Vivado Design Suite User Guide: Synthesis (UG901)

HDL Coding Techniques

HDL Coding Techniques

Introduction

Hardware Description Language (HDL) coding techniques let you:

- Describe the most common functionality found in digital logic circuits.
- Take advantage of the architectural features of AMD devices.
- Templates are available from the AMD Vivado™ Design Suite Integrated Design Environment (IDE). To access the templates, in the Window Menu, select Language Templates.

Coding examples are included in this chapter. Download the coding example files from Coding Examples .

Advantages of VHDL

- Enforces stricter rules, in particular strongly typed, less permissive and error-prone
- Initialization of RAM components in the HDL source code is easier (Verilog initial blocks are less convenient)
- Package support
- Custom types
- Enumerated types
- No `reg` versus `wire` confusion

Advantages of Verilog

- C-like syntax
- More compact code
- Block commenting
- No heavy component instantiation as in VHDL

Advantages of SystemVerilog

- More compact code compared to Verilog
- Structures and enumerated types for better scalability
- Interfaces for higher level of abstraction
- Supported in Vivado synthesis

Flip-Flops, Registers, and Latches

Vivado synthesis recognizes Flip-Flops, Registers with the following control signals:

- Rising or falling-edge clocks
- Asynchronous Set/Reset
- Synchronous Set/Reset
- Clock Enable

Flip-Flops, Registers, and Latches are described with:

- sequential process (VHDL)
- `always` block (Verilog)
- `always_ff` for flip-flops, `always_latch` for Latches (SystemVerilog)

The `process` or `always` block sensitivity list should list:

- The clock signal
- All asynchronous control signals

Flip-Flops and Registers Control Signals

Flip-Flops and Registers control signals include:

- Clocks
- Asynchronous and synchronous set and reset signals
- Clock enable

Coding Guidelines

- Do not asynchronously set or reset registers.
    - Control set remapping becomes impossible.
    - Sequential functionality in device resources, such as block RAM components and DSP blocks, can be set or reset synchronously only.
    - If you use asynchronously set or reset registers, you cannot leverage device resources or are configured sub-optimally.
- Do not describe flip-flops with both a set and a reset.
    - No flip-flop primitives feature both a set and a reset, whether synchronous or asynchronous.
    - Flip-flop primitives featuring both a set and a reset can adversely affect area and performance.
- Avoid operational set/reset logic whenever possible. There can be other, less expensive, ways to achieve the desired effect, such as taking advantage of the circuit global reset by defining an initial content.
- Always describe the clock enable, set, and reset control inputs of flip-flop primitives as active-High. If they are described as active-Low, the resulting inverter logic penalizes circuit performance.

Flip-Flops and Registers Inference

Vivado synthesis infers four types of register primitives depending on how the HDL code is written:

FDCE
    D flip-flop with Clock Enable and Asynchronous Clear

FDPE
    D flip-flop with Clock Enable and Asynchronous Preset

FDSE
    D flip-flop with Clock Enable and Synchronous Set

FDRE
    D flip-flop with Clock Enable and Synchronous Reset

Flip-Flops and Registers Initialization

To initialize the content of a Register at circuit power-up, specify a default value for the signal during declaration.

Flip-Flops and Registers Reporting

- Registers are inferred and reported during HDL synthesis.
- The number of Registers inferred during HDL synthesis might not precisely equal the number of Flip-Flop primitives in the Design Summary section.
- The number of Flip-Flop primitives depends on the following processes:
    - Absorption of Registers into DSP blocks or block RAM components
    - Register duplication
    - Removal of constant or equivalent Flip-Flops

Flip-Flops and Registers Reporting Example

```
---------------------------------------------------------------------------
-------
RTL Component Statistics
---------------------------------------------------------------------------
-------
Detailed RTL Component Info :
+---Registers :
8 Bit Registers := 1

Report Cell Usage:
-----+----+-----
|Cell|Count
-----+----+-----
3 |FDCE| 8
-----+----+-----
```

Flip-Flops and Registers Coding Examples

The following subsections provide VHDL and Verilog examples of coding for flip-flops and registers. Download the coding example files from Coding Examples.

Register with Rising-Edge Coding Verilog Example

Filename: registers_1.v

```
// 8-bit Register with
// Rising-edge Clock
// Active-high Synchronous Clear
// Active-high Clock Enable
// File: registers_1.v

module registers_1(d_in,ce,clk,clr,dout);
input [7:0] d_in;
input ce;
input clk;
input clr;
output [7:0] dout;
reg [7:0] d_reg;
always @ (posedge clk)
begin
if(clr)
d_reg <= 8'b0;
else if(ce)
d_reg <= d_in;
end
assign dout = d_reg;
endmodule
```

Flip-Flop Registers with Rising-Edge Clock Coding VHDL Example

Filename: registers_1.vhd

```
-- Flip-Flop with
-- Rising-edge Clock
-- Active-high Synchronous Clear
-- Active-high Clock Enable
-- File: registers_1.vhd

library IEEE;
use IEEE.std_logic_1164.all;

entity registers_1 is
port(
clr, ce, clk : in std_logic;
d_in : in std_logic_vector(7 downto 0);
dout : out std_logic_vector(7 downto 0)
);
end entity registers_1;
```

```
architecture rtl of registers_1 is
begin
process(clk) is
begin
if rising_edge(clk) then
if clr = '1' then
dout <= "00000000";
elsif ce = '1' then
dout <= d_in;
end if;
end if;
end process;
end architecture rtl;
```

Latches

The Vivado log file reports the type and size of recognized Latches.
Inferred Latches are often the result of HDL coding mistakes, such as incomplete if or case statements.
Vivado synthesis issues a warning for the instance shown in the following reporting example. This warning lets you verify that the inferred Latch functionality was intended.

Latches Reporting Example

```
=========================================================================
* Vivado.log *
=========================================================================

WARNING: [Synth 8-327] inferring latch for variable 'Q_reg'

=========================================================================
Report Cell Usage:
-----+----+-----
|Cell|Count
-----+----+-----
2 |LD | 1
-----+----+-----
===================================================================
```

Latch With Positive Gate and Asynchronous Reset Coding Verilog Example

Filename: latches.v

```
// Latch with Positive Gate and Asynchronous Reset
// File: latches.v
module latches (
input G,
input D,
input CLR,
output reg Q
);
always @ *
begin
if(CLR)
Q = 0;
else if(G)
Q = D;
end

endmodule
```

Latch With Positive Gate and Asynchronous Reset Coding VHDL Example

Filename: latches.vhd

```
-- Latch with Positive Gate and Asynchronous Reset
-- File: latches.vhd
library ieee;
use ieee.std_logic_1164.all;

entity latches is
port(
G, D, CLR : in std_logic;
Q : out std_logic
);
end latches;
```

```
architecture archi of latches is
begin
process(CLR, D, G)
begin
if (CLR = '1') then
Q <= '0';
elsif (G = '1') then
Q <= D;
end if;
end process;
end archi;
```

Tristates

- Tristate buffers are usually modeled by a signal or an if-else construct.
- This applies whether the buffer drives an internal bus or an external bus on the board on which the device resides.
- The signal is assigned a high impedance value in one branch of the if-else. Download the coding example files from Coding Examples.

Tristate Implementation

Inferred Tristate buffers are implemented with different device primitives when driving the following:

- An external pin of the circuit (OBUFT)
- An Internal bus (BUFT):
  - An inferred BUFT is converted automatically to logic realized in LUTs by Vivado synthesis.
  - When an internal bus inferring a BUFT is driving an output of the top module, the Vivado synthesis infers an OBUF.

Tristate Reporting Example

Tristate buffers are inferred and reported during synthesis.

```
=========================================================================
* Vivado log file *
=========================================================================
Report Cell Usage:
```

```
-----+-----+-----
|Cell |Count
-----+-----+-----
1 |OBUFT| 1
-----+-----+-----
==========================================================================
```

Tristate Description Using Concurrent Assignment Coding Verilog Example

Filename: tristates_2.v

```
// Tristate Description Using Concurrent Assignment
// File: tristates_2.v
//
module tristates_2 (T, I, O);
input T, I;
output O;
assign O = (~T) ? I: 1'bZ;
endmodule
```

Tristate Description Using Combinatorial Process Implemented with OBUFT Coding VHDL Example

Filename: tristates_1.vhd

```
-- Tristate Description Using Combinatorial Process
-- Implemented with an OBUFT (IO buffer)
-- File: tristates_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;
entity tristates_1 is
port(
T : in std_logic;
I : in std_logic;
O : out std_logic
);
end tristates_1;
architecture archi of tristates_1 is
```

```
begin
process(I, T)
begin
if (T = '0') then
O <= I;
else
O <= 'Z';
end if;
end process;
end archi;
```

Tristate Description Using Combinatorial Always Block Coding Verilog Example

Filename: tristates_1.v

```
// Tristate Description Using Combinatorial Always Block
// File: tristates_1.v
//
module tristates_1 (T, I, O);
input T, I;
output O;
reg O;

always @(T or I)
begin
if (~T)
O = I;
else
O = 1'bZ;
end

endmodule
```

Shift Registers

A Shift Register is a chain of Flip-Flops allowing propagation of data across a fixed (static) number of latency stages. In contrast, in Dynamic Shift Registers, the length of the propagation chain varies dynamically during circuit operation. Download the coding example files from Coding Examples.

Static Shift Register Elements

A static Shift Register usually involves:

- A clock
- An optional clock enable
- A serial data input
- A serial data output

Shift Registers SRL-Based Implementation

Vivado synthesis implements inferred Shift Registers on SRL-type resources such as:

- `SRL16E`
- `SRLC32E`

Depending on the length of the Shift Register, Vivado synthesis does one of the following:

- Implements it on a single SRL-type primitive
- Takes advantage of the cascading capability of SRLC-type primitives
- Attempts to take advantage of this cascading capability if the rest of the design uses some intermediate positions of the Shift Register

Shift Registers Coding Examples

The following sections provide VHDL and Verilog coding examples for shift registers.

32-Bit Shift Register Coding Example One (VHDL)

This coding example uses the concatenation coding style.

Filename: shift_registers_0.vhd

```
-- 32-bit Shift Register
-- Rising edge clock
-- Active high clock enable
-- Concatenation-based template
-- File: shift_registers_0.vhd
```

```
library ieee;
use ieee.std_logic_1164.all;
entity shift_registers_0 is
generic(
DEPTH : integer := 32
);
port(
clk : in std_logic;
clken : in std_logic;
SI : in std_logic;
SO : out std_logic
);

end shift_registers_0;

architecture archi of shift_registers_0 is
signal shreg : std_logic_vector(DEPTH - 1 downto 0);
begin
process(clk)
begin
if rising_edge(clk) then
if clken = '1' then
shreg <= shreg(DEPTH - 2 downto 0) & SI;
end if;
end if;
end process;
SO <= shreg(DEPTH - 1);
end archi;
```

32-Bit Shift Register Coding Example Two (VHDL)

The same functionality can also be described as follows:

Filename: shift_registers_1.vhd

```
-- 32-bit Shift Register
-- Rising edge clock
-- Active high clock enable
-- foor loop-based template
-- File: shift_registers_1.vhd

library ieee;
use ieee.std_logic_1164.all;
entity shift_registers_1 is
generic(
DEPTH : integer := 32
);
port(
clk : in std_logic;
clken : in std_logic;
SI : in std_logic;
SO : out std_logic
);

end shift_registers_1;

architecture archi of shift_registers_1 is
signal shreg : std_logic_vector(DEPTH - 1 downto 0);
begin
process(clk)
begin
if rising_edge(clk) then
if clken = '1' then
for i in 0 to DEPTH - 2 loop
shreg(i + 1) <= shreg(i);
end loop;
shreg(0) <= SI;
end if;
end if;
end process;
SO <= shreg(DEPTH - 1);
end archi;
```

8-Bit Shift Register Coding Example One (Verilog)

This coding example uses a concatenation to describe the Register chain.

Filename: shift_registers_0.v

```
// 8-bit Shift Register
// Rising edge clock
// Active high clock enable
// Concatenation-based template
// File: shift_registers_0.v

module shift_registers_0 (clk, clken, SI, SO);
parameter WIDTH = 32;
input clk, clken, SI;
output SO;

reg [WIDTH-1:0] shreg;

always @(posedge clk)
begin
if (clken)
shreg = {shreg[WIDTH-2:0], SI};
end

assign SO = shreg[WIDTH-1];

endmodule
```

32-Bit Shift Register Coding Example Two (Verilog)

Filename: shift_registers_1.v

```
// 32-bit Shift Register
// Rising edge clock
// Active high clock enable
// For-loop based template
// File: shift_registers_1.v

module shift_registers_1 (clk, clken, SI, SO);
parameter WIDTH = 32;
input clk, clken, SI;
output SO;
```

```
reg [WIDTH-1:0] shreg;

integer i;
always @(posedge clk)
begin
if (clken)
begin
for (i = 0; i < WIDTH-1; i = i+1)
shreg[i+1] <= shreg[i];
shreg[0] <= SI;
end
end
assign SO = shreg[WIDTH-1];
endmodule
```

SRL Based Shift Registers Reporting

```
Report Cell Usage:
-----+-------+-----
|Cell |Count
-----+-------+-----
1 |SRLC32E| 1
```

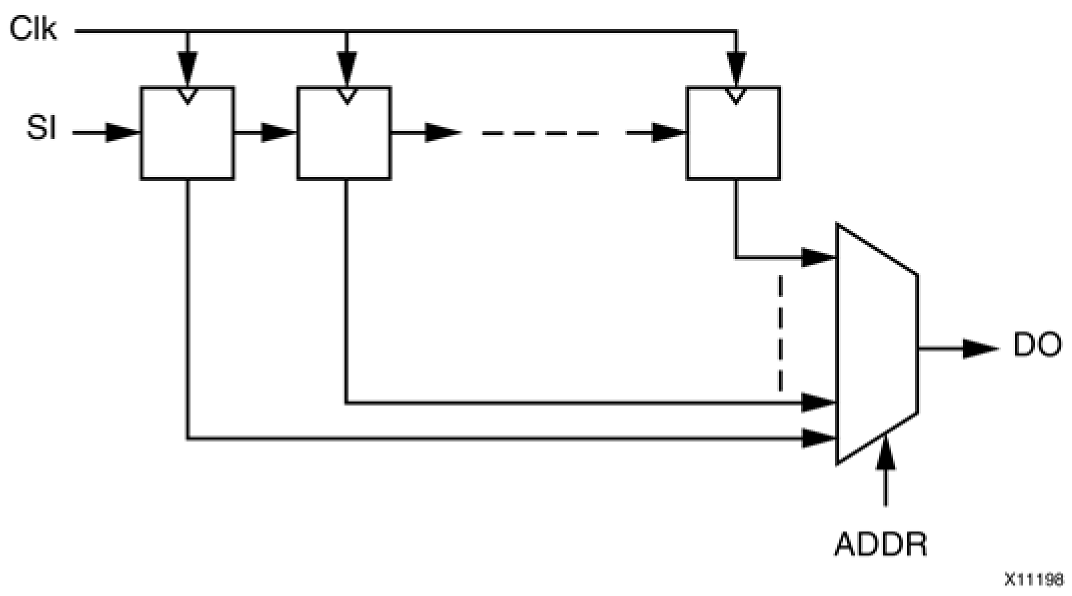
Dynamic Shift Registers

A Dynamic Shift register is a Shift register the length of which can vary dynamically during circuit operation.
A Dynamic Shift register can be seen as:

- A chain of Flip-Flops of the maximum length that it can accept during circuit operation.
- A Multiplexer that selects, in a given clock cycle, the stage at which data is to be extracted from the propagation chain.

The Vivado synthesis tool can infer Dynamic Shift registers of any maximal length.
Vivado synthesis tool can implement Dynamic Shift registers optimally using the SRL-type primitives available in the device family. The following figure illustrates the functionality of the Dynamic Shift register.

Figure: Dynamic Shift Registers Diagram



Dynamic Shift Registers Coding Examples

Download the coding example files from Coding Examples.

32-Bit Dynamic Shift Registers Coding Verilog Example

Filename: dynamic_shift_registers_1.v

```
// 32-bit dynamic shift register.
// Download:
// File: dynamic_shift_registers_1.v

module dynamic_shift_register_1 (CLK, CE, SEL, SI, DO);
parameter SELWIDTH = 5;
input CLK, CE, SI;
input [SELWIDTH-1:0] SEL;
output DO;

localparam DATAWIDTH = 2**SELWIDTH;
reg [DATAWIDTH-1:0] data;

assign DO = data[SEL];

always @(posedge CLK)
begin
```

```
if (CE == 1'b1)
data <= {data[DATAWIDTH-2:0], SI};
end
endmodule
```

32-Bit Dynamic Shift Registers Coding VHDL Example

Filename: dynamic_shift_registers_1.vd

```
-- 32-bit dynamic shift register.
-- File:dynamic_shift_registers_1.vhd
-- 32-bit dynamic shift register.
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity dynamic_shift_register_1 is
generic(
DEPTH : integer := 32;
SEL_WIDTH : integer := 5
);
port(
CLK : in std_logic;
SI : in std_logic;
CE : in std_logic;
A : in std_logic_vector(SEL_WIDTH - 1 downto 0);
DO : out std_logic
);

end dynamic_shift_register_1;

architecture rtl of dynamic_shift_register_1 is
type SRL_ARRAY is array (DEPTH - 1 downto 0) of std_logic;

signal SRL_SIG : SRL_ARRAY;

begin
process(CLK)
begin
if rising_edge(CLK) then
if CE = '1' then
SRL_SIG <= SRL_SIG(DEPTH - 2 downto 0) & SI;
end if;
end if;
end process;

DO <= SRL_SIG(conv_integer(A));

end rtl;
```

Multipliers

Vivado synthesis infers multiplier macros from multiplication operators in the source code. The resulting signal width equals the sum of the two operand sizes. For example, multiplying a 16-bit signal by an 8-bit signal produces a result of 24 bits.

Recommended: If you do not intend to use all most significant bits of a device, AMD recommends that you reduce the size of operands to the minimum needed, especially if the Multiplier macro is implemented on slice logic.

Multipliers Implementation

Multiplier macros can be implemented on:

- Slice logic
- DSP blocks

The implementation choice is:

- Driven by the size of operands
- Aimed at maximizing performance

To force implementation of a Multiplier to slice logic or DSP block, set the `USE_DSP` attribute on the appropriate signal, entity, or module to either:

- no (slice logic)
- yes (DSP block)

DSP Block Implementation

When implementing a Multiplier in a single DSP block, Vivado synthesis tries to take advantage of the pipelining capabilities of DSP blocks. Vivado synthesis pulls up to two levels of registers present: On the multiplication operands, and after the multiplication.
When a Multiplier does not fit on a single DSP block, Vivado synthesis decomposes the macro to implement it. In that case, Vivado synthesis uses either of the following:

- Several DSP blocks
- A hybrid solution involving both DSP blocks and slice logic

Use the `KEEP` attribute to restrict absorption of Registers into DSP blocks. For example, if a Register is present on an operand of the multiplier, place `KEEP` on the output of the Register to prevent the Register from being absorbed into the DSP block.

Multipliers Coding Examples

Unsigned 16x24-Bit Multiplier Coding Verilog Example

Filename: mult_unsigned.v

```
// Unsigned 16x24-bit Multiplier
// 1 latency stage on operands
// 3 latency stage after the multiplication
// File: multipliers2.v
//
module mult_unsigned (clk, A, B, RES);

parameter WIDTHA = 16;
parameter WIDTHB = 24;
input clk;
input [WIDTHA-1:0] A;
input [WIDTHB-1:0] B;
output [WIDTHA+WIDTHB-1:0] RES;

reg [WIDTHA-1:0] rA;
reg [WIDTHB-1:0] rB;
reg [WIDTHA+WIDTHB-1:0] M [3:0];

integer i;
always @(posedge clk)
begin
rA <= A;
rB <= B;
M[0] <= rA * rB;
for (i = 0; i < 3; i = i+1)
M[i+1] <= M[i];
end

assign RES = M[3];

endmodule
```

Unsigned 16x16-Bit Multiplier Coding VHDL Example

Filename: mult_unsigned.vhd

```
-- Unsigned 16x16-bit Multiplier
-- File: mult_unsigned.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity mult_unsigned is
generic(
WIDTHA : integer := 16;
WIDTHB : integer := 16
);
port(
A : in std_logic_vector(WIDTHA - 1 downto 0);
B : in std_logic_vector(WIDTHB - 1 downto 0);
RES : out std_logic_vector(WIDTHA + WIDTHB - 1 downto 0)
);
end mult_unsigned;

architecture beh of mult_unsigned is
```

```
begin
RES <= A * B;
end beh;
```

Multiply-Add and Multiply-Accumulate

The following macros are inferred:

- Multiply-Add
- Multiply-Sub
- Multiply-Add/Sub
- Multiply-Accumulate

The macros are inferred by aggregation of:

- A Multiplier
- An Adder/Subtractor
- Registers

Multiply-Add and Multiply-Accumulate Implementation

During Multiply-Add and Multiply-Accumulate implementation:

- Vivado synthesis can implement an inferred Multiply-Add or Multiply-Accumulate macro on DSP block resources.
- Vivado synthesis attempts to take advantage of the pipelining capabilities of DSP blocks.
- Vivado synthesis pulls up to:
    - Two register stages are present on the multiplication operands.
    - One register stage present after the multiplication.
    - One register stage found after the Adder, Subtractor, or Adder/Subtractor.
    - One register stage on the add/sub-selection signal.
    - One register stage on the Adder optional carry input.
- Vivado synthesis can implement a Multiply Accumulate in a DSP block if its implementation requires only a single DSP resource.
- If the macro exceeds the limits of a single DSP, Vivado synthesis does the following:
    - Processes it as two separate Multiplier and Accumulate macros.
    - Makes independent decisions on each macro.

Macro Implementation on DSP Block Resources

Macro implementation on DSP block resources is inferred by default in Vivado synthesis.

- In default mode, Vivado synthesis:
    - Implements Multiply-Add and Multiply-Accumulate macros.
    - Takes into account DSP block resources availability in the targeted device.
    - Uses all available DSP resources.
    - Attempts to maximize circuit performance by leveraging all the pipelining capabilities of DSP blocks.
    - Scans for opportunities to absorb registers into a Multiply-Add or Multiply-Accumulate macro.

Use the `KEEP` attribute to restrict absorption of Registers into DSP blocks. For example, to exclude a register present on an operand of the Multiplier from absorption into the DSP block, apply `KEEP` on the output of the register. For more information about the `KEEP` attribute, see KEEP.
Download the coding example files from Coding Examples.

Complex Multiplier Examples

The following examples show complex multiplier examples in VHDL and Verilog. The coding example files also include a complex multiplier with accumulation example that uses three DSP blocks for the AMD UltraScale™ architecture.

Complex Multiplier Verilog Example

Fully pipelined complex multiplier using three DSP blocks.

Filename: cmult.v

```
//
// Complex Multiplier (pr+i.pi) = (ar+i.ai)*(br+i.bi)
// file: cmult.v

module cmult # (parameter AWIDTH = 16, BWIDTH = 18)
```

```
(
input clk,
input signed [AWIDTH-1:0] ar, ai,
input signed [BWIDTH-1:0] br, bi,
output signed [AWIDTH+BWIDTH:0] pr, pi
);

reg signed [AWIDTH-1:0] ai_d, ai_dd, ai_ddd, ai_dddd ;
reg signed [AWIDTH-1:0] ar_d, ar_dd, ar_ddd, ar_dddd ;
reg signed [BWIDTH-1:0] bi_d, bi_dd, bi_ddd, br_d, br_dd, br_ddd ;
reg signed [AWIDTH:0] addcommon ;
reg signed [BWIDTH:0] addr, addi ;
reg signed [AWIDTH+BWIDTH:0] mult0, multr, multi, pr_int, pi_int ;
reg signed [AWIDTH+BWIDTH:0] common, commonr1, commonr2 ;

always @(posedge clk)
begin
ar_d <= ar;
ar_dd <= ar_d;
ai_d <= ai;
ai_dd <= ai_d;
br_d <= br;
br_dd <= br_d;
br_ddd <= br_dd;
bi_d <= bi;
bi_dd <= bi_d;
bi_ddd <= bi_dd;
end

// Common factor (ar ai) x bi, shared for the calculations of the real and
imaginary final products
//
always @(posedge clk)
begin
addcommon <= ar_d - ai_d;
mult0 <= addcommon * bi_dd;
common <= mult0;
end

// Real product
//
always @(posedge clk)
```

```
begin
ar_ddd <= ar_dd;
ar_dddd <= ar_ddd;
addr <= br_ddd - bi_ddd;
multr <= addr * ar_dddd;
commonr1 <= common;
pr_int <= multr + commonr1;
end

// Imaginary product
//
always @(posedge clk)
begin
ai_ddd <= ai_dd;
ai_dddd <= ai_ddd;
addi <= br_ddd + bi_ddd;
multi <= addi * ai_dddd;
commonr2 <= common;
pi_int <= multi + commonr2;
end

assign pr = pr_int;
assign pi = pi_int;

endmodule // cmult
```

Complex Multiplier Examples (VHDL)

Fully pipelined complex multiplier using three DSP blocks.

Filename: cmult.vhd

```
-- Complex Multiplier (pr+i.pi) = (ar+i.ai)*(br+i.bi)
--
--
-- cumult.vhd
--

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

```
entity cmult is
generic(AWIDTH : natural := 16;
BWIDTH : natural := 16);
port(clk : in std_logic;
ar, ai : in std_logic_vector(AWIDTH - 1 downto 0);
br, bi : in std_logic_vector(BWIDTH - 1 downto 0);
pr, pi : out std_logic_vector(AWIDTH + BWIDTH downto 0));
end cmult;

architecture rtl of cmult is
signal ai_d, ai_dd, ai_ddd, ai_dddd : signed(AWIDTH - 1 downto 0);
signal ar_d, ar_dd, ar_ddd, ar_dddd : signed(AWIDTH - 1 downto 0);
signal bi_d, bi_dd, bi_ddd, br_d, br_dd, br_ddd : signed(BWIDTH - 1 downto
0);
signal addcommon : signed(AWIDTH downto 0);
signal addr, addi : signed(BWIDTH downto 0);
signal mult0, multr, multi, pr_int, pi_int : signed(AWIDTH + BWIDTH downto
0);
signal common, commonr1, commonr2 : signed(AWIDTH + BWIDTH downto 0);

begin
process(clk)
begin
if rising_edge(clk) then
ar_d <= signed(ar);
ar_dd <= signed(ar_d);
ai_d <= signed(ai);
ai_dd <= signed(ai_d);
br_d <= signed(br);
br_dd <= signed(br_d);
br_ddd <= signed(br_dd);
bi_d <= signed(bi);
bi_dd <= signed(bi_d);
bi_ddd <= signed(bi_dd);
end if;
end process;

-- Common factor (ar - ai) x bi, shared for the calculations
-- of the real and imaginary final products.
--
process(clk)
```

```
begin
if rising_edge(clk) then
addcommon <= resize(ar_d, AWIDTH + 1) - resize(ai_d, AWIDTH + 1);
mult0 <= addcommon * bi_dd;
common <= mult0;
end if;
end process;

-- Real product
--
process(clk)
begin
if rising_edge(clk) then
ar_ddd <= ar_dd;
ar_dddd <= ar_ddd;
addr <= resize(br_ddd, BWIDTH + 1) - resize(bi_ddd, BWIDTH + 1);
multr <= addr * ar_dddd;
commonr1 <= common;
pr_int <= multr + commonr1;
end if;
end process;

-- Imaginary product
--
process(clk)
begin
if rising_edge(clk) then
ai_ddd <= ai_dd;
ai_dddd <= ai_ddd;
addi <= resize(br_ddd, BWIDTH + 1) + resize(bi_ddd, BWIDTH + 1);
multi <= addi * ai_dddd;
commonr2 <= common;
pi_int <= multi + commonr2;
end if;
end process;

--
-- VHDL type conversion for output
--
pr <= std_logic_vector(pr_int);
pi <= std_logic_vector(pi_int);
```

```
end rtl;
```

Pre-Adders in the DSP Block

When coding for inference and targeting the DSP block, it is recommended to use signed arithmetic and it is a requirement to have one extra bit of width for the pre-adder result so that it can be packed into the DSP block.

Pre-Adder Dynamically Configured Followed by Multiplier and Post-Adder (Verilog)

Filename: dynpreaddmultadd.v

```
// Pre-add/subtract select with Dynamic control
// dynpreaddmultadd.v
module dynpreaddmultadd # (parameter SIZEIN = 16)
(
input clk, ce, rst, subadd,
input signed [SIZEIN-1:0] a, b, c, d,
output signed [2*SIZEIN:0] dynpreaddmultadd_out
);

// Declare registers for intermediate values
reg signed [SIZEIN-1:0] a_reg, b_reg, c_reg;
reg signed [SIZEIN:0] add_reg;
reg signed [2*SIZEIN:0] d_reg, m_reg, p_reg;

always @(posedge clk)
begin
if (rst)
begin
a_reg <= 0;
b_reg <= 0;
c_reg <= 0;
d_reg <= 0;
add_reg <= 0;
m_reg <= 0;
p_reg <= 0;
end
```

```
else if (ce)
begin
a_reg <= a;
b_reg <= b;
c_reg <= c;
d_reg <= d;
if (subadd)
add_reg <= a_reg - b_reg;
else
add_reg <= a_reg + b_reg;
m_reg <= add_reg * c_reg;
p_reg <= m_reg + d_reg;
end
end

// Output accumulation result
assign dynpreaddmultadd_out = p_reg;

endmodule // dynpreaddmultadd
```

Pre-Adder Dynamically Configured Followed by Multiplier and Post-Adder (VHDL)

Filename: dynpreaddmultadd.vhd

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

entity dynpreaddmultadd is
generic(
AWIDTH : natural := 12;
BWIDTH : natural := 16;
CWIDTH : natural := 17
);
port(
clk : in std_logic;
subadd : in std_logic;
ain : in std_logic_vector(AWIDTH - 1 downto 0);
bin : in std_logic_vector(BWIDTH - 1 downto 0);
cin : in std_logic_vector(CWIDTH - 1 downto 0);
din : in std_logic_vector(BWIDTH + CWIDTH downto 0);
pout : out std_logic_vector(BWIDTH + CWIDTH downto 0)
);
end dynpreaddmultadd;

architecture rtl of dynpreaddmultadd is
signal a : signed(AWIDTH - 1 downto 0);
signal b : signed(BWIDTH - 1 downto 0);
signal c : signed(CWIDTH - 1 downto 0);
signal add : signed(BWIDTH downto 0);
signal d, mult, p : signed(BWIDTH + CWIDTH downto 0);

begin
process(clk)
begin
if rising_edge(clk) then
a <= signed(ain);
b <= signed(bin);
c <= signed(cin);
d <= signed(din);
if subadd = '1' then
add <= resize(a, BWIDTH + 1) - resize(b, BWIDTH + 1);
else
add <= resize(a, BWIDTH + 1) + resize(b, BWIDTH + 1);
end if;
mult <= add * c;
p <= mult + d;
end if;
end process;

--
-- Type conversion for output
--
pout <= std_logic_vector(p);

end rtl;
```

Using the Squarer in the UltraScale DSP

Block

The UltraScale DSP block (DSP48E2) primitive can compute the square of an input or the output of the pre-adder.
Download the coding example files from Coding Examples .
The following are examples of the square of a difference; this can be used to efficiently replace calculations on absolute values of differences.
It fits into a single DSP block and runs at full speed. The coding example files mentioned previously also include an accumulator of the square of differences which also fits into a single DSP block for the UltraScale architecture.

Square of a Difference (Verilog)

Filename: squarediffmult.v

```
// Squarer support for DSP block (DSP48E2) with
// pre-adder configured
// as subtractor
// File: squarediffmult.v

module squarediffmult # (parameter SIZEIN = 16)
(
input clk, ce, rst,
input signed [SIZEIN-1:0] a, b,
output signed [2*SIZEIN+1:0] square_out
);

// Declare registers for intermediate values
reg signed [SIZEIN-1:0] a_reg, b_reg;
reg signed [SIZEIN:0] diff_reg;
reg signed [2*SIZEIN+1:0] m_reg, p_reg;

always @(posedge clk)
begin
if (rst)
begin
a_reg <= 0;
b_reg <= 0;
diff_reg <= 0;
m_reg <= 0;
```

```
p_reg <= 0;
end
else
if (ce)
begin
a_reg <= a;
b_reg <= b;
diff_reg <= a_reg - b_reg;
m_reg <= diff_reg * diff_reg;
p_reg <= m_reg;
end
end

// Output result
assign square_out = p_reg;

endmodule // squarediffmult
```

Square of a Difference (VHDL)

Filename: squarediffmult.vhd

```
-- Squarer support for DSP block (DSP48E2) with pre-adder
-- configured
-- as subtractor
-- File: squarediffmult.vhd

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

entity squarediffmult is
generic(
SIZEIN : natural := 16
);
port(
clk, ce, rst : in std_logic;
ain, bin : in std_logic_vector(SIZEIN - 1 downto 0);
square_out : out std_logic_vector(2 * SIZEIN + 1 downto 0)
);
end squarediffmult;
```

```
architecture rtl of squarediffmult is

-- Declare intermediate values
signal a_reg, b_reg : signed(SIZEIN - 1 downto 0);
signal diff_reg : signed(SIZEIN downto 0);
signal m_reg, p_reg : signed(2 * SIZEIN + 1 downto 0);

begin
process(clk)
begin
if rising_edge(clk) then
if rst = '1' then
a_reg <= (others => '0');
b_reg <= (others => '0');
diff_reg <= (others => '0');
m_reg <= (others => '0');
p_reg <= (others => '0');
else
a_reg <= signed(ain);
b_reg <= signed(bin);
diff_reg <= resize(a_reg, SIZEIN + 1) - resize(b_reg, SIZEIN + 1);
m_reg <= diff_reg * diff_reg;
p_reg <= m_reg;
end if;
end if;
end process;

--
-- Type conversion for output
--
square_out <= std_logic_vector(p_reg);

end rtl;
```

FIR Filters

Vivado synthesis infers cascades of multiply-add to compose FIR filters directly from RTL.

There are several possible implementations of such filters; one example is the systolic filter described in the *7 Series DSP48E1 Slice User Guide* (UG479) and shown in the 8-Tap Even Symmetric Systolic FIR (Verilog).
Download the coding example files from Coding Examples .

8-Tap Even Symmetric Systolic FIR (Verilog)

Filename: sfir_even_symetric_systolic_top.v

```
// sfir_even_symmetric_systolic_top.v
// FIR Symmetric Systolic Filter, Top module is
sfir_even_symmetric_systolic_top

// sfir_shifter - sub module which is used in top level
(* dont_touch = "yes" *)
module sfir_shifter #(parameter dsize = 16, nbtap = 4)
(input clk, [dsize-1:0] datain, output [dsize-1:0] dataout);

(* srl_style = "srl_register" *) reg [dsize-1:0] tmp [0:2*nbtap-1];
integer i;

always @(posedge clk)
begin
tmp[0] <= datain;
for (i=0; i<=2*nbtap-2; i=i+1)
tmp[i+1] <= tmp[i];
end

assign dataout = tmp[2*nbtap-1];

endmodule

// sfir_even_symmetric_systolic_element - sub module which is used in top
module sfir_even_symmetric_systolic_element #(parameter dsize = 16)
(input clk, input signed [dsize-1:0] coeffin, datain, datazin, input
signed [2*dsize-1:0] cascin,
output signed [dsize-1:0] cascdata, output reg signed [2*dsize-1:0]
cascout);

reg signed [dsize-1:0] coeff;
reg signed [dsize-1:0] data;
reg signed [dsize-1:0] dataz;
reg signed [dsize-1:0] datatwo;
reg signed [dsize:0] preadd;
reg signed [2*dsize-1:0] product;

assign cascdata = datatwo;

always @(posedge clk)
begin
coeff <= coeffin;
data <= datain;
datatwo <= data;
dataz <= datazin;
preadd <= datatwo + dataz;
product <= preadd * coeff;
cascout <= product + cascin;
end

endmodule


module sfir_even_symmetric_systolic_top #(parameter nbtap = 4, dsize = 16,
psize = 2*dsize)
(input clk, input signed [dsize-1:0] datain, output signed [2*dsize-1:0]
firout);

wire signed [dsize-1:0] h [nbtap-1:0];
wire signed [dsize-1:0] arraydata [nbtap-1:0];
wire signed [psize-1:0] arrayprod [nbtap-1:0];

wire signed [dsize-1:0] shifterout;
reg signed [dsize-1:0] dataz [nbtap-1:0];

assign h[0] = 7;
assign h[1] = 14;
assign h[2] = -138;
assign h[3] = 129;

assign firout = arrayprod[nbtap-1]; // Connect last product to output

sfir_shifter #(dsize, nbtap) shifter_inst0 (clk, datain, shifterout);
```

```
generate
genvar I;
for (I=0; I<nbtap; I=I+1)
if (I==0)
sfir_even_symmetric_systolic_element #(dsize) fte_inst0 (clk, h[I],
datain, shifterout, {32{1'b0}}, arraydata[I], arrayprod[I]);
else
sfir_even_symmetric_systolic_element #(dsize) fte_inst (clk, h[I],
arraydata[I-1], shifterout, arrayprod[I-1], arraydata[I], arrayprod[I]);
endgenerate

endmodule // sfir_even_symmetric_systolic_top
```

8-Tap Even Symmetric Systolic FIR (VHDL)

Filename: sfir_even_symetric_systolic_top.vhd

```
--
-- FIR filter top
-- File: sfir_even_symmetric_systolic_top.vhd


-- FIR filter shifter
-- submodule used in top (sfir_even_symmetric_systolic_top)
library ieee;
use ieee.std_logic_1164.all;

entity sfir_shifter is
generic(
DSIZE : natural := 16;
NBTAP : natural := 4
);
port(
clk : in std_logic;
datain : in std_logic_vector(DSIZE - 1 downto 0);
dataout : out std_logic_vector(DSIZE - 1 downto 0)
);
end sfir_shifter;

architecture rtl of sfir_shifter is
```

```
-- Declare signals
--
type CHAIN is array (0 to 2 * NBTAP - 1) of std_logic_vector(DSIZE - 1
downto 0);
signal tmp : CHAIN;

begin
process(clk)
begin
if rising_edge(clk) then
tmp(0) <= datain;
looptmp : for i in 0 to 2 * NBTAP - 2 loop
tmp(i + 1) <= tmp(i);
end loop;
end if;
end process;

dataout <= tmp(2 * NBTAP - 1);

end rtl;

--
-- FIR filter engine (multiply with pre-add and post-add)
-- submodule used in top (sfir_even_symmetric_systolic_top)
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity sfir_even_symmetric_systolic_element is
generic(DSIZE : natural := 16);
port(clk : in std_logic;
coeffin, datain, datazin : in std_logic_vector(DSIZE - 1 downto 0);
cascin : in std_logic_vector(2 * DSIZE downto 0);
cascdata : out std_logic_vector(DSIZE - 1 downto 0);
cascout : out std_logic_vector(2 * DSIZE downto 0));
end sfir_even_symmetric_systolic_element;

architecture rtl of sfir_even_symmetric_systolic_element is

-- Declare signals
--
signal coeff, data, dataz, datatwo : signed(DSIZE - 1 downto 0);
signal preadd : signed(DSIZE downto 0);
signal product, cascouttmp : signed(2 * DSIZE downto 0);

begin
process(clk)
begin
if rising_edge(clk) then
coeff <= signed(coeffin);
data <= signed(datain);
datatwo <= data;
dataz <= signed(datazin);
preadd <= resize(datatwo, DSIZE + 1) + resize(dataz, DSIZE + 1);
product <= preadd * coeff;
cascouttmp <= product + signed(cascin);
end if;
end process;

-- Type conversion for output
--
cascout <= std_logic_vector(cascouttmp);
cascdata <= std_logic_vector(datatwo);

end rtl;

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

entity sfir_even_symmetric_systolic_top is
generic(NBTAP : natural := 4;
DSIZE : natural := 16;
PSIZE : natural := 33);
port(clk : in std_logic;
datain : in std_logic_vector(DSIZE - 1 downto 0);
firout : out std_logic_vector(PSIZE - 1 downto 0));
end sfir_even_symmetric_systolic_top;

architecture rtl of sfir_even_symmetric_systolic_top is

-- Declare signals
--
type DTAB is array (0 to NBTAP - 1) of std_logic_vector(DSIZE - 1 downto
0);
type HTAB is array (0 to NBTAP - 1) of std_logic_vector(0 to DSIZE - 1);
type PTAB is array (0 to NBTAP - 1) of std_logic_vector(PSIZE - 1 downto
0);

signal arraydata, dataz : DTAB;
signal arrayprod : PTAB;
signal shifterout : std_logic_vector(DSIZE - 1 downto 0);

-- Initialize coefficients and a "zero" for the first filter element
--
constant h : HTAB := ((std_logic_vector(TO_SIGNED(63, DSIZE))),
(std_logic_vector(TO_SIGNED(18, DSIZE))),
(std_logic_vector(TO_SIGNED(-100, DSIZE))),
(std_logic_vector(TO_SIGNED(1, DSIZE))));
constant zero_psize : std_logic_vector(PSIZE - 1 downto 0) := (others =>
'0');

begin

-- Connect last product to output
--
firout <= arrayprod(nbtap - 1);

-- Shifter
--
shift_u0 : entity work.sfir_shifter
generic map(DSIZE, NBTAP)
port map(clk, datain, shifterout);

-- Connect the arithmetic building blocks of the FIR
--
gen : for I in 0 to NBTAP - 1 generate
begin
g0 : if I = 0 generate
element_u0 : entity work.sfir_even_symmetric_systolic_element
generic map(DSIZE)
port map(clk, h(I), datain, shifterout, zero_psize, arraydata(I),
arrayprod(I));
end generate g0;
gi : if I /= 0 generate
element_ui : entity work.sfir_even_symmetric_systolic_element
```

```
generic map(DSIZE)
port map(clk, h(I), arraydata(I - 1), shifterout, arrayprod(I - 1),
arraydata(I), arrayprod(I));
end generate gi;
end generate gen;

end rtl;
```

Convergent Rounding (LSB Correction Technique)

The DSP block primitive leverages a pattern detect circuitry to compute convergent rounding (either to even, or to odd).
The following are examples of the convergent rounding inference, which infers at the block full performance, and also infers a 2-input AND gate (1 LUT) to implement the LSB correction.

Rounding to Even (Verilog)

Filename: convergentRoundingEven.v

```
// Convergent rounding(Even) Example which makes use of pattern detect
// File: convergentRoundingEven.v
module convergentRoundingEven (
input clk,
input [23:0] a,
input [15:0] b,
output reg signed [23:0] zlast
);

reg signed [23:0] areg;
reg signed [15:0] breg;
reg signed [39:0] z1;

reg pattern_detect;
wire [15:0] pattern = 16'b0000000000000000;
wire [39:0] c = 40'b0000000000000000000000000111111111111111; // 15 ones

wire signed [39:0] multadd;
wire signed [15:0] zero;
reg signed [39:0] multadd_reg;

// Convergent Rounding: LSB Correction Technique
// ---------------------------------------------
// For static convergent rounding, the pattern detector can be used
// to detect the midpoint case. For example, in an 8-bit round, if
// the decimal place is set at 4, the C input should be set to
// 0000.0111. Round to even rounding should use CARRYIN = "1" and
// check for PATTERN "XXXX.0000" and replace the units place with 0
// if the pattern is matched. See UG193 for more details.

assign multadd = z1 + c + 1'b1;

always @(posedge clk)
begin
areg <= a;
breg <= b;
z1 <= areg * breg;
pattern_detect <= multadd[15:0] == pattern ? 1'b1 : 1'b0;
multadd_reg <= multadd;
end

// Unit bit replaced with 0 if pattern is detected
always @(posedge clk)
zlast <= pattern_detect ? {multadd_reg[39:17],1'b0} : multadd_reg[39:16];

endmodule // convergentRoundingEven
```

Rounding to Even (VHDL)

Filename: convergentRoundingEven.vhd

```
-- Convergent rounding(Even) Example which makes use of pattern detect
-- File: convergentRoundingEven.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity convergentRoundingEven is
port (clk : in std_logic;
a : in std_logic_vector (23 downto 0);
b : in std_logic_vector (15 downto 0);
zlast : out std_logic_vector (23 downto 0));
end convergentRoundingEven;

architecture beh of convergentRoundingEven is

signal ar : signed(a'range);
signal br : signed(b'range);
signal z1 : signed(a'length + b'length - 1 downto 0);

signal multaddr : signed(a'length + b'length - 1 downto 0);
signal multadd : signed(a'length + b'length - 1 downto 0);
signal pattern_detect : boolean;

constant pattern : signed(15 downto 0) := (others => '0');
constant c : signed := "0000000000000000000000000111111111111111";

-- Convergent Rounding: LSB Correction Technique
-- ---------------------------------------------
-- For static convergent rounding, the pattern detector can be used
-- to detect the midpoint case. For example, in an 8-bit round, if
-- the decimal place is set at 4, the C input should be set to
-- 0000.0111. Round to even rounding should use CARRYIN = "1" and
-- check for PATTERN "XXXX.0000" and replace the units place with 0
-- if the pattern is matched. See UG193 for more details.

begin

multadd <= z1 + c + 1;

process(clk)
begin
if rising_edge(clk) then
ar <= signed(a);
br <= signed(b);
z1 <= ar * br;
multaddr <= multadd;
if multadd(15 downto 0) = pattern then
pattern_detect <= true;
else
pattern_detect <= false;
```

```
end if;
end if;
end process;

-- Unit bit replaced with 0 if pattern is detected
process(clk)
begin
if rising_edge(clk) then
if pattern_detect = true then
zlast <= std_logic_vector(multaddr(39 downto 17)) & "0";
else
zlast <= std_logic_vector(multaddr(39 downto 16));
end if;
end if;
end process;

end beh;
```

Rounding to Odd (Verilog)

Filename: convergentRoundingOdd.v

```
// Convergent rounding(Odd) Example which makes use of pattern detect
// File: convergentRoundingOdd.v
module convergentRoundingOdd (
input clk,
input [23:0] a,
input [15:0] b,
output reg signed [23:0] zlast
);

reg signed [23:0] areg;
reg signed [15:0] breg;
reg signed [39:0] z1;

reg pattern_detect;
wire [15:0] pattern = 16'b1111111111111111;
wire [39:0] c = 40'b0000000000000000000000000111111111111111; // 15 ones

wire signed [39:0] multadd;
wire signed [15:0] zero;
```

```
reg signed [39:0] multadd_reg;

// Convergent Rounding: LSB Correction Technique
// ---------------------------------------------
// For static convergent rounding, the pattern detector can be
// used to detect the midpoint case. For example, in an 8-bit
// round, if the decimal place is set at 4, the C input should
// be set to 0000.0111. Round to odd rounding should use
// CARRYIN = "0" and check for PATTERN "XXXX.1111" and then
// replace the units place bit with 1 if the pattern is
// matched. See UG193 for details

assign multadd = z1 + c;

always @(posedge clk)
begin
areg <= a;
breg <= b;
z1 <= areg * breg;
pattern_detect <= multadd[15:0] == pattern ? 1'b1 : 1'b0;
multadd_reg <= multadd;
end

always @(posedge clk)
zlast <= pattern_detect ? {multadd_reg[39:17],1'b1} : multadd_reg[39:16];

endmodule // convergentRoundingOdd
```

Rounding to Odd (VHDL)

Filename: convergentRoundingOdd.vhd

```
-- Convergent rounding(Odd) Example which makes use of pattern detect
-- File: convergentRoundingOdd.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity convergentRoundingOdd is
port (clk : in std_logic;
a : in std_logic_vector (23 downto 0);
b : in std_logic_vector (15 downto 0);
zlast : out std_logic_vector (23 downto 0));
end convergentRoundingOdd;

architecture beh of convergentRoundingOdd is

signal ar : signed(a'range);
signal br : signed(b'range);
signal z1 : signed(a'length + b'length - 1 downto 0);

signal multadd, multaddr : signed(a'length + b'length - 1 downto 0);
signal pattern_detect : boolean;

constant pattern : signed(15 downto 0) := (others => '1');
constant c : signed := "000000000000000000000000111111111111111";

-- Convergent Rounding: LSB Correction Technique
-- ------------------------------------------------
-- For static convergent rounding, the pattern detector can be
-- used to detect the midpoint case. For example, in an 8-bit
-- round, if the decimal place is set at 4, the C input should
-- be set to 0000.0111. Round to odd rounding should use
-- CARRYIN = "0" and check for PATTERN "XXXX.1111" and then
-- replace the units place bit with 1 if the pattern is
-- matched. See UG193 for details

begin

multadd <= z1 + c;

process(clk)
begin
if rising_edge(clk) then
ar <= signed(a);
br <= signed(b);
z1 <= ar * br;
multaddr <= multadd;
if multadd(15 downto 0) = pattern then
pattern_detect <= true;
else
pattern_detect <= false;
end if;
end if;
end process;

process(clk)
begin
if rising_edge(clk) then
if pattern_detect = true then
zlast <= std_logic_vector(multaddr(39 downto 17)) & "1";
else
zlast <= std_logic_vector(multaddr(39 downto 16));
end if;
end if;
end process;

end beh;
```

RAM HDL Coding Techniques

Vivado synthesis can interpret various RAM coding styles, and maps them into distributed RAMs or block RAMs. This action does the following:

- Makes it unnecessary to manually instantiate RAM primitives
- Saves time
- Keeps HDL source code portable and scalable

Download the coding example files from Coding Examples.

Choosing Between Distributed RAM and Dedicated Block RAM

Data is written synchronously into the RAM for both types. The primary difference between distributed RAM and dedicated block RAM lies in the way data is read from the RAM. See the following table.

Table: Distributed RAM versus Dedicated Block RAM

Action	Distributed RAM	Dedicated Block RAM
Write	Synchronous	Synchronous
Read	Asynchronous	Synchronous

Whether to use distributed RAM or dedicated block RAM can depend upon the characteristics of the RAM described in the HDL source code, the availability of block RAM resources, and whether you have forced a specific implementation style using RAM_STYLE attribute.

Memory Inference Capabilities

Memory inference capabilities include the following:

- Support for any size and data width. Vivado synthesis maps the memory description to one or several RAM primitives
- Single-port, simple-dual port, true dual port
- Up to two write ports
- Multiple read ports

Provided that only one write port is described, Vivado synthesis can identify RAM descriptions with two or more read ports that access the RAM contents at addresses different from the write address.

- Write enable
- RAM enable (block RAM)
- Data output reset (block RAM)
- Optional output register (block RAM)
- Byte write enable (block RAM)
- Each RAM port can be controlled by its distinct clock, port enable, write enable, and data output reset
- Initial contents specification
- Vivado synthesis can use parity bits as regular data bits to accommodate the described data widths

---

Note: For more information on parity bits see the user guide for the device you are targeting.

UltraRAM Coding Templates

UltraRAM is described in "Chapter 2, UltraRAM Resources" of the *UltraScale Architecture Memory Resources User Guide* (UG573) as follows:
UltraRAM is a single-clocked, two port, synchronous memory available in AMD UltraScale+™ devices. Because UltraRAM is compatible with the columnar architecture, multiple UltraRAMs can be instantiated and directly cascaded in an UltraRAM column for the entire height of the device. A column in a single clock

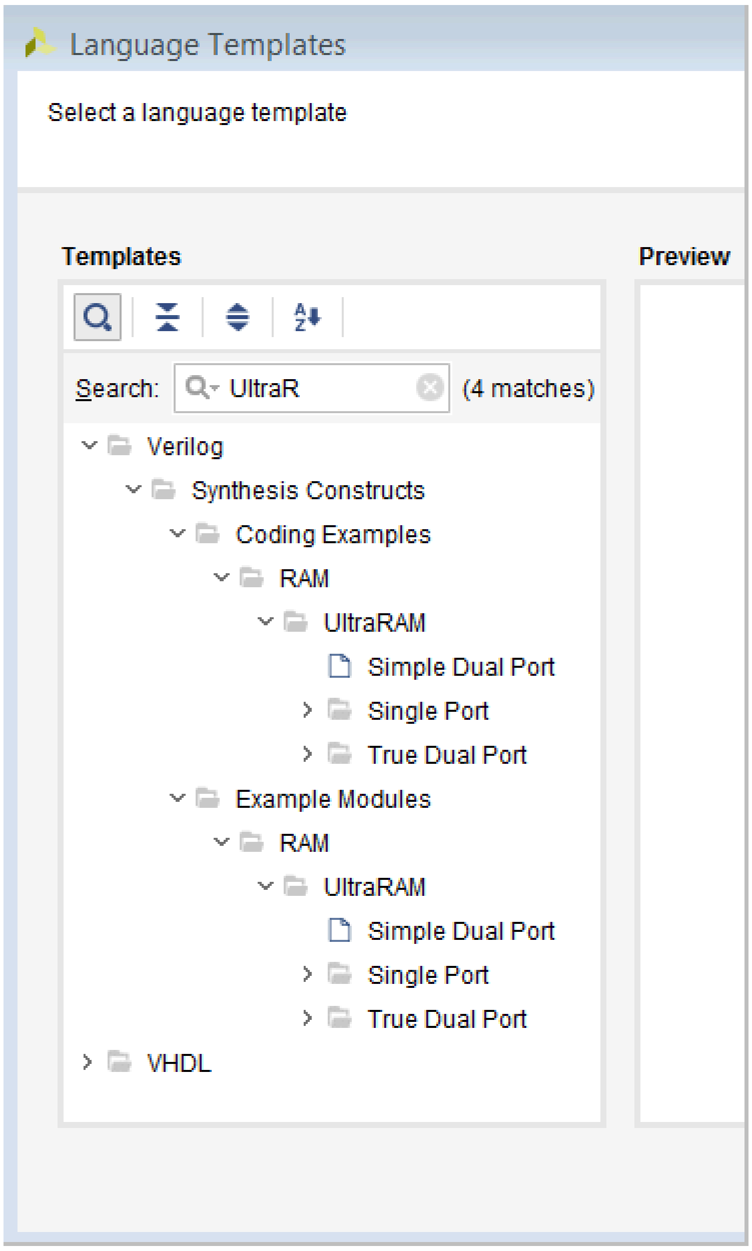
region contains 16 UltraRAM blocks. Devices with UltraRAM include multiple UltraRAM columns distributed in the device. Most of the devices in the UltraScale+ family include UltraRAM blocks. For the available quantity of UltraRAM in specific device families, see the *UltraScale Architecture and Product Data Sheet: Overview* (DS890).

The following files are included in the Coding Examples:

- `xilinx_ultraram_single_port_no_change.v`
- `xilinx_ultraram_single_port_no_change.vhd`
- `xilinx_ultraram_single_port_read_first.v`
- `xilinx_ultraram_single_port_read_first.vhd`
- `xilinx_ultraram_single_port_write_first.v`
- `xilinx_ultraram_single_port_write_first.vhd`

The Vivado tool includes templates of UltraRAM VHDL and Verilog code. The following figure shows the template files.

Figure: UltraRAM Coding Templates



- Verilog
  - Synthesis Constructs
    - Coding Examples
      - RAM
        - UltraRAM
          - Simple Dual Port
          - Single Port
          - True Dual Port
    - Example Modules
      - RAM
        - UltraRAM
          - Simple Dual Port
          - Single Port
          - True Dual Port
- VHDL

See the *UltraScale Architecture Memory Resources User Guide* (UG573) for more information.

Inferring UltraRAM in Vivado Synthesis

Overview of the UltraRAM Primitive

UltraRAM is a new dedicated memory primitive available in the UltraScale+ devices from AMD. This is a large memory that is designed to be cascaded for very large RAM blocks. For more information, see the *UltraScale Architecture Memory Resources User Guide* (UG573).

Description of the UltraRAM Primitive

The UltraRAM primitive is a dual port memory with a single clock. A single primitive is configured as 4 K x 72. The UltraRAM has two ports, which can access all 4 K of the RAM. This allows for a single port, simple dual port, and true dual-port behavior. There are also multiple pipeline registers for each port of the primitive. The UltraRAM has one clock, global enable, an output register reset, a write enable, and byte write enable support for control signals.

Differences between UltraRAM and Block RAM

There are a few notable differences between UltraRAM and block RAM to consider, as follows:

- The UltraRAM only has one clock, so while true dual port operation is supported, both ports are synchronous to each other.
- The aspect ratio of the UltraRAM is not configurable like block RAM, it is always configured as 4 K x 72.
- The resets on the output registers can only be reset to 0.
- The write modes (`read_first`, `write_first`, `no_change`) do not exist in this primitive. The regular UltraRAM behaves like `no_change`; however, if you describe `read_first` or `write_first` in RTL, the Vivado synthesis creates the correct logic.
- Finally, the `INIT` for RAM does not exist, the UltraRAM powers up in a 0 condition.

Using UltraRAM Inference

There are three ways of getting UltraRAM primitives, as follows:

Direct instantiation
Provides you the most control but is the hardest to perform.

XPM flow
Allows you to specify the type of RAM you want along with the behavior, but gives no access to the RTL.

Inference RAM
Is in the middle of the two, relatively easy, and gives more control to the user on how the RAM is created.

Attributes for Controlling UltraRAM

There are two attributes needed to control UltraRAM in Vivado synthesis: `RAM_STYLE` and `CASCADE_HEIGHT`.

RAM_STYLE

The RAM_STYLE attribute has a value called ultra. By default, Vivado synthesis uses a heuristic to determine what type of RAM to infer, URAM, block RAM or LUTRAM. If you want to force the RAM into an UltraRAM, you can use the RAM_STYLE attribute to tell Vivado synthesis to infer the URAM primitives. More information is available in RAM_STYLE.

RAM_STYLE Verilog Example

```
(* ram_style = "ultra" *) reg [data_size-1:0] myram [2**addr_size-1:0];
```

RAM_STYLE VHDL Example

```
attribute ram_style : string;
attribute ram_style of myram : signal is "ultra";
```

CASCADE_HEIGHT

When cascading multiple UltraRAMs (URAMs) together to create a larger RAM, Vivado synthesis limits the height of the chain to eight to provide flexibility to the place and route tool. To change this limit, you can use the `CASCADE_HEIGHT` attribute to change the default behavior.

Note: This option is only applicable to UltraScale architecture block RAMs and URAMs.

CASCADE_HEIGHT Verilog Example

```
(* cascade_height = 16 *) reg [data_size-1:0] myram [2**addr_size-1:0];
```

CASCADE_HEIGHT VHDL Example

```
attribute cascade_height : integer;
attribute cascade_height of my_ram signal is 16;
```

In addition to the attributes that only affect the specific RAMs on which they are put, there is also a global setting which affects all RAMs in the design.
The Synthesis Settings menu has the `-max_uram_cascade_height` setting. The default value is -1 which means that the Vivado synthesis tool determines the best course of action, but this can be overridden by other values. In case of a conflict between the global setting and a `CASCADE_HEIGHT` attribute, the attribute is used for that specific RAM.

Inference Capabilities

The Vivado Synthesis tool can do many types of memories using the UltraRAM primitives. For examples, see the Coding Guidelines.

- In single port memory, the same port that reads the memory also writes to it. All three of the write modes for the block RAM are supported, but it should be noted that the UltraRAM itself acts like a `NO_CHANGE` memory. If `WRITE_FIRST` or `READ_FIRST` behavior is described in the RTL, the UltraRAM created is set in simple dual-port mode.
- In a simple dual port memory, one port reads from the RAM while another writes to it. Vivado synthesis can infer these memories into UltraRAM.

  ★ **Tip:**One stipulation is that both ports must have the same clock.
- In True Dual Port mode, both ports can read from and write to the memory. In this mode, only the `NO_CHANGE` mode is supported.

⚠ **CAUTION!** Care should also be taken when simulating the true dual port RAM. In the previous versions of block RAM, there was address collision that was taken care of by the simulation models; with the UltraRAM, it is different. In the UltraRAM, port A always happens before port B. If Port A has a write and Port B is a read from that address, the memory is written to and read from, but if Port A has the read and Port B has the write, the old value is seen during the read.

⚠ **CAUTION!** Be sure to never read and write to the same address during the same clock cycle on a true dual-port memory because the RTL and post-synthesis simulations could be different.

For both the simple dual-port memory and the true dual-port memory, the clocks have to be the same for both ports.
In addition to the different styles of RAMs, there are also a few other features of the UltraRAM that can be inferred. The RAM has a global enable signal that

precedes the write enable. It has the standard write enable, and byte write enable support. The data output also has a reset like the previous block RAM; however, in this case, there is no `SRVAL` that can be set. Only resets of 0 are supported.

Pipelining the RAM

The UltraRAM (URAM) supports pipelining registers into the RAM. This becomes especially useful when multiple UltraRAMs are used to create a very large RAM. To fully pipeline the RAM, you must add extra registers to the RAM output in RTL. To calculate the number of pipeline registers to use, add together the number of rows and columns in the RAM matrix.

Note: The tool does not create the pipeline registers for you; they must be in the RTL code for Vivado synthesis to make use of them.

The synthesis log file has a section about URAMs and how many rows and columns are used to create the RAM matrix. You can use this section to add pipeline registers in the RTL.
To calculate the number of rows and columns of the matrix yourself, remember that the UltraRAM is configured as a 4 K x 72.
To calculate the number of rows, take your address space of the RAM in RTL and divide by 4 K. If this number is higher than the number specified by `CASCADE_HEIGHT`, remove the extra RAMs, and start them on a new column in the log.

Creating Pipeline Example 1: 8K x 72

In this example, 8 K divided by 4 K is two, so there are 2 rows. If the `CASCADE_HEIGHT` is set higher than 2, it is a 2 x 1 matrix. There should be three pipeline stages added to the output of the RAM (2 + 1).

Creating Pipeline Example 2 : 8K x 80

In this example, 8 K divided by 4 K is two, so there are two rows. The data space does not matter for this calculation, so the matrix would be two rows and 1 column resulting in three pipeline registers again.

Note: The whole matrix is reproduced to get the extra 8 bits of data space needed to create the RAM, but that does not matter to the calculation of pipeline registers.

Creating Pipeline Example 3: 16K x 70 CASCADE_HEIGHT Set to 3

In this example, 16 K divided by 4 K is four; however, because the `CASCADE_HEIGHT` is 3, this would be a 3 x 2 matrix. This would result in 5 pipeline registers that can be used.

RAM HDL Coding Guidelines

Download the coding example files from Coding Examples .

Block RAM Read/Write Synchronization Modes

You can configure block RAM resources to provide the following synchronization modes for a given read/write port:

Read-first
: Old content is read before new content is loaded.

Write-first
: New content is immediately made available for reading Write-first is also known as read-through.

No-change
: Data output does not change as new content is loaded into RAM.

Vivado synthesis provides inference support for all of these synchronization modes. You can describe a different synchronization mode for each port of the RAM.

Distributed RAM Examples

The following sections provide VHDL and Verilog coding examples for distributed RAM.

Dual-Port RAM with Asynchronous Read Coding Verilog Example

Filename: rams_dist.v

```
// Dual-Port RAM with Asynchronous Read (Distributed RAM)
// File: rams_dist.v
```

```
module rams_dist (clk, we, a, dpra, di, spo, dpo);

input clk;
input we;
input [5:0] a;
input [5:0] dpra;
input [15:0] di;
output [15:0] spo;
output [15:0] dpo;
reg [15:0] ram [63:0];

always @(posedge clk)
begin
if (we)
ram[a] <= di;
end

assign spo = ram[a];
assign dpo = ram[dpra];

endmodule
```

Single-Port RAM with Asynchronous Read Coding Example (VHDL)

Filename: rams_dist.vhd

```
-- Single-Port RAM with Asynchronous Read (Distributed RAM)
-- File: rams_dist.vhd

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

entity rams_dist is
port(
clk : in std_logic;
we : in std_logic;
a : in std_logic_vector(5 downto 0);
di : in std_logic_vector(15 downto 0);
do : out std_logic_vector(15 downto 0)
);
end rams_dist;

architecture syn of rams_dist is
type ram_type is array (63 downto 0) of std_logic_vector(15 downto 0);
signal RAM : ram_type;
begin
process(clk)
begin
if (clk'event and clk = '1') then
if (we = '1') then
RAM(to_integer(unsigned(a))) <= di;
end if;
end if;
end process;

do <= RAM(to_integer(unsigned(a)));

end syn;
```

Single-Port Block RAMs

The following sections provide VHDL and Verilog coding examples for Single-Port Block RAM.

Single-Port Block RAM with Resettable Data Output (Verilog)

Filename: rams_sp_rf_rst.v

```
// Block RAM with Resettable Data Output
// File: rams_sp_rf_rst.v

module rams_sp_rf_rst (clk, en, we, rst, addr, di, dout);
input clk;
input en;
input we;
input rst;
input [9:0] addr;
input [15:0] di;
output [15:0] dout;
```

```
reg [15:0] ram [1023:0];
reg [15:0] dout;

always @(posedge clk)
begin
if (en) //optional enable
begin
if (we) //write enable
ram[addr] <= di;
if (rst) //optional reset
dout <= 0;
else
dout <= ram[addr];
end
end

endmodule
```

Single Port Block RAM with Resettable Data Output (VHDL)

Filename: rams_sp_rf_rst.vhd

```
-- Block RAM with Resettable Data Output
-- File: rams_sp_rf_rst.vhd

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

entity rams_sp_rf_rst is
port(
clk : in std_logic;
en : in std_logic;
we : in std_logic;
rst : in std_logic;
addr : in std_logic_vector(9 downto 0);
di : in std_logic_vector(15 downto 0);
do : out std_logic_vector(15 downto 0)
);
end rams_sp_rf_rst;
```

```
architecture syn of rams_sp_rf_rst is
type ram_type is array (1023 downto 0) of std_logic_vector(15 downto 0);
signal ram : ram_type;
begin
process(clk)
begin
if clk'event and clk = '1' then
if en = '1' then -- optional enable
if we = '1' then -- write enable
ram(to_integer(unsigned(addr))) <= di;
end if;
if rst = '1' then -- optional reset
do <= (others => '0');
else
do <= ram(to_integer(unsigned(addr)));
end if;
end if;
end if;
end process;

end syn;
```

Single-Port Block RAM Write-First Mode (Verilog)

Filename: rams_sp_wf.v

```
// Single-Port Block RAM Write-First Mode (recommended template)
// File: rams_sp_wf.v
module rams_sp_wf (clk, we, en, addr, di, dout);
input clk;
input we;
input en;
input [9:0] addr;
input [15:0] di;
output [15:0] dout;
reg [15:0] RAM [1023:0];
reg [15:0] dout;

always @(posedge clk)
```

```
begin
if (en)
begin
if (we)
begin
RAM[addr] <= di;
dout <= di;
end
else
dout <= RAM[addr];
end
end
endmodule
```

Single-Port Block RAM Write-First Mode (VHDL)

Filename: rams_sp_wf.vhd

```
-- Single-Port Block RAM Write-First Mode (recommended template)
--
-- File: rams_sp_wf.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity rams_sp_wf is
port(
clk : in std_logic;
we : in std_logic;
en : in std_logic;
addr : in std_logic_vector(9 downto 0);
di : in std_logic_vector(15 downto 0);
do : out std_logic_vector(15 downto 0)
);
end rams_sp_wf;

architecture syn of rams_sp_wf is
type ram_type is array (1023 downto 0) of std_logic_vector(15 downto 0);
signal RAM : ram_type;
```

```
begin
process(clk)
begin
if clk'event and clk = '1' then
if en = '1' then
if we = '1' then
RAM(to_integer(unsigned(addr))) <= di;
do <= di;
else
do <= RAM(to_integer(unsigned(addr)));
end if;
end if;
end if;
end process;

end syn;
```

Single-Port RAM with Read First (VHDL)

Filename: rams_sp_rf.vhd

```
-- Single-Port Block RAM Read-First Mode
-- rams_sp_rf.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity rams_sp_rf is
port(
clk : in std_logic;
we : in std_logic;
en : in std_logic;
addr : in std_logic_vector(9 downto 0);
di : in std_logic_vector(15 downto 0);
do : out std_logic_vector(15 downto 0)
);
end rams_sp_rf;

architecture syn of rams_sp_rf is
```

```
type ram_type is array (1023 downto 0) of std_logic_vector(15 downto 0);
signal RAM : ram_type;
begin
process(clk)
begin
if clk'event and clk = '1' then
if en = '1' then
if we = '1' then
RAM(to_integer(unsigned(addr))) <= di;
end if;
do <= RAM(to_integer(unsigned(addr)));
end if;
end if;
end process;

end syn;
```

Single-Port Block RAM No-Change Mode (Verilog)

Filename: rams_sp_nc.v

```
// Single-Port Block RAM No-Change Mode
// File: rams_sp_nc.v

module rams_sp_nc (clk, we, en, addr, di, dout);

input clk;
input we;
input en;
input [9:0] addr;
input [15:0] di;
output [15:0] dout;

reg [15:0] RAM [1023:0];
reg [15:0] dout;

always @(posedge clk)
begin
if (en)
begin
```

```
if (we)
RAM[addr] <= di;
else
dout <= RAM[addr];
end
end
endmodule
```

Single-Port Block RAM No-Change Mode (VHDL)

Filename: rams_sp_nc.vhd

```
-- Single-Port Block RAM No-Change Mode
-- File: rams_sp_nc.vhd
--

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

entity rams_sp_nc is
port(
clk : in std_logic;
we : in std_logic;
en : in std_logic;
addr : in std_logic_vector(9 downto 0);
di : in std_logic_vector(15 downto 0);
do : out std_logic_vector(15 downto 0)
);
end rams_sp_nc;

architecture syn of rams_sp_nc is
type ram_type is array (1023 downto 0) of std_logic_vector(15 downto 0);
signal RAM : ram_type;
begin
process(clk)
begin
if clk'event and clk = '1' then
if en = '1' then
if we = '1' then
```

```
RAM(to_integer(unsigned(addr))) <= di;
else
do <= RAM(to_integer(unsigned(addr)));
end if;
end if;
end if;
end process;

end syn;
```

Simple Dual-Port Block RAM Examples

The following sections provide VHDL and Verilog coding examples for Simple Dual-Port Block RAM.

Simple Dual-Port Block RAM with Single Clock (Verilog)

Filename: simple_dual_one_clock.v

```
// Simple Dual-Port Block RAM with One Clock
// File: simple_dual_one_clock.v

module simple_dual_one_clock (clk,ena,enb,wea,addra,addrb,dia,dob);

input clk,ena,enb,wea;
input [9:0] addra,addrb;
input [15:0] dia;
output [15:0] dob;
reg [15:0] ram [1023:0];
reg [15:0] doa,dob;

always @(posedge clk) begin
if (ena) begin
if (wea)
ram[addra] <= dia;
end
end

always @(posedge clk) begin
if (enb)
```

```
dob <= ram[addrb];
end

endmodule
```

Simple Dual-Port Block RAM with Single Clock (VHDL)

Filename: simple_dual_one_clock.vhd

```
-- Simple Dual-Port Block RAM with One Clock
-- Correct Modelization with a Shared Variable
-- File:simple_dual_one_clock.vhd

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity simple_dual_one_clock is
port(
clk : in std_logic;
ena : in std_logic;
enb : in std_logic;
wea : in std_logic;
addra : in std_logic_vector(9 downto 0);
addrb : in std_logic_vector(9 downto 0);
dia : in std_logic_vector(15 downto 0);
dob : out std_logic_vector(15 downto 0)
);
end simple_dual_one_clock;

architecture syn of simple_dual_one_clock is
type ram_type is array (1023 downto 0) of std_logic_vector(15 downto 0);
shared variable RAM : ram_type;
begin
process(clk)
begin
if clk'event and clk = '1' then
if ena = '1' then
if wea = '1' then
RAM(conv_integer(addra)) := dia;
```

```
end if;
end if;
end if;
end process;

process(clk)
begin
if clk'event and clk = '1' then
if enb = '1' then
dob <= RAM(conv_integer(addrb));
end if;
end if;
end process;

end syn;
```

Simple Dual-Port Block RAM with Dual Clocks (Verilog)

Filename: simple_dual_two_clocks.v

```
// Simple Dual-Port Block RAM with Two Clocks
// File: simple_dual_two_clocks.v

module simple_dual_two_clocks (clka,clkb,ena,enb,wea,addra,addrb,dia,dob);

input clka,clkb,ena,enb,wea;
input [9:0] addra,addrb;
input [15:0] dia;
output [15:0] dob;
reg [15:0] ram [1023:0];
reg [15:0] dob;

always @(posedge clka)
begin
if (ena)
begin
if (wea)
ram[addra] <= dia;
end
end
```

```
always @(posedge clkb)
begin
if (enb)
begin
dob <= ram[addrb];
end
end

endmodule
```

Simple Dual-Port Block RAM with Dual Clocks (VHDL)

Filename: simple_dual_two_clocks.vhd

```
-- Simple Dual-Port Block RAM with Two Clocks
-- Correct Modelization with a Shared Variable
-- File: simple_dual_two_clocks.vhd
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity simple_dual_two_clocks is
port(
clka : in std_logic;
clkb : in std_logic;
ena : in std_logic;
enb : in std_logic;
wea : in std_logic;
addra : in std_logic_vector(9 downto 0);
addrb : in std_logic_vector(9 downto 0);
dia : in std_logic_vector(15 downto 0);
dob : out std_logic_vector(15 downto 0)
);
end simple_dual_two_clocks;

architecture syn of simple_dual_two_clocks is
type ram_type is array (1023 downto 0) of std_logic_vector(15 downto 0);
shared variable RAM : ram_type;
begin
```

```
process(clka)
begin
if clka'event and clka = '1' then
if ena = '1' then
if wea = '1' then
RAM(conv_integer(addra)) := dia;
end if;
end if;
end if;
end process;

process(clkb)
begin
if clkb'event and clkb = '1' then
if enb = '1' then
dob <= RAM(conv_integer(addrb));
end if;
end if;
end process;

end syn;
```

True Dual-Port Block RAM Examples

The following sections provide VHDL and Verilog coding examples for True Dual-Port Block RAM.

Dual-Port Block RAM with Two Write Ports in Read First Mode Verilog Example

Filename: ram_tdp_rf_rf.v

```
// Dual-Port Block RAM with Two Write Ports
// File: rams_tdp_rf_rf.v

module rams_tdp_rf_rf
(clka,clkb,ena,enb,wea,web,addra,addrb,dia,dib,doa,dob);

input clka,clkb,ena,enb,wea,web;
input [9:0] addra,addrb;
input [15:0] dia,dib;
output [15:0] doa,dob;
```

```
reg [15:0] ram [1023:0];
reg [15:0] doa,dob;

always @(posedge clka)
begin
if (ena)
begin
if (wea)
ram[addra] <= dia;
doa <= ram[addra];
end
end

always @(posedge clkb)
begin
if (enb)
begin
if (web)
ram[addrb] <= dib;
dob <= ram[addrb];
end
end

endmodule
```

Dual-Port Block RAM with Two Write Ports in Read-First Mode (VHDL)

Filename: ram_tdp_rf_rf.vhd

```
-- Dual-Port Block RAM with Two Write Ports
-- Correct Modelization with a Shared Variable
-- File: rams_tdp_rf_rf.vhd

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

entity rams_tdp_rf_rf is
port(
clka : in std_logic;
clkb : in std_logic;
ena : in std_logic;
enb : in std_logic;
wea : in std_logic;
web : in std_logic;
addra : in std_logic_vector(9 downto 0);
addrb : in std_logic_vector(9 downto 0);
dia : in std_logic_vector(15 downto 0);
dib : in std_logic_vector(15 downto 0);
doa : out std_logic_vector(15 downto 0);
dob : out std_logic_vector(15 downto 0)
);
end rams_tdp_rf_rf;

architecture syn of rams_tdp_rf_rf is
type ram_type is array (1023 downto 0) of std_logic_vector(15 downto 0);
shared variable RAM : ram_type;
begin
process(CLKA)
begin
if CLKA’event and CLKA = ‘1’ then
if ENA = ‘1’ then
DOA <= RAM(to_integer(unsigned(ADDRA)));
if WEA = ‘1’ then
RAM(to_integer(unsigned(ADDRA))) := DIA;
end if;
end if;
end if;
end process;

process(CLKB)
begin
if CLKB’event and CLKB = ‘1’ then
if ENB = ‘1’ then
DOB <= RAM(to_integer(unsigned(ADDRB)));
if WEB = ‘1’ then
RAM(to_integer(unsigned(ADDRB))) := DIB;
end if;
end if;
end if;
end process;
end syn;
```

Block RAM with Optional Output Registers (Verilog)

Filename: rams_pipeline.v

```
// Block RAM with Optional Output Registers
// File: rams_pipeline

module rams_pipeline (clk1, clk2, we, en1, en2, addr1, addr2, di, res1,
res2);
input clk1;
input clk2;
input we, en1, en2;
input [9:0] addr1;
input [9:0] addr2;
input [15:0] di;
output [15:0] res1;
output [15:0] res2;
reg [15:0] res1;
reg [15:0] res2;
reg [15:0] RAM [1023:0];
reg [15:0] do1;
reg [15:0] do2;

always @(posedge clk1)
begin
if (we == 1'b1)
RAM[addr1] <= di;
do1 <= RAM[addr1];
end

always @(posedge clk2)
begin
do2 <= RAM[addr2];
end

always @(posedge clk1)
begin
```

```
if (en1 == 1'b1)
res1 <= do1;
end

always @(posedge clk2)
begin
if (en2 == 1'b1)
res2 <= do2;
end
endmodule
```

Block RAM with Optional Output Registers (VHDL)

Filename: rams_pipeline.vhd

```
-- Block RAM with Optional Output Registers
-- File: rams_pipeline.vhd
library IEEE;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity rams_pipeline is
port(
clk1, clk2 : in std_logic;
we, en1, en2 : in std_logic;
addr1 : in std_logic_vector(9 downto 0);
addr2 : in std_logic_vector(9 downto 0);
di : in std_logic_vector(15 downto 0);
res1 : out std_logic_vector(15 downto 0);
res2 : out std_logic_vector(15 downto 0)
);
end rams_pipeline;

architecture beh of rams_pipeline is
type ram_type is array (1023 downto 0) of std_logic_vector(15 downto 0);
signal ram : ram_type;
signal do1 : std_logic_vector(15 downto 0);
signal do2 : std_logic_vector(15 downto 0);
begin
```

```
process(clk1)
begin
if rising_edge(clk1) then
if we = '1' then
ram(to_integer(unsigned(addr1))) <= di;
end if;
do1 <= ram(to_integer(unsigned(addr1)));
end if;
end process;

process(clk2)
begin
if rising_edge(clk2) then
do2 <= ram(to_integer(unsigned(addr2)));
end if;
end process;

process(clk1)
begin
if rising_edge(clk1) then
if en1 = '1' then
res1 <= do1;
end if;
end if;
end process;

process(clk2)
begin
if rising_edge(clk2) then
if en2 = '1' then
res2 <= do2;
end if;
end if;
end process;

end beh;
```

Byte Write Enable (Block RAM)

AMD supports byte write enable in block RAM. Use byte write enable in block RAM to:

- Exercise advanced control over writing data into RAM
- Separately specify the writeable portions of 8 bits of an addressed memory

From the standpoint of HDL modeling and inference, the concept is best described as a column-based write:

- The RAM is seen as a collection of equal size columns
- During a write cycle, you separately control writing into each of these columns

Vivado synthesis inference lets you take advantage of the block RAM byte write enable feature. The described RAM is implemented on block RAM resources, using the byte write enable capability, provided that the following requirements are met:

- Write columns of equal widths
- Allowed write column widths: 8-bit, 9-bit, 16-bit, 18-bit (multiple of 8-bit or 9-bit)

For other write column widths, such as 5-bit or 12-bit (non multiple of 8-bit or 9-bit), Vivado synthesis uses separate RAMs for each column:

- Number of write columns: any
- Supported read-write synchronizations: read-first, write-first, no-change

Byte Write Enable—True Dual Port with Byte-Wide Write Enable (Verilog)

Filename: bytewrite_tdp_ram_rf.v

```
// True-Dual-Port BRAM with Byte-wide Write Enable
// Read-First mode
// bytewrite_tdp_ram_rf.v
//

module bytewrite_tdp_ram_rf
#(
//--------------------------------------------------------------------------
--
parameter NUM_COL = 4,
parameter COL_WIDTH = 8,
parameter ADDR_WIDTH = 10,
// Addr Width in bits : 2 *ADDR_WIDTH = RAM Depth
parameter DATA_WIDTH = NUM_COL*COL_WIDTH // Data Width in bits
//----------------------------------------------------------------------
) (
input clkA,
input enaA,
input [NUM_COL-1:0] weA,
input [ADDR_WIDTH-1:0] addrA,
input [DATA_WIDTH-1:0] dinA,
output reg [DATA_WIDTH-1:0] doutA,

input clkB,
input enaB,
input [NUM_COL-1:0] weB,
input [ADDR_WIDTH-1:0] addrB,
input [DATA_WIDTH-1:0] dinB,
output reg [DATA_WIDTH-1:0] doutB
);


// Core Memory
reg [DATA_WIDTH-1:0] ram_block [(2**ADDR_WIDTH)-1:0];

integer i;
// Port-A Operation
always @ (posedge clkA) begin
if(enaA) begin
for(i=0;i<NUM_COL;i=i+1) begin
if(weA[i]) begin
ram_block[addrA][i*COL_WIDTH +: COL_WIDTH] <= dinA[i*COL_WIDTH +:
COL_WIDTH];
end
end
doutA <= ram_block[addrA];
end
end

// Port-B Operation:
always @ (posedge clkB) begin
if(enaB) begin
for(i=0;i<NUM_COL;i=i+1) begin
if(weB[i]) begin
```

```
ram_block[addrB][i*COL_WIDTH +: COL_WIDTH] <= dinB[i*COL_WIDTH +:
COL_WIDTH];
end
end

doutB <= ram_block[addrB];
end
end

endmodule // bytewrite_tdp_ram_rf
```

Byte Write Enable—True Dual Port READ_FIRST Mode (VHDL)

Filename: bytewrite_tdp_ram_rf.vhd

```
-- True-Dual-Port BRAM with Byte-wide Write Enable
-- Read First mode
--
-- bytewrite_tdp_ram_rf.vhd
--
-- READ_FIRST ByteWide WriteEnable Block RAM Template

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity bytewrite_tdp_ram_rf is
generic(
SIZE : integer := 1024;
ADDR_WIDTH : integer := 10;
COL_WIDTH : integer := 9;
NB_COL : integer := 4
);

port(
clka : in std_logic;
ena : in std_logic;
wea : in std_logic_vector(NB_COL - 1 downto 0);
addra : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
dia : in std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
doa : out std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
clkb : in std_logic;
enb : in std_logic;
web : in std_logic_vector(NB_COL - 1 downto 0);
addrb : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
dib : in std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
dob : out std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0)
);

end bytewrite_tdp_ram_rf;

architecture byte_wr_ram_rf of bytewrite_tdp_ram_rf is
type ram_type is array (0 to SIZE - 1) of std_logic_vector(NB_COL *
COL_WIDTH - 1 downto 0);
shared variable RAM : ram_type := (others => (others => '0'));

begin

------- Port A -------
process(clka)
begin
if rising_edge(clka) then
if ena = '1' then
doa <= RAM(conv_integer(addra));
for i in 0 to NB_COL - 1 loop
if wea(i) = '1' then
RAM(conv_integer(addra))((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH) :=
dia((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH);
end if;
end loop;
end if;
end if;
end process;

------- Port B -------
process(clkb)
begin
if rising_edge(clkb) then
if enb = '1' then
dob <= RAM(conv_integer(addrb));
for i in 0 to NB_COL - 1 loop
```

```
if web(i) = '1' then
RAM(conv_integer(addrb))((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH) :=
dib((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH);
end if;
end loop;
end if;
end if;
end process;
end byte_wr_ram_rf;
```

Byte Write Enable—WRITE_FIRST Mode (VHDL)

Filename: bytewrite_tdp_ram_wf.vhd

```
-- True-Dual-Port BRAM with Byte-wide Write Enable
-- Write First mode
--
-- bytewrite_tdp_ram_wf.vhd
-- WRITE_FIRST ByteWide WriteEnable Block RAM Template

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity bytewrite_tdp_ram_wf is
generic(
SIZE : integer := 1024;
ADDR_WIDTH : integer := 10;
COL_WIDTH : integer := 9;
NB_COL : integer := 4
);

port(
clka : in std_logic;
ena : in std_logic;
wea : in std_logic_vector(NB_COL - 1 downto 0);
addra : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
dia : in std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
doa : out std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
clkb : in std_logic;
enb : in std_logic;
web : in std_logic_vector(NB_COL - 1 downto 0);
addrb : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
dib : in std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
dob : out std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0)
);

end bytewrite_tdp_ram_wf;

architecture byte_wr_ram_wf of bytewrite_tdp_ram_wf is
type ram_type is array (0 to SIZE - 1) of std_logic_vector(NB_COL *
COL_WIDTH - 1 downto 0);
shared variable RAM : ram_type := (others => (others => '0'));

begin

------- Port A -------
process(clka)
begin
if rising_edge(clka) then
if ena = '1' then
for i in 0 to NB_COL - 1 loop
if wea(i) = '1' then
RAM(conv_integer(addra))((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH) :=
dia((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH);
end if;
end loop;
doa <= RAM(conv_integer(addra));
end if;
end if;

end process;

------- Port B -------
process(clkb)
begin
if rising_edge(clkb) then
if enb = '1' then
for i in 0 to NB_COL - 1 loop
if web(i) = '1' then
RAM(conv_integer(addrb))((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH) :=
dib((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH);
```

```
end if;
end loop;
dob <= RAM(conv_integer(addrb));
end if;
end if;
end process;
end byte_wr_ram_wf;
```

Byte-Wide Write Enable—NO_CHANGE Mode (Verilog)

Filename: bytewrite_tdp_ram_nc.v

```
//
// True-Dual-Port BRAM with Byte-wide Write Enable
// No-Change mode
//
// bytewrite_tdp_ram_nc.v
//
// ByteWide Write Enable, - NO_CHANGE mode template - Vivado recomended
module bytewrite_tdp_ram_nc
#(
//---------------------------------------------------------------
parameter NUM_COL = 4,
parameter COL_WIDTH = 8,
parameter ADDR_WIDTH = 10, // Addr Width in bits : 2**ADDR_WIDTH = RAM
Depth
parameter DATA_WIDTH = NUM_COL*COL_WIDTH // Data Width in bits
//---------------------------------------------------------------
) (
input clkA,
input enaA,
input [NUM_COL-1:0] weA,
input [ADDR_WIDTH-1:0] addrA,
input [DATA_WIDTH-1:0] dinA,
output reg [DATA_WIDTH-1:0] doutA,

input clkB,
input enaB,
input [NUM_COL-1:0] weB,
input [ADDR_WIDTH-1:0] addrB,
input [DATA_WIDTH-1:0] dinB,
output reg [DATA_WIDTH-1:0] doutB
);


// Core Memory
reg [DATA_WIDTH-1:0] ram_block [(2**ADDR_WIDTH)-1:0];

// Port-A Operation
generate
genvar i;
for(i=0;i<NUM_COL;i=i+1) begin
always @ (posedge clkA) begin
if(enaA) begin
if(weA[i]) begin
ram_block[addrA][i*COL_WIDTH +: COL_WIDTH] <= dinA[i*COL_WIDTH +:
COL_WIDTH];
end
end
end
end
endgenerate

always @ (posedge clkA) begin
if(enaA) begin
if (~|weA)
doutA <= ram_block[addrA];
end
end


// Port-B Operation:
generate
for(i=0;i<NUM_COL;i=i+1) begin
always @ (posedge clkB) begin
if(enaB) begin
if(weB[i]) begin
ram_block[addrB][i*COL_WIDTH +: COL_WIDTH] <= dinB[i*COL_WIDTH +:
COL_WIDTH];
end
end
end
```

```
end
endgenerate

always @ (posedge clkB) begin
if(enaB) begin
if (~|weB)
doutB <= ram_block[addrB];
end
end

endmodule // bytewrite_tdp_ram_nc
```

Byte-Wide Write Enable—NO_CHANGE Mode (VHDL)

Filename: bytewrite_tdp_ram_nc.vhd

```
--
-- True-Dual-Port BRAM with Byte-wide Write Enable
-- No change mode
--
-- bytewrite_tdp_ram_nc.vhd
--
-- NO_CHANGE ByteWide WriteEnable Block RAM Template

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity bytewrite_tdp_ram_nc is
generic(
SIZE : integer := 1024;
ADDR_WIDTH : integer := 10;
COL_WIDTH : integer := 9;
NB_COL : integer := 4
);

port(
clka : in std_logic;
ena : in std_logic;
wea : in std_logic_vector(NB_COL - 1 downto 0);
addra : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
dia : in std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
doa : out std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
clkb : in std_logic;
enb : in std_logic;
web : in std_logic_vector(NB_COL - 1 downto 0);
addrb : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
dib : in std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0);
dob : out std_logic_vector(NB_COL * COL_WIDTH - 1 downto 0)
);

end bytewrite_tdp_ram_nc;

architecture byte_wr_ram_nc of bytewrite_tdp_ram_nc is
type ram_type is array (0 to SIZE - 1) of std_logic_vector(NB_COL *
COL_WIDTH - 1 downto 0);
shared variable RAM : ram_type := (others => (others => '0'));

begin

------- Port A -------
process(clka)
begin
if rising_edge(clka) then
if ena = '1' then
if (wea = (wea'range => '0')) then
doa <= RAM(conv_integer(addra));
end if;
for i in 0 to NB_COL - 1 loop
if wea(i) = '1' then
RAM(conv_integer(addra))((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH) :=
dia((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH);
end if;
end loop;
end if;
end if;
end process;

------- Port B -------
process(clkb)
begin
if rising_edge(clkb) then
if enb = '1' then
if (web = (web'range => '0')) then
dob <= RAM(conv_integer(addrb));
end if;
for i in 0 to NB_COL - 1 loop
if web(i) = '1' then
RAM(conv_integer(addrb))((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH) :=
dib((i + 1) * COL_WIDTH - 1 downto i * COL_WIDTH);
end if;
end loop;
end if;
end if;
end process;
end byte_wr_ram_nc;
```

Asymmetric RAMs

The following sections provide VHDL and Verilog coding examples for asymmetric RAMs.

Note: Asymmetric RAMs with byte-write enables are not supported with RTL inference. Please use the XPM flow if this is needed.

Simple Dual-Port Asymmetric RAM When Read is Wider than Write (VHDL)

Filename: asym_ram_sdp_read_wider.vhd

```
-- Asymmetric port RAM
-- Read Wider than Write
-- asym_ram_sdp_read_wider.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asym_ram_sdp_read_wider is
generic(
WIDTHA : integer := 4;
SIZEA : integer := 1024;
ADDRWIDTHA : integer := 10;
WIDTHB : integer := 16;
SIZEB : integer := 256;
ADDRWIDTHB : integer := 8
);

port(
clkA : in std_logic;
clkB : in std_logic;
enA : in std_logic;
enB : in std_logic;
weA : in std_logic;
addrA : in std_logic_vector(ADDRWIDTHA - 1 downto 0);
addrB : in std_logic_vector(ADDRWIDTHB - 1 downto 0);
diA : in std_logic_vector(WIDTHA - 1 downto 0);
doB : out std_logic_vector(WIDTHB - 1 downto 0)
);

end asym_ram_sdp_read_wider;

architecture behavioral of asym_ram_sdp_read_wider is
function max(L, R : INTEGER) return INTEGER is
begin
if L > R then
return L;
else
return R;
end if;
end;

function min(L, R : INTEGER) return INTEGER is
begin
if L < R then
return L;
else
return R;
end if;
end;

function log2(val : INTEGER) return natural is
variable res : natural;
begin
for i in 0 to 31 loop
```

```
if (val <= (2 ** i)) then
res := i;
exit;
end if;
end loop;
return res;
end function Log2;

constant minWIDTH : integer := min(WIDTHA, WIDTHB);
constant maxWIDTH : integer := max(WIDTHA, WIDTHB);
constant maxSIZE : integer := max(SIZEA, SIZEB);
constant RATIO : integer := maxWIDTH / minWIDTH;

-- An asymmetric RAM is modeled in a similar way as a symmetric RAM, with
an
-- array of array object. Its aspect ratio corresponds to the port with
the
-- lower data width (larger depth)
type ramType is array (0 to maxSIZE - 1) of std_logic_vector(minWIDTH - 1
downto 0);

signal my_ram : ramType := (others => (others => '0'));

signal readB : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');
signal regA : std_logic_vector(WIDTHA - 1 downto 0) := (others => '0');
signal regB : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');

begin

-- Write process
process(clkA)
begin
if rising_edge(clkA) then
if enA = '1' then
if weA = '1' then
my_ram(conv_integer(addrA)) <= diA;
end if;
end if;
end if;
end process;

-- Read process
```

```
process(clkB)
begin
if rising_edge(clkB) then
for i in 0 to RATIO - 1 loop
if enB = '1' then
readB((i + 1) * minWIDTH - 1 downto i * minWIDTH) <=
my_ram(conv_integer(addrB & conv_std_logic_vector(i, log2(RATIO))));
end if;
end loop;
regB <= readB;
end if;
end process;

doB <= regB;

end behavioral;
```

Dual-Port Asymmetric RAM When Read is Wider than Write (Verilog)

Filename: asym_ram_sdp_read_wider.v

```
// Asymmetric port RAM
// Read Wider than Write. Read Statement in loop
//asym_ram_sdp_read_wider.v

module asym_ram_sdp_read_wider (clkA, clkB, enaA, weA, enaB, addrA, addrB,
diA, doB);
parameter WIDTHA = 4;
parameter SIZEA = 1024;
parameter ADDRWIDTHA = 10;

parameter WIDTHB = 16;
parameter SIZEB = 256;
parameter ADDRWIDTHB = 8;
input clkA;
input clkB;
input weA;
input enaA, enaB;
input [ADDRWIDTHA-1:0] addrA;
input [ADDRWIDTHB-1:0] addrB;
input [WIDTHA-1:0] diA;
output [WIDTHB-1:0] doB;
`define max(a,b) {(a) > (b) ? (a) : (b)}
`define min(a,b) {(a) < (b) ? (a) : (b)}

function integer log2;
input integer value;
reg [31:0] shifted;
integer res;
begin
if (value < 2)
log2 = value;
else
begin
shifted = value-1;
for (res=0; shifted>0; res=res+1)
shifted = shifted>>1;
log2 = res;
end
end
endfunction

localparam maxSIZE = `max(SIZEA, SIZEB);
localparam maxWIDTH = `max(WIDTHA, WIDTHB);
localparam minWIDTH = `min(WIDTHA, WIDTHB);

localparam RATIO = maxWIDTH / minWIDTH;
localparam log2RATIO = log2(RATIO);

reg [minWIDTH-1:0] RAM [0:maxSIZE-1];
reg [WIDTHB-1:0] readB;

always @(posedge clkA)
begin
if (enaA) begin
if (weA)
RAM[addrA] <= diA;
end
end


always @(posedge clkB)
```

```
begin : ramread
integer i;
reg [log2RATIO-1:0] lsbaddr;
if (enaB) begin
for (i = 0; i < RATIO; i = i+1) begin
lsbaddr = i;
readB[(i+1)*minWIDTH-1 -: minWIDTH] <= RAM[{addrB, lsbaddr}];
end
end
end
assign doB = readB;

endmodule
```

Simple Dual-Port Asymmetric RAM When Write is Wider than Read (Verilog)

Filename: asym_ram_sdp_write_wider.v

```
// Asymmetric port RAM
// Write wider than Read. Write Statement in a loop.
// asym_ram_sdp_write_wider.v

module asym_ram_sdp_write_wider (clkA, clkB, weA, enaA, enaB, addrA,
addrB, diA, doB);
parameter WIDTHB = 4;
parameter SIZEB = 1024;
parameter ADDRWIDTHB = 10;

parameter WIDTHA = 16;
parameter SIZEA = 256;
parameter ADDRWIDTHA = 8;
input clkA;
input clkB;
input weA;
input enaA, enaB;
input [ADDRWIDTHA-1:0] addrA;
input [ADDRWIDTHB-1:0] addrB;
input [WIDTHA-1:0] diA;
```

```
output [WIDTHB-1:0] doB;
`define max(a,b) {(a) > (b) ? (a) : (b)}
`define min(a,b) {(a) < (b) ? (a) : (b)}

function integer log2;
input integer value;
reg [31:0] shifted;
integer res;
begin
if (value < 2)
log2 = value;
else
begin
shifted = value-1;
for (res=0; shifted>0; res=res+1)
shifted = shifted>>1;
log2 = res;
end
end
endfunction

localparam maxSIZE = `max(SIZEA, SIZEB);
localparam maxWIDTH = `max(WIDTHA, WIDTHB);
localparam minWIDTH = `min(WIDTHA, WIDTHB);

localparam RATIO = maxWIDTH / minWIDTH;
localparam log2RATIO = log2(RATIO);

reg [minWIDTH-1:0] RAM [0:maxSIZE-1];
reg [WIDTHB-1:0] readB;

always @(posedge clkB) begin
if (enaB) begin
readB <= RAM[addrB];
end
end
assign doB = readB;

always @(posedge clkA)
begin : ramwrite
integer i;
reg [log2RATIO-1:0] lsbaddr;
```

```
for (i=0; i< RATIO; i= i+ 1) begin : write1
lsbaddr = i;
if (enaA) begin
if (weA)
RAM[{addrA, lsbaddr}] <= diA[(i+1)*minWIDTH-1 -: minWIDTH];
end
end
end

endmodule
```

Simple Dual Port Asymmetric RAM When Write Wider than Read (VHDL)

Filename: asym_ram_sdp_write_wider.vhd

```
-- Asymmetric port RAM
-- Write Wider than Read
-- asym_ram_sdp_write_wider.vhd


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asym_ram_sdp_write_wider is
generic(
WIDTHA : integer := 4;
SIZEA : integer := 1024;
ADDRWIDTHA : integer := 10;
WIDTHB : integer := 16;
SIZEB : integer := 256;
ADDRWIDTHB : integer := 8
);

port(
clkA : in std_logic;
clkB : in std_logic;
enA : in std_logic;
enB : in std_logic;
weB : in std_logic;
addrA : in std_logic_vector(ADDRWIDTHA - 1 downto 0);
addrB : in std_logic_vector(ADDRWIDTHB - 1 downto 0);
diB : in std_logic_vector(WIDTHB - 1 downto 0);
doA : out std_logic_vector(WIDTHA - 1 downto 0)
);

end asym_ram_sdp_write_wider;

architecture behavioral of asym_ram_sdp_write_wider is
function max(L, R : INTEGER) return INTEGER is
begin
if L > R then
return L;
else
return R;
end if;
end;

function min(L, R : INTEGER) return INTEGER is
begin
if L < R then
return L;
else
return R;
end if;
end;

function log2(val : INTEGER) return natural is
variable res : natural;
begin
for i in 0 to 31 loop
if (val <= (2 ** i)) then
res := i;
exit;
end if;
end loop;
return res;
end function Log2;

constant minWIDTH : integer := min(WIDTHA, WIDTHB);
constant maxWIDTH : integer := max(WIDTHA, WIDTHB);
constant maxSIZE : integer := max(SIZEA, SIZEB);
constant RATIO : integer := maxWIDTH / minWIDTH;

-- An asymmetric RAM is modeled in a similar way as a symmetric RAM, with
an
-- array of array object. Its aspect ratio corresponds to the port with
the
-- lower data width (larger depth)
type ramType is array (0 to maxSIZE - 1) of std_logic_vector(minWIDTH - 1
downto 0);

signal my_ram : ramType := (others => (others => '0'));

signal readA : std_logic_vector(WIDTHA - 1 downto 0) := (others => '0');
signal readB : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');
signal regA : std_logic_vector(WIDTHA - 1 downto 0) := (others => '0');
signal regB : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');

begin

-- read process
process(clkA)
begin
if rising_edge(clkA) then
if enA = '1' then
readA <= my_ram(conv_integer(addrA));
end if;
regA <= readA;
end if;
end process;

-- Write process
process(clkB)
begin
if rising_edge(clkB) then
for i in 0 to RATIO - 1 loop
if enB = '1' then
if weB = '1' then
my_ram(conv_integer(addrB & conv_std_logic_vector(i, log2(RATIO)))) <=
diB((i + 1) * minWIDTH - 1 downto i * minWIDTH);
end if;
end if;
end loop;
regB <= readB;
end if;
end process;

doA <= regA;

end behavioral;
```

True Dual Port Asymmetric RAM Read First (Verilog)

Filename: asym_ram_tdp_read_first.v

```
// Asymetric RAM - TDP
// READ_FIRST MODE.
// asym_ram_tdp_read_first.v


module asym_ram_tdp_read_first (clkA, clkB, enaA, weA, enaB, weB, addrA,
addrB, diA, doA, diB, doB);
parameter WIDTHB = 4;
parameter SIZEB = 1024;
parameter ADDRWIDTHB = 10;
parameter WIDTHA = 16;
parameter SIZEA = 256;
parameter ADDRWIDTHA = 8;
input clkA;
input clkB;
input weA, weB;
input enaA, enaB;

input [ADDRWIDTHA-1:0] addrA;
input [ADDRWIDTHB-1:0] addrB;
input [WIDTHA-1:0] diA;
input [WIDTHB-1:0] diB;

output [WIDTHA-1:0] doA;
output [WIDTHB-1:0] doB;

`define max(a,b) {(a) > (b) ? (a) : (b)}
```

```
`define min(a,b) {(a) < (b) ? (a) : (b)}

function integer log2;
input integer value;
reg [31:0] shifted;
integer res;
begin
if (value < 2)
log2 = value;
else
begin
shifted = value-1;
for (res=0; shifted>0; res=res+1)
shifted = shifted>>1;
log2 = res;
end
end
endfunction

localparam maxSIZE = `max(SIZEA, SIZEB);
localparam maxWIDTH = `max(WIDTHA, WIDTHB);
localparam minWIDTH = `min(WIDTHA, WIDTHB);

localparam RATIO = maxWIDTH / minWIDTH;
localparam log2RATIO = log2(RATIO);

reg [minWIDTH-1:0] RAM [0:maxSIZE-1];
reg [WIDTHA-1:0] readA;
reg [WIDTHB-1:0] readB;

always @(posedge clkB)
begin
if (enaB) begin
readB <= RAM[addrB] ;
if (weB)
RAM[addrB] <= diB;
end
end


always @(posedge clkA)
begin : portA
```

```
integer i;
reg [log2RATIO-1:0] lsbaddr ;
for (i=0; i< RATIO; i= i+ 1) begin
lsbaddr = i;
if (enaA) begin
readA[(i+1)*minWIDTH -1 -: minWIDTH] <= RAM[{addrA, lsbaddr}];

if (weA)
RAM[{addrA, lsbaddr}] <= diA[(i+1)*minWIDTH-1 -: minWIDTH];
end
end
end

assign doA = readA;
assign doB = readB;

endmodule
```

True Dual Port Asymmetric RAM Read First (VHDL)

Filename: asym_ram_tdp_read_first_first.vhd

```
-- asymmetric port RAM
-- True Dual port read first
-- asym_ram_tdp_read_first_first.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asym_ram_tdp_read_first is
generic(
WIDTHA : integer := 4;
SIZEA : integer := 1024;
ADDRWIDTHA : integer := 10;
WIDTHB : integer := 16;
SIZEB : integer := 256;
ADDRWIDTHB : integer := 8
);
```

```
port(
clkA : in std_logic;
clkB : in std_logic;
enA : in std_logic;
enB : in std_logic;
weA : in std_logic;
weB : in std_logic;
addrA : in std_logic_vector(ADDRWIDTHA - 1 downto 0);
addrB : in std_logic_vector(ADDRWIDTHB - 1 downto 0);
diA : in std_logic_vector(WIDTHA - 1 downto 0);
diB : in std_logic_vector(WIDTHB - 1 downto 0);
doA : out std_logic_vector(WIDTHA - 1 downto 0);
doB : out std_logic_vector(WIDTHB - 1 downto 0)
);

end asym_ram_tdp_read_first;

architecture behavioral of asym_ram_tdp_read_first is
function max(L, R : INTEGER) return INTEGER is
begin
if L > R then
return L;
else
return R;
end if;
end;

function min(L, R : INTEGER) return INTEGER is
begin
if L < R then
return L;
else
return R;
end if;
end;

function log2(val : INTEGER) return natural is
variable res : natural;
begin
for i in 0 to 31 loop
if (val <= (2 ** i)) then
```

```
res := i;
exit;
end if;
end loop;
return res;
end function Log2;

constant minWIDTH : integer := min(WIDTHA, WIDTHB);
constant maxWIDTH : integer := max(WIDTHA, WIDTHB);
constant maxSIZE : integer := max(SIZEA, SIZEB);
constant RATIO : integer := maxWIDTH / minWIDTH;

-- An asymmetric RAM is modeled in a similar way as a symmetric RAM, with
an
-- array of array object. Its aspect ratio corresponds to the port with
the
-- lower data width (larger depth)
type ramType is array (0 to maxSIZE - 1) of std_logic_vector(minWIDTH - 1
downto 0);

signal my_ram : ramType := (others => (others => '0'));

signal readA : std_logic_vector(WIDTHA - 1 downto 0) := (others => '0');
signal readB : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');
signal regA : std_logic_vector(WIDTHA - 1 downto 0) := (others => '0');
signal regB : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');

begin
process(clkA)
begin
if rising_edge(clkA) then
if enA = '1' then
readA <= my_ram(conv_integer(addrA));
if weA = '1' then
my_ram(conv_integer(addrA)) <= diA;
end if;
end if;
regA <= readA;
end if;
end process;

process(clkB)
```

```
begin
if rising_edge(clkB) then
for i in 0 to RATIO - 1 loop
if enB = '1' then
readB((i + 1) * minWIDTH - 1 downto i * minWIDTH) <=
my_ram(conv_integer(addrB & conv_std_logic_vector(i, log2(RATIO))));
if weB = '1' then
my_ram(conv_integer(addrB & conv_std_logic_vector(i, log2(RATIO)))) <=
diB((i + 1) * minWIDTH - 1 downto i * minWIDTH);
end if;
end if;
end loop;
regB <= readB;
end if;
end process;

doA <= regA;
doB <= regB;

end behavioral;
```

True Dual Port Asymmetric RAM Write First (Verilog)

Filename: asym_ram_tdp_write_first.v

```
// Asymmetric port RAM - TDP
// WRITE_FIRST MODE.
// asym_ram_tdp_write_first.v


module asym_ram_tdp_write_first (clkA, clkB, enaA, weA, enaB, weB, addrA,
addrB, diA, doA, diB, doB);
parameter WIDTHB = 4;
parameter SIZEB = 1024;
parameter ADDRWIDTHB = 10;
parameter WIDTHA = 16;
parameter SIZEA = 256;
parameter ADDRWIDTHA = 8;
input clkA;
input clkB;
input weA, weB;
input enaA, enaB;

input [ADDRWIDTHA-1:0] addrA;
input [ADDRWIDTHB-1:0] addrB;
input [WIDTHA-1:0] diA;
input [WIDTHB-1:0] diB;

output [WIDTHA-1:0] doA;
output [WIDTHB-1:0] doB;

`define max(a,b) {(a) > (b) ? (a) : (b)}
`define min(a,b) {(a) < (b) ? (a) : (b)}

function integer log2;
input integer value;
reg [31:0] shifted;
integer res;
begin
if (value < 2)
log2 = value;
else
begin
shifted = value-1;
for (res=0; shifted>0; res=res+1)
shifted = shifted>>1;
log2 = res;
end
end
endfunction

localparam maxSIZE = `max(SIZEA, SIZEB);
localparam maxWIDTH = `max(WIDTHA, WIDTHB);
localparam minWIDTH = `min(WIDTHA, WIDTHB);

localparam RATIO = maxWIDTH / minWIDTH;
localparam log2RATIO = log2(RATIO);

reg [minWIDTH-1:0] RAM [0:maxSIZE-1];
reg [WIDTHA-1:0] readA;
reg [WIDTHB-1:0] readB;
```

```
always @(posedge clkB)
begin
if (enaB) begin
if (weB)
RAM[addrB] = diB;
readB = RAM[addrB] ;
end
end


always @(posedge clkA)
begin : portA
integer i;
reg [log2RATIO-1:0] lsbaddr ;
for (i=0; i< RATIO; i= i+ 1) begin
lsbaddr = i;
if (enaA) begin

if (weA)
RAM[{addrA, lsbaddr}] = diA[(i+1)*minWIDTH-1 -: minWIDTH];

readA[(i+1)*minWIDTH -1 -: minWIDTH] = RAM[{addrA, lsbaddr}];
end
end
end

assign doA = readA;
assign doB = readB;

endmodule
```

True Dual Port Asymmetric RAM Write First (VHDL)

Filename: asym_ram_tdp_write_first.vhd

```
--Asymmetric RAM
--True Dual Port write first mode.
--asym_ram_tdp_write_first.vhd
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asym_ram_tdp_write_first is
generic(
WIDTHA : integer := 4;
SIZEA : integer := 1024;
ADDRWIDTHA : integer := 10;
WIDTHB : integer := 16;
SIZEB : integer := 256;
ADDRWIDTHB : integer := 8
);

port(
clkA : in std_logic;
clkB : in std_logic;
enA : in std_logic;
enB : in std_logic;
weA : in std_logic;
weB : in std_logic;
addrA : in std_logic_vector(ADDRWIDTHA - 1 downto 0);
addrB : in std_logic_vector(ADDRWIDTHB - 1 downto 0);
diA : in std_logic_vector(WIDTHA - 1 downto 0);
diB : in std_logic_vector(WIDTHB - 1 downto 0);
doA : out std_logic_vector(WIDTHA - 1 downto 0);
doB : out std_logic_vector(WIDTHB - 1 downto 0)
);

end asym_ram_tdp_write_first;

architecture behavioral of asym_ram_tdp_write_first is
function max(L, R : INTEGER) return INTEGER is
begin
if L > R then
return L;
else
return R;
end if;
end;
```

```
function min(L, R : INTEGER) return INTEGER is
begin
if L < R then
return L;
else
return R;
end if;
end;

function log2(val : INTEGER) return natural is
variable res : natural;
begin
for i in 0 to 31 loop
if (val <= (2 ** i)) then
res := i;
exit;
end if;
end loop;
return res;
end function Log2;

constant minWIDTH : integer := min(WIDTHA, WIDTHB);
constant maxWIDTH : integer := max(WIDTHA, WIDTHB);
constant maxSIZE : integer := max(SIZEA, SIZEB);
constant RATIO : integer := maxWIDTH / minWIDTH;

-- An asymmetric RAM is modeled in a similar way as a symmetric RAM, with
an
-- array of array object. Its aspect ratio corresponds to the port with
the
-- lower data width (larger depth)
type ramType is array (0 to maxSIZE - 1) of std_logic_vector(minWIDTH - 1
downto 0);

signal my_ram : ramType := (others => (others => '0'));

signal readA : std_logic_vector(WIDTHA - 1 downto 0) := (others => '0');
signal readB : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');
signal regA : std_logic_vector(WIDTHA - 1 downto 0) := (others => '0');
signal regB : std_logic_vector(WIDTHB - 1 downto 0) := (others => '0');

begin
```

```
process(clkA)
begin
if rising_edge(clkA) then
if enA = '1' then
if weA = '1' then
my_ram(conv_integer(addrA)) <= diA;
readA <= diA;
else
readA <= my_ram(conv_integer(addrA));
end if;
end if;
regA <= readA;
end if;
end process;

process(clkB)
begin
if rising_edge(clkB) then
for i in 0 to RATIO - 1 loop
if enB = '1' then
if weB = '1' then
my_ram(conv_integer(addrB & conv_std_logic_vector(i, log2(RATIO)))) <=
diB((i + 1) * minWIDTH - 1 downto i * minWIDTH);
end if;
-- The read statement below is placed after the write statement -- on
purpose
-- to ensure write-first synchronization through the variable
-- mechanism
readB((i + 1) * minWIDTH - 1 downto i * minWIDTH) <=
my_ram(conv_integer(addrB & conv_std_logic_vector(i, log2(RATIO))));
end if;
end loop;
regB <= readB;
end if;
end process;

doA <= regA;
doB <= regB;

end behavioral;
```

Initializing RAM Contents

RAM can be initialized in following ways:

- Specifying RAM Initial Contents in the HDL Source Code
- Specifying RAM Initial Contents in an External Data File

Specifying RAM Initial Contents in the HDL Source Code

Use the signal default value mechanism to describe initial RAM contents directly in the HDL source code.

VHDL Coding Examples

```
type ram_type is array (0 to 31) of std_logic_vector(19 downto 0);
signal RAM : ram_type :=
(
X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300",
X"08602", X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500",
X"04001", X"02500", X"00340", X"00241", X"04002", X"08300", X"08201",
X"00500", X"08101", X"00602", X"04003", X"0241E", X"00301", X"00102",
X"02122", X"02021", X"0030D", X"08201"
);
```

All bit positions are initialized to the same value:

```
type ram_type is array (0 to 127) of std_logic_vector (15 downto 0);
signal RAM : ram_type := (others => (others => '0'));
```

Verilog Coding Example

All addressable words are initialized to the same value.

```
reg [DATA_WIDTH-1:0] ram [DEPTH-1:0];
integer i;
initial for (i=0; i<DEPTH; i=i+1) ram[i] = 0;
end
```

Specifying RAM Initial Contents in an External Data File

Use the file read function in the HDL source code to load the RAM initial contents from an external data file.

- The external data file is an ASCII text file with any name.
- Each line in the external data file describes the initial content at an address position in the RAM.
- There must be as many lines in the external data file as there are rows in the RAM array. An insufficient number of lines is flagged.
- The addressable position related to a given line is defined by the direction of the primary range of the signal modeling the RAM.
- You can represent RAM content in either binary or hexadecimal. You cannot mix both.
- The external data file cannot contain any other content, such as comments.

The following external data file initializes an 8 x 32-bit RAM with binary values:

```
00001110110000011001111011000110
00101011001011010101001000100011
01110100010100011000011100001111
01000001010000100101001110010100
00001001101001111111101000101011
00101101001011111110101010100111
11101111000100111000111101101101
10001111010010011001000011101111
00000001100011100011110010011111
11011111001110101011111001001010
11100111010100111110110011001010
11000100000100110110011110010100 1
```

```
11100111110001111010101100001101
01110100000011101111111000011111
00010011110101111000111001011101
01101110001111100011010101101111
10111110000000001001110101101011
11000001001101001101111100010000
00011111110010110110011111010101
011001001000000011100100101110000
10001000000100111011001010001111
11001000100011101001010001100001
10000000100111010011100111100011
11011111010010100010101010000111
10000000110111101000111110111011
10110011010111101111000110011001
00010111100001001010110111011100
10011100101110101111011010110011
01010011101101010001110110011010
01111011011100010101000101000001
10001000000110010110111001101010
11101000001101010000111001010110
11100011111100000111110101110101
01001010000000001111111101101111
00100011000011001000000010001111
10011000111010110001001011100100
11111111111011110101000101000111
11000011000101000011100110100000
01101101001011111010100011101001
10000111101100101001110011010111
11010110100100101110110010100100
01001111111001101101011111001011
11011001001101110110000100110111
10110110110111100101110011100110
10011100111001000010111111010110
00000000010110111111001010110010
10100110011010000010001000011011
11001010111111001001110001110101
00100001100010000111000101001000
00111100101111110001101101111010
11000010001010000000010100100001
110000001000110001101000101001110
```

```
10010011010100010001100100100111
```

Verilog Code Example

```
reg [31:0] ram [0:63];

initial begin
$readmemb("rams_20c.data", ram, 0, 63);
end
```

VHDL Code Example

Load the data as follows:

```
type RamType is array(0 to 7) of bit_vector(31 downto 0);
impure function InitRamFromFile (RamFileName : in string) return RamType
is
FILE RamFile : text is in RamFileName;
variable RamFileLine : line;
variable RAM : RamType;
begin
for I in RamType'range loop
readline (RamFile, RamFileLine);
read (RamFileLine, RAM(I));
end loop;
return RAM;
end function;
signal RAM : RamType := InitRamFromFile("rams_20c.data");
```

Initializing Block RAM (Verilog)

Filename: rams_sp_rom.v

```
// Initializing Block RAM (Single-Port Block RAM)
// File: rams_sp_rom
module rams_sp_rom (clk, we, addr, di, dout);
input clk;
input we;
input [5:0] addr;
input [19:0] di;
output [19:0] dout;

reg [19:0] ram [63:0];
reg [19:0] dout;

initial
begin
ram[63] = 20'h0200A; ram[62] = 20'h00300; ram[61] = 20'h08101;
ram[60] = 20'h04000; ram[59] = 20'h08601; ram[58] = 20'h0233A;
ram[57] = 20'h00300; ram[56] = 20'h08602; ram[55] = 20'h02310;
ram[54] = 20'h0203B; ram[53] = 20'h08300; ram[52] = 20'h04002;
ram[51] = 20'h08201; ram[50] = 20'h00500; ram[49] = 20'h04001;
ram[48] = 20'h02500; ram[47] = 20'h00340; ram[46] = 20'h00241;
ram[45] = 20'h04002; ram[44] = 20'h08300; ram[43] = 20'h08201;
ram[42] = 20'h00500; ram[41] = 20'h08101; ram[40] = 20'h00602;
ram[39] = 20'h04003; ram[38] = 20'h0241E; ram[37] = 20'h00301;
ram[36] = 20'h00102; ram[35] = 20'h02122; ram[34] = 20'h02021;
ram[33] = 20'h00301; ram[32] = 20'h00102; ram[31] = 20'h02222;
ram[30] = 20'h04001; ram[29] = 20'h00342; ram[28] = 20'h0232B;
ram[27] = 20'h00900; ram[26] = 20'h00302; ram[25] = 20'h00102;
ram[24] = 20'h04002; ram[23] = 20'h00900; ram[22] = 20'h08201;
ram[21] = 20'h02023; ram[20] = 20'h00303; ram[19] = 20'h02433;
ram[18] = 20'h00301; ram[17] = 20'h04004; ram[16] = 20'h00301;
ram[15] = 20'h00102; ram[14] = 20'h02137; ram[13] = 20'h02036;
ram[12] = 20'h00301; ram[11] = 20'h00102; ram[10] = 20'h02237;
ram[9] = 20'h04004; ram[8] = 20'h00304; ram[7] = 20'h04040;
ram[6] = 20'h02500; ram[5] = 20'h02500; ram[4] = 20'h02500;
ram[3] = 20'h0030D; ram[2] = 20'h02341; ram[1] = 20'h08201;
ram[0] = 20'h0400D;
end

always @(posedge clk)
begin
if (we)
ram[addr] <= di;
dout <= ram[addr];
end

endmodule
```

Initializing Block RAM (VHDL)

Filename: rams_sp_rom.vhd

```
-- Initializing Block RAM (Single-Port Block RAM)
-- File: rams_sp_rom.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity rams_sp_rom is
port(
clk : in std_logic;
we : in std_logic;
addr : in std_logic_vector(5 downto 0);
di : in std_logic_vector(19 downto 0);
do : out std_logic_vector(19 downto 0)
);
end rams_sp_rom;

architecture syn of rams_sp_rom is
type ram_type is array (63 downto 0) of std_logic_vector(19 downto 0);
signal RAM : ram_type := (X"0200A", X"00300", X"08101", X"04000",
X"08601", X"0233A",
X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
X"0030D", X"02341", X"08201", X"0400D");

begin
process(clk)
begin
if rising_edge(clk) then
if we = '1' then
RAM(to_integer(unsigned(addr))) <= di;
```

```
end if;
do <= RAM(to_integer(unsigned(addr)));
end if;
end process;

end syn;
```

Initializing Block RAM From an External Data File (Verilog)

Filename: rams_init_file.v

```
// Initializing Block RAM from external data file
// Binary data
// File: rams_init_file.v

module rams_init_file (clk, we, addr, din, dout);
input clk;
input we;
input [5:0] addr;
input [31:0] din;
output [31:0] dout;

reg [31:0] ram [0:63];
reg [31:0] dout;

initial begin
$readmemb("rams_init_file.data",ram);
end

always @(posedge clk)
begin
if (we)
ram[addr] <= din;
dout <= ram[addr];
end
endmodule
```

Note: The external file initializing the RAM needs to be in bit vector form. External files in integer or hex format do not work.

Initializing Block RAM From an External Data File (VHDL)

Filename: rams_init_file.vhd

```
-- Initializing Block RAM from external data file
-- File: rams_init_file.vhd

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

entity rams_init_file is
port(
clk : in std_logic;
we : in std_logic;
addr : in std_logic_vector(5 downto 0);
din : in std_logic_vector(31 downto 0);
dout : out std_logic_vector(31 downto 0)
);
end rams_init_file;

architecture syn of rams_init_file is
type RamType is array (0 to 63) of bit_vector(31 downto 0);

impure function InitRamFromFile(RamFileName : in string) return RamType is
FILE RamFile : text is in RamFileName;
variable RamFileLine : line;
variable RAM : RamType;
begin
for I in RamType'range loop
readline(RamFile, RamFileLine);
read(RamFileLine, RAM(I));
end loop;
return RAM;
end function;

signal RAM : RamType := InitRamFromFile("rams_init_file.data");
begin
process(clk)
begin
```

```
if clk'event and clk = '1' then
if we = '1' then
RAM(to_integer(unsigned(addr))) <= to_bitvector(din);
end if;
dout <= to_stdlogicvector(RAM(to_integer(unsigned(addr))));
end if;
end process;

end syn;
```

Note:
The external file initializing the RAM needs to be in bit vector form. External files in integer or hex format do not work.

3D RAM Inference

RAMs Using 3D Arrays

The following examples show inference of RAMs using 3D arrays.

3D RAM Inference Single Port (Verilog)

filename: rams_sp_3d.sv

```
// 3-D Ram Inference Example (Single port)
// File:rams_sp_3d.sv
module rams_sp_3d #(
parameter NUM_RAMS = 2,
A_WID = 10,
D_WID = 32
)
(
input clk,
input [NUM_RAMS-1:0] we,
input [NUM_RAMS-1:0] ena,
input [A_WID-1:0] addr [NUM_RAMS-1:0],
input [D_WID-1:0] din [NUM_RAMS-1:0],
output reg [D_WID-1:0] dout [NUM_RAMS-1:0]
);
```

```
reg [D_WID-1:0] mem [NUM_RAMS-1:0][2**A_WID-1:0];
genvar i;

generate
for(i=0;i<NUM_RAMS;i=i+1)
begin:u
always @ (posedge clk)
begin
if (ena[i]) begin
if(we[i])
begin
mem[i][addr[i]] <= din[i];
end
dout[i] <= mem[i][addr[i]];
end
end
end
endgenerate

endmodule
```

3D RAM Inference Single Port (VHDL)

Filename: ram_sp_3d.vhd

```
-- 3-D Ram Inference Example (Single port)
-- Compile this file in VHDL2008 mode
-- File:rams_sp_3d.vhd

library ieee;
use ieee.std_logic_1164.all;
package mypack is
type myarray_t is array(integer range<>) of std_logic_vector;
type mem_t is array(integer range<>) of myarray_t;
end package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.mypack.all;
```

```
entity rams_sp_3d is generic (
NUM_RAMS : integer := 2;
A_WID : integer := 10;
D_WID : integer := 32
);
port (
clk : in std_logic;
we : in std_logic_vector(NUM_RAMS-1 downto 0);
ena : in std_logic_vector(NUM_RAMS-1 downto 0);
addr : in myarray_t(NUM_RAMS-1 downto 0)(A_WID-1 downto 0);
din : in myarray_t(NUM_RAMS-1 downto 0)(D_WID-1 downto 0);
dout : out myarray_t(NUM_RAMS-1 downto 0)(D_WID-1 downto 0)
);
end rams_sp_3d;

architecture arch of rams_sp_3d is
signal mem : mem_t(NUM_RAMS-1 downto 0)(2**A_WID-1 downto 0)(D_WID-1
downto 0);
begin
process(clk)
begin
if(clk' event and clk=' 1' ) then
for i in 0 to NUM_RAMS-1 loop
if(ena(i) =  ‘1’ ) then
if(we(i) =  ‘1’ ) then
mem(i)(to_integer(unsigned(addr(i)))) <= din(i);
end if;
dout(i) <= mem(i)(to_integer(unsigned(addr(i))));
end if;
end loop;
end if;
end process;

end arch;
```

3D RAM Inference Simple Dual Port (Verilog)

Filename: rams_sdp_3d.sv

```
// 3-D Ram Inference Example (Simple Dual port)
// File:rams_sdp_3d.sv
module rams_sdp_3d #(
parameter NUM_RAMS = 2,
A_WID = 10,
D_WID = 32
)
(
input clka,
input clkb,
input [NUM_RAMS-1:0] wea,
input [NUM_RAMS-1:0] ena,
input [NUM_RAMS-1:0] enb,
input [A_WID-1:0] addra [NUM_RAMS-1:0],
input [A_WID-1:0] addrb [NUM_RAMS-1:0],
input [D_WID-1:0] dina [NUM_RAMS-1:0],
output reg [D_WID-1:0] doutb [NUM_RAMS-1:0]
);

reg [D_WID-1:0] mem [NUM_RAMS-1:0][2**A_WID-1:0];
// PORT_A
genvar i;
generate
for(i=0;i<NUM_RAMS;i=i+1)
begin:port_a_ops
always @ (posedge clka)
begin
if (ena[i]) begin
if(wea[i])
begin
mem[i][addra[i]] <= dina[i];
end
end
end
end
endgenerate
```

```
//PORT_B
generate
for(i=0;i<NUM_RAMS;i=i+1)
begin:port_b_ops
always @ (posedge clkb)
begin
if (enb[i])
doutb[i] <= mem[i][addrb[i]];
end
end
endgenerate

endmodule
```

3D RAM Inference - Simple Dual Port (VHDL)

filename: rams_sdp_3d.vhd

```
-- 3-D Ram Inference Example ( Simple Dual port)
-- Compile this file in VHDL2008 mode
-- File:rams_sdp_3d.vhd

library ieee;
use ieee.std_logic_1164.all;
package mypack is
type myarray_t is array(integer range<>) of std_logic_vector;
type mem_t is array(integer range<>) of myarray_t;
end package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.mypack.all;
entity rams_sdp_3d is generic (
NUM_RAMS : integer := 2;
A_WID : integer := 10;
D_WID : integer := 32
);
port (
clka : in std_logic;
clkb : in std_logic;
wea : in std_logic_vector(NUM_RAMS-1 downto 0);
ena : in std_logic_vector(NUM_RAMS-1 downto 0);
enb : in std_logic_vector(NUM_RAMS-1 downto 0);
addra : in myarray_t(NUM_RAMS-1 downto 0)(A_WID-1 downto 0);
addrb : in myarray_t(NUM_RAMS-1 downto 0)(A_WID-1 downto 0);
dina : in myarray_t(NUM_RAMS-1 downto 0)(D_WID-1 downto 0);
doutb : out myarray_t(NUM_RAMS-1 downto 0)(D_WID-1 downto 0)
);
end rams_sdp_3d;

architecture arch of rams_sdp_3d is
signal mem : mem_t(NUM_RAMS-1 downto 0)(2**A_WID-1 downto 0)(D_WID-1
downto 0);
begin
process(clka)
begin
if(clka'event and clka='1') then
for i in 0 to NUM_RAMS-1 loop
if(ena(i) = '1') then
if(wea(i) = '1') then
mem(i)(to_integer(unsigned(addra(i)))) <= dina(i);
end if;
end if;
end loop;
end if;
end process;

process(clkb)
begin
if(clkb'event and clkb='1') then
for i in 0 to NUM_RAMS-1 loop
if(enb(i) = '1') then
doutb(i) <= mem(i)(to_integer(unsigned(addrb(i))));
end if;
end loop;
end if;
end process;

end arch;
```

3D RAM Inference True Dual Port (Verilog)

Filename: rams_tdp_3d.sv

```
// 3-D Ram Inference Example (True Dual port)
// File:rams_tdp_3d.sv
module rams_tdp_3d #(
parameter NUM_RAMS = 2,
A_WID = 10,
D_WID = 32
)
(
input clka,
input clkb,
input [NUM_RAMS-1:0] wea,
input [NUM_RAMS-1:0] web,
input [NUM_RAMS-1:0] ena,
input [NUM_RAMS-1:0] enb,
input [A_WID-1:0] addra [NUM_RAMS-1:0],
input [A_WID-1:0] addrb [NUM_RAMS-1:0],
input [D_WID-1:0] dina [NUM_RAMS-1:0],
input [D_WID-1:0] dinb [NUM_RAMS-1:0],
output reg [D_WID-1:0] douta [NUM_RAMS-1:0],
output reg [D_WID-1:0] doutb [NUM_RAMS-1:0]
);

reg [D_WID-1:0] mem [NUM_RAMS-1:0][2**A_WID-1:0];
// PORT_A
genvar i;
generate
for(i=0;i<NUM_RAMS;i=i+1)
begin:port_a_ops
always @ (posedge clka)
begin
if (ena[i]) begin
if(wea[i])
begin
mem[i][addra[i]] <= dina[i];
end
douta[i] <= mem[i][addra[i]];
end
```

```
end
end
endgenerate

//PORT_B
generate
for(i=0;i<NUM_RAMS;i=i+1)
begin:port_b_ops
always @ (posedge clkb)
begin
if (enb[i]) begin
if(web[i])
begin
mem[i][addrb[i]] <= dinb[i];
end
doutb[i] <= mem[i][addrb[i]];
end
end
end
endgenerate

endmodule
```

RAM Inference Using Structures and Records

The following examples show inference of RAMs using Structures and Records.

RAM Inference Single Port Structure (Verilog)

Filename: rams_sp_struct.sv

```
// RAM Inference using Struct in SV(Simple Dual port)
// File:rams_sdp_struct.sv
typedef struct packed {
logic [3:0] addr;
logic [27:0] data;
} Packet;


module rams_sdp_struct #(
parameter A_WID = 10,
D_WID = 32
)
(
input clk,
input we,
input ena,
input [A_WID-1:0] raddr, waddr,
input Packet din,
output Packet dout
);

Packet mem [2**A_WID-1:0];

always @ (posedge clk)
begin
if (ena) begin
if(we)
mem[waddr] <= din;
end
end

always @ (posedge clk)
begin
if (ena) begin
dout <= mem[raddr];
end
end
endmodule
```

RAM Inference Single Port Structure (VHDL)

Filename: rams_sp_record.vhd

```
-- Ram Inference Example using Records (Single port)
-- File:rams_sp_record.vhd

library ieee;
use ieee.std_logic_1164.all;
package mypack is
type Packet is record
```

```
addr : std_logic_vector(3 downto 0);
data : std_logic_vector(27 downto 0);
end record Packet;
type mem_t is array(integer range<>) of Packet;
end package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.mypack.all;
entity rams_sp_record is generic (
A_WID : integer := 10;
D_WID : integer := 32
);
port (
clk : in std_logic;
we : in std_logic;
ena : in std_logic;
addr : in std_logic_vector(A_WID-1 downto 0);
din : in Packet;
dout : out Packet
);
end rams_sp_record;

architecture arch of rams_sp_record is
signal mem : mem_t(2**A_WID-1 downto 0);
begin
process(clk)
begin
if(clk'event and clk='1') then
if(ena = '1') then
if(we = '1') then
mem(to_integer(unsigned(addr))) <= din;
end if;
dout <= mem(to_integer(unsigned(addr)));
end if;
end if;
end process;

end arch;
```

RAM Inference - Simple Dual Port Structure (SystemVerilog)

Filename: rams_sdp_struct.sv

```
// RAM Inference using Struct in SV(Simple Dual port)
// File:rams_sdp_struct.sv
typedef struct packed {
logic [3:0] addr;
logic [27:0] data;
} Packet;


module rams_sdp_struct #(
parameter A_WID = 10,
D_WID = 32
)
(
input clk,
input we,
input ena,
input [A_WID-1:0] raddr, waddr,
input Packet din,
output Packet dout
);

Packet mem [2**A_WID-1:0];

always @ (posedge clk)
begin
if (ena) begin
if(we)
mem[waddr] <= din;
end
end

always @ (posedge clk)
begin
if (ena) begin
dout <= mem[raddr];
end
```

```
end
endmodule
```

RAM Inference - Simple Dual Port Record (VHDL)

Filename: rams_sdp_record.vhd

```
-- Ram Inference Example using Records (Simple Dual port)
-- File:rams_sdp_record.vhd

library ieee;
use ieee.std_logic_1164.all;
package mypack is
type Packet is record
addr : std_logic_vector(3 downto 0);
data : std_logic_vector(27 downto 0);
end record Packet;
type mem_t is array(integer range<>) of Packet;
end package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.mypack.all;
entity rams_sdp_record is generic (
A_WID : integer := 10;
D_WID : integer := 32
);
port (
clk : in std_logic;
we : in std_logic;
ena : in std_logic;
raddr : in std_logic_vector(A_WID-1 downto 0);
waddr : in std_logic_vector(A_WID-1 downto 0);
din : in Packet;
dout : out Packet
);
end rams_sdp_record;

architecture arch of rams_sdp_record is
```

```
signal mem : mem_t(2**A_WID-1 downto 0);
begin
process(clk)
begin
if(clk'event and clk='1') then
if(ena = '1') then
if(we = '1') then
mem(to_integer(unsigned(waddr))) <= din;
end if;
end if;
end if;
end process;

process(clk)
begin
if(clk'event and clk='1') then
if(ena = '1') then
dout <= mem(to_integer(unsigned(raddr)));
end if;
end if;
end process;

end arch;
```

RAM Inference True Dual Port Structure (SystemVerilog)

Filename: rams_tdp_struct.sv

```
// RAM Inference using Struct in SV(True Dual port)
// File:rams_tdp_struct.sv
typedef struct packed {
logic [3:0] addr;
logic [27:0] data;
} Packet;

module rams_tdp_struct #(
parameter A_WID = 10,
D_WID = 32
)
(
input clka,
input clkb,
input wea,
input web,
input ena,
input enb,
input [A_WID-1:0] addra,
input [A_WID-1:0] addrb,
input Packet dina, dinb,
output Packet douta, doutb
);

Packet mem [2**A_WID-1:0];

always @ (posedge clka)
begin
if (ena)
begin
douta <= mem[addra];
if(wea)
mem[addra] <= dina;
end
end

always @ (posedge clkb)
begin
if (enb)
begin
doutb <= mem[addrb];
if(web)
mem[addrb] <= dinb;
end
end

endmodule
```

RAM Inference True Dual Port Record (VHDL)

Filename: rams_tdp_record.vhd

```
-- Ram Inference Example using Records (True Dual port)
-- File:rams_tdp_record.vhd

library ieee;
use ieee.std_logic_1164.all;
package mypack is
type Packet is record
addr : std_logic_vector(3 downto 0);
data : std_logic_vector(27 downto 0);
end record Packet;
type mem_t is array(integer range<>) of Packet;
end package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.mypack.all;
entity rams_tdp_record is generic (
A_WID : integer := 10;
D_WID : integer := 32
);
port (
clka : in std_logic;
clkb : in std_logic;
wea : in std_logic;
web : in std_logic;
ena : in std_logic;
enb : in std_logic;
addra : in std_logic_vector(A_WID-1 downto 0);
addrb : in std_logic_vector(A_WID-1 downto 0);
dina : in Packet;
dinb : in Packet;
douta : out Packet;
doutb : out Packet
);
end rams_tdp_record;
```

```
architecture arch of rams_tdp_record is
signal mem : mem_t(2**A_WID-1 downto 0);
begin

process(clka)
begin
if(clka'event and clka='1') then
if(ena = '1') then
douta <= mem(to_integer(unsigned(addra)));
if(wea = '1') then
mem(to_integer(unsigned(addra))) <= dina;
end if;
end if;
end if;
end process;


process(clkb)
begin
if(clkb'event and clkb='1') then
if(enb = '1') then
doutb <= mem(to_integer(unsigned(addrb)));
if(web = '1') then
mem(to_integer(unsigned(addrb))) <= dinb;
end if;
end if;
end if;
end process;

end arch;
```

Black Boxes

A design can contain EDIF files generated by:

- Synthesis tools
- Schematic text editors
- Any other design entry mechanism

These modules must be instantiated to be connected to the rest of the design. Use `BLACK_BOX` instantiation in the HDL source code.

Vivado synthesis lets you apply specific constraints to these BLACK_BOX instantiations. After you make a design a BLACK_BOX, each instance of that design is a BLACK_BOX. Download the coding example files from Coding Examples.

Black Box Verilog Example

Filename: black_box_1.v

```
// Black Box
// black_box_1.v
//
(* black_box *) module black_box1 (in1, in2, dout);
input in1, in2;
output dout;
endmodule

module black_box_1 (DI_1, DI_2, DOUT);
input DI_1, DI_2;
output DOUT;

black_box1 U1 (
.in1(DI_1),
.in2(DI_2),
.dout(DOUT)
);

endmodule
```

Black Box VHDL Example

Filename: black_box_1.vhd

```
-- Black Box
-- black_box_1.vhd
library ieee;
use ieee.std_logic_1164.all;

entity black_box_1 is
port(DI_1, DI_2 : in std_logic;
DOUT : out std_logic);
```

```
end black_box_1;
architecture rtl of black_box_1 is
component black_box1
port(I1 : in std_logic;
I2 : in std_logic;
O : out std_logic);
end component;

attribute black_box : string;
attribute black_box of black_box1 : component is "yes";

begin
U1 : black_box1 port map(I1 => DI_1, I2 => DI_2, O => DOUT);
end rtl;
```

FSM Components

Vivado Synthesis Features

- Specific inference capabilities for synchronous Finite State Machine (FSM) components.
- Built-in FSM encoding strategies to accommodate your optimization goals.
- FSM extraction is enabled by default.
- Use `-fsm_extraction off` to disable FSM extraction.

FSM Description

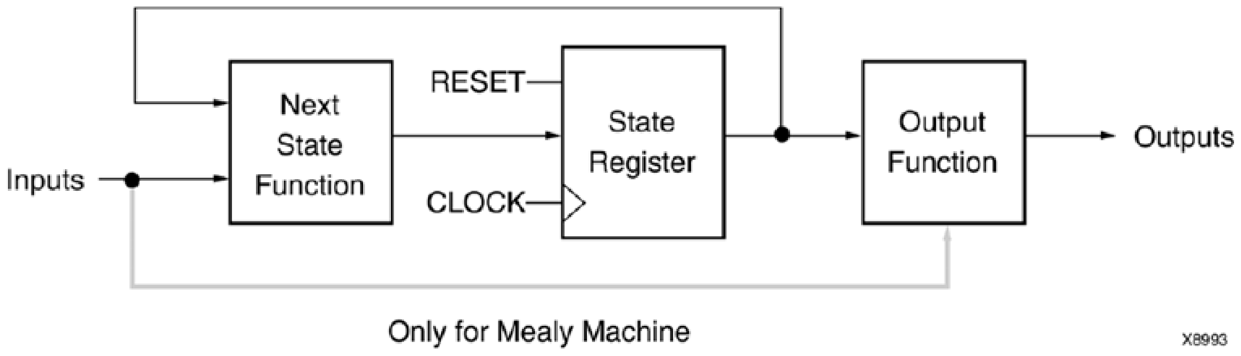
Vivado synthesis supports specification of Finite State Machine (FSM) in both Moore and Mealy form. An FSM consists of the following:

- A state register
- A next state function
- An outputs function

FSM Diagrams

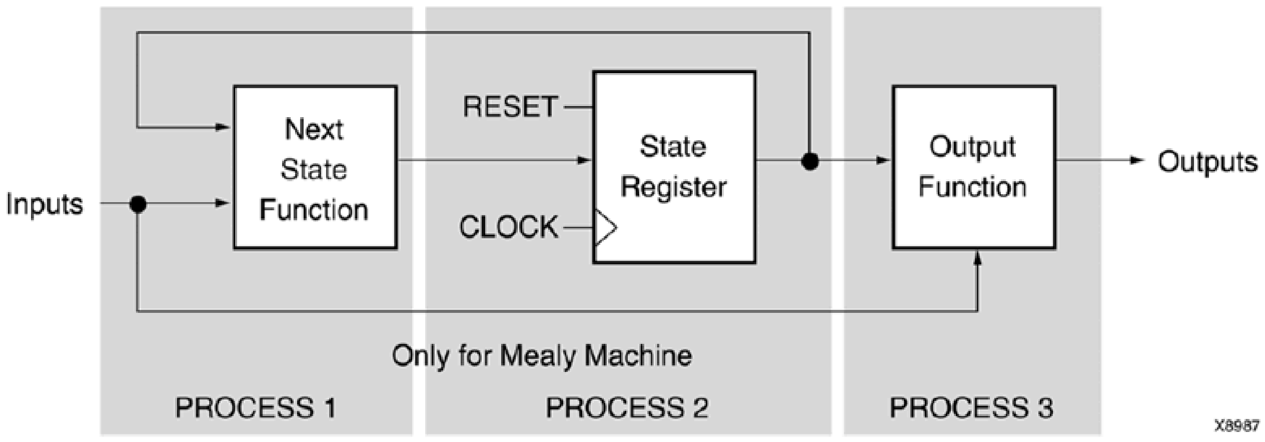
The following diagram shows an FSM representation that incorporates Mealy and Moore machines.

Figure: FSM Representation Incorporating Mealy and Moore Machines Diagram



The following diagram shows an FSM diagram with three processes.

Figure: FSM with Three Processes Diagram



FSM Registers

- Specify a reset or power-up state for Vivado synthesis to identify a Finite State Machine (FSM) or set the value of `FSM_ENCODING` to "`none`".
- The State Register can be asynchronously or synchronously reset to a particular state.

Note: Use synchronous reset logic over asynchronous reset logic for an FSM.

Auto State Encoding

When `FSM_ENCODING` is set to "`auto`", the Vivado synthesis attempts to select the best-suited encoding method for a given FSM.

One-Hot State Encoding

One-Hot State encoding has the following attributes:

- Is the default encoding scheme for a state machine, up to 32 states.
- Is usually a good choice for optimizing speed or reducing power dissipation.
- Assigns a distinct bit of code to each FSM state.
- Implements the State Register with one flip-flop for each state.
- In a given clock cycle during operation, only one bit of the State Register is asserted.
- Only two bits toggle during a transition between two states.

Gray State Encoding

Gray State encoding has the following attributes:

- Guarantees that only one bit switches between two consecutive states.
- Is appropriate for controllers exhibiting long paths without branching.
- Minimizes hazards and glitches.
- Can be used to minimize power dissipation.

Johnson State Encoding

Johnson State encoding is beneficial when using state machines containing long paths with no branching (as in Gray State Encoding).

Sequential State Encoding

Sequential State encoding has the following attributes:

- Identifies long paths
- Applies successive radix two codes to the states on these paths.
- Minimizes next state equations.

FSM Example (Verilog)

Filename: fsm_1.v

```
// State Machine with single sequential block
//fsm_1.v
module fsm_1(clk,reset,flag,sm_out);
input clk,reset,flag;
output reg sm_out;
```

```
parameter s1 = 3'b000;
parameter s2 = 3'b001;
parameter s3 = 3'b010;
parameter s4 = 3'b011;
parameter s5 = 3'b111;

reg [2:0] state;

always@(posedge clk)
begin
if(reset)
begin
state <= s1;
sm_out <= 1'b1;
end
else
begin
case(state)
s1: if(flag)
begin
state <= s2;
sm_out <= 1'b1;
end
else
begin
state <= s3;
sm_out <= 1'b0;
end
s2: begin state <= s4; sm_out <= 1'b0; end
s3: begin state <= s4; sm_out <= 1'b0; end
s4: begin state <= s5; sm_out <= 1'b1; end
s5: begin state <= s1; sm_out <= 1'b1; end
endcase
end
end
endmodule
```

FSM Example with Single Sequential Block (VHDL)

Filename: fsm_1.vhd

```
-- State Machine with single sequential block
-- File: fsm_1.vhd
library IEEE;
use IEEE.std_logic_1164.all;

entity fsm_1 is
port(
clk, reset, flag : IN std_logic;
sm_out : OUT std_logic
);
end entity;

architecture behavioral of fsm_1 is
type state_type is (s1, s2, s3, s4, s5);
signal state : state_type;
begin
process(clk)
begin
if rising_edge(clk) then
if (reset = '1') then
state <= s1;
sm_out <= '1';

else
case state is
when s1 => if flag = '1' then
state <= s2;
sm_out <= '1';

else
state <= s3;
sm_out <= '0';

end if;
when s2 => state <= s4;
sm_out <= '0';
when s3 => state <= s4;
```

```
sm_out <= '0';
when s4 => state <= s5;
sm_out <= '1';
when s5 => state <= s1;
sm_out <= '1';
end case;
end if;
end if;
end process;

end behavioral;
```

FSM Reporting

The Vivado synthesis flags INFO messages in the log file, giving information about Finite State Machine (FSM) components and their encoding. The following are example messages:

```
INFO: [Synth 8-802] inferred FSM for state register 'state_reg' in module
'fsm_test'
INFO: [Synth 8-3354] encoded FSM with state register 'state_reg' using
encoding 'sequential' in module 'fsm_test'
```

ROM HDL Coding Techniques

Read-only memory (ROM) closely resembles random access memory (RAM) with respect to HDL modeling and implementation. Use the `ROM_STYLE` attribute to implement a properly-registered ROM on block RAM resources. See ROM_STYLE for more information.

ROM Using Block RAM Resources (Verilog)

Filename: rams_sp_rom_1.v

```
// ROMs Using Block RAM Resources.
// File: rams_sp_rom_1.v
//
module rams_sp_rom_1 (clk, en, addr, dout);
input clk;
input en;
input [5:0] addr;
output [19:0] dout;

(*rom_style = "block" *) reg [19:0] data;

always @(posedge clk)
begin
if (en)
case(addr)
6'b000000: data <= 20'h0200A; 6'b100000: data <= 20'h02222;
6'b000001: data <= 20'h00300; 6'b100001: data <= 20'h04001;
6'b000010: data <= 20'h08101; 6'b100010: data <= 20'h00342;
6'b000011: data <= 20'h04000; 6'b100011: data <= 20'h0232B;
6'b000100: data <= 20'h08601; 6'b100100: data <= 20'h00900;
6'b000101: data <= 20'h0233A; 6'b100101: data <= 20'h00302;
6'b000110: data <= 20'h00300; 6'b100110: data <= 20'h00102;
6'b000111: data <= 20'h08602; 6'b100111: data <= 20'h04002;
6'b001000: data <= 20'h02310; 6'b101000: data <= 20'h00900;
6'b001001: data <= 20'h0203B; 6'b101001: data <= 20'h08201;
6'b001010: data <= 20'h08300; 6'b101010: data <= 20'h02023;
6'b001011: data <= 20'h04002; 6'b101011: data <= 20'h00303;
6'b001100: data <= 20'h08201; 6'b101100: data <= 20'h02433;
6'b001101: data <= 20'h00500; 6'b101101: data <= 20'h00301;
6'b001110: data <= 20'h04001; 6'b101110: data <= 20'h04004;
6'b001111: data <= 20'h02500; 6'b101111: data <= 20'h00301;
6'b010000: data <= 20'h00340; 6'b110000: data <= 20'h00102;
6'b010001: data <= 20'h00241; 6'b110001: data <= 20'h02137;
6'b010010: data <= 20'h04002; 6'b110010: data <= 20'h02036;
6'b010011: data <= 20'h08300; 6'b110011: data <= 20'h00301;
6'b010100: data <= 20'h08201; 6'b110100: data <= 20'h00102;
6'b010101: data <= 20'h00500; 6'b110101: data <= 20'h02237;
6'b010110: data <= 20'h08101; 6'b110110: data <= 20'h04004;
6'b010111: data <= 20'h00602; 6'b110111: data <= 20'h00304;
6'b011000: data <= 20'h04003; 6'b111000: data <= 20'h04040;
6'b011001: data <= 20'h0241E; 6'b111001: data <= 20'h02500;
6'b011010: data <= 20'h00301; 6'b111010: data <= 20'h02500;
6'b011011: data <= 20'h00102; 6'b111011: data <= 20'h02500;
6'b011100: data <= 20'h02122; 6'b111100: data <= 20'h0030D;
6'b011101: data <= 20'h02021; 6'b111101: data <= 20'h02341;
6'b011110: data <= 20'h00301; 6'b111110: data <= 20'h08201;
6'b011111: data <= 20'h00102; 6'b111111: data <= 20'h0400D;
endcase
```

```
end

assign dout = data;

endmodule
```

ROM Inference on an Array (VHDL)

Filename: roms_1.vhd

```
-- ROM Inference on array
-- File: roms_1.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity roms_1 is
port(
clk : in std_logic;
en : in std_logic;
addr : in std_logic_vector(5 downto 0);
data : out std_logic_vector(19 downto 0)
);
end roms_1;

architecture behavioral of roms_1 is
type rom_type is array (63 downto 0) of std_logic_vector(19 downto 0);
signal ROM : rom_type := (X"0200A", X"00300", X"08101", X"04000",
X"08601", X"0233A",
X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
X"08201", X"00500", X"04001", X"02500", X"00340", X"00241", X"04002",
X"08300", X"08201", X"00500" X"08101", X"00602", X"04003", X"0241E",
X"00301", X"00102", X"02122", X"02021", X"00301", X"00102", X"02222",
X"04001", X"00342", X"0232B", X"00900", X"00302", X"00102", X"04002",
X"00900", X"08201", X"02023", X"00303", X"02433", X"00301", X"04004"
X"00301",X"00102", X"02137", X"02036", X"00301", X"00102",
X"02237",X"04004", X"00304", X"04040", X"02500", X"02500",
X"02500",X"0030D", X"02341", X"08201", X"0400D");
attribute rom_style : string;
attribute rom_style of ROM : signal is "block";
```

```
begin
process(clk)
begin
if rising_edge(clk) then
if (en = '1') then
data <= ROM(conv_integer(addr));
end if;
end if;
end process;

end behavioral;
```