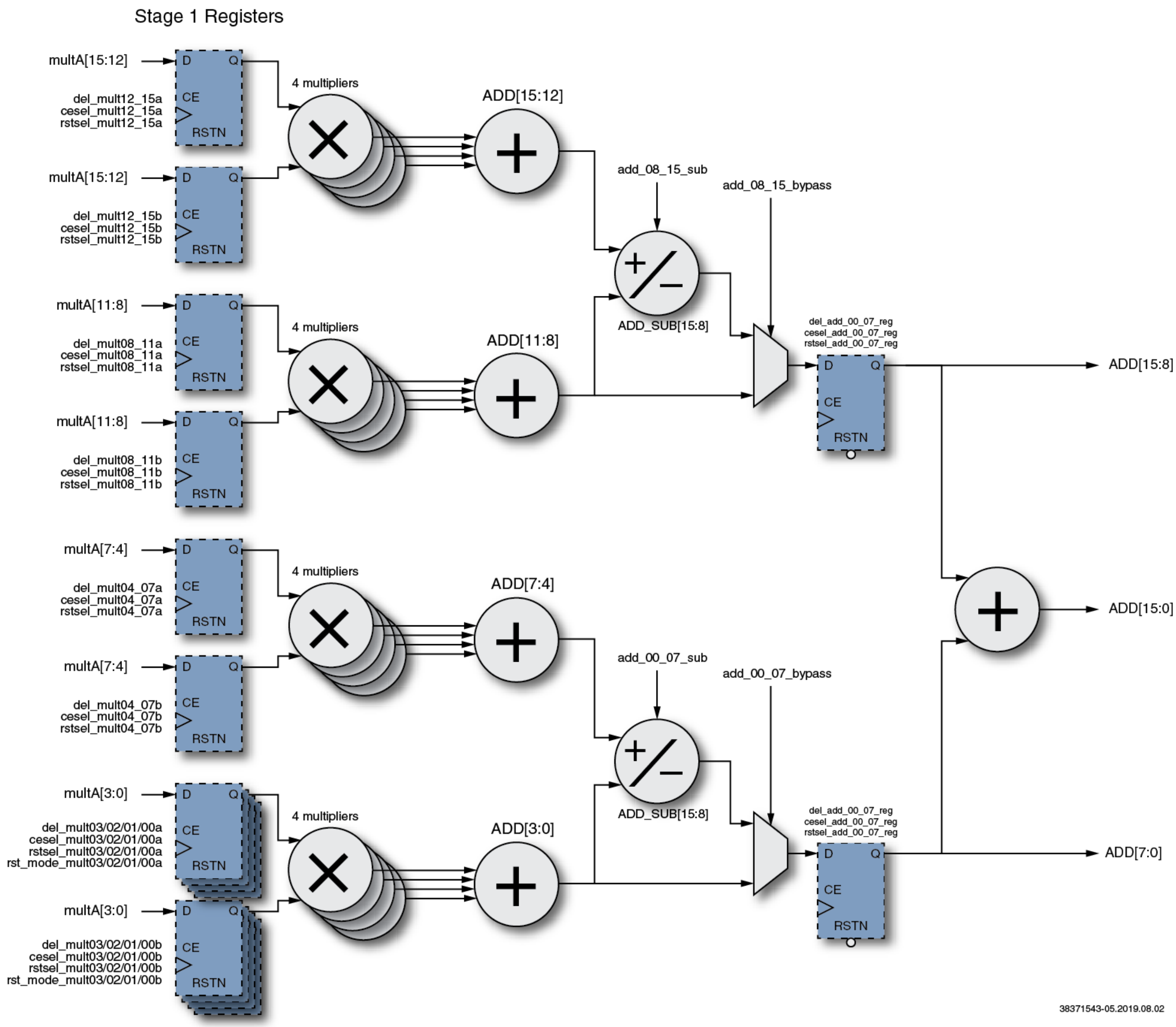

Figure 43: *Multiplication Stage Structure (Integer)*

Parallel Multiplications

The MLP72 combines multipliers in appropriate structures based on the selected number formats. Each multiplier natively supports an Int8 x Int8 multiplication with a 16 bit result. These multipliers can then also be split to perform two parallel Int4 x Int4, or Int3 x 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 121: ***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 MLP72, refer to Number formats (see page 79) . In addition to the actual number formats listed, there is a further processed format, Sign - No Add, that is specific to the MLP72

Sign - No ADD (SNOADD)

This 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 x Int4, or Int3 x 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 MLP72 output stage.

Format Consistency

Between the Input Selection (see page) and the Integer Multiplier Stage (see page) the MLP72 selects the appropriate slice of the input bus to route to each multiplier. This slice selection is dependant upon the input number format, and is controlled by the `bytesel_xx_xx` parameters, and detailed in Byte Selection (see page 103). Equally the multiplier modes are controlled by the `multmode_xx_xx` parameters, which are dependant 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 Figure 3 (see page 109) . Their operation is described in Common Signals. (see page)

Table 122: ***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

Parameter	Supported Values	Default Value	Description
			• 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 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

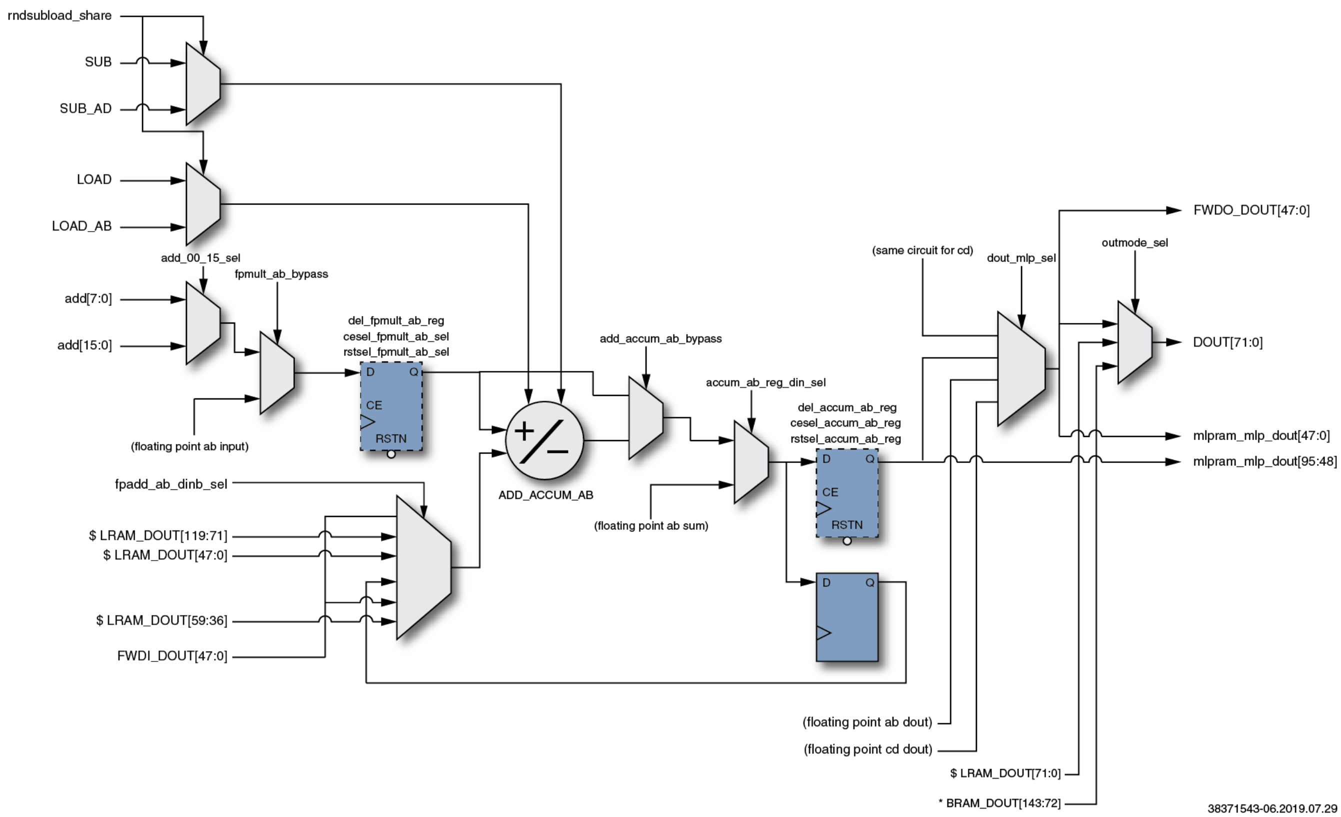

Figure 44: *Output Stage*

Parameters

Table 123: *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
			Select to bypass (C*D) Floating-Point Multiplier • 1'b0 – Floating-Point Multiplier is enabled

Parameter	Supported Values	Default Value	Description
fpmult_cd_bypass	1'b0 - 1'b1	1'b0	• 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. This selector is not shown on the diagram above. • 1'b0 – Selection the value from (C*D) Floating-Point-Multiplier • 1'b1 – Selection the value from (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_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] • 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 will be 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 will be used • 1'b1 – Bypass integer CD accumulator
			Select out_reg input. This selector is not shown on the diagram above.

Parameter	Supported Values	Default Value	Description
out_reg_din_sel[2:0]	3'b000 - 3'b110	2'b00	• 3'b000 – Value will be 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 will be Mult16×2
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
dout_mlp_sel[1:0]	2'b00 - 2'b11	2'b00	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[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][(†)] • 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 Note

† LRAM_DOUT is not a physical port on the MLP72. It is an internal only connection from the associated tightly coupled LRAM

Ports

Table 124: ***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 MLP72 below
fwdo_dout[47:0]	Output	MLP72 internally calculated results, cascaded up to MLP72 above
mlpram_mlp_dout[95:0]	Output	Bits[47:0] 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 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 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 the user requires an LRAM independent of the MLP72, they are recommended to use the dedicated Speedster7t LRAM2K_SDP (see page 227) or Speedster7t LRAM2K_FIFO (see page 222) primitive, appropriate to the application. These primitives have only the required LRAM2K ports and parameters, simplifying instantiation.

> **Note**
>
> When an LRAM2K is instantiated directly, the associated 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 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 BRAM72K. Using the `wrmsel` and `rdmsel` address enables on the co-sited BRAM72K, the LRAM operates as an extension to the BRAM72K, supporting additional address space. The data, read and write signals are connected from the 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 MLP72 calculation, and its output can be routed back into the MLP72 Input Selection (see page) stage. For details of how the MLP72 inputs can be re-purposed to the LRAM, see LRAM Virtual Ports. (see page 117)
- **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 119)). 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 MLP72, several of the 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 125: *LRAM Virtual Port Mapping*

Virtual Port Name	Description	External Pin		
		Mode 0	Mode 1	Mode 2
lram_wraddr	Write address	mlpram_wraddr	expb[7:2]	6'h0
lram_wren	Write enable	mlpram_wren	ce[7]	ce[7]
lram_rdaddr	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:

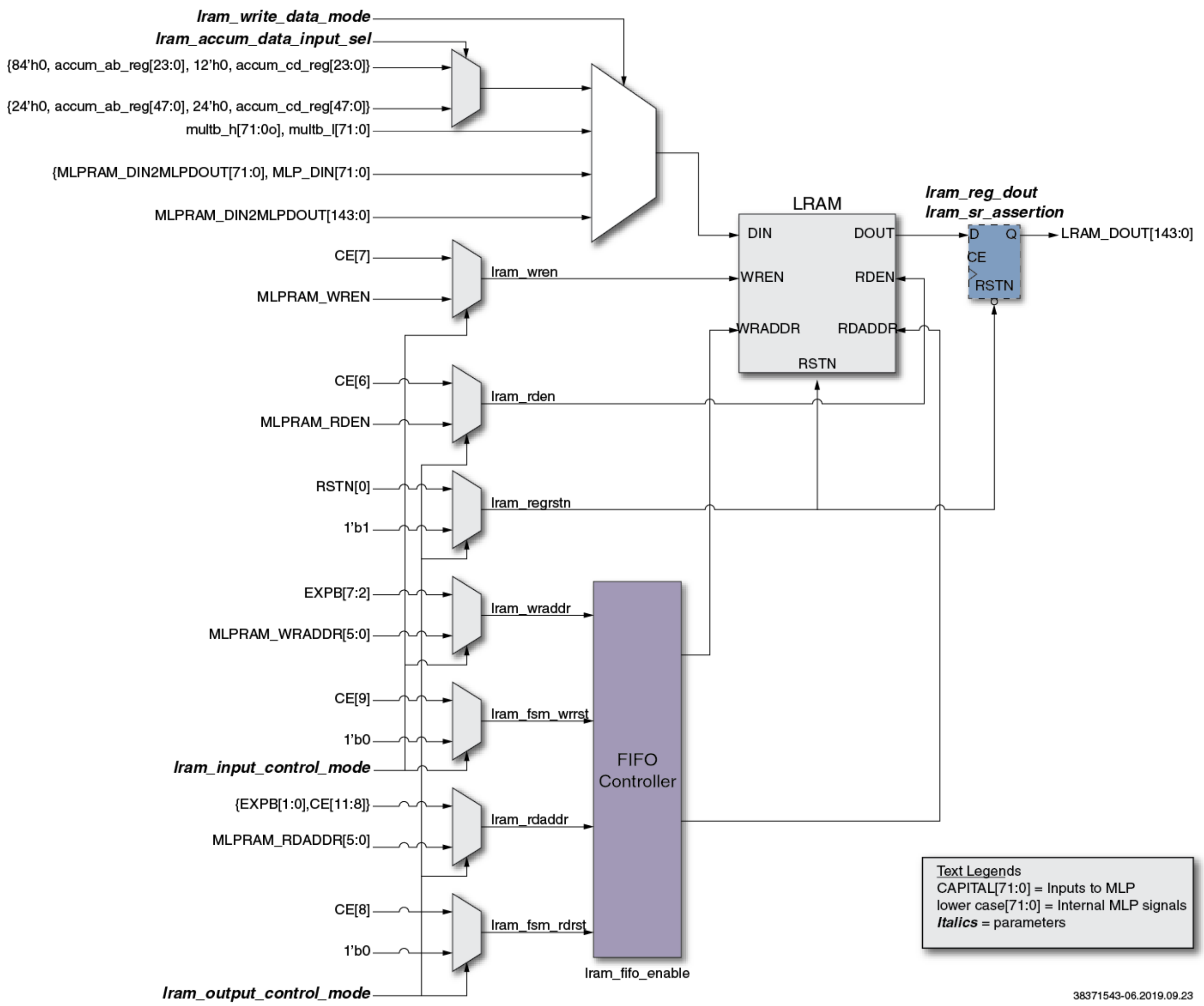

Figure Notes

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 45: ***LRAM connectivity***

FIFO Address Generators

The LRAM is designed with particular flexibility around its FIFO address generators. Most FIFOs support address generators that increment from zero to the maximum memory address boundary, and then wrap back to zero. In addition, it is normally not possible to write to a full FIFO, and reading from an empty FIFO is usually undefined.

Length Adjustment

The LRAM FIFO within the MLP72 supports programmable start and end locations for both the write and read address generators. These thresholds are set using the `lram_fifo_wrptr_rstval`, `lram_fifo_rdptr_rstval`, `lram_fifo_wrptr_maxval`, `and` `lram_fifo_rdptr_maxval` parameters. This paramters allow the FIFO to operate with a programmable length that may better match the length of the group of results or values that are stored in the FIFO.

Mode 2 Pointer Reset

In addition, in Mode 2 (reqiurement 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 the value of `lram_fifo_wrptr_rstval` on the next active edge of `lram_wrclk`.
- Asserting `lram_fsm_rdrst` resets the FIFO read pointer to the value of `lram_fifo_rdptr_rstval` on the next active edge of `lram_rdclk`.

These additional signals allow the FIFO's read or write pointers to be dynamically reset. This capability can be particularly useful at the end of a set of calculations when the remainder of the number of results or values does not match the pre-programmed length of the FIFO.

Ignore Flags

A further feature of the LRAM FIFO address generators is to override the normal operation at a boundary condition. When the `lram_fifo_ignore_flags` parameter is set, the following conditions are allowed:

- Writes when the FIFO is full (write address = read address -1) – The data value will be written into the write address, overwriting the value that is already there. The write address pointer will not be incremented. However, the `write_error` flag will still be asserted.
- Reads when the FIFO is empty,(read address = write address) – The value at the read address can be read from the FIFO repeatably. However, the `read_error` flag will still be asserted.

These additional modes allow for situations where it may be necessary to reuse a value in the FIFO, basically reading the value multiple times.

Parameters

Table 126: *LRAM Parameters*

Parameter	Supported Values	Default Value	Description
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. If lram_reg_dout = 1'b0, then this parameter has no effect: • 1'b0 – Synchronous reset mode • 1'b1 – Asynchronous reset mode
lram_fifo_enable	1'b0 - 1'b1	1'b0	Enable LRAM FIFO mode: • 1'b0 – LRAM is not in FIFO mode • 1'b1 – LRAM is in FIFO mode
lram_clear_enable	1'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. **Note** The LRAM output register is always reset when `lram_regrstn` is asserted low, independent of the state of `lram_clear_enable`
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

Parameter	Supported Values	Default Value	Description
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. This controls the source of wraddr and wren: • 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
lram_output_control_mode[1:0]	2'b00 - 2'b11	2'b00	Select LRAM output control mode. This controls the source of rdaddr, rden and regrstn: • 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
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_rstval[6:0]	7'h00 - 7'h7F	7'h00	LRAM FIFO write pointer reset value

Parameter	Supported Values	Default Value	Description
lram_fifo_rdptr_rstval[6:0]	7'h00 - 7'h7F	7'h00	LRAM FIFO read pointer reset value
lram_fifo_wrptr_maxval[6:0]	7'h00 - 7'h7F	7'h7F	LRAM FIFO write pointer maximum value
lram_fifo_rdptr_maxval[6:0]	7'h00 - 7'h7F	7'h7F	LRAM FIFO read pointer maximum value
lram_fifo_sync_mode	1'b0 - 1'b1	1'b0	Enable LRAM FIFO synchronous mode: • 1'b0 – LRAM FIFO is in asynchronous mode • 1'b1 – LRAM FIFO is in synchronous mode
lram_fifo_afull_threshold[6:0]	7'h00 - 7'h3F	7'h3F	Set LRAM FIFO almost full threshold. User-defined configuration bit. Recommended values are less than 7'h3F.
lram_fifo_aempty_threshold	7'h00 - 7'h0F	7'h00	Set LRAM FIFO almost empty threshold. User-defined configuration bit. Recommended values are not less than 7'h01.
lram_fifo_ignore_flags	1'b0 - 1'b1	1'b0	Enable LRAM FIFO to ignore error flags: • 1'b0 – LRAM FIFO will not write when the FIFO is full (asserting `write_error`) and will not read when the FIFO is empty (asserting `read_error`). • 1'b1 – LRAM FIFO will write to last location in the FIFO (set by `lram_fifo_wrptr_maxval`), even when full, and will re-read lowest location in FIFO (set by `lram_fifo_rdptr_rstval`), even when the FIFO is empty. The LRAM FIFO will not assert `write_error` or `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.
lram_clk_sel_wr	1'b0 - 1'b1	1'b0	Select MLP clock for LRAM write clock: • 1'b0 – LRAM write clock driven by lram_wrclk • 1'b0 – LRAM write clock driven by clk
			Select MLP clock for LRAM read clock: • 1'b0 – LRAM read clock driven by lram_rdclk

Parameter	Supported Values	Default Value	Description
lram_clk_sel_rd	1'b0 - 1'b1	1'b0	• 1'b0 – LRAM read clock driven by clk

Ports

Table 127: *LRAM Ports*

Name	Direction	Description
lram_wrclk	Input	Write side clock input for LRAM. Usage controlled by `lram_clk_sel_wr`
lram_rdclk	Input	Read side clock input for LRAM. Usage controlled by `lram_clk_sel_rd`
mlpram_din2mlpdout [143:0]	Input	Connects BRAM data input, either BRAM_DIN or BRAM internal din, to LRAM_DIN. (†)
mlpram_rdaddr[5:0]	Input	Allows BRAM to control LRAM read address. (†)
mlpram_wraddr[5:0]	Input	Allows BRAM to control LRAM write address. (†)
mlpram_rden	Input	Allows BRAM to control LRAM read enable. (†)
mlpram_wren	Input	Allows BRAM to control LRAM write enable. (†)
mlpram_sbit_error	Input	Allows BRAM to pass through single bit error indication. (†)
mlpram_dbit_error	Input	Allows BRAM to pass through double bit error indication. (†)
sbit_error	Output	Co-sited BRAM72K dedicated pass through of `mlpram_sbit_error`
dbit_error	Output	Co-sited BRAM72K dedicated pass through of `mlpram_dbit_error`
empty	Output	LRAM FIFO empty flag
full	Output	LRAM FIFO full flag
almost_empty	Output	LRAM FIFO almost empty flag
almost_full	Output	LRAM FIFO almost full flag
write_error	Output	Asserted when LRAM in FIFO mode, and write enable is asserted when LRAM FIFO is full
read_error	Output	Asserted when LRAM in FIFO mode, and read enable is asserted when LRAM FIFO is empty

Name	Direction	Description

Table Note

† All inputs prefixed with `mlpram_` are a dedicated path from the co-sited 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 BRAM72K and cannot be driven directly by fabric logic. The user must instantiate a BRAM72K to use these connections. If used, same site placement constraints must be used for the paired BRAM72K and MLP72.

Block Floating-Point Modes

The 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 is the same as for integer; refer to Input Selection (see page 125) 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 128: ***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 129: ***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		b3	b2	b1	b0				
multa_h	Unused								
multb_h	Unused								

> **Table Note**
>
> `eb = expb[7:0]`

Table 130: ***Int8 6 Multiplications (×2 Mode Split – bytesel_00_07 = 'h03; bytesel_08_15 = 'h23)***

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						ec	a10	a9	a8
multb_h		ed	b10	b9	b8				

> **Table Note**
>
> A and B inputs data fields are numbered to reflect the multiplier to which they are applied

Table 131: ***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		b3	b2	b1	b0				
multa_h	ec					a11	a10	a9	a8
multb_h		b11	b10	b9	b8				

Table Note

`eb = ed = expb[7:0]`

Table 132: ***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 133: ***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 134: ***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 135: ***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 136: ***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 137: ***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 138: ***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 139: ***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			b4	b3	b2	b1	b0					
multa_h	Unused											
multb_h	Unused											

Table Note

`eb = expb[7:0]`

Table 140: ***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 141: ***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			b4	b3	b2	b1	b0					
multa_h	ec							a12	a11	a10	a9	a8
multb_h			b12	b11	b10	b9	b8					

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]

Table Note

`eb = ed = expb[7:0]`

Table 142: ***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 143: ***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 will use four multipliers, mult0, .., mult3, in the same way that four multipliers are required for integer Int16 multiplication.

Table 144: ***Int6 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		b1	b0		
multa_h	Unused				

Input Bus	[71:64]	[63:48]	[47:32]	[31:16]	[15:0]
multb_h	Unused				

Table Note

eb = expb[7:0]

Table 145: ***Int6 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		b1	b0		
multa_h	ec			a3	a2
multb_h		b3	b2		

Table Note

`eb = ed = expb[7:0]`

Table 146: ***Int6 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
multa_h	ec	a3	a2		
multb_h	ed	b3	b2		

Ports

Table 147: ***Block Floating-Point Inputs***

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

Parameters

Table 148: ***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)

Parameter	Supported Values	Default Value	Description
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
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_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, the user is recommended to use the existing Speedster7t MLP72 Floating-Point Library (see page 165). This library consists of macros which instantiate the 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 will enable the user to configure the base 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 MLP72 for floating-point operation, the user is recommended to understand how the differing types of floating point numbers are represented and manipulated within the MLP72 as detailed in Speedster7t Number Formats (see page 79).

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 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 149: ***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 150: ***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 151: ***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 152: ***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 153: ***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 154: ***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 155: ***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	

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

Table 156: ***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 157: ***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 158: ***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 159: ***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 160: ***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 161: ***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 162: ***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

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

Parameters

Table 163: ***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 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 selected number format. If they are inconsistent, then 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.

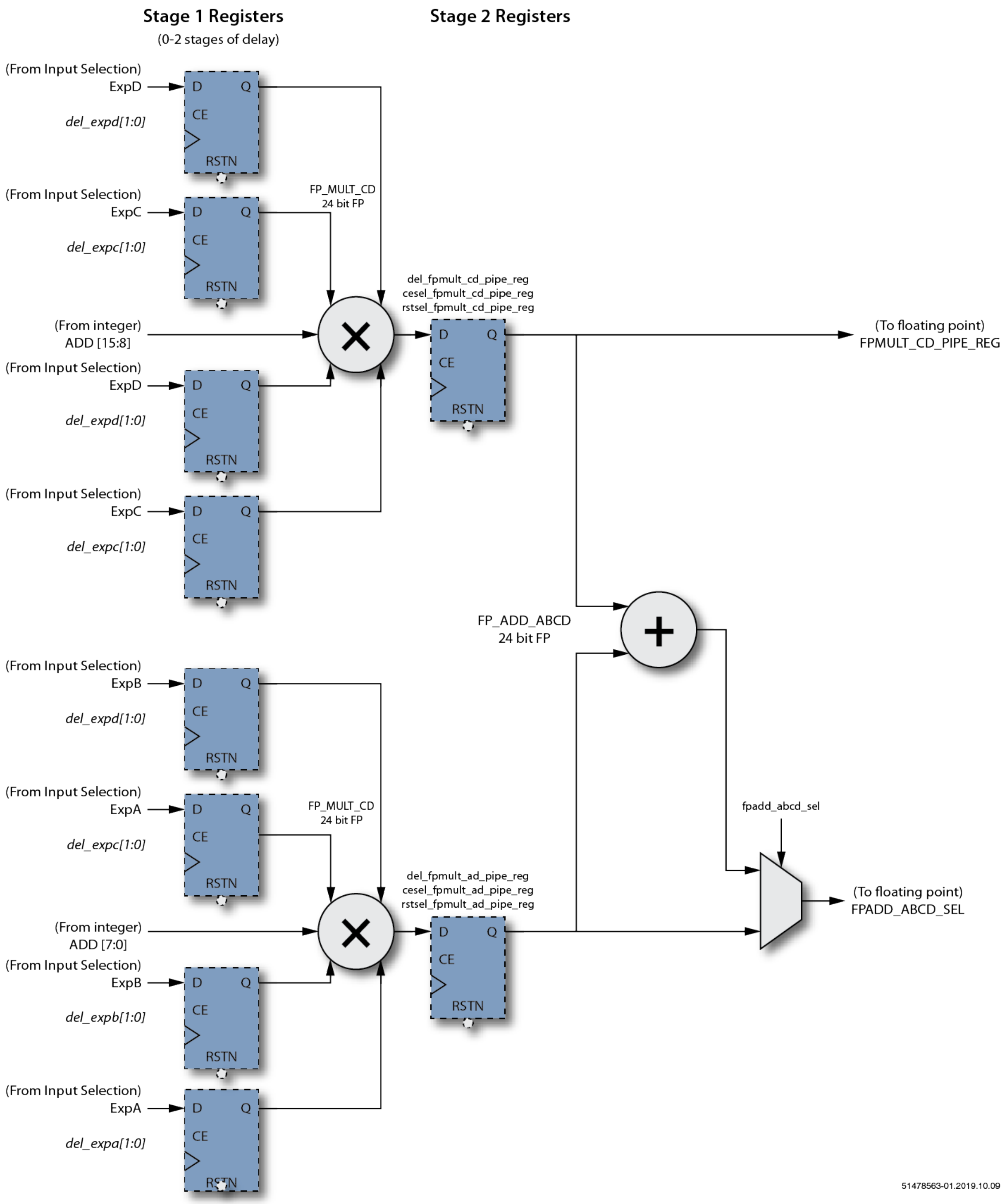

Figure 46: *Floating-Point Multiplier Stage*

Parameters

Table 164: ***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 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 MLP72. This ability allows the MLP72 to externally support consistently sized floating-point inputs and outputs (such as fp16 or bfloat16), while internally performing all calculations at fp24.

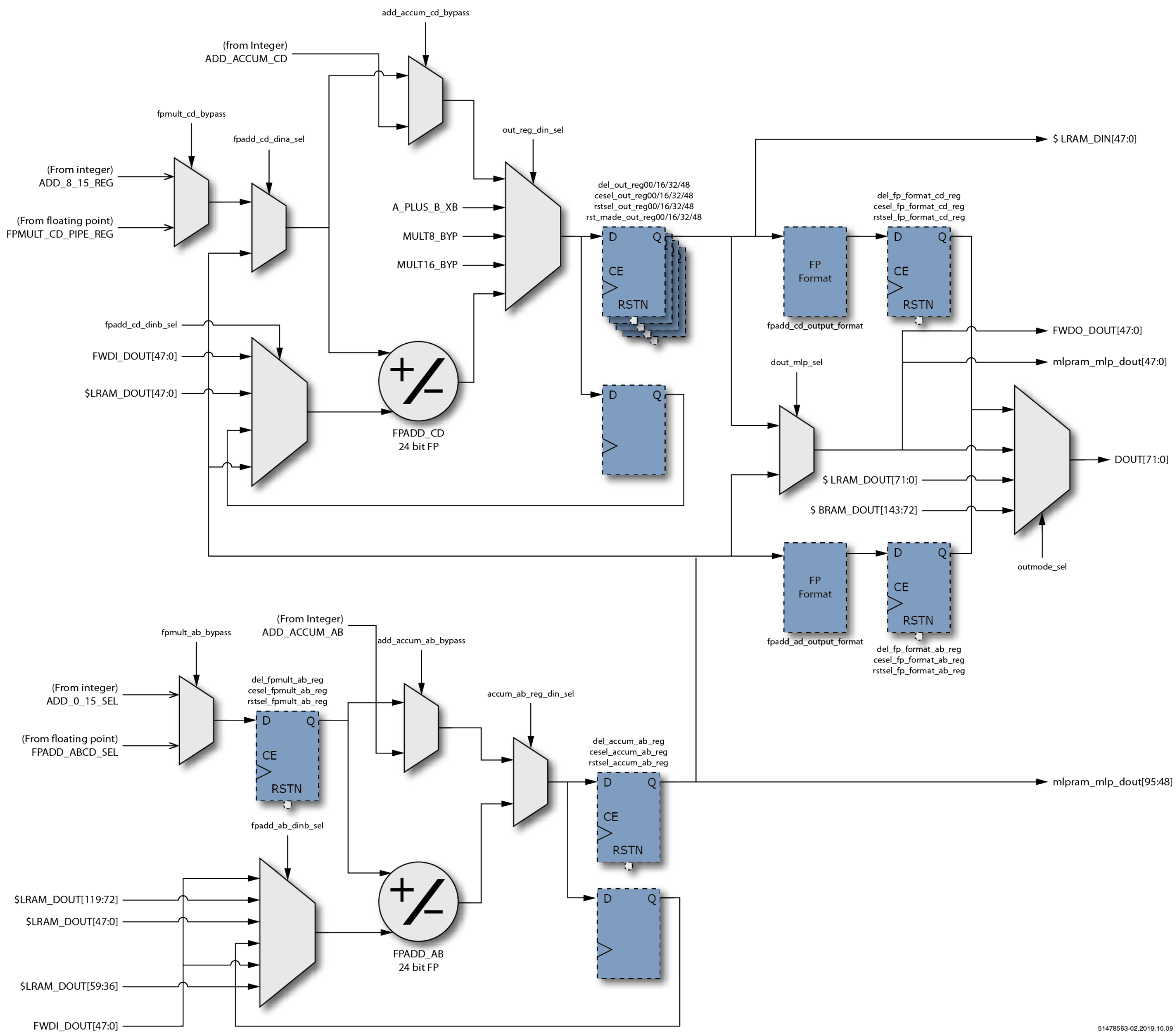

Figure 47: *Floating-Point Output Stage*

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 165: ***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 will be 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 will be 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)
			Select the addend, or subtrahend for the FPADD_AB adder /accumulator: • 3'b000 – 48-bit ACCUM_AB_REG input (always registered)

Parameter	Supported Values	Default Value	Description
fpadd_ab_dinb_sel[2:0]	3'b000 - 3'b101	3'b000	• 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 will be FP24 • 2'b01 – Output format will be BFLOAT16 • 2'b10 – Output format will be 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)
fpadd_cd_output_format [1:0]	2'b00 - 2'b10	2'b00	Selection of floating-point output format from FPADD_CD floating-point adder/accumulator: • 2'b00 – Output format will be FP24 • 2'b01 – Output format will be BFLOAT16 • 2'b10 – Output format will be FP16
			Select to bypass (A×B) floating-point multiplier:

Parameter	Supported Values	Default Value	Description
fpmult_ab_bypass	1'b0 - 1'b1	1'b0	• 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 will be 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 will be 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)
        );
```

MLP72_INT

The 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.

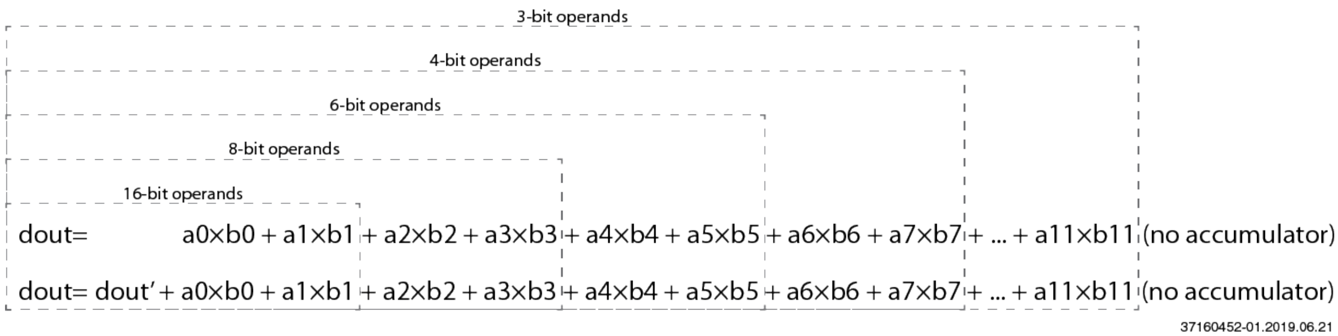

Figure 48: ***MLP72_INT Arithmetic Expressions***

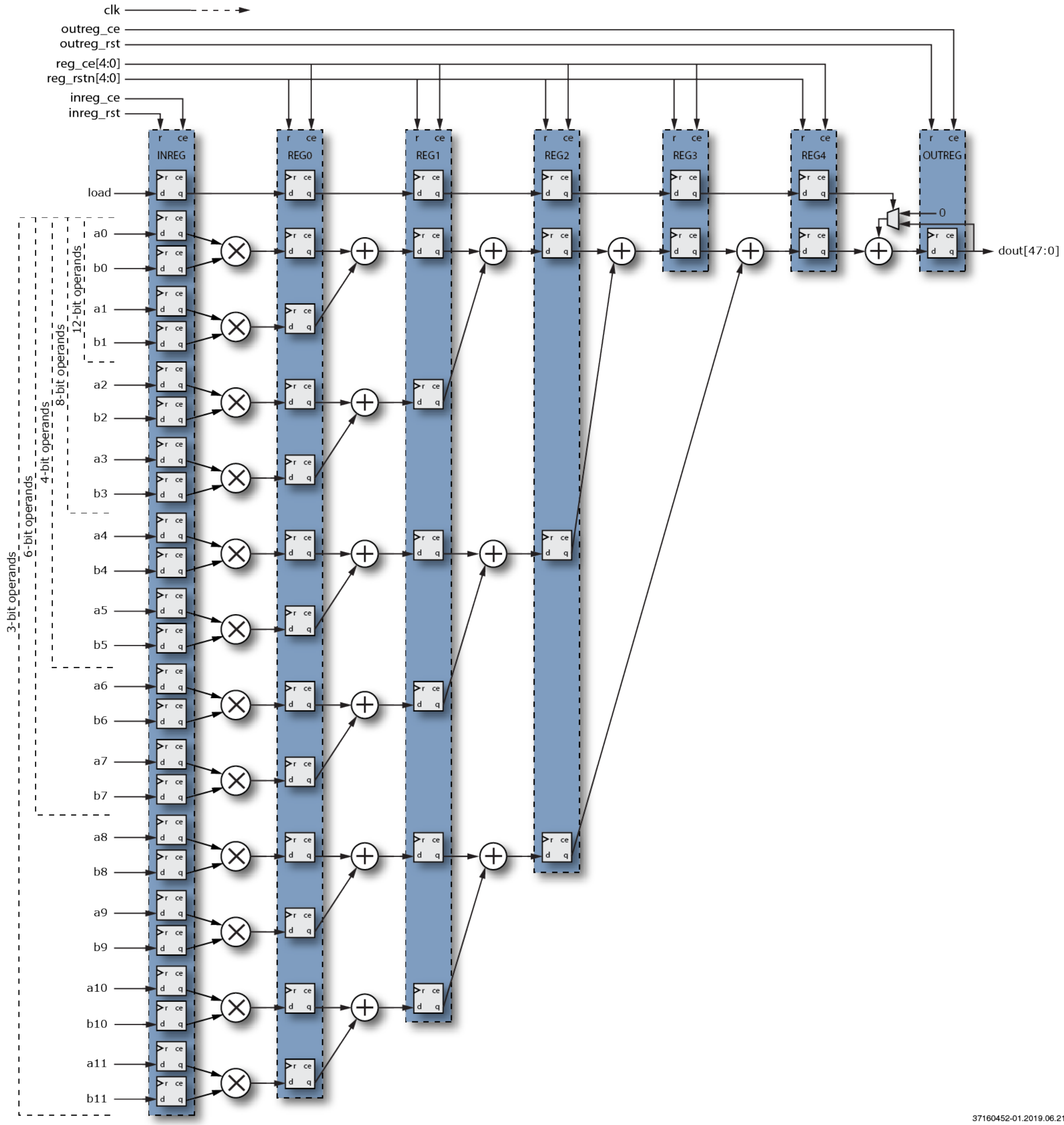

Figure 49: ***MLP72_INT Block Diagram***

Parameters

Table 166: *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 outreg_enable	0, 1	0	Controls whether or not the input register 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 153) below for more details.

Parameter	Supported Values	Default Value	Description
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 167: *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 signal load is also registered if inreg_enable is enabled.
din[71:0]	Input	Data inputs.
inreg_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 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 168: ***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 169: ***B Operand Input Data Mapping***

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

Output Formatting and Error Conditions

The number format of the data output is the same as the format of the data input, as controlled by the number_format parameter. The output register is always 48 bits wide, regardless of the number format or inputs 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 "clocked" inreg_sr_assertion. Therefore, to use "unclocked" inreg_sr_assertion, either:

- Tie off the upper multipliers and not use them, or;
- Set inreg_enable to 0 and keep the input registers out of the MLP (as 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 to "clocked"; "unclocked" inreg_sr_assertion is not supported when using the accumulator feature.

Below is a table which describes valid scenarios when inreg_sr_assertion can be set to "unclocked" when the input register is enabled.

Table 170: ***MLP72_INT Asynchronous Reset Rules***

operand_width	accumulator_enable	Multipliers that can be Used with inreg_sr_assertion of "unclocked"	Multipliers Which Must 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 MLP72_INT8 cannot currently be inferred.

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        ),
    .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    ),
        .outreg_rstn             (outreg_rstn ),
        .outreg_ce               (outreg_ce   ),
        .dout                    (dout        )
    );
```

MLP72_INT8_MULT_4X

The 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.

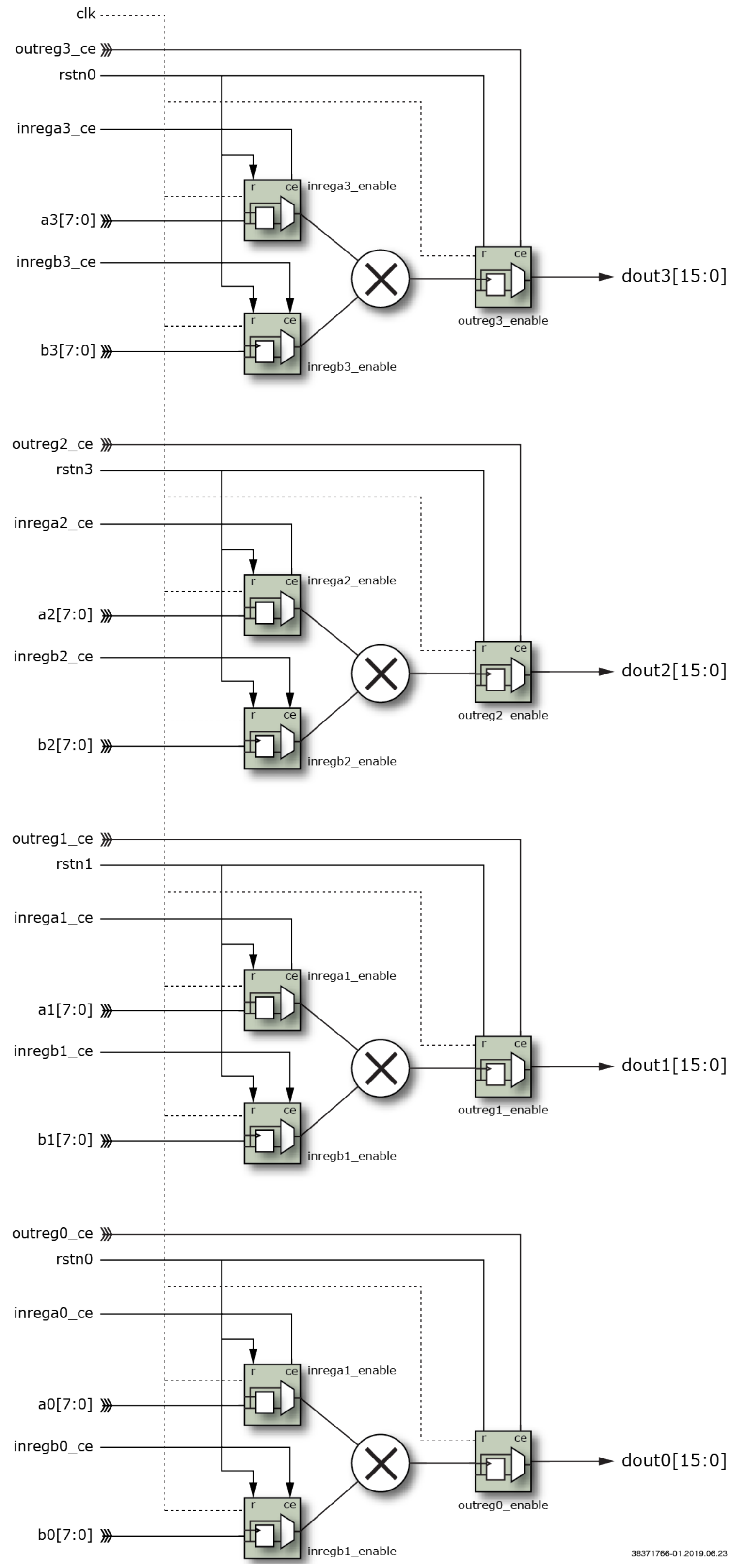

Figure 50: *MLP72_INT8_MULT_4X Block Diagram*

Parameters

Table 171: *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 172: ***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 MLP72_INT8_MULT_4X, where both `inreg_enable` and `outreg_enable` are true, and all control inputs are active high.

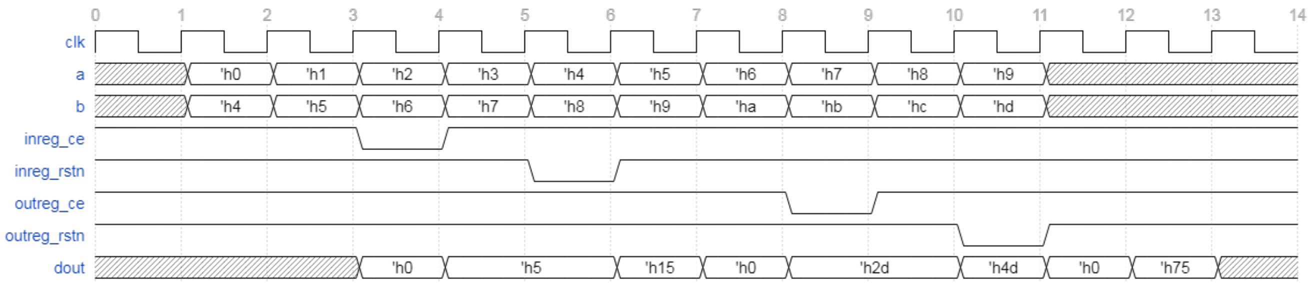

Figure 51: ***Timing Diagram for Single Multiplier Channel***

Inference

The MLP72_INT8_MULT_4X cannot currently be inferred

Instantiation Template

Verilog

```
ACX_MLP72_INT8_MULT_4X
  #(
    .clk_polarity         (clk_polarity          ),
    .number_format        (number_format         ),
    .inrega3_enable       (inrega3_enable        ),
    .inregb3_enable       (inregb3_enable        ),
    .inrega2_enable       (inrega2_enable        ),
    .inregb2_enable       (inregb2_enable        ),
    .inrega1_enable       (inrega1_enable        ),
    .inregb1_enable       (inregb1_enable        ),
    .inrega0_enable       (inrega0_enable        ),
    .inregb0_enable       (inregb0_enable        ),
    .outreg3_enable       (outreg3_enable        ),
    .outreg2_enable       (outreg2_enable        ),
    .outreg1_enable       (outreg1_enable        ),
    .outreg0_enable       (outreg0_enable        ),
    .inrega3_sr_assertion (inrega3_sr_assertion  ),
    .inregb3_sr_assertion (inregb3_sr_assertion  ),
    .inrega2_sr_assertion (inrega2_sr_assertion  ),
    .inregb2_sr_assertion (inregb2_sr_assertion  ),
    .inrega1_sr_assertion (inrega1_sr_assertion  ),
    .inregb1_sr_assertion (inregb1_sr_assertion  ),
    .inrega0_sr_assertion (inrega0_sr_assertion  ),
    .inregb0_sr_assertion (inregb0_sr_assertion  ),
    .outreg3_sr_assertion (outreg3_sr_assertion  ),
    .outreg2_sr_assertion (outreg2_sr_assertion  ),
    .outreg1_sr_assertion (outreg1_sr_assertion  ),
    .outreg0_sr_assertion (outreg0_sr_assertion  )
) instance_name (
   .clk                       (clk        ),
```

```
        .a0                         (a0         ),
        .a1                         (a1         ),
        .a2                         (a2         ),
        .a3                         (a3         ),
        .b0                         (b0         ),
        .b1                         (b1         ),
        .b2                         (b2         ),
        .b3                         (b3         ),
        .rstn0                      (rstn0      ),
        .rstn1                      (rstn1      ),
        .rstn2                      (rstn2      ),
        .rstn3                      (rstn3      ),
        .inrega3_ce                 (inrega3_ce ),
        .inregb3_ce                 (inregb3_ce ),
        .inrega2_ce                 (inrega2_ce ),
        .inregb2_ce                 (inregb2_ce ),
        .inrega1_ce                 (inrega1_ce ),
        .inregb1_ce                 (inregb1_ce ),
        .inrega0_ce                 (inrega0_ce ),
        .inregb0_ce                 (inregb0_ce ),
        .outreg3_ce                 (outreg3_ce ),
        .outreg2_ce                 (outreg2_ce ),
        .outreg1_ce                 (outreg1_ce ),
        .outreg0_ce                 (outreg0_ce ),
        .dout0                      (dout0      ),
        .dout1                      (dout1      ),
        .dout2                      (dout2      ),
        .dout3                      (dout3      )
    );
```

MLP72_INT16_MULT_2X

The 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.

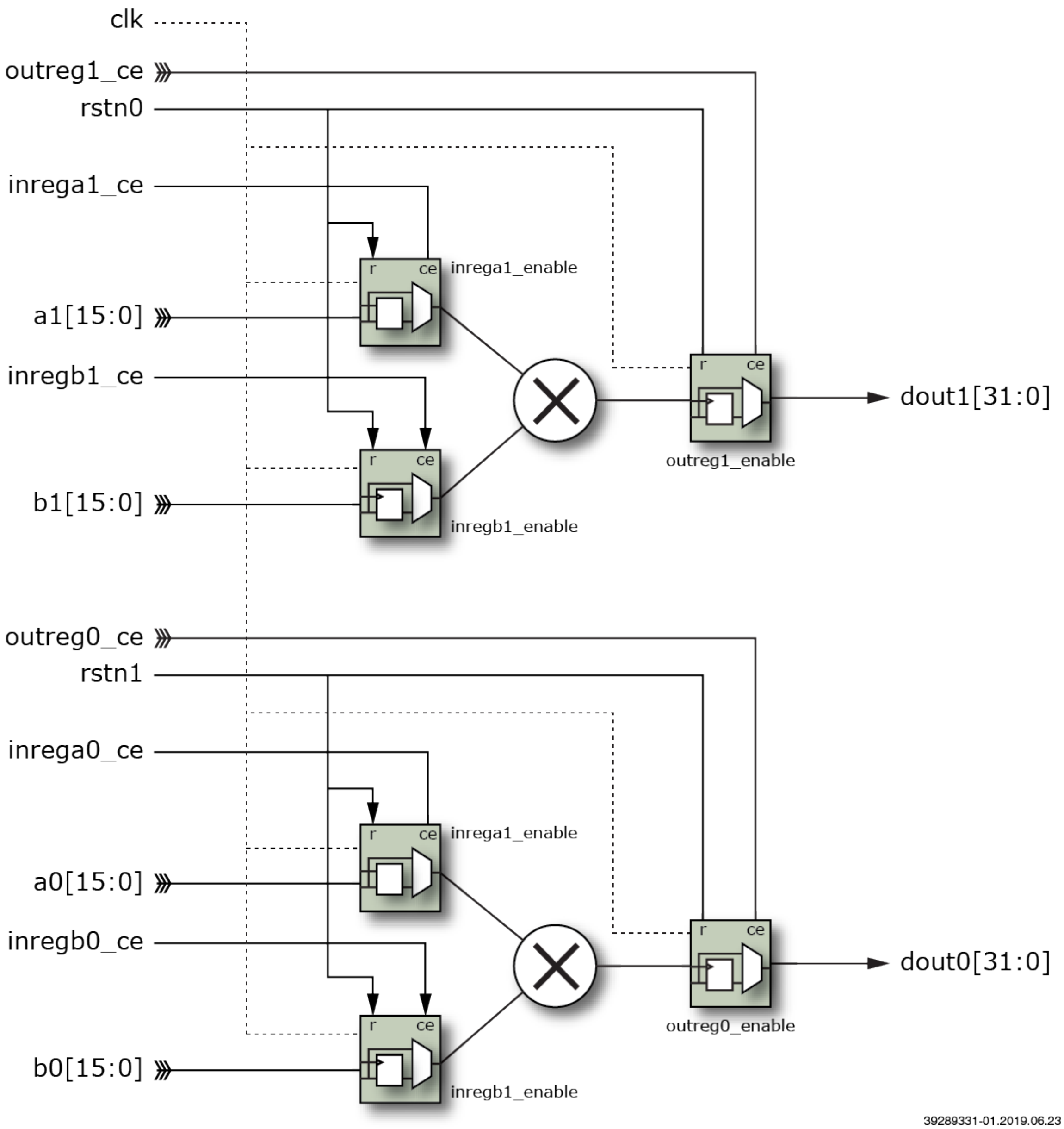

Figure 52: *MLP72_INT16_MULT_2X Block Diagram*

Parameters

Table 173: ***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 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.
inrega1_sr_assertion inregb1_sr_assertion	"clocked"	"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. **Note** For optimal MLP performance on upper multipliers, use synchronous ("clocked") resets.

Ports

Table 174: *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 MLP72_INT16_MULT2X, where both `inreg_enable` and `outreg_enable` are true, and all control inputs are active high.

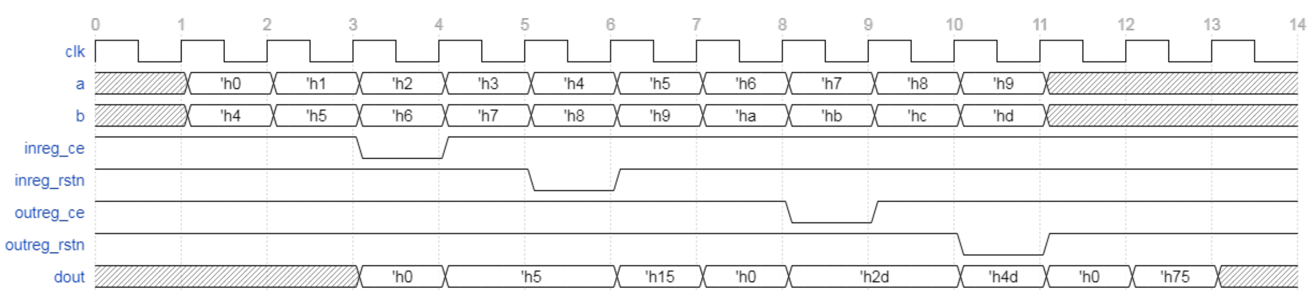

Figure 53: *Timing Diagram for a Single Multiplier Channel*

Inference

The MLP72_INT16_MULT_2X is not currently inferable

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                )
);
```

Floating-Point Library

Introduction

The Achronix floating-point library provides macros that instantiate the MLP72 to perform common floating-point operations.

Verilog Include File

To use the library, include the following in the Verilog source code:

```
`include "speedster7t/common/acx_floating_point.v"
```

Most macros have common parameters and ports; these are described below.

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, the following registers are supported. By default, all registers are disabled (bypassed). To enable a register, the corresponding enable parameter must be set to 1'b1. Not all macros support all registers because some registers are not applicable to every operation.

in_reg_enable

This parameter enables registers on all the input signals. Enabling these registers allows for improved timing for input signals routed through the fabric to the macro. All input registers enabled have a dedicated clock enable (`ce`, active high) and a shared synchronous reset (`rstn`, active low). If the input register is not enabled, `ce` and `rstn` are ignored.

mult_reg_enable

This parameter enables a register after any multiplication stage. This register will improve timing particularly if an addition (or accumulation) follows the multiplication.

> **Note**
>
> The addition only macros support `mult_reg_enable`. In all macros the input data signals pass through a multiplier stage, in the case of the addition only macros, this multiplier is set to multiply the input data signal by 1.0. Hence `mult_reg_enable` is still valid for addition only macros.

add_reg_enable

This parameter enables a register following any addition stage. This register is particularly useful if a second addition (namely, accumulation) follows the first addition. If there is no accumulation, (`accumulate` = 1'b0), the output of the addition stage is directly connected to the output stage.

out_reg_enable

This parameter enables the output register. If enabled this allows for improved timing for output signals routed from the macro through the fabric. The output register is driven by either the adder stage, (if accumulation is not enabled), or by the accumulation stage. See Double Delay Stage (see page 166) with regard to enabling `add_reg_enable` and `out_reg_enable`.

> **Note**
>
> In this library, the input registers are the only registers with `ce` and `rstn`.

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 of the accumulator is this sum.

The accumulator is located after any adder stage, and before the output stage. Setting add_reg_enable = 1'b1, will enable a register stage between the adder stage and the accumulator which will improve timing a higher frequencies.

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.

Double Delay Stage

If accumulation is not required, then the adder result is passed directly to the output register. In this instance if both add_reg_enable and out_reg_enable are set, then there will be two back-to-back delay stages in the output. Although this is a valid combination, the double stage only result in extra latency through the macro, rather than improved timing. Therefore it is generally recommended that add_reg_enable and out_reg_enable are only both set when accumulation is also set. In this configuration, the accumulator stage is then present between the adder and the output register.

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 79) 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 175: *Output Status Bits*

Status	Description
2'b00	Normal.
2'b01	Result is ± 0.0.
2'b11	Last operation had underflow, and hence, the result is ± 0.0.

Status	Description
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.

FP_ADD

This module computes A+B with optional accumulation. Accumulation is enabled with parameter `accumulate` and cleared by asserting `i_load`. Internal delay stages can be enabled to meet either the required latency, or to achieve timing closure.

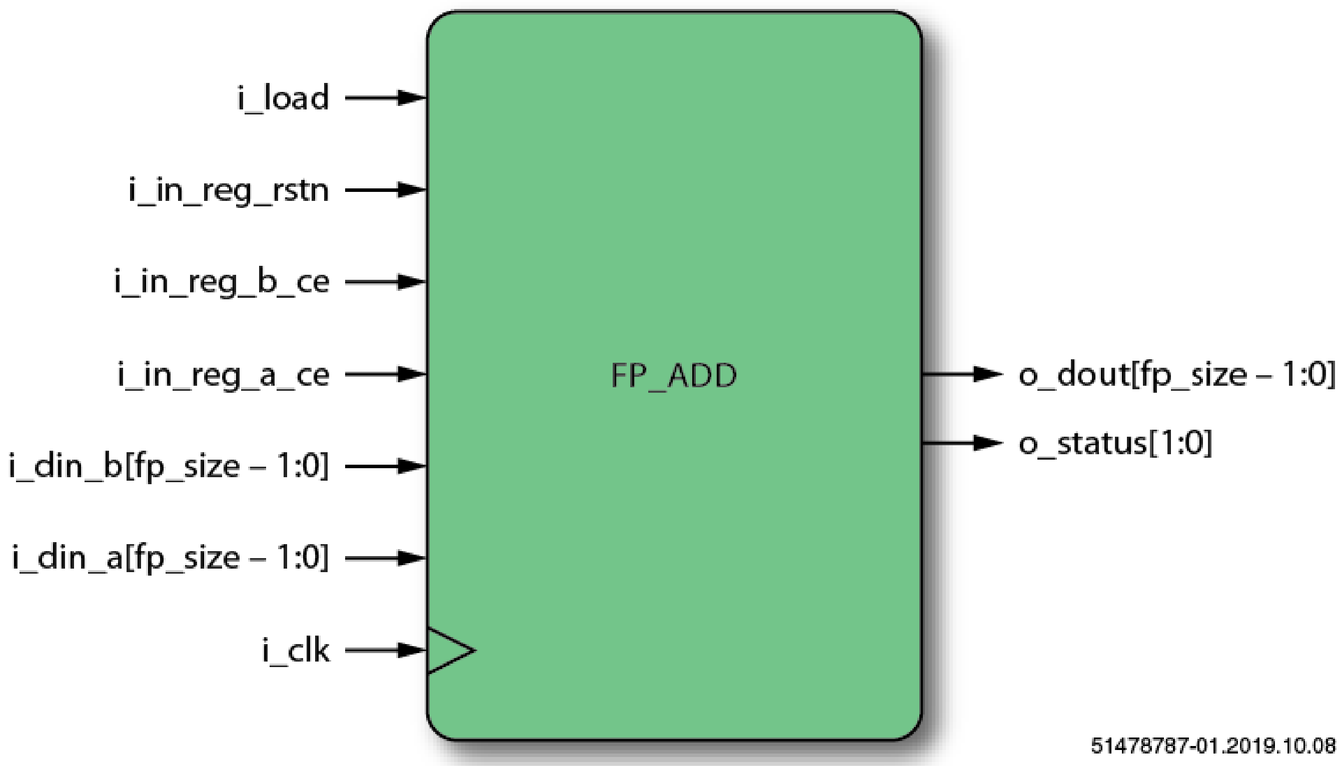

Figure 54: *Floating-Point Adder with Optional Accumulate*

Parameters

Table 176: *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.
accumulate	0, 1	0	0 – No accumulation. 1 – Accumulate the sum on each output cycle. `i_load` used to reset accumulation.
in_reg_enable	0, 1	0	0 – No input registers. 1 – Ports `i_din_a` and `i_din_b` registered. Registers controlled by `i_in_reg_a/b_ce` and `i_in_reg_rstn`. Adds a cycle of latency.
mult_reg_enable	0, 1	0	0 – Multiplier output used directly 1 – Multiplier output registered before next stage. Adds a cycle of latency. Although FP_ADD performs only addition, internally it computes (1.0 × i_din_a) + (1.0 × i_din_b). This setting enables a register after the multiplication with 1.0.
add_reg_enable	0, 1	0	0 – Adder output used directly. 1 – Adder output registered before next stage. Adds a cycle of latency. This setting is primarily useful when accumulation is enabled.
out_reg_enable	0, 1	0	0 – Port `o_dout` sourced from either adder or accumulator outputs directly. 1 – Port `o_dout` registered internally. Adds a cycle of latency.

Ports

Table 177: *FP_ADD 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 adder/accumulator.(1)
i_din_b[fp_size − 1:0]	Input	B data input to adder/accumulator.(1)
i_in_reg_a_ce	Input	Clock enable for `i_din_a` input when parameter `in_reg_enable` = 1'b1.(2)

Name	Direction	Description
i_in_reg_b_ce	Input	Clock enable for `i_din_b` input registers.(2)
i_in_reg_rstn	Input	Synchronous active-low reset for input registers.(2)
i_load	Input	Synchronous load of accumulator. Resets accumulator value to (i_din_a + i_din_b).
o_dout[fp_size − 1: 0]	Output	Result of addition and accumulation.
o_status[1:0]	Output	Indicates if errors occurred with o_dout.(3)

Table Notes

1. Width determined by fp_size parameter.
2. Wen parameter `in_reg_enable` = 1'b1. Unused when `in_reg_enable` = 1'b0.
3. See Output Status (see page 165) for details.

Inference

ACX_FP_ADD cannot be inferred. It has to be directly instantiated.

Instantiation Templates

Verilog

```
// Verilog template for ACX_FP_ADD
ACX_FP_ADD #(
    .fp_size             (fp_size),
    .fp_exp_size         (fp_exp_size),
    .accumulate          (accumulate),
    .in_reg_enable       (in_reg_enable),
    .mult_reg_enable     (mult_reg_enable),
    .add_reg_enable      (add_reg_enable),
    .out_reg_enable      (out_reg_enable)
) 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_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;
    accumulate           : integer := 0;
    in_reg_enable        : integer := 0;
    mult_reg_enable      : integer := 0;
    add_reg_enable       : integer := 0;
    out_reg_enable       : 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_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,
    accumulate           => accumulate,
    in_reg_enable        => in_reg_enable,
    mult_reg_enable      => mult_reg_enable,
    add_reg_enable       => add_reg_enable,
    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_load               => user_i_load,
    o_dout               => user_o_dout,
    o_status             => user_o_status
);
```

FP_MULT

This module computes A × B with optional accumulation. Accumulation is enabled with parameter `accumulate` and cleared by asserting `i_load`. Internal delay stages can be enabled to meet either the required latency, or to achieve timing closure.

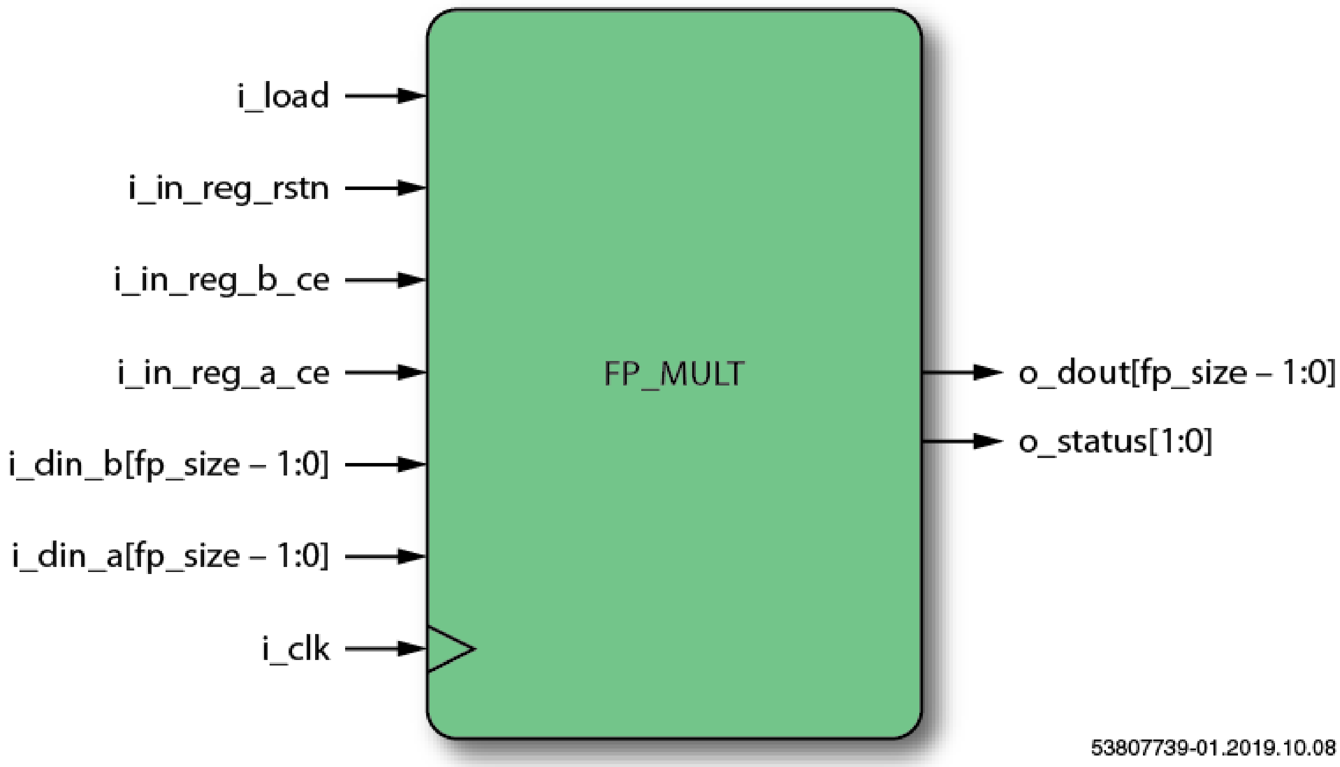

Figure 55: *Floating-Point Multiplier with Optional Accumulate*

Parameters

Table 178: *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. 1 – Accumulate the sum on each output cycle. `i_load` used to reset accumulation.
in_reg_enable	0, 1	0	0 – No input registers. 1 – Ports `i_din_a` and `i_din_b` registered. Registered controlled by `i_in_reg_a/b_ce` and `i_in_reg_rstn`. Adds a cycle of latency.

mult_reg_enable	0, 1	0	0 – Multiplier output used directly. 1 – Multiplier output registered before next stage. Adds a cycle of latency. This setting is primarily useful when accumulation is enabled.
out_reg_enable	0, 1	0	0 – Port `o_dout` sourced from either multiplier or accumulator outputs directly. 1 –Port `o_dout` registered internally. Adds a cycle of latency.

Ports

Table 179: *FP_MULT 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.(1)
i_din_b[fp_size − 1:0]	Input	B data input to multiplier.(1)
i_in_reg_a_ce	Input	Clock enable for `i_din_a` input.(2)
i_in_reg_b_ce	Input	Clock enable for `i_din_b` input.(2)
i_in_reg_rstn	Input	Synchronous active low reset for input registers.(2)
i_load	Input	Synchronous load of accumulator. Will reset accumulator value to (i_din_a * i_din_b).
o_dout[fp_size − 1:0]	Output	Result of multiplication and accumulation.
o_status[1:0]	Output	Indicates if errors occurred with `o_dout`.(3)

Table Notes

1. Width determined by fp_size parameter.
2. Wen parameter `in_reg_enable` = 1'b1. Unused when `in_reg_enable` = 1'b0.
3. See Output Status (see page 165) for details.

Inference

ACX_FP_MULT cannot be inferred. It has to be directly instantiated.

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),
    .mult_reg_enable      (mult_reg_enable),
    .out_reg_enable       (out_reg_enable)
) 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_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;
    mult_reg_enable       : integer := 0;
    out_reg_enable        : 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_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,
```

```
        mult_reg_enable       => mult_reg_enable,
        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_load                => user_i_load,
        o_dout                => user_o_dout,
        o_status              => user_o_status
    );
```

FP_MULT_PLUS

This module computes A×B+C, with optional accumulation. Accumulation is enabled with parameter `accumulate` and is cleared by asserting `i_load`. Internal delay stages can be enabled to meet either the required latency, or to achieve timing closure.

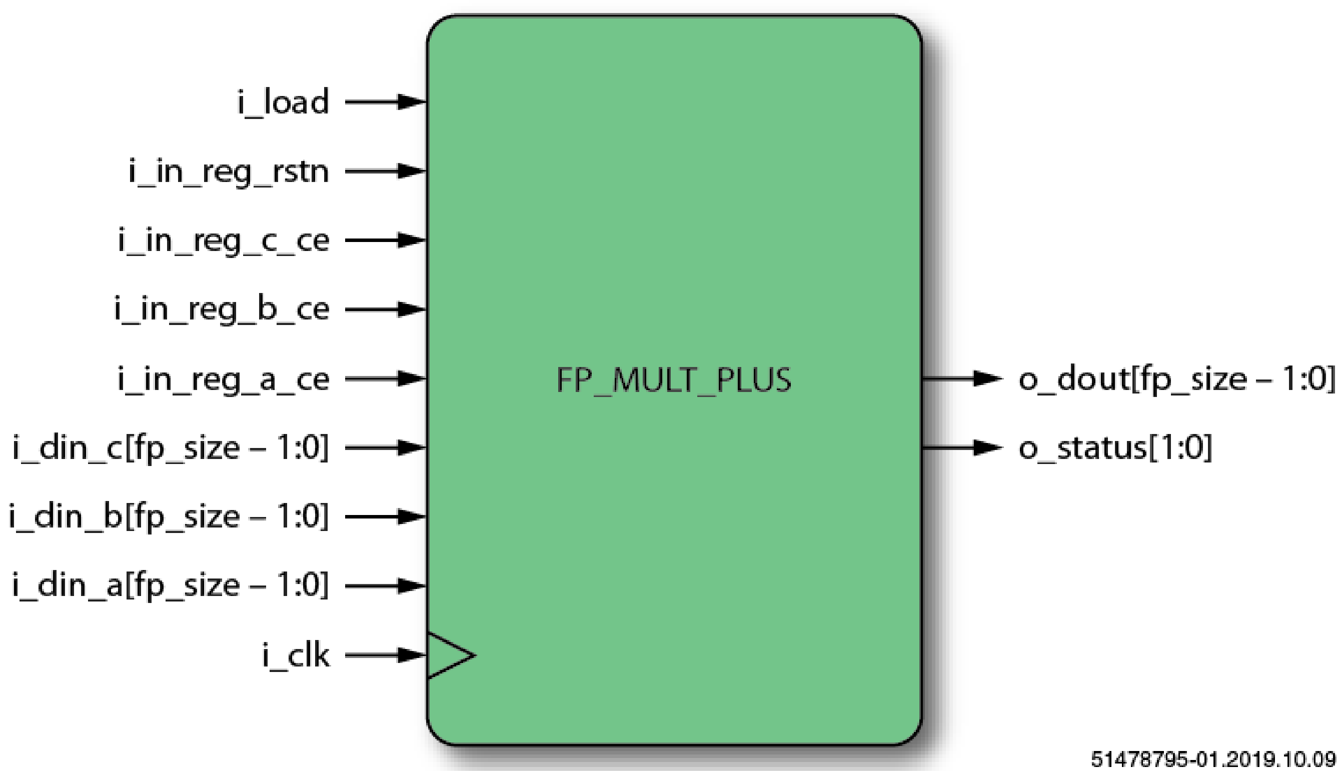

Figure 56: ***Floating-Point Multiplier Plus Adder with Optional Accumulate***

Parameters

Table 180: *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.
accumulate	0, 1	0	0 – No accumulation. 1 – Accumulate the sum on each output cycle. `i_load` used to reset accumulation.
in_reg_enable	0, 1	0	0 – No input registers. 1 – Ports `i_din_a/b/c` registered. Registers controlled by `i_in_reg_a/b/c_ce` and `i_in_reg_rstn`. Adds a cycle of latency.
mult_reg_enable	0, 1	0	0 – Multiplier output used directly. 1 – Multiplier output registered before next stage. Adds a cycle of latency.
add_reg_enable(†)	0, 1	0	0 – Adder output is used directly. 1 – Adder output is registered before next stage. Adds a cycle of latency. This setting is primarily useful when accumulation is enabled.
out_reg_enable	0, 1	0	0 – Port `o_dout` sourced from either adder or accumulator outputs directly. 1 – Port `o_dout` registered internally. Adds a cycle of latency.

Table Note

† The parameter `add_reg_enable` controls the delay stage after the internal floating-point adder. It does not control the delay stage to the add input (i_din_c). Control of the delay stage for the C input is shared with the A & B input delay stages control performed by `in_reg_enable`.

Ports

Table 181: *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.(1)
i_din_b[fp_size − 1:0]	Input	B data input to multiplier.(1)
i_din_c[fp_size − 1:0]	Input	C data input direct to adder.(1)
i_in_reg_a_ce	Input	Clock enable for `i_din_a` input.(2)
i_in_reg_b_ce	Input	Clock enable for `i_din_b` input.(2)
i_in_reg_c_ce	Input	Clock enable for `i_din_c` input.(2)
i_in_reg_rstn	Input	Synchronous active-low reset for input registers when parameter `in_reg_enable` = 1'b1.(2)
i_load	Input	Synchronous load of accumulator. Will reset accumulator value to (`i_din_a` × `i_din_b`).
o_dout[fp_size − 1:0]	Output	Result of multiplication and accumulation.
o_status[1:0]	Output	Indicates if errors occurred with `o_dout`.(3)

Table Notes

1. Width determined by fp_size parameter.
2. When parameter `in_reg_enable` = 1'b1. Unused when `in_reg_enable` = 1'b0.
3. See Output Status (see page 165) for details.

Inference

ACX_FP_MULT_PLUS cannot be inferred. It has to be directly instantiated.

Instantiation Templates

Verilog

```
// Verilog template for ACX_FP_MULT_PLUS
ACX_FP_MULT_PLUS #(
    .fp_size            (fp_size),
    .fp_exp_size        (fp_exp_size),
    .accumulate         (accumulate),
    .in_reg_enable      (in_reg_enable),
    .mult_reg_enable    (mult_reg_enable),
    .add_reg_enable     (add_reg_enable),
    .out_reg_enable     (out_reg_enable)
) 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_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;
    accumulate          : integer := 0;
    in_reg_enable       : integer := 0;
    mult_reg_enable     : integer := 0;
    add_reg_enable      : integer := 0;
    out_reg_enable      : 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_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,
    accumulate          => accumulate,
    in_reg_enable       => in_reg_enable,
    mult_reg_enable     => mult_reg_enable,
    add_reg_enable      => add_reg_enable,
    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_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_load              => user_i_load,
    o_dout              => user_o_dout,
    o_status            => user_o_status
);
```

FP_MULT_2X

The FP_MULT_2X macro is similar to FP_MULT (see page 171), but uses a single MLP72 to compute two products in parallel with optional accumulation. The two operations are:

- `dout_ab = i_din_a × i_din_b`
- `dout_cd = i_din_c × i_din_d`

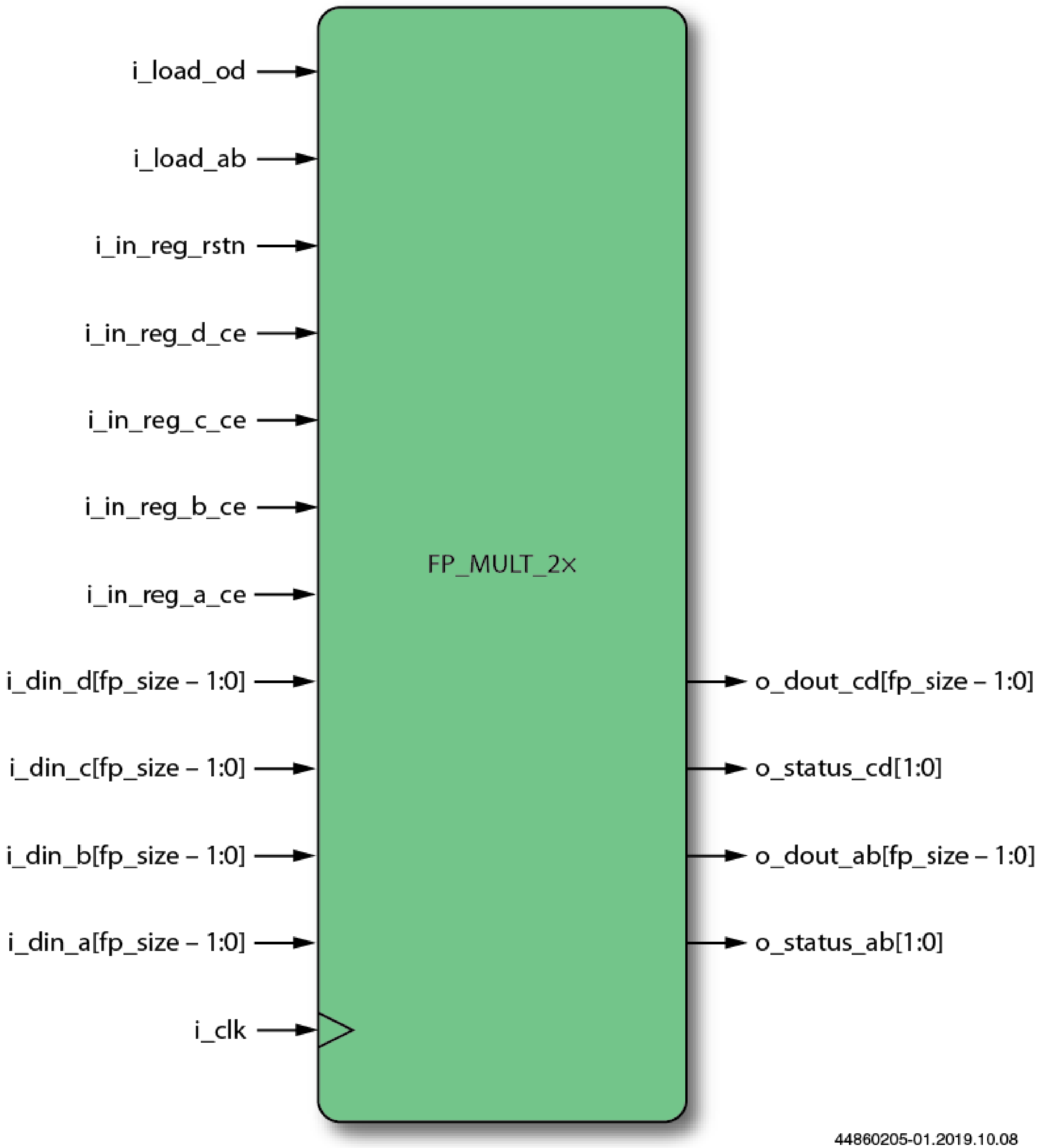

Figure 57: ***Twin Floating-Point Multipliers with Optional Accumulate***

Parameters

Table 182: *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 1 – Accumulate the sum on each output cycle. `i_load_ab/i_load_cd` used to reset accumulation.
in_reg_enable	0, 1	0	0 – No input registers 1 – Ports `i_din_a/b/c/d` registered. Registers controlled by `i_in_reg_a/b/c/d_ce` and `i_in_reg_rstn`. Adds a cycle of latency.
mult_reg_enable	0, 1	0	0 – Multiplier output used directly. 1 – Multiplier output registered before next stage. Adds a cycle of latency. This setting is primarily useful when accumulation is enabled.
out_reg_enable	0, 1	0	0 – Port `o_dout` sourced from either multiplier or accumulator outputs directly. 1 – Port `o_dout` registered internally. Adds a cycle of latency.

Ports

Table 183: *FP_MULT_2X 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 AB multiplier.(1)
i_din_b[fp_size – 1:0]	Input	B data input to AB multiplier.(1)
i_din_c[fp_size – 1:0]	Input	C data input to CD multiplier.(1)
i_din_d[fp_size – 1:0]	Input	D data input to CD multiplier.(1)
i_in_reg_a_ce	Input	Clock enable for `i_din_a` input.(2)

Name	Direction	Description
i_in_reg_b_ce	Input	Clock enable for `i_din_b` input.(2)
i_in_reg_c_ce	Input	Clock enable for `i_din_c` input.(2)
i_in_reg_d_ce	Input	Clock enable for `i_din_d` input.(2)
i_in_reg_rstn	Input	Synchronous active low reset for input registers.(2)
i_load_ab	Input	Synchronous load of accumulator following A × B multiplier. Resets the accumulator value to (i_din_a × i_din_b).
i_load_cd	Input	Synchronous load of accumulator following C × D multiplier. Resets the accumulator value to (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]	Output	Indicates if errors occurred with `o_dout_ab`.(3)
o_status_cd[1:0]	Output	Indicates if errors occurred with `o_dout_cd`.(3)

Table Notes

1. Width determined by fp_size parameter.
2. When parameter `in_reg_enable` = 1'b1. Unused when `in_reg_enable` = 1'b0.
3. See Output Status (see page 165) for details.

Inference

The ACX_FP_MULT_2X cannot be inferred. It has to be directly instantiated.

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),
    .mult_reg_enable    (mult_reg_enable),
    .out_reg_enable     (out_reg_enable)
) 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_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;
    mult_reg_enable      : integer := 0;
    out_reg_enable       : 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_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,
    mult_reg_enable      => mult_reg_enable,
    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_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_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
);
```

FP_MULT_ADD

This macro computes (A×B) + (C×D) with optional accumulation. Accumulation is enabled with parameter `accumulate` and is cleared by asserting `i_load`. Internal delay stages can be enabled to meet either the required latency or to achieve timing closure.

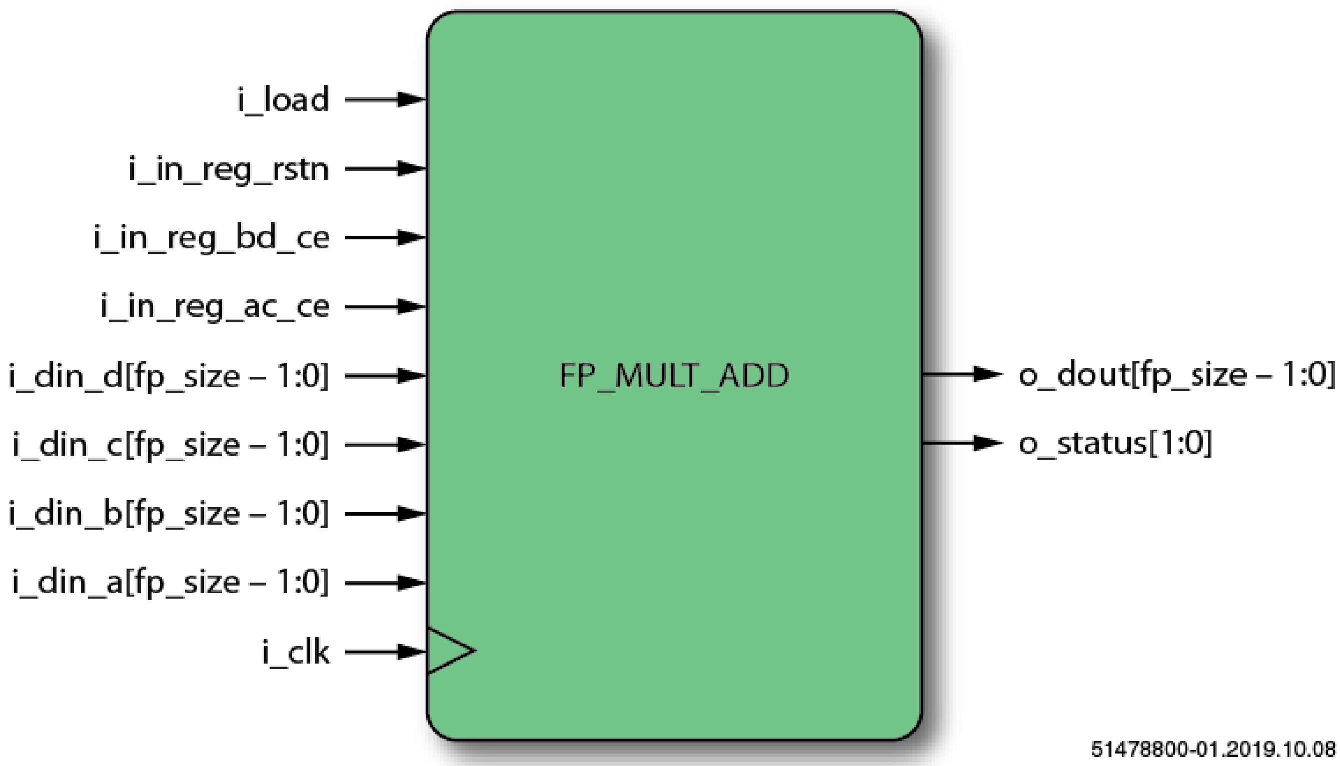

Figure 58: ***Twin Floating-Point Multiplies with Addition***

Parameters

Table 184: *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.
accumulate	0, 1	0	0 – No accumulation. 1 – Accumulate the sum on each output cycle. `i_load` used to reset accumulation.
in_reg_enable	0, 1	0	0 – No input registers. 1 – Ports `i_din_a/b/c/d` registered. Registers controlled by `i_in_reg_ac/bd_ce` and `i_in_reg_rstn`. Adds a cycle of latency.
mult_reg_enable	0, 1	0	0 – Multiplier outputs used directly. 1 – Multiplier outputs registered before adder stage. Adds a cycle of latency.
add_reg_enable	0, 1	0	0 – Adder output is used directly. 1 – Adder output is registered before next stage. Adds a cycle of latency. This setting is primarily useful when accumulation is enabled.
out_reg_enable	0, 1	0	0 – Port `o_dout` sourced from adder/accumulator output directly. 1 – Port `o_dout` registered internally. Adds a cycle of latency.

Ports

Table 185: *FP_MULT_ADD 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 AB multiplier.(1)
i_din_b[fp_size − 1:0]	Input	B data input to AB multiplier.(1)
i_din_c[fp_size − 1:0]	Input	C data input to CD multiplier.(1)
i_din_d[fp_size − 1:0]	Input	D data input to CD multiplier.(1)

Name	Direction	Description
i_in_reg_ac_ce	Input	Clock enable for `i_din_a` and `i_din_c` inputs.(2)
i_in_reg_bd_ce	Input	Clock enable for `i_din_b` and `i_din_d` inputs.(2)
i_in_reg_rstn	Input	Synchronous active-low reset for input registers.(2)
i_load	Input	Synchronous load of adder/accumulator. Will reset accumulator value to (`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]	Output	Indicates if errors occurred with `o_dout`.(3)

Table Notes

1. Width determined by fp_size parameter.
2. Wen parameter `in_reg_enable` = 1'b1. Unused when `in_reg_enable` = 1'b0.
3. See Output Status (see page 165) for details.

Inference

ACX_FP_MULT_ADD cannot be inferred. It has to be directly instantiated

Instantiation Templates

Verilog

```
// Verilog template for ACX_FP_MULT_ADD
ACX_FP_MULT_ADD #(
    .fp_size             (fp_size),
    .fp_exp_size         (fp_exp_size),
    .accumulate          (accumulate),
    .in_reg_enable       (in_reg_enable),
    .mult_reg_enable     (mult_reg_enable),
    .add_reg_enable      (add_reg_enable),
    .out_reg_enable      (out_reg_enable)
) 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_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;
    accumulate            : integer := 0;
    in_reg_enable         : integer := 0;
    mult_reg_enable       : integer := 0;
    add_reg_enable        : integer := 0;
    out_reg_enable        : 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_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,
    accumulate            => accumulate,
    in_reg_enable         => in_reg_enable,
    mult_reg_enable       => mult_reg_enable,
    add_reg_enable        => add_reg_enable,
    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_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_load                => user_i_load,
    o_dout                => user_o_dout,
    o_status              => user_o_status
);
```

Integer Library

The Achronix integer library provides macros that use the MLP72 to to perform common integer operations. To use the library, include the following in the Verilog source code:

```
`include "speedster7t/common/acx_integer.v"
```

Most macros have common parameters and ports, which are described below. Full descriptions of each macro then follow.

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 integer library, the following registers are supported. By default, all registers are disabled (bypassed). To enable a register, the corresponding enable parameter must be set to 1'b1. Not all macros support all registers because some registers are not applicable to every operation.

in_reg_enable

This parameter enables registers on all the input signals. Enabling these registers allows for improved timing for input signals routed through the fabric to the macro. All input registers enabled have a dedicated clock enable (`ce` , active high) and a shared synchronous reset (`rstn`, active low). If the input register is not enabled, `ce` and `rstn` are ignored.

mult_reg_enable

This parameter enables a register after any multiplication stage. This register will improve timing particularly if an addition (or accumulation) follows the multiplication.

out_reg_enable

This parameter enables the output register. If enabled this allows for improved timing for output signals routed from the macro through the fabric. The output register is driven by either the adder stage, (if accumulation is not enabled), or by the accumulation stage.

> **Note**
>
> In this library, the input registers are the only registers with `ce` and `rstn`.

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 starts a new accumulation, then load must be high when those inputs are presented. If accumulation is not enabled, the load signal is ignored.

The internal accumulation register is 48 bits wide.

Integer Format

The number of bits per input is specified with parameter int_size. The MLP72 supports the following values: 3, 4, 6, 7, 8, and 16.

The parameter `int_unsigned` specifies whether the integer uses two's complement format (default), or unsigned format.

If addition is performed, the output format is a 48-bit integer, signed or unsigned as determined by `int_unsigned`. Parameter `dout_size` may be used to reduce the number of bits, if it is known that the result will fit.

If only if multiplication is performed, without addition or accumulation, the output has 2 × int_size bits, meaning that the result always fits without possibility of overflow.

INT_MULT

This module computes A × B with optional accumulation. Accumulation is enabled with parameter `accumulate` and cleared by asserting `i_load`. Internal delay stages can be enabled to meet either the required latency, or to achieve timing closure.

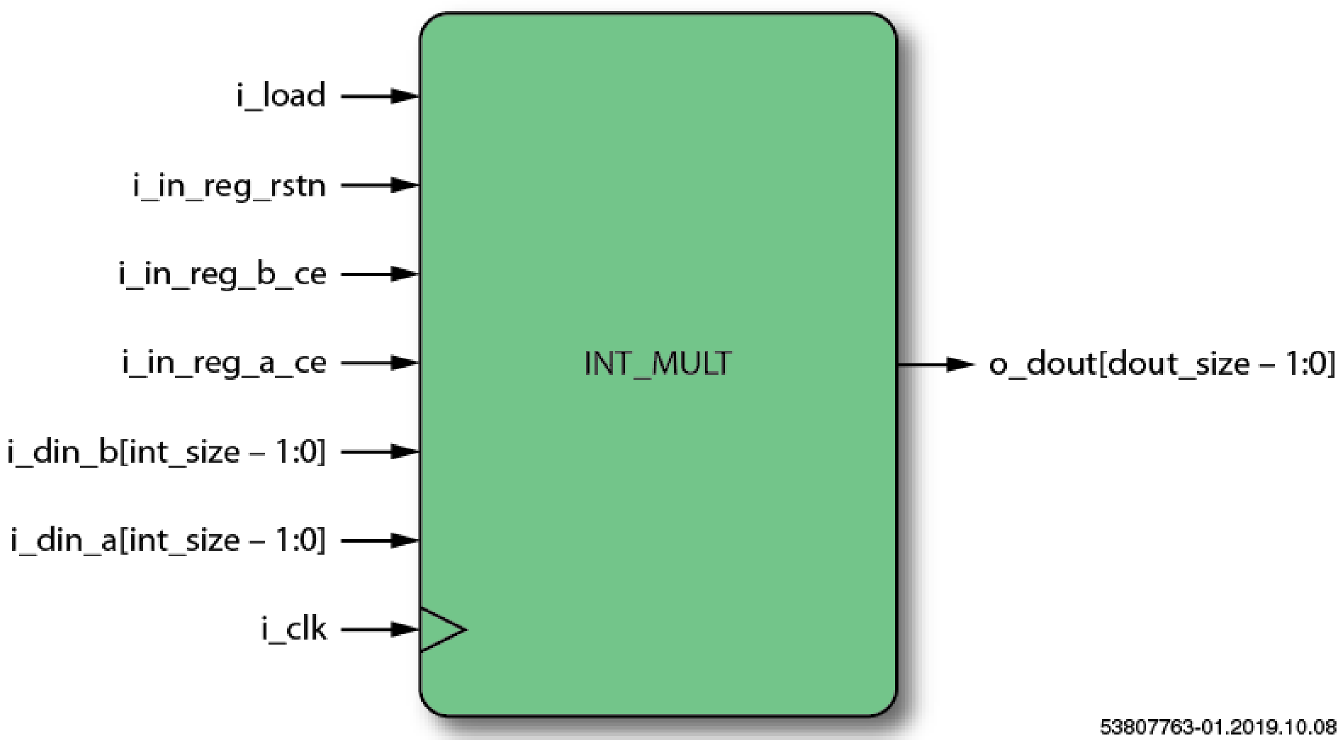

Figure 59: ***Integer Multiplier with Optional Accumulate***

Parameters

Table 186: *INT_MULT Parameters*

Parameter	Supported Values	Default	Description
int_size	3,4,6,7,8,16	8	Width of integer inputs.
int_unsigned	0, 1	0	0 – Integer values are unsigned (both inputs and outputs). 1 – Integer values are signed two's compliment (both inputs and outputs).
accumulate	0, 1	0	0 – No accumulation. 1 – Accumulate the sum on each output cycle. `i_load` used to reset accumulation.
in_reg_enable	0, 1	0	0 – No input registers. 1 – `i_din_a` and `i_din_b` registered. Registered controlled by `i_in_reg_a/b_ce` and `i_in_reg_rstn`. Adds a cycle of latency.
mult_reg_enable	0, 1	0	0 – Multiplier output used directly. 1 – Multiplier output registered before next stage. Adds a cycle of latency. This setting is primarily useful when accumulation is enabled.
out_reg_enable	0, 1	0	0 – `o_dout` sourced from either multiplier or accumulator outputs directly. 1 – `o_dout` registered internally. Adds a cycle of latency.
dout_size	48, 2 × int_size	[See description]	Width of o_dout output. Varies according to `accumulate` and `int_size` parameters. `accumulate` = 1 – dout_size = 48 `accumulate` = 0 – dout_size = 2 × `int_size`. Similar to all parameters, `dout_size` can also be overridden at instantiation.

Ports

Table 187: *INT_MULT 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[int_size-1:0]	Input	A data input to multiplier.(1)

Name	Direction	Description
i_din_b[int_size-1:0]	Input	B data input to multiplier.(1)
i_in_reg_a_ce	Input	Clock enable for `i_din_a` input.(2)
i_in_reg_b_ce	Input	Clock enable for `i_din_b` input.(2)
i_in_reg_rstn	Input	Synchronous active low reset for input registers.(2)
i_load	Input	Synchronous load of accumulator. Will reset accumulator value to (i_din_a * i_din_b).
o_dout[dout_size-1:0]	Output	Result of multiplication and accumulation
o_status[1:0]	Output	Indicates if errors occurred with o_dout.(3)

Table Notes

1. Width determined by fp_size parameter.
2. Wen parameter `in_reg_enable` = 1'b1. Unused when `in_reg_enable` = 1'b0.
3. See Output Status (see page 165) for details.

Inference

ACX_INT_MULT cannot be inferred. It has to be directly instantiated

Instantiation Templates

Verilog

```
// Verilog template for ACX_INT_MULT
ACX_INT_MULT #(
    .int_size             (int_size),
    .int_unsigned         (int_unsigned),
    .accumulate           (accumulate),
    .in_reg_enable        (in_reg_enable),
    .mult_reg_enable      (mult_reg_enable),
    .out_reg_enable       (out_reg_enable),
    .dout_size            (dout_size)
) 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_load               (user_i_load),
    .o_dout               (user_o_dout)
);
```

VHDL

```
-- VHDL Component template for ACX_INT_MULT
component ACX_INT_MULT is
generic (
    int_size            : integer := 8;
    int_unsigned        : integer := 0;
    accumulate          : integer := 0;
    in_reg_enable       : integer := 0;
    mult_reg_enable     : integer := 0;
    out_reg_enable      : integer := 0;
    dout_size           : integer := 48
);
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_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        => int_unsigned,
    accumulate          => accumulate,
    in_reg_enable       => in_reg_enable,
    mult_reg_enable     => mult_reg_enable,
    out_reg_enable      => out_reg_enable,
    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_load              => user_i_load,
    o_dout              => user_o_dout
);
```

INT_MULT_N

This macro computes N parallel multiplications with all numbers using the same format. There is no accumulation option.

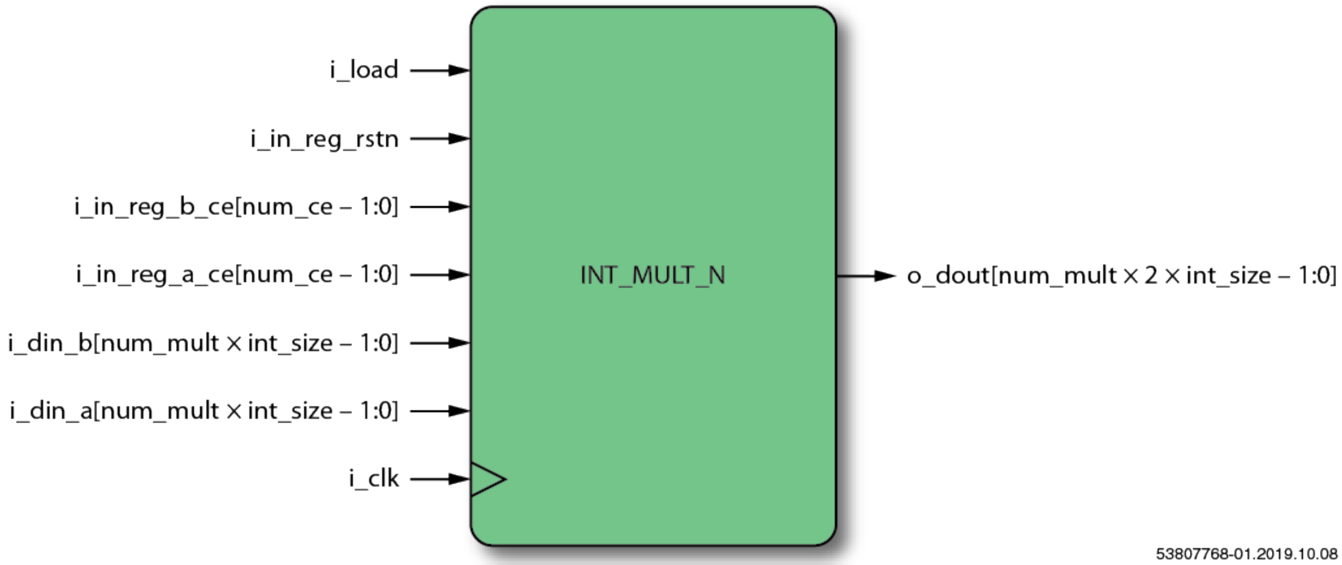

Figure 60: ***N Integer Parallel Multiplications***

Input Packing

Inputs and outputs are packed in the 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 +: int_size];
```

Clock Enables

If the input register is enabled, each input has its own clock enable if int_size = 6, 7, 8, or 16. 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.

Table 188: ***Maximum Parallel Multiplications***

int_size	Max Signed Multiplications	Max Unsigned Multiplications
3	8	8
4	8	4

int_size	Max Signed Multiplications	Max Unsigned Multiplications
6,7,8	4	4
16	2	2

Parameters

Table 189: *INT_MULT_N Parameters*

Parameter	Supported Values	Default	Description
int_size	3,4,6,7,8,16	8	Width of integer inputs.
num_mult	1-8	1	Number of parallel multiplications. Refer to Maximum Parallel Multiplications (see page 192) for the limits per number format.
int_unsigned	0, 1	0	0 – Integer values are unsigned (both inputs and outputs). 1 – Integer values are signed two's compliment (both inputs and outputs).
in_reg_enable	0, 1	0	0 – No input registers. 1 – Ports `i_din_a` and `i_din_b` registered. Registered controlled by `i_in_reg_a/b_ce` and `i_in_reg_rstn`. Adds a cycle of latency.
out_reg_enable	0, 1	0	0 – Port `o_dout` sourced from the multiplier output directly. 1 – Port `o_dout` registered internally. Adds a cycle of latency.

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 190: *INT_MULT_N 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 [num_mult × int_size − 1:0]	Input	Packed vector of A data input to multipliers.(1)
i_din_b [num_mult × int_size − 1:0]	Input	Packed vector of B data input to multipliers.(1)
i_in_reg_a_ce [num_ce − 1:0]	Input	Clock enable for `i_din_a` input(2).

Name	Direction	Description
i_in_reg_b_ce [num_ce − 1:0]	Input	Clock enable for `i_din_b` input[(2)].
i_in_reg_rstn	Input	Synchronous active-low reset for input registers[(2)].
o_dout [num_mult × 2 × int_size − 1:0]	Output	Aggregated vector of all multiplications.[(1)]

Table Notes

1. Width determined by `num_mult` and `int_size` parameters.
2. Wen parameter `in_reg_enable` = 1'b1. Unused when `in_reg_enable` = 1'b0.

Inference

ACX_INT_MULT_N cannot be inferred. It has to be directly instantiated

Instantiation Templates

Verilog

```
// Verilog template for ACX_INT_MULT_N
ACX_INT_MULT_N #(
    .int_size           (int_size),
    .num_mult           (num_mult),
    .int_unsigned       (int_unsigned),
    .in_reg_enable      (in_reg_enable),
    .out_reg_enable     (out_reg_enable)
) 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),
    .o_dout             (user_o_dout)
);
```

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        : integer := 0;
    in_reg_enable       : integer := 0;
    out_reg_enable      : 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;
    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        => int_unsigned,
    in_reg_enable       => in_reg_enable,
    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,
    o_dout              => user_o_dout
);
```

INT_MULT_ADD

This module computes a sum of products, SUM a(i) × b(*), with optional accumulation.

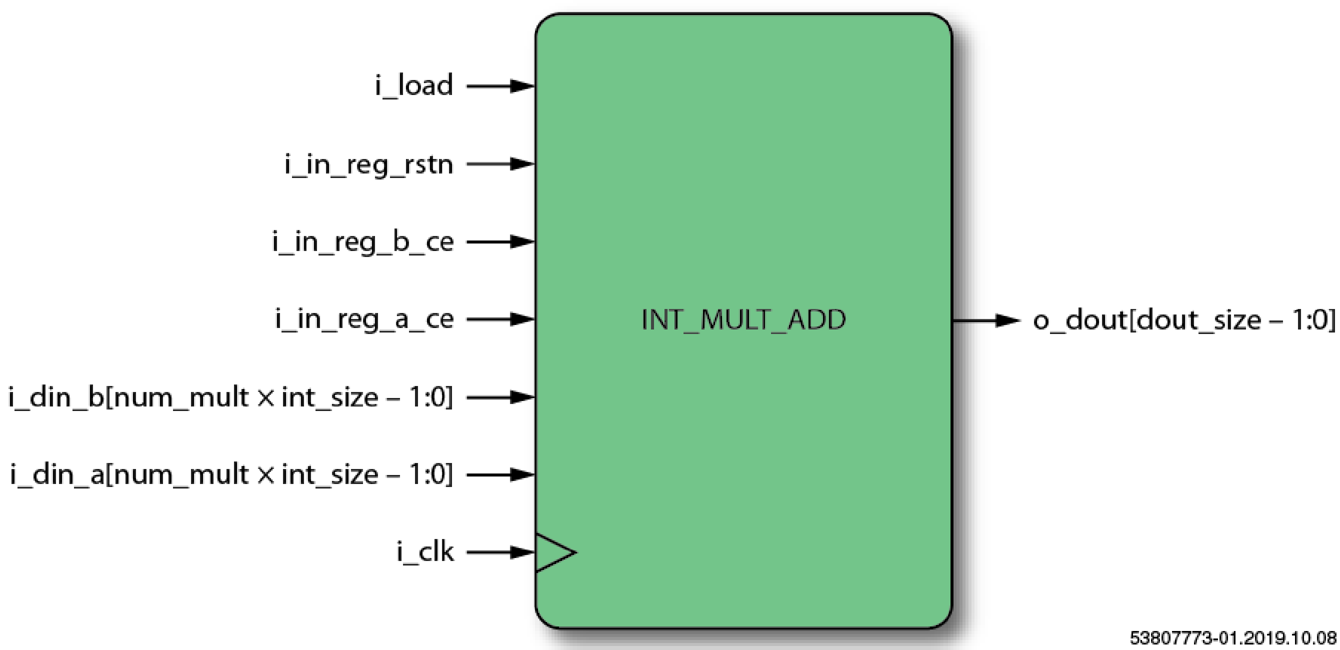

Figure 61: ***N Integer Sum of Products with Optional Accumulation***

Input Packing

Inputs are packed in the single input vectors

```
a(i) = i_din_a[i*int_size +: int_size];
b(i) = i_din_b[i*int_size +: int_size];
```

Clock Enables

If the input register is enabled, there is one 'ce' for all 'a' inputs, and one 'ce' for all 'b' inputs.

Maximum Parallel Multiplications

Parameter num_mult specifies the number of parallel multiplications. The maximum is determined by the input format:

Table 191: ***Maximum Parallel Multiplications***

int_size	Max Signed Multiplications	Max Unsigned Multiplications
3	24	16
4	16	12
6	12	10
7	10	8
8	8	8
16	4	4

Parameters

Table 192: ***INT_MULT_ADD Parameters***

Parameter	Supported Values	Default	Description
int_size	3,4,6,7,8,16	8	Width of integer inputs.
num_mult	1-8	1	Number of parallel multiplications. Refer to Maximum Parallel Multiplications (see page 192) for the limits per number format.
int_unsigned	0, 1	0	0 – Integer values are unsigned (both inputs and outputs). 1 – Integer values are signed two's compliment (both inputs and outputs).
accumulate	0, 1	0	0 – No accumulation. 1 – Accumulate the sum on each output cycle. `i_load` used to reset accumulation..

Parameter	Supported Values	Default	Description
in_reg_enable	0, 1	0	0 – No input registers. 1 – Ports `i_din_a` and `i_din_b` registered. Registered controlled by `i_in_reg_a/b_ce` and `i_in_reg_rstn`. Adds a cycle of latency.
mult_reg_enable	0, 1	0	0 – Multiplier output used directly. 1 – Multiplier output registered before next stage. Adds a cycle of latency.
out_reg_enable	0, 1	0	0 – Port `o_dout` sourced from either multiplier-adder tree or accumulator outputs directly. 1 – Port `o_dout` registered internally. Adds a cycle of latency.
dout_size	8-48	48	Width of o_dout output.

Ports

Table 193: *INT_MULT_ADD 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 [num_mult × int_size − 1:0]	Input	Packed vector of A data input to multipliers.(1)
i_din_b [num_mult × int_size − 1:0]	Input	Packed vector of B data input to multipliers.(1)
i_in_reg_a_ce	Input	Clock enable for `i_din_a` input(2).
i_in_reg_b_ce	Input	Clock enable for `i_din_b` input(2).
i_in_reg_rstn	Input	Synchronous active-low reset for input registers(2).
i_load	Input	Synchronous load of accumulator. Resets accumulator value to sum of the array of (`i_din_a(i)` * `i_din_b(i)`) multiplications.
o_dout [dout_size − 1:0]	Output	Sum and optional accumulation of all multiplications.(1)

Table Notes

1. Width determined by `num_mult` and `int_size` parameters.
2. Wen parameter `in_reg_enable` = 1'b1. Unused when `in_reg_enable` = 1'b0.

Inference

ACX_INT_MULT_ADD cannot be inferred. It has to be directly instantiated

Instantiation Templates

Verilog

```
// Verilog template for ACX_INT_MULT_ADD
ACX_INT_MULT_ADD #(
    .int_size           (int_size),
    .num_mult           (num_mult),
    .int_unsigned       (int_unsigned),
    .accumulate         (accumulate),
    .in_reg_enable      (in_reg_enable),
    .mult_reg_enable    (mult_reg_enable),
    .out_reg_enable     (out_reg_enable),
    .dout_size          (dout_size)
) 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_load             (user_i_load),
    .o_dout             (user_o_dout)
);
```

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        : integer := 0;
    accumulate          : integer := 0;
    in_reg_enable       : integer := 0;
    mult_reg_enable     : integer := 0;
    out_reg_enable      : 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_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         => int_unsigned,
    accumulate           => accumulate,
    in_reg_enable        => in_reg_enable,
    mult_reg_enable      => mult_reg_enable,
    out_reg_enable       => out_reg_enable,
    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_load               => user_i_load,
    o_dout               => user_o_dout
);
```

Chapter - 6: Memories

BRAM72K_FIFO

The BRAM72K_FIFO implements a 72-kb FIFO. Each port width can be independently configured and can be on different clock domains. For higher performance operation, an additional output register can be enabled.

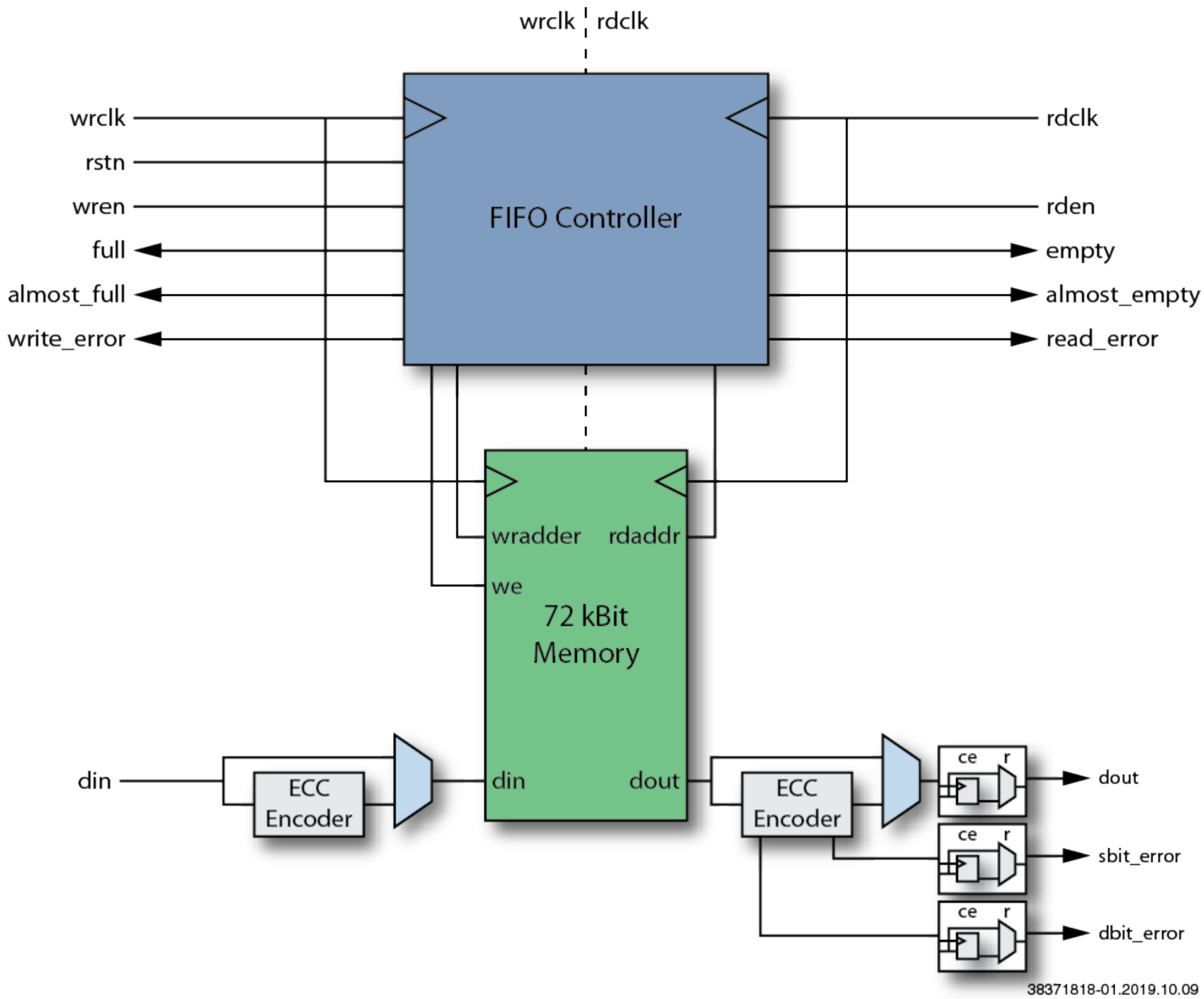

Figure 62: ***BRAM72K_FIFO Block Diagram***

Parameters

Table 194: *BRAM72K_FIFO Parameters*

Parameter	Supported Values	Default Value	Description
read_width	4, 8, 9, 16, 18, 32, 36, 64, 72, 128, 144	72	Controls the width of the read port.(†)
write_width	4, 8, 9, 16, 18, 32, 36, 64, 72, 128, 144	72	Controls the width of the write port.(†)
rdclk_polarity	"rise", "fall"	"rise"	Controls whether the rdclk signal uses the falling edge or the rising edge: • "rise" – rising edge • "fall" – falling edge
wrclk_polarity	"rise", "fall"	"rise"	Controls whether the wrclk signal uses the falling edge 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.
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	Almost full threshold. Controls the fill level above which which the almost_full output is asserted.
aempty_threshold	0 .. 14'h3FFF	14'h10	Almost empty threshold. Controls the fill level below which the almost_empty output is asserted.

Parameter	Supported Values	Default Value	Description
fwft_mode	0, 1	0	First-word fall through. 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. • 1 – data is presented at the output of the FIFO as soon as it is available, coincident with the de-assertion of empty. The data is held until rden is asserted.
ecc_encoder_enable	0, 1	0	Enables the ECC encoder, which calculates the ECC syndrome and stores it in the 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	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.

Table Note

† read_widths/write_widths of 128 and 144 consume the adjacent MLP site by using it as a route through to get the wide data in and out.

Ports

Table 195: ***BRAM72K_FIFO Pin Descriptions***

Name	Direction	Description
rstn	Input	Asynchronous reset input. This resets the entire FIFO.
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 push data to the FIFO.
din[143:0]	Input	The din signal determines the data to push to the memory array during a push operation. The data must be bottom justified, with upper bits tied to 0s.
full	Output	Asserted high when the FIFO is full.
almost_full	Output	Asserted high when remaining space in the FIFO is less than aempty_threshold.
write_error	Output	Asserted the cycle after a push 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 clock 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.
empty	Output	Asserted high when the FIFO is empty.
almost_empty	Output	Asserted high when the FIFO contains more than aempty_threshold elements.
read_error	Output	Asserted the cycle after a pop from the FIFO when the FIFO is already empty.
sbit_error[1:0]	Output	Asserted high when the data on dout[71:0] includes a single-bit error that was corrected.
dbit_error[1:0]	Output	Asserted high when the data on dout[71:0] includes an error or errors that were not corrected.
dout[71:0]	Output	Read port data output. For read operations, the dout output is updated with the memory contents addressed by the read pointer. The data must be bottom justified for widths < 36.

Memory Organization and Data Input/Output Pin Assignments

The BRAM72K_FIFO block supports memory widths from four to thirty-six bits wide. The width of the din data input is determined by the write_width parameter while the width of the dout data output is determined by the read_width parameter. The width of read port may be set to be different from the width of the write port. The data must be bottom justified.

Read and Write Operations

Timing Options

The BRAM72K_FIFO has two options for interface timing, controlled by the outreg_enable parameter:

- Latched mode – When outreg_enable is 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 is 1, the port is in registered mode. In registered mode, there is an additional register after the latch, supporting higher-frequency designs, providing a read operation with two cycles of latency.

Read Operation

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

Table 196: ***BRAM72K_FIFO Output Function Table for Latched Mode (Assumes Rising-Edge Clock and Active-High Port Enable)***

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 197: ***BRAM72K_FIFO Output Function Table for Registered Mode (Assumes Active-High Clock, Output Register Clock Enable, and Output Register Reset)***

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

Write Operation

Write operations are signalled by asserting the `wren` signal. The values of the `din` signal is stored in the memory array at the address indicated by the `wraddr` signal on the next active clock edge.

Inference

The 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)
);
```

BRAM72K_SDP

The block RAM primitive BRAM72K_SDP 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 a 512 × 144, 128 × 512, 1024 × 72, 1024 × 64, 2048 × 36, 2048 × 32, 4096 × 18, 4096 × 16, 8192 × 9, 8192 × 8, or 16384 × 4. 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. 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 specified by the user from either parameters or a memory initialization file.

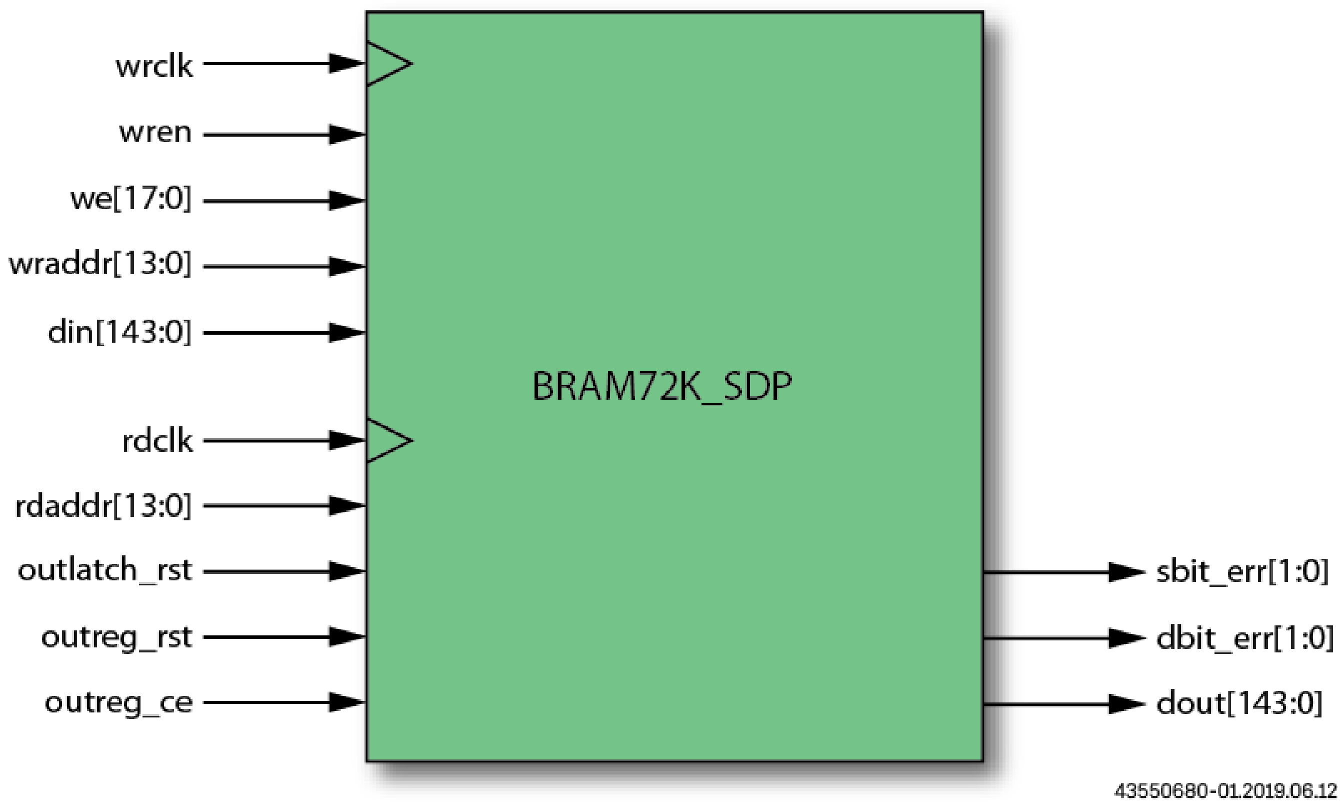

Figure 63: ***72-Kb Simple Dual-Port Memory with Error Correction***

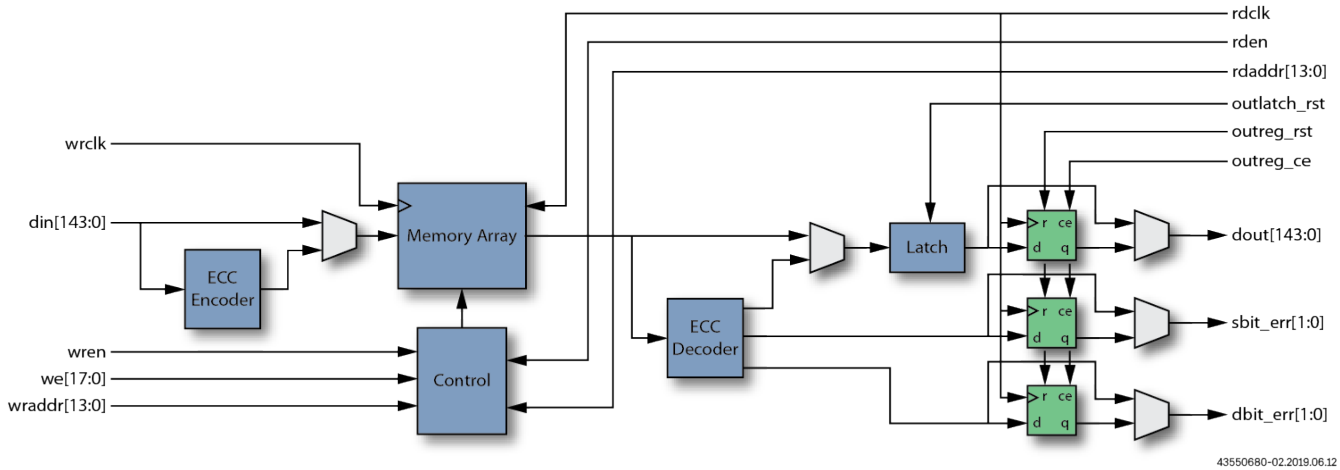

Figure 64: *BRAM72K_SDP Block Diagram*

Parameters

Table 198: *BRAM72K_SDP Parameters*

Parameter	Supported Values	Default Value	Description
read_width	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_widths of 72 or 144. **Note** Setting read_widths to 128 or 144 consumes the adjacent MLP site by using it as a route-through to get the wide data in and out.
write_width	4, 8, 9, 16, 18, 32, 36, 64, 72, 128, 144	72	Data width of write port. **Note** Setting write_widths to 128 or 144 consumes the adjacent MLP site by using it as a route-through to get the wide data in and out.
rdclk_polarity	"rise", "fall"	"rise"	Determines whether the rdclk signal uses the falling edge or the rising edge: • "rise" – Rising edge • "fall" – Falling edge
			Determines whether the wrclk signal uses the falling edge or the rising edge: • "rise" – Rising edge

Parameter	Supported Values	Default Value	Description
wrclk_polarity	"rise", "fall"	"rise"	• "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 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; where the output register is reset immediately following the assertion of the outreg_rstn input.
byte_width(†)	8,9	9	Determines whether the the we[] signal applies as 8-bit bytes or 9-bit bytes: • 8 – The 144-bit din signal is to be thought of as eighteen 8-bit bytes. During a write operation, we[17:0] selects which of the 8-bit bytes to write to, where we[0] implies that din[7:0] is written to memory, and we[17] implies that din[143:136] is written. • 9 – The 144-bit din signal is to be thought of as sixteen 9-bit bytes. During a write operation, we[7:0] selects which of the lower 9-bit bytes to write to and we[16:9] selects which of the higher 9-bit bytes to write to, 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 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. • If the mem_init_file is left at the default value of “”, the initial contents are defined by the values of the initd_0 - initd_1023 parameters. • If the memory initialization parameters and the mem_init_file parameters are not defined, the contents of the BRAM remain uninitialized.

Parameter	Supported Values	Default Value	Description
initd_0 to initd_1023	72 bit hex number	72'hX	The initd_0 through initd_1023 parameters define the initial contents of the memory contents associated with dout[71:0], as defined in defined in Memory Initialization (see page 217).
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.
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, port will present up to 1024 locations. • 1 - Enable remap. With read_width = 4, 8, 16, 32 or 64, when rdmsel=1'b1, rdaddr[11]=1'b0, port will present up to 1152 locations, reading the higher order data bits as extended memory address locations. See Advanced Modes (see page 220) for full details.
write_remap	0, 1	0	Enable write port to be remapped: • 0 - Disable remap. In byte_mode=8, port will present up to 1024 locations. • 1 - Enable remap. With write_width = 4, 8, 16, 32 or 64, when wrmsel=1'b1, wraddr[11]=1'b0, port will present up to 1152 locations, writing the extended memory address locations to the higher order data bits. See Advanced Modes (see page 220) for full details.

Table Note

† The following write and read port widths require byte_width to be set to 8:

read/write_width = 4, 8, 16, 32, 64 or 128.

The following write and read port widths require byte_width to be set to 9

read/write_width = 9, 18 or 36.

The following write and read port widths are allowed to use either byte_width of 8 or 9

read/write_width = 72 or 144.

Ports

Table 199: ***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.
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 – LRAM slave mode. Write-side control signals sent to MLP integrated LRAM, configured to 72 × 32. See Advanced Modes (see page 220) 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.
rdclk	Input	Read clock input. Read operations are fully synchronous and occur upon the active edge of the `rdclk` clock input when the `rden` 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.
rdaddr[13:0]	Input	The `rdaddr` signal determines which memory location is being read from. See the read port address and data bus mapping tables below.
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 – LRAM slave mode. Read-side control signals sent to MLP integrated LRAM, configured to 72 × 32.

Name	Direction	Description
		See Advanced Modes (see page 220) 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.
sbit_error[1:0]	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 72-bit width, only the lower bit must be connected. ECC is done on each 72-bits separately.
dbit_error[1:0]	Output	Dual-bit error (active high). The `dbit_error` signal is asserted during a read operation when `ecc_decoder_enable` = 1 and a 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 72-bit width, only the lower bit must be connected. ECC is done on each 72-bits separately.

Memory Organization and Data Input/Output Pin Assignments

Supported Width Combinations

The BRAM72K_SDP block supports a variety of memory width combinations, as shown in the following table.

Table 200: *BRAM72K_SDP Supported Data Widths*

	Write Data Width										
Read Data Width	**144**	**72**	**36**	**18**	**9**	**128**	**64**	**32**	**16**	**8**	**4**
144	✓	✓	✓	✓	✓		✓				
72	✓	✓	✓	✓	✓		⚠				
36			✓	✓	✓		⚠				
18			✓	✓	✓		⚠				
9			✓	✓	✓		⚠				
128						✓	✓	✓	✓	✓	✓
64	⚠	⚠	⚠	⚠	⚠	✓	✓	✓	✓	✓	✓
32						✓	✓	✓	✓	✓	✓
16						✓	✓	✓	✓	✓	✓
8						✓	✓	✓	✓	✓	✓
4						✓	✓	✓	✓	✓	✓

Table Notes

1. It's possible to configure the write width to be one of 32, 16, 8, or 4 bits, and the read width to be one of 36, 18, or 9 bits. In this configuration, the high-order data bit of every 9-bit group will be unknown.
2. It's possible to configure the write width to be one of 36, 18, or 9 bits, and the read width to be one of 32, 16, 8, or 4 bits. In this configuration, the high-order data bit of every 9-bit group can never be read back.

Write Data Port Usage

Table 201: ***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] wraddr[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 202: ***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(†)	user_dout[35:0] <= dout[35:0]	rdaddr[13:3] <= user_rdaddr[10:0] rdaddr[2:0] <= 3'b0
32 × 2048(†)	user_dout[31:0] <= dout[31:0]	rdaddr[13:3] <= user_rdaddr[10:0] rdaddr[2:0] <= 3'b0
18 × 4096(†)	user_dout[17:0] <= dout[17:0]	rdaddr[13:2] <= user_rdaddr[11:0] rdaddr[1:0] <= 2'b0
16 × 4096(†)	user_dout[15:0] <= dout[15:0]	rdaddr[13:2] <= user_rdaddr[11:0] rdaddr[1:0] <= 2'b0
9 × 8192(†)	user_dout[8:0] <= dout[8:0]	rdaddr[13:1] <= user_rdaddr[12:0] rdaddr[0] <= 1'b0
8 × 8192(†)	user_dout[7:0] <= dout[7:0]	rdaddr[13:1] <= user_rdaddr[12:0] rdaddr[0] <= 1'b0
4 × 16384(†)	user_dout[3:0] <= dout[3:0]	rdaddr[13:0] <= user_rdaddr[13:0]

Table Note

† Not supported for Write Width of 72 or 144

Read and Write Operations

Timing Options

The BRAM72K_SDP has two options for interface timing, controlled by the outreg_enable parameter:

- Latched mode – When outreg_enable is 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 is 1, the port is in registered mode. In registered mode, there is an additional register after the latch, supporting higher-frequency designs, providing a read operation with two cycles of latency.

Read Operation

Read operations are signaled by driving the `rdaddr` signal with the address to be read and asserting the `rden` signal. The requested read data arrives on the `dout` signal on the following clock cycle or the cycle after depending on the outreg_enable parameter value.

Table 203: ***BRAM72K_SDP Output Function Table for Latched Mode (Assumes Rising-Edge Clock and Active-High Port Enable)***

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 204: ***BRAM72K_SDP Output Function Table for Registered Mode (Assumes Active-High Clock, Output Register Clock Enable, and Output Register Reset)***

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

Write Operation

Write operations are signaled by asserting the `wren` signal. The values 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 of the ports access the same memory location(s) within the same clock cycle (both ports connected to the same clock), or within a fixed time window (if each port is connected to a different clock). If one of the ports is writing an address while the other port is reading the same address (qualified with overlapping write enables per bit), the write operation takes precedence, but the read data is invalid. The user may reliably read the data 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 BRAM72K_SDP instance with the output register disabled. The following describes the behavior of the 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 `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, it 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 data is output on the following cycle.
    - outreg_enable = 1 – The data 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 data is output on the following cycle.
    - outreg_enable = 1 – The data 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.

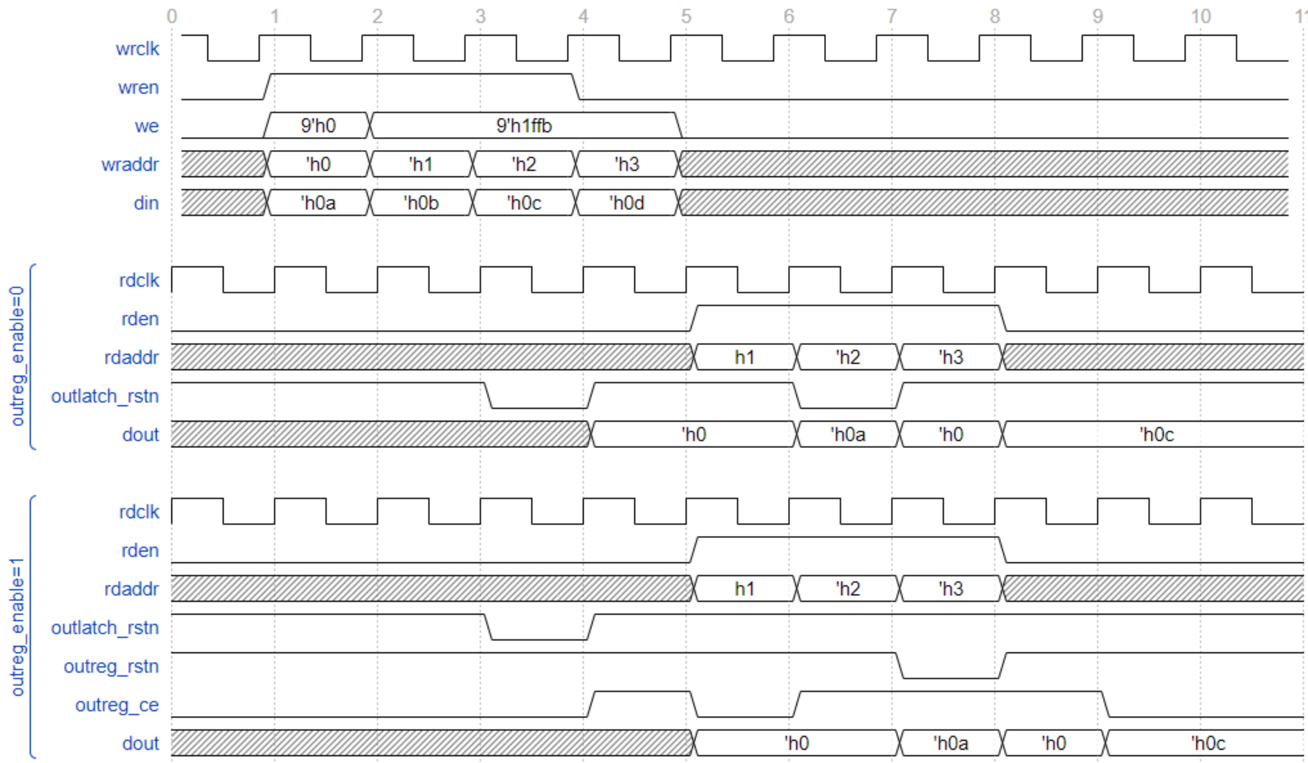

Figure 65: ***BRAM72K_SDP Timing Diagram***

Memory Initialization

Initializing with Parameters

The data portion 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 0 mapped to bit zero of parameter initd_0 and bit 72'703 mapped to bit 71 of parameter initd_1023.

Initializing with Memory Initialization File

A 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 BRAM72K_SDP memory organization is configured with the byte_mode parameter as either 9-bit byte or 8-bit byte mode. For read and write data widths, the mem_init_file will contain 1024 lines with 72 bits of init data per line, organized as follows:

Table 205: ***9-bit Byte Mode (byte_width == 9)***

Line in mem_init_file	Corresponding initd* Parameter	Bits:	71:63	62:54	53:45	44:36	35:27	26:1
1st line in file	initd_0		9byte7	9byte6	9byte5	9byte4	9byte3	9byte2

Line in mem_init_file	Corresponding initd* Parameter	Bits:	71:63	62:54	53:45	44:36	35:27	26:1
2nd line in file	initd_1		9byte15	9byte14	9byte13	9byte12	9byte11	9byte1
...	...		...	...	...	...	...	...
1024th line in file	initd_1023		9byte8191	9byte8190	9byte8189	9byte8188	9byte8187	9byte8

Table 206: ***8-bit Byte Mode (byte_width == 8)***

Line in mem_init_file	Corresponding initd* Parameter	Bits:	71: 64	63:56	55:48	47:40	39:32	31:24	23:1
1st line in file	initd_0		8'h0	byte7	byte6	byte5	byte4	byte3	byte2
2nd line in file	initd_1		8'h0	byte15	byte14	byte13	byte12	byte11	byte1(
...	...			...	...	...	...	...	...
1024th line in file	initd_1023		8'h0	byte8191	byte8190	byte8189	byte8188	byte8187	byte81

Table 207: ***8-bit Byte Mode (byte_width == 8) Special Case for read_width of 72 or 144***

Line in mem_init_file	Corresponding initd* Parameter	Bits:	71:64	63:56	55:48	47:40	39:32	31:24	
1st line in file	initd_0		byte8	byte7	byte6	byte5	byte4	byte3	b
2nd line in file	initd_1		byte17	byte16	byte15	byte14	byte13	byte12	b
...	...			...	...	...	...	...	..
1024th line in file	initd_1023		byte9215	byte9214	byte9213	byte9212	byte9211	byte9210	b

A number entry may 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 BRAM72K_SDP is initialized with the values in the file referenced by the mem_init_file parameter. If the mem_init_file is left at the default value of “”, the initial contents are defined by the values of the initd_0 - initd_1023. 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 BRAM72K_SDP defined by the enable_ecc_encoder and enable_ecc_decoder parameters shown in the table below

Table 208: *BRAM72K_SDP ECC Modes of Operation*

enable_ecc_decoder	enable_ecc_decoder	ECC Operation Mode
0	0	ECC encoder and decoder disabled; standard 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 BRAM72K_SDPW 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 `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.

ECC Encode-Only Operation Mode

The ECC encode-only operation has the ECC encoder enabled and the ECC decoder disabled. This mode allows the user to write 64 bits of data and have 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 have error correction for off-chip memories.

ECC Decode-Only Operation Mode

The ECC decode-only operation has the ECC encoder disabled and the ECC decoder enabled. This mode bypasses the ECC encoder and allows the user to write 72-bit data directly into the memory array during write operations. Read operations places 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 have error correction for off-chip memories.

Using BRAM72K_SDP as a Read-Only Memory (ROM)

The 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 217)) and tying the `wren` signal to its de-asserted value. All signals on the read-side of the BRAM72K_SDP operate as described above. This configuration allows the user to read from the memory, but not write to it.

Advanced Modes

The BRAM72K_SDP supports two advanced modes that allows 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 MLP72, (see Speedster7t MLP 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

(wr/rdmsel = 1'b1, wr/rdaddr[11]=1'b0)

The BRAM72K_SDP is natively configured as a 72x1024 bit memory, with 9-bit bytes. However users may require to access the memory using traditional 8-bit byte access, 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 BRAM72K_SDP uniquely offers an 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, with the memory only supporting a maximum of 1024 word depth.

To enable the remap mode for the either port, the respective parameter, `write_remap` and `read_remap` must be set to 1'b1.

With the appropriate parameter enabled, `wr/rdmsel = 1'b1` , and `wr/rdaddr[11] = 1'b0` , the relevant BRAM72K_SDP port will operate 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 `wr/rdaddr[11]` set to 1'b0, the further address bits `wr/rdaddr[6:0]` are used to access the additional 128 words of memory.

LRAM Slave Mode

(wr/rdmsel = 1'b1, wr/rdaddr[11]=1'b1)

This mode corresponds to Mode 0 in Speedster7t MLP LRAM, and is when the LRAM is slaved to the BRAM72K_SDP operation. Accesses to BRAM72K_SDP are transferred to the LRAM72, which is configured in 72x32-bit mode. Address bits `wr/rdaddr[4:0]` control access to the LRAM72 memory locations.

Inference

The 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.

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    )
);
```

LRAM2K_FIFO

The LRAM2K_FIFO implements a 2-kb FIFO. 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.

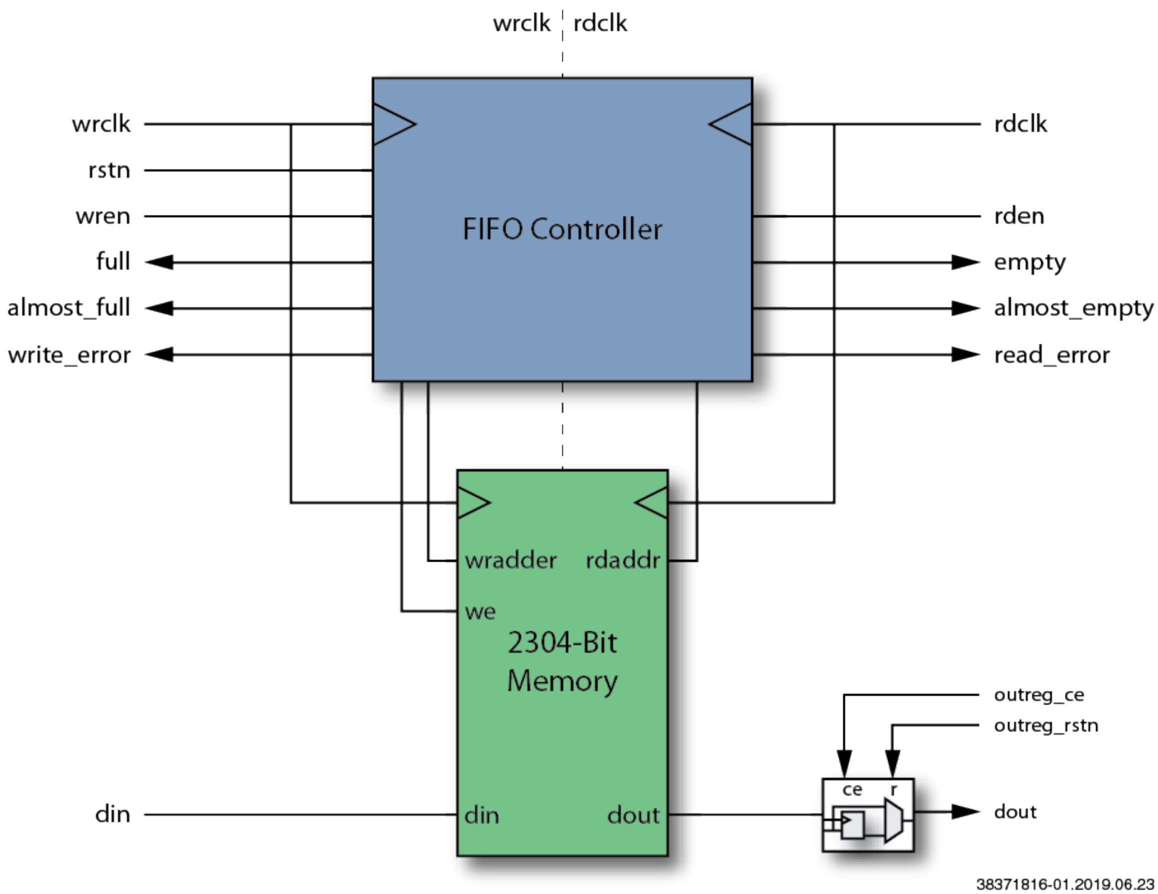

Figure 66: ***LRAM2K_FIFO Block Diagram***

Parameters

Table 209: *LRAM2K_FIFO Parameters*

Parameter	Supported Values	Default Value	Description
read_width	36,72	72	Controls the width of the read port.
write_width	36,72	72	Controls the width of the write port.
rdclk_polarity	"rise", "fall"	"rise"	Controls whether the rdclk signal uses the falling edge or the rising edge: • "rise" – Rising edge • "fall" – Falling edge
wrclk_polarity	"rise", "fall"	"rise"	Controls whether the wrclk signal uses the falling edge 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.
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 .. 6'h3F	6'h4	Almost full threshold. Controls the fill level above which which the almost_full output is asserted.
aempty_threshold	0 .. 6'h3F	6'h4	Almost empty threshold. Controls the fill level below which the almost_empty output is asserted.
			First-word fall through. 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.

Parameter	Supported Values	Default Value	Description
fwft_mode	0, 1	0	• 1 – Data is presented at the output of the FIFO as soon as it is available, coincident with the de-assertion of empty. The data is held until rden is asserted.

Ports

Table 210: *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 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 push data to the FIFO.
din[71:0]	Input	The din signal determines the data to push to the memory array during a push operation. The data must be bottom justified for widths < 36.
full	Output	Asserted high when the FIFO is full.
almost_full	Output	Asserted high when remaining space in the FIFO is less than aempty_threshold.
write_error	Output	Asserted the cycle after a push 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 clock 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 output register is reset to 0.
outreg_ce	Input	Output register clock enable (active-high). When outreg_enable is 1, de-asserting outreg_ce causes the BRAM to hold the dout signal unchanged, independent of a read operation. When outreg_enable is 0, outregce input is ignored.
empty	Output	Asserted high when the FIFO is empty.
almost_empty	Output	Asserted high when the FIFO contains more than aempty_threshold elements.
read_error	Output	Asserted the cycle after a read request to the FIFO when the FIFO is already empty.

Name	Direction	Description
dout[71:0]	Output	Read port data output. For read operations, the dout output is updated with the memory contents addressed by the read pointer. The data must be bottom justified for widths < 36.

Memory Organization and Data Input/Output Pin Assignments

The LRAM2K_FIFO block supports memory widths from 36 or 72. The width of the `din` data input is determined by the write_width parameter while the width of the `dout` data output is determined by the read_width parameter. The width of read port may be set to be different from the width of the write port.

Read and Write Operations

Timing Options

The LRAM2K_FIFO has two options for interface timing, controlled by the outreg_enable parameter:

- Latched mode – When outreg_enable is 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 is 1, the port is in registered mode. In registered mode, there is an additional register after the latch, supporting higher-frequency designs, providing a read operation with two cycles of latency.

Read Operation

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

Table 211: ***LRAM2K_FIFO Output Function Table for Latched Mode (Assumes Rising-Edge Clock and Active-High Port Enable)***

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 212: ***LRAM2K_FIFO Output Function Table for Registered Mode (Assumes Active-High Clock, Output Register Clock Enable, and Output Register Reset)***

Operation	rdclk	outreg_rstn	outregce	dout
Hold	X	X	X	Previous dout

Operation	rdclk	outreg_rstn	outregce	dout
Reset Output	↑	0	1	0
Hold	↑	1	0	Previous dout
Update Output	↑	1	1	Registered from latch output

Write Operation

Write operations are signaled by asserting the `wren` signal. The values of the `din` signal is stored in the memory array at the address indicated by the `wraddr` signal on the next active clock edge.

Inference

The LRAM2K_FIFO is not inferrable.

Instantiation Template

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),
    rstn                 (rstn),
    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)
);
```

LRAM2K_SDP

LRAM2K_SDP implements a 2-kb simple-dual-port (SDP) memory block with one write port and one read port. Each port can be configured as either a 32 × 72 or a 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.

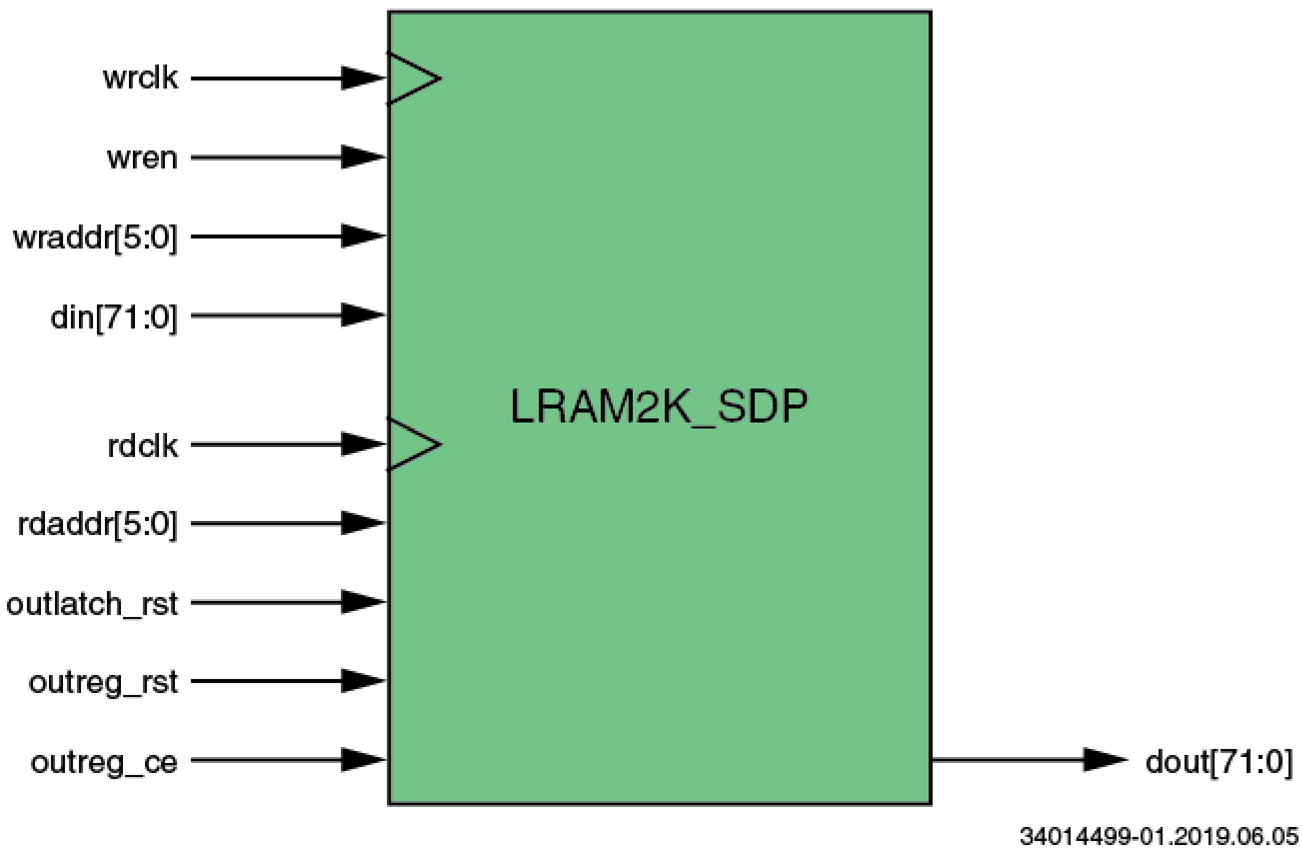

Figure 67: ***2-kb Simple Dual-Port Memory***

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

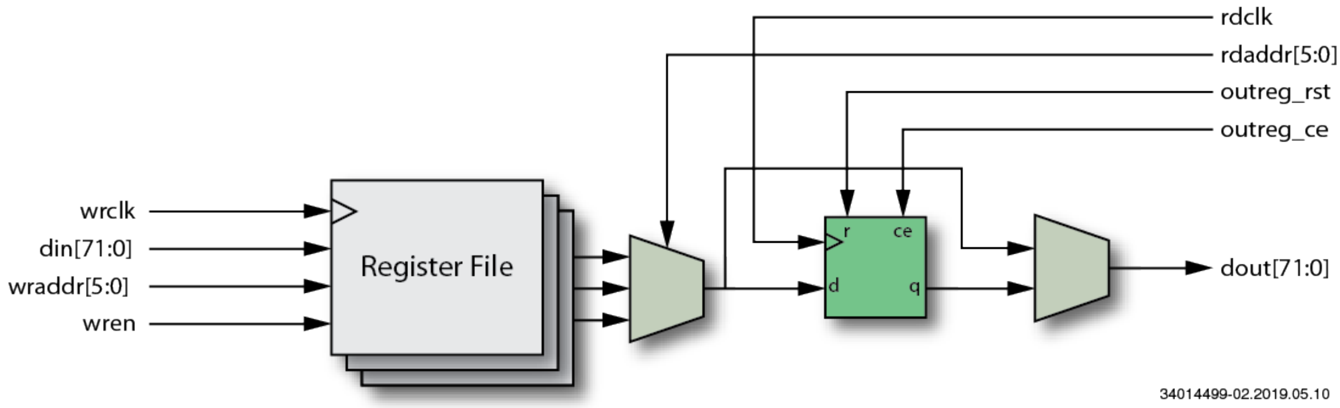

Figure 68: ***LRAM2K_SDP Block Diagram***

Parameters

Table 213: ***LRAM2K_SDP Parameters***

Parameter	Supported Values	Default Value	Description
read_width	36, 72	72	Read port data width.
write_width	36, 72	72	Write port data width.
wrclk_polarity	"rise", "fall"	0	Determines whether the wrclk signal uses the falling edge or the rising edge: • "rise" – Rising edge. • "fall" – Falling edge.
rdclk_polarity	"rise", "fall"	0	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 one cycle. • 1 – Enables the output register and results in a read latency of two cycles.
			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.

Parameter	Supported Values	Default Value	Description
outreg_sr_assertion	"clocked", "unclocked"	"clocked"	• "unclocked" – Asynchronous reset; where 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 231). • If the mem_init_file is left at the default value of "", the initial contents are defined by the values of the initd_0 – 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 to initd_31	72-bit hex number	X	The initd_0 through initd_31 parameters define the initial contents of the memory contents associated with dout[71:0] as defined in Memory Initialization (see page 231).

Ports

Table 214: *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 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 do 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[71:0]	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 `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.

Name	Direction	Description
outreg_rstn	Input	Output register reset. When outreg_rstn is asserted low, the value of the output registers are reset to 0. The parameter `outreg_sr_assertion` controls whether this reset is synchronous or asynchronous.
outreg_ce	Input	Output register clock enable (active high). When `outreg_enable` = 1, de-asserting `outreg_ce` causes the LRAM2K_SDP to hold the `dout` signal unchanged, independent of a read operation. When `outreg_enable` = 0, the `outreg_ce` input is ignored.
dout[71:0]	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 LRAM2K_SDP block supports a 72-bit wide memory, with each port arranged as a 32 × 72 or 64 × 36 memory array.

Read and Write Operations

Timing Options

The LRAM2K_SDP has two options for interface timing, controlled by the `outreg_enable` parameter:

- Combinatorial mode – When the parameter `outreg_enable` is 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 the parameter `outreg_enable` is 1, the port is in registered mode. In registered mode, there is an additional register supporting higher-frequency designs, providing 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). 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 user may reliably read the data on the next cycle if there is no longer a write collision.

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 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 LRAM2K_SDP memory organization is always configured in 9-bit byte mode (equivalent to the BRAM72K_SDP when byte_width == 9). For both ×72 and ×36 modes, the mem_init_file will contain 32 lines with 72-bits of initialization data per line, organized as detailed in the following table.

Table 215: *Initialization File Organization*

Line in mem_init_file	Corresponding initd* Parameter	Bits							
		71:63	62:54	53:45	44:36	35:27	26:18	17:9	8:0
1st line in file	initd_0	9byte7	9byte6	9byte5	9byte4	9byte3	9byte2	9byte1	9byte0
2nd line in file	initd_1	9byte15	9byte14	9byte13	9byte12	9byte11	9byte10	9byte9	9byte8
...	...	...	...	...	...	...	...	...	...
32nd line in file	initd_31	9byte255	9byte254	9byte253	9byte252	9byte251	9byte250	9byte249	9byte248

A number entry may 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 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 LRAM2K_SDP as a Read-Only Memory (ROM)

The 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 231)) and never asserting the `wren` signal. All signals on the read side of the LRAM2K_SDP operate as described above. This configuration allows the user to read from the memory, but not write to it.

Inference

The 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.

Instantiation Template

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       )
);
```

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

NAP_AXI_MASTER

The 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.

Parameters

Table 216: *Parameters*

Parameter	Supported Values	Default Value	Description
n2s_arbitration_schedule s2n_arbitration_schedule	Any 32-bit value	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 a competing flit 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 high priority on every fourth cycle. A value of 32'hAAAA_AAAA means that the local node has high priority on every second cycle. A value of 32'hFFFF_FFFF means that the local node always has high priority.
ns_router_id		4'h1	TBD

Ports

Table 217: *Port Descriptions*

Name	Direction	Description
rstn	Input	Asynchronous reset input. This signal resets the NAP. This signal does not affect the NOC router.
output_rstn	Output	Reset output from NAP to fabric logic.
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[40:0]	Output	
awlen[7:0]	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	
bid[7:0]	Input	AXI write response channel.
bresp[1:0]	Input	
bvalid	Input	
bready	Output	

Name	Direction	Description
arid[7:0]	Output	AXI read command channel.
araddr[40:0]	Output	
arlen[7:0]	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	
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 flit type does not match node configuration. • 3'b001 – Received flit destination ID does not match node ID. • Others – Reserved.

NAP_AXI_SLAVE

The 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.

Parameters

Table 218: *Parameters*

<table>
<tr><th>Parameter</th><th>Supported Values</th><th>Default Value</th><th>Description</th></tr>
<tr><td>e2w_arbitration_schedule</td><td rowspan="2">Any 32-bit value</td><td rowspan="2">32'hAAAA_AAAA</td><td rowspan="2">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 a competing flit 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 high priority on every fourth cycle. A value of 32'hAAAA_AAAA means that the local node has high priority on every second cycle. A value of 32'hFFFF_FFFF means that the local node always has high priority.</td></tr>
<tr><td>w2e_arbitration_schedule</td></tr>
<tr><td>ew_router_id</td><td></td><td>4'h1</td><td>TBD</td></tr>
<tr><td>timeout_value</td><td>Any 32-bit value</td><td>32'h2000_0000</td><td>The number of FPGA clock cycles to wait for a transaction response before issuing a timeout error.</td></tr>
</table>

Ports

Table 219: *Port Descriptions*

Name	Direction	Description
rstn	Input	Asynchronous reset input. This signal resets the NAP. This signal does not affect the NOC router.
output_rstn	Output	Reset output from NAP to fabric logic.
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[40:0]	Output	
awlen[7:0]	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	
bid[7:0]	Input	AXI write response channel.
bresp[1:0]	Input	
bvalid	Input	
bready	Output	

Name	Direction	Description
arid[7:0]	Output	AXI read command channel.
araddr[40:0]	Output	
arlen[7:0]	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	
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 flit type doesn't match node configuration. • 3'b001 – Received flit destination ID doesn't match node ID. • 3'b010 – Transaction timeout. • Others – Reserved.

Instantiation Template

Verilog

```
ACX_NAP_AXI_SLAVE #(

) 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.
```

NAP_HORIZONTAL

The NAP_HORIZONTAL component provides for 288-bit bidirectional data transport. When the user presents a word of data 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 row indicated by the `tx_dest[3:0]` signal, which then presents it to the FPGA logic using the destination NAP's receiver interface.

Parameters

Table 220: *Parameters*

Parameter	Supported Values	Default Value	Description
e2w_arbitration_schedule	Any 32-bit value	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 a competing flit 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 high priority on every fourth cycle. A value of 32'hAAAA_AAAA means that the local node has high priority on every second cycle. A value of 32'hFFFF_FFFF means that the local node always has high priority.
w2e_arbitration_schedule			
ew_router_id		4'h1	

Ports

Table 221: *Port Descriptions*

Name	Direction	Description
rstn	Input	Asynchronous reset input. This signal resets the NAP. This signal does not affect the NOC router.
output_rstn	Output	Reset output from NAP to fabric logic
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.

Name	Direction	Description
tx_dest[3:0]	Input	The 16-bit destination ID of the row (NAP_VERTICAL) or column (NAP_HORIZONTAL) that data is to be transmitted to.
tx_sop tx_eop	Input	Transmitted data start and end of packet indicators.
tx_data[287:0]	Input	Data transmitted to the destination node.
rx_ready	Input	Assert high to indicate 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 row (NAP_VERTICAL) or column (NAP_HORIZONTAL) that the data word was transmitted from.
rx_sop rx_eop	Output	Received data start and end of packet indicators.
rx_data[287:0]	Output	Received data.

NAP_VERTICAL

The NAP_VERTICAL component provides for 293-bit (vertical) or 288-bit (horizontal) 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 are 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.

Parameters

Table 222: *Parameters*

Parameter	Supported Values	Default Value	Description
n2s_arbitration_schedule s2n_arbitration_schedule	Any 32-bit value	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 a competing flit 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 high priority on every fourth cycle. A value of 32'hAAAA_AAAA means that the local node has high

Parameter	Supported Values	Default Value	Description
			priority on every second cycle. A value of 32'hFFFF_FFFF means that the local node always has high priority.
ns_router_id		4'h1	TBD

Ports

Table 223: *Port Descriptions*

Name	Direction	Description
rstn	Input	Asynchronous reset input. This signal resets the NAP. This signal does not affect the NoC router.
output_rstn	Output	Reset output from NAP to fabric logic
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 16-bit destination ID of the row (NAP_VERTICAL) or column (NAP_HORIZONTAL) that this word of data should be sent to.
tx_sop tx_eop	Input	Start of packet and end of packet indicators. These values are sent unmodified to the destination.
tx_data[292:0]	Input	Data transmitted to the destination node.
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 (NAP_VERTICAL) or column (NAP_HORIZONTAL) originating the data.
rx_sop rx_eop	Output	Start of packet and end of packet indicators. These values are unmodified from the source.
rx_data[292:0]	Output	Received data.

Revision History

Version	Date	Description
0.9	24 Jun 2019	• Initial preliminary release.
0.91	10 Oct 2019	• Added full details on MLP72 including LRAM, integer, block floating-point and floating-point modes. • Updated BRAM72K advanced modes. • Add RLB description and mutually exclusive modes. • Restructured document to move Primitives under Fabric Architecture (see page 10). • Added ACX_ prefix to all templates. • Added diagrams and tables for all floating point and integer library macro paths. • Added diagram for ALU8.