# Efinity® Synthesis User Guide

**UG-EFN-SYNTH-v3.8**
**December 2023**
**www.efinixinc.com**

# Contents

# Introduction

The Efinity® software is a complete tool for creating RTL designs. The first stage after you complete your RTL design is synthesis. During synthesis, the compiler takes your design and turns it into a gate-level netlist.

The software supports the synthesizable subset of the following languages:

- SystemVerilog and Verilog HDL
- VHDL
- Mixed languages (any combination of the above)

## SystemVerilog and Verilog HDL Support

The Efinity® software supports the complete IEEE-1800 standard (2017, 2012, 2009, 2005) and includes regular Verilog (IEEE 1164).

- Supports full SystemVerilog IEEE 1800
- Supports Verilog 2001 and Verilog 1995
- Provides 100% language coverage for analysis
- Supports mixed-language with VHDL

## VHDL Support

The Efinity® software supports supports the complete IEEE-1076 standard (2008, 1993, 1987) for analysis.

- Supports all of VHDL IEEE 1076
- Includes specialized packages for mixed -1993 / -2008 support
- Provides 100% language coverage for analysis
- Supports mixed-language with SystemVerilog and Verilog HDL
- Supports VHDL libraries (Efinity® software v2020.2 and higher)

## Specifying Language Support

You can set the language support at the project level or at the file level. In both cases, you make this setting in the **Design** tab of the Project Editor dialog box. Open the Project Editor by choosing **File > Edit Project** or by clicking the toolbar button.

| Verilog HDL Choices | VHDL Choices |
|---|---|
| verilog_2k | vhdl_2008 |
| verilog_95 | vhdl_1993 |
| SystemVerilog2005 | |
| SystemVerilog2009 | |

# Synthesis Project Settings

You set project-specific synthesis options in the **Project Editor > Synthesis tab**.

*Table 1: Synthesis Project Settings*

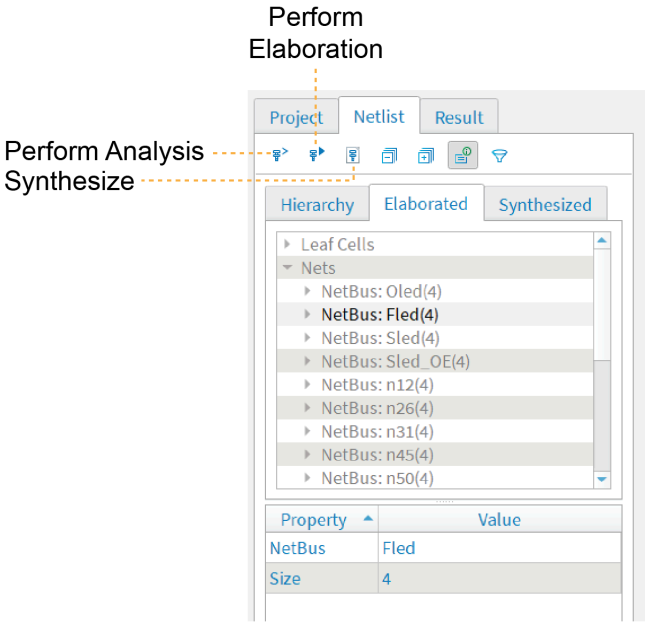
| Setting | Description |
|---|---|
| **Work Directory** | Specify a custom directory or use the default (**work_syn**). |
| **Generate post synthesis netlist** | Choose whether the software should create this netlist.<br>Default: On |
| **Synthesis Options** | See Synthesis Options on page 21. |
| **Include Dir** | Specify directories to include in your project. If you use the IP Manager to add IP, the **ip/**<*module*> directory is listed here. The software searches these locations when you use include statements. |
| **Dynamic Parameter** | Use this area to add parameters and values that apply to the top-level module or entity in your project. The value passed into the Dynamic Parameter field must be the same format as that you would use for any variable in VHDL or Verilog HDL. For example, string should be in quotation marks. |
| **Verilog `define Macro** | Use this area to add `define macros to your project.<br>Some FPGA EDA tools automatically create a SYNTHESIS macro. If you want to use the same behavior in the Efinity software, you need to create it here. For example, click the Add Verilog `define Macro button and then enter `SYNTHESIS` in the **NAME** field and `1` in the **Value** field. Then if you want to include simulation only code, use this format:<br>`` `ifndef SYNTHESIS<br>$display(...)<br>... some other simulation directives ...<br>`endif ``<br>You can also use the translate_on and translate_off directives to accomplish similar functionality. |

# Netlist Pane

The Netlist pane, which is under the Dashboard, shows the design hierarchy and helps you browse through the elaborated design and synthesized netlist. You can only view the synthesized netlist after you have performed synthesis. You can right-click the items in the Netlist pane to open a context-sensitive menu with shortcut actions.

**Tip:** In Efinity® v2020.2 and higher you can resize the Netlist pane. Grab the blank space between the Netlist pane and the Console and drag to resize.

*Figure 1: Using the Netlist Pane*



# Design Guidelines

The following sections provide some general design guidelines.

## DSP

Titanium FPGAs have DSP Blocks that support arithmetic functions such as multiplication, addition, subtraction, accumulation, and 4-bit variable right shifting. The full functionality of the DSP Block is represented with the EFX_DSP48 primitive. Additionally, the DSP Block supports a concept of *fracturing*, in which the software packs two or more multiplications into a single DSP Block. These fractured blocks are represented with the EFX_DSP24 and EFX_DSP12 primitives.

**Note:** Refer to the Quantum® Titanium Primitives User Guide for information on the DSP Block primitives.

### Inferring DSP

The Titanium DSP Block supports multiply, add, shift, and cascade functions. The following examples show code to infer them.

*Figure 2: Inferring Multipliers*

When inferring multipliers, synthesis uses different primitives, depending on the width:

- < = 4 infers an EFX_DSP12
- < = 8 infers an EFX_DSP24
- < = 19,18, infers an EFX_DSP48

```
`define AWIDTH 4
`define BWIDTH 4
module mult(a, b, x);
   input signed [`AWIDTH-1:0] a;
   input signed [`BWIDTH-1:0] b;
   output signed [`AWIDTH+`BWIDTH-1:0] x;

   assign x = a * b;
endmodule
```

*Figure 3: Inferring Multiply-Accumulate*

This multiply-add can be packed into a single EFX_DSP48 primitive because the width of C fits into tbe C port.

```
module dsp_multadd_s(a, b, c, o);
   input signed [17:0] a;
   input signed [17:0] b;
   input signed [17:0] c;

   output signed [35:0] o;

   wire signed [35:0] p;

   assign p = a * b;
   assign o = p + c;

endmodule
```

*Figure 4: Inferring Multiply-Accumulate with Cascading*

Generally, a multiply-add cannot be packed into a single EFX_DSP48 primitive because the width of C does not fit into the C port. With the **syn_use_dsp** attribute, synthesis uses the A:B:C cascade of another DSP Block to produce a multiply-accumulate.

```
module dsp_multadd_s(a, b, c, o);
    input signed [17:0] a;
    input signed [17:0] b;
    input signed [35:0] c;

    output signed [35:0] o;

    wire signed [35:0] p;
    (* syn_use_dsp = "yes" *) wire [35:0] sum;

    assign p = a * b;
    assign sum = p + c;
    assign o = sum;

endmodule
```

*Figure 5: Inferring Multiply-Accumulate with Output Feeding Back to Accumulate*

The feedback path can take output of the DSP Block and feed it back for accumulation. You enable it with the **syn_use_dsp** attribute.

```
module dsp_multadd_s(a, b, clk, o);
    input signed [17:0] a;
    input signed [17:0] b;
    input clk;

    output signed [35:0] o;

    wire signed [35:0] p;
    (* syn_use_dsp = "yes" *) reg [35:0] sum;

    assign p = a * b;
    always @(posedge clk) begin
      sum <= p + sum;
    end
    assign o = sum;

endmodule
```

## Using the DSP Block Effectively

Fracturing lets the synthesis tool optimize how it packs operations into the DSP block. The packing density is the percentage of DSP Blocks that are fully occupied with EFX_DSP24 and/or EFX_DSP12 primitives. Theoretically, two EFX_DSP24 primitives or four EFX_DSP12 primitives pack into one EFX_DSP48. In practice, there are some restrictions on how well they can pack: *legality* constraints and *packing quality* constraints.

### Legality Constraints

Synthesis can only pack EFX_DSP24 and EFX_DSP12 primitives into the same EFX_DSP48 if they are the same except for the datapath inputs/outputs. Generally:

- Clock, CE, RST, SHIFT_ENA, and OP inputs to the EFX_DSP24 and EFX_DSP12 to be packed together must be identical nets. You can use VCC, GND, or disconnected for these ports, as long as they match.
- All DSP primitive Verilog HDL parameters must match, including registered ports (A_REG, B_REG, W_REG) and the mode.

If you are not happy with the DSP Block packing density, you need to modify your design to allow them to pack more effectively.

### Quality Constraints

The software avoids packing random DSP Blocks together because it can have a negative impact on $f_{MAX}$. Instead, it tries to pack DSP blocks that are logically related—for example, ones that share neighbouring blocks and input nets—resulting in the best clustering that fits on the FPGA comfortably.

### Improving Packing Density

If you want better DSP Block packing, there are some things you can try:

- Tweak the design such that more DSP blocks have identical control signals and mode settings, and are thus packable. For example, try to avoid letting synthesis infer unique clock enables for every EFX_DSP24 and/or EFX_DSP12 primitive.
- Setting the synthesis options --dspinout-regs-packing, --dsp-output-regs-packing, and --dsp-mac-packing to 0 may improve packing density at a cost of increased flipflop and adder consumption. You set these options in the **Project Editor > Synthesis tab**.

You can also use these reports to help with debugging:

- **place.rpt**—This report includes a DSP packing summary that shows the number of DSP Blocks that would be packable if synthesis ignored control signal and attribute-related legality constraints. This number helps you understand whether tweaking the design to improve the packing density is even feasible. In practice, you can get about 50% packing improvement by tweaking the design. The report also lists the control set and attributes for each DSP Block. To be packed, EFX_DSP24 and EFX_DSP12 primitives must have matching control sets and attributes. Look for DSP Blocks that do not share control sets or attributes with other blocks, and then look at dsp_control_sets.csv for more detailed information on how to potentially adjust them.
- **dsp_control_sets.csv**—This file is a table in **.csv** format that lists every control signal and attribute for every DSP Block. You can use a spreadsheet application to review the data to identify EFX_DSP24 and EFX_DSP12 primitives that are inferred with unique settings that make packing illegal.

## *Closing Timing with High DSP Block Utilization*

If your design has a >50% of the DSP Blocks implemented with EFX_DSP24 or EFX_DSP12 primitives, the $f_{MAX}$ can vary significantly depending on the placement seed. Therefore, it is a good idea to try 3 or 4 seeds to see if it helps with timing closure, more so than for a typical design.

**Learn more:** Refer to the **Efinity Timing Closure User Guide** for information on how to perform seed sweeping.

# Flip-Flops

The Efinity® synthesis tool recognizes flip-flops (or registers) while processing the RTL. Flip-flops can have these control signals:

- Rising or falling edge clocks
- Asynchronous set/reset
- Synchronous set/reset
- Clock enable

*Figure 6: EFX_FF Symbol*

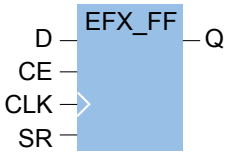



*Table 2: EFX_FF Ports*

| Port | Direction | Description |
|---|---|---|
| D | Input | Input data. |
| CE | Input | Clock Enable. |
| CLK | Input | Clock. |
| SR | Input | Asynchronous/synchronous set/reset. |
| Q | Output | Output data. |

Flip-flops with active-high synchronous resets and active-high clock enables are described as sequential processes in VHDL.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity D_FF_VHDL is
  port (clk, rst, ce, d  : in std_logic; q : out std_logic );
end entity D_FF_VHDL;

architecture Behavioral of D_FF_VHDL is
begin
  process (clk) is
  begin
     if rising_edge(clk) then
        if (rst='1') then
           q <= '0';
        elsif (ce='1') then
           q <= d;
        end if;
     end if;
  end process;
end architecture Behavioral;
```

They are described with an always statement in Verilog HDL.

```
module D_FF_VERILOG (d, ce, clk, reset, out);
  input d;
  input ce;
  input clk;
  input reset;
  output out;
  reg q;

  always @(posedge clk) begin
    if (reset) q <= 1'b0;
    else if (ce) q <= d;
  end
  assign out = q;
endmodule
```

### Flip-Flop Reporting

The synthesis report (**map.rpt**) shows flip-flop utilization and optimization. For example:

```
Equivalent flip-flop removal reporting:

FF|OPT : Flip-flop optimization by equivalence checking

@ "./zipcpu/idecode.v (350)" removed instance : thecpu/instruction_decoder/dff_195/i7
@ "./zipcpu/idecode.v (350)" representative instance : thecpu/instruction_decoder/dff_196/i7

Clock Enable reporting:

Total number of enable signals: 1199
Enable signal <vcc>, number of controlling flip flops: 256
Enable signal <o_dbg_stall>, number of controlling flip flops: 35

Flip-flop resource summary:

### ### ### Resource Summary (begin) ### ### ###

EFX_FF : 35132
```

### Flip-Flop Guidelines

For best optimization during synthesis, follow these guidelines:

- Avoid using flip-flops with asynchronous set/reset. These structures limit the synthesis tool's ability to optimize the code.
- Avoid flip-flops with both a set and a reset signal. The Trion® and Titanium flip-flop only has a single set/reset signal. Therefore, to construct a register with both a set and reset signal, the synthesis must infer additional control logic.

## Latches

If you do not assign an output for all possible conditions in an `if` or `case` statement (that is, incomplete assignment), the software infers a latch. Trion® and Titanium FPGAs do not support latches natively in hardware. The Efinity® synthesis tool infers look-up tables (LUTs) to provide latch behaviour.

Because latches are turned into LUTs, they can use up resources you could be using for something else. So you want to avoid them when possible.

# RAM

Efinix® FPGAs have embedded RAM blocks that support simple dual-port memory and true dual-port memory. The read and write ports are registered. Asynchronous memory reads (e.g. in asynchronous FIFO or buffer implementation) can be bit-blasted into logic but may cause high device resource utilization. If you do not use the write port, the primitive acts as a ROM.

- *Trion FPGAs*—During synthesis, the memory is mapped to EFX_RAM_5K (simple dual port) or EFX_DPRAM_5K (true dual port) primitives.
- *Titanium FPGAs*—During synthesis, the memory is mapped to EFX_RAM10 (simple dual port) or EFX_DPRAM10 (true dual port) primitives.

The following sections provide code example for inferring these memories.

**Learn more:** Refer to the **Quantum® Trion Primitives User Guide** for detailed information on the EFX_RAM_5K and EFX_DPRAM_5K primitives.

Refer to the **Quantum® Titanium Primitives User Guide** for information on the EFX_RAM10 and EFX_DPRAM10 primitives.

## *Inferring RAM*

The following sections provide example for simple and true dual-port inferencing.

### Simple Dual-Port Memory Examples

The following example infers a 512 x 8 RAM. Because both writes and reads are performed with a blocking statement and the write occurs before the read in the `always` block, the software infers a simple dual ported RAM in WRITE_FIRST mode.

*Figure 7: Simple Dual-Port RAM in WRITE_FIRST Mode*

```
module ram512x8_sp(wdata, addr, clk, we, rdata);
   parameter AWIDTH = 9;
   parameter DWIDTH = 8;
   localparam DEPTH = 1 << AWIDTH;
   localparam MAX_DATA = (1<<DWIDTH)-1;
   input [DWIDTH-1:0] wdata;
   input [AWIDTH-1:0] addr;
   input  clk, we;
   output reg [DWIDTH-1:0] rdata;

   reg [DWIDTH-1:0] mem [DEPTH-1:0];

   // Blocking Statement and order indicates write before read
   always@(posedge clk) begin
      if (we) begin
          mem[addr] = wdata;
      end
      rdata = mem[addr];
   end
endmodule
```

If the read and write clocks are different, the software configures the memory primitive as READ_UNKNOWN. That is, if you read and write to the same address at the same time, the read data is indeterministic.

*Figure 8: Simple Dual-Port RAM in READ_UNKNOWN Mode*

```
module ram512x8_sp(wdata, addr, rclk, re, wclk, we, rdata);
   parameter AWIDTH = 9;
   parameter DWIDTH = 8;
   localparam DEPTH = 1 << AWIDTH;
   localparam MAX_DATA = (1<<DWIDTH)-1;
   input [DWIDTH-1:0] wdata;
   input [AWIDTH-1:0] addr;
   input      rclk, re, wclk, we;
   output reg [DWIDTH-1:0] rdata;

   reg [DWIDTH-1:0]  mem [DEPTH-1:0];

   // different read and write clock, forces READ_UNKNOWN mode
   always@(posedge wclk) begin
      if (we) begin
          mem[addr] = wdata;
      end
   end
   always@(posedge rclk) begin
      if (re) begin
          rdata = mem[addr];
      end
   end
endmodule
```

*Figure 9: Simple Dual-Port RAM with Byte Enable (Verilog HDL) (Titanium)*

```
// 16-bit wide, 512 depth, byte-enabled
// fits into 1 Titanium 10K blockram
module ram10_be1 #(
        parameter integer wrAddressWidth = 9,   // 512 depth
        parameter integer wrDataWidth = 16,     // 16-bit wide
        parameter integer wrMaskWidth = 2,
        parameter integer rdAddressWidth = 9,
        parameter integer rdDataWidth  = 16
    )(
        input wr_clk,
        input wr_en,
        input [wrMaskWidth-1:0] wr_mask,
        input [wrAddressWidth-1:0] wr_addr,
        input [wrDataWidth-1:0] wr_data,
        input rd_clk,
        input rd_en,
        input [rdAddressWidth-1:0] rd_addr,
        output [rdDataWidth-1:0] rd_data
    );

    reg [wrDataWidth-1:0] ram_block [(2**wrAddressWidth)-1:0];
    integer i;
    localparam COL_WIDTH = wrDataWidth/wrMaskWidth;
    always @ (posedge wr_clk) begin
        if(wr_en) begin
            for(i=0;i<wrMaskWidth;i=i+1) begin
                if(wr_mask[i]) begin    // byte-enable
                    ram_block[wr_addr][i*COL_WIDTH +: COL_WIDTH] <=
 wr_data[i*COL_WIDTH +:COL_WIDTH];
                end
            end
        end
    end

    reg [rdDataWidth-1:0] ram_rd_data;
    always @ (posedge rd_clk) begin
        if(rd_en) begin
            ram_rd_data <= ram_block[rd_addr];
        end
    end
    assign rd_data = ram_rd_data;
endmodule
```

*Figure 10: Simple Dual-Port RAM with Byte Enable (VHDL) (Titanium)*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- 512 x 32, with 4 byte-enable signals

entity Memory is
    generic (ADDR_WIDTH: integer := 9);
    Port ( DBOut : out  STD_LOGIC_VECTOR (31 downto 0);
           DBIn : in  STD_LOGIC_VECTOR (31 downto 0);
           AdrBus : in  STD_LOGIC_VECTOR (ADDR_WIDTH-1 downto 0);
           ENA : in  STD_LOGIC;
           WREN : in  STD_LOGIC_VECTOR (3 downto 0);
           CLK : in  STD_LOGIC
         );
end Memory;

architecture Behavioral of Memory is

constant SIZE : natural := 2**ADDR_WIDTH;
type tRam is array (0 to SIZE-1) of STD_LOGIC_VECTOR (31 downto 0);
subtype tWord is std_logic_vector(31 downto 0);

signal ram : tRam;
signal DOA,DIA : tWord;
signal WEA : STD_LOGIC_VECTOR (3 downto 0);

begin

  DIA<=DBIn;
  DBOut<=DOA;
  WEA<=WREN;

  process(clk)
  variable adr : integer;
  begin
    if rising_edge(clk) then
        if ena = '1' then
           adr :=  to_integer(unsigned(AdrBus));

           for i in 0 to 3 loop
             if WEA(i) = '1' then
                ram(adr)((i+1)*8-1 downto i*8)<= DIA((i+1)*8-1 downto i*8);
             end if;
           end loop;

           DOA <= ram(adr);

         end if;
     end if;

  end process;

end Behavioral;
```

## True Dual-Port Memory Examples

In true dual-port RAM, the two ports have independent read and write functions. Each port supports different write modes. The following example shows how to implement a 512 x 8 RAM block with READ_FIRST write mode for port A and WRITE_FIRST mode for port B.

```
module ram512x8_tdp_mix (wdataA, addrA, clkA, weA, rdataA, wdataB, addrB, clkB, weB, rdataB);
   parameter AWIDTH = 9;
   parameter DWIDTH = 8;
   localparam DEPTH = 1 << AWIDTH;
   localparam MAX_DATA = (1<<DWIDTH)-1;

   input [DWIDTH-1:0] wdataA, wdataB;
   input [AWIDTH-1:0] addrA, addrB;
   input    clkA, weA;
   input    clkB, weB;
   output reg [DWIDTH-1:0] rdataA, rdataB;

   reg [DWIDTH-1:0]   mem [DEPTH-1:0];

   integer   i;
   initial begin
      // The memory is initialized with
      // decreasing values startingfrom MAX_DATA
      for (i=0;i<DEPTH;i=i+1)
         mem[i] = MAX_DATA - i;
   end

   always@(posedge clkA) begin
      // Use blocking assignments to for read-first
      rdataA = mem[addrA];
      if (weA) begin
         mem[addrA] = wdataA;
      end
   end

   always@(posedge clkB) begin
      // Use blocking assignments to force write-first
      if (weB) begin
         mem[addrB] = wdataB;
      end
      rdataB = mem[addrB];
   end
endmodule
```

## Initializing RAM

Initialize the memory content with the Verilog HDL `$readmemh` or `$readmemb` routines.

*Figure 11: Initializing RAM in Verilog HDL*

```
module ram_256x16 (wdata,  waddr, wclk, we, raddr, rclk, re, rdata);
   localparam addr_width = 8;
   localparam data_width = 16;
   input [data_width-1:0] wdata;
   input [addr_width-1:0] waddr, raddr;
   input    wclk, we;
   input    rclk, re;
   output reg [data_width-1:0] rdata;

   reg [data_width-1:0]   mem [(1<<addr_width)-1:0];

   integer   i;
   initial begin
      // Initialize memory with external file
      $readmemh("ram256x16b.inithex", mem);
   end

   always@(posedge wclk) begin
       if (we)
     mem[waddr] <= wdata;
       end
   always@(posedge rclk) begin
       if (re)
           rdata <= mem[raddr];
       end

endmodule // ram_256x16
```

The memory file should be simple hexadecimal numbers (`$readmemh`) or binary numbers (`$readmemb`) without any comments or prefixes.

*Figure 12: Example Memory File*

```
FE
FD
FC
FB
FA
F9
F8
F7
F6
F5
...
```

## Inferring Output Registers

Synthesis packs registers that immediately follow the read data into the output registers of the BRAM if the control logic is compatible:

- The read clock is the same as the register clock signal.
- Enables:
  - Trion FPGAs—The register must always be enabled (no explicit clock enable control).
  - Titanium FPGAs—The register's clock enable must be the same as the BRAM's read enable signal.
- Resets:
  - Trion FPGAs—The register cannot have a reset signal.
  - Titanium FPGAs—If the read port has a reset signal, and if the register has a reset signal, they must match. Additionally, the Titanium output register only supports asynchronous reset logic.

## Address Enable (Titanium)

The Titanium BRAM supports an address enable feature. However, synthesis does not infer these signals;for inferred BRAM these signals are always tied high. To use the address enable, instantiate the primitive (EFX_RAM10 and EFX_DPRAM10),

## Resetting RAM (Titanium)

Titanium RAM supports a reset option on the RAM output and output register.

*Figure 13: RAM Output with Asynchronous Reset*

```
module ram10_arst (wdata, waddr, wclk, wclke, rclk, we, re, raddr, rdata,
 rst);
   parameter AWIDTH = 11;   // 2048 depth
   parameter DWIDTH = 4;    // 4-bit wide
   localparam DEPTH = 1 << AWIDTH;
   input [DWIDTH-1:0] wdata;
   input [AWIDTH-1:0] waddr, raddr;
   input              wclk, wclke, we, rclk, re, rst;
   output reg [DWIDTH-1:0] rdata;

   reg [DWIDTH-1:0]        mem [DEPTH-1:0];

   always@(posedge wclk) begin
      if (wclke) begin
        if (we)
            mem[waddr] <= wdata;
      end
   end
   always@(posedge rclk or posedge rst) begin
      if (rst)
        rdata <= 0;
      else if (re)
        rdata <= mem[raddr];
    end
endmodule
```

## Estimating Block RAM Resources

The Efinity® software v2020.2 (patch 2020.2.299.2.6) and higher includes a Block RAM Resource Estimator that helps you determine how many block RAM resources the software needs for a given memory size. You run this tool at the command line using the `efx_map_ramest` command. The estimator uses these options:

*Table 3: Block RAM Resource Estimator Options*

| Option | Description |
|---|---|
| --help | Display the help. |
| --family <*family name*> | Specify the family:<br>Quantum: for Quantum® cores.<br>Trion: for Trion FPGAs.<br>Titanium: for Titanium FPGAs. |
| --device <*FPGA name*> | Optional. Specify the FPGA you are targeting. |
| --mode <*mode*> | Specify the decomposition mode, speed, area, or power. |
| --size <*memory size*> | Specify the memory size as <*depth*>x<*width*>. |
| --size2 <*memory size*> | If using true dual-port, specify the second port's memory size as <*depth*>x<*depth*>. |

The following code examples show how to estimate RAM for Trion and Titanium RAM blocks of varying sizes.

### Simple Dual-Port RAM Example (Trion)

The following example command runs the estimator for a Trion FPGA, 10240 X 16 RAM size, and optimizing for area:

```
efx_map_ramest --family Trion --mode area --size 10240x16
```

The command outputs:

```
Efinix Block Ram Resource Estimator
Version: 2020.2.299
Compiled: Dec 30 2020.

Copyright (C) 2013 - 2020 Efinix Inc. All rights reserved.

FPGA Family           : Trion
Block Ram Size        : 5K
Input Memory Size     : 10240x16
Mode                  : area

Result                : 33 block rams required to implement the above memory size.
```

### True Dual-Port RAM Example (Trion)

The following example command runs the estimator for a Trion FPGA, 10240 X 16 RAM size, 5120 x 32 RAM size, and optimizing for power:

```
efx_map_ramest --family Trion --mode power --size 10240x16 --size2 5120x32
```

The command outputs:

```
Efinix Block Ram Resource Estimator
Version: 2020.2.299
Compiled: Dec 30 2020.

Copyright (C) 2013 - 2020 Efinix Inc. All rights reserved.

FPGA Family           : Trion
Block Ram Size        : 5K
Input Memory Size     : 10240x16
2nd Memory Port Size  : 5120x32
Mode                  : power

Result                : 33 block rams required to implement the above memory size.
```

### Simple Dual-Port RAM Example (Titanium)

The following example command runs the estimator for a Titanium FPGA, 10240 X 16 RAM size, and optimizing for area:

```
efx_map_ramest --family Titanium --mode area --size 10240x16
```

The command outputs:

```
Efinix Block Ram Resource Estimator
Version: 2020.2.299
Compiled: Dec 30 2020.

Copyright (C) 2013 - 2020 Efinix Inc. All rights reserved.

FPGA Family           : Titanium
Block Ram Size        : 10K
Input Memory Size     : 10240x16
Mode                  : area

Result                : 17 block rams required to implement the above memory size.
```

### True Dual-Port RAM Example (Titanium)

The following example command runs the estimator for a Titanium FPGA, 10240 X 16 RAM size, 5120 x 32 RAM size, and optimizing for area:

```
efx_map_ramest --family Titanium --mode area --size 10240x16 --size2 5120x32
```

The command outputs:

```
Efinix Block Ram Resource Estimator
Version: 2020.2.299
Compiled: Dec 30 2020.

Copyright (C) 2013 - 2020 Efinix Inc. All rights reserved.

FPGA Family           : Titanium
Block Ram Size        : 10K
Input Memory Size     : 10240x16
2nd Memory Port Size  : 5120x32
Mode                  : area

Result                : 17 block rams required to implement the above memory size.
```

# Inferring Shift Registers

Efinity® synthesis can infer shift register functions that use the XLR cell's 8-bit shift register function. You do not need to set any synthesis options. For example, synthesis infers the following code as a simple shift register:

```
// 12-bit shift register example
module srl12 (CLK, SI, DO);
input CLK, SI;
output DO;
localparam DATAWIDTH = 12;
reg [DATAWIDTH-1:0] data;

// initializing the shift register content
initial begin
  data = 12'h800;
end

always @(posedge CLK)
  begin
    data <= {data[DATAWIDTH-2:0], data[DATAWIDTH-1]};
  end
assign DO = data[0];

endmodule
```

The shift register reset is constructed with extra flipflops.

The following example shows a shift register with a reset:

```
// 12-bit shift register with reset example
module srl12_rst (CLK, DI, RST, DO);
input CLK, DI, RST;
output DO;
localparam DATAWIDTH = 12;
reg [DATAWIDTH-1:0] data;
initial begin
    data = 12'h800;
end

always @(posedge CLK)
 begin
     if (RST)
        data <= 0;
     else
        data <= {data[DATAWIDTH-2:0], DI};
 end
assign DO = data[DATAWIDTH-1];

endmodule
```

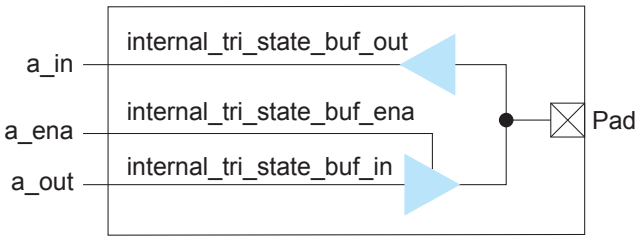
# Tri-State Buffers

Typically, you infer a tri-state buffer in your RTL code. For example:

```
module tri_state_buf_original (a);
    inout a;
    wire b;
    wire oe;
    assign a = (oe ? b : 1'bZ);
endmodule
```

In the Efinity® software, however, tri-state buffers are implemented as GPIO blocks in the Interface Designer. The following figure shows a tri-state buffer model in the Efinity® software. The Interface Designer promotes all of the internal signals of tri-state buffer to input and output ports of the RTL design (`a_ena`, `a_in`, and `a_out`).

*Figure 14: Tri-State Buffer*




To target Trion® and Titanium FPGAs, you map the inout port to three internal nodes (`a_ena`, `a_in`, and `a_out`), export all signals to the top-level module, and then assign the ports to GPIO. The following Verilog HDL code creates the tri-state buffer:

```
module tri_state_buf (a_in, a_out, a_ena,
          internal_tri_state_buf_in,
          internal_tri_state_buf_out,
          internal_tri_state_buf_ena);

// Split the initial inout port a to three wrapper ports of tri-state buffer
input a_in;
output a_out;
output a_ena;

// Act as the internal tri state buffer signals.
output internal_tri_state_buf_out;
input internal_tri_state_buf_in;
input internal_tri_state_buf_ena;

// Modification from traditional way of inferring

assign internal_tri_state_buf_out = a_in;
assign a_out = internal_tri_state_buf_in;
assign a_ena = internal_tri_state_buf_ena;

endmodule
```

**Download:** To download a code example, go to the **How do I create a Tri-State Buffer?** topic in the Support Center Knowledgebase.

# Synthesis Options

You can set project-wide synthesis options to control the design flow. You set these options in the Project Editor's **Synthesis** tab. Most options apply to all FPGA families; however, some are specific to Trion or Titanium FPGAs as shown in the following tables.

*Table 4: Synthesis Options (All Families)*

| Name | Choices | Description |
|---|---|---|
| --allow-const-ram-index | 0, 1 | Infer RAM if an array is accessed through constant indices. This option can be useful if memory is written such that a constant index refers to each segment (e.g., in a byte-enable read/write). See example.<br>0: Default. Do not infer.<br>1: Infer. |
| --blackbox-error | 0, 1 | Generate an error when synthesis encounters an undefined instance or entity.<br>0: No error.<br>1: Default. Generate error. |
| --blast_const_operand_adders | 0, 1 | If one of the operands to an arithmetic operation is constant, implement it as logic instead of adders.<br>0: Disable.<br>1: Default. Enable. |
| --bram_output_regs_packing | 0, 1 | Enables the software to pack registers into the output of BRAM.<br>0: Disable.<br>1: Default. Enable |
| --create-onehot-fsms | 0, 1 | Create onehot encoded state machine when appropriate. Synthesis can only create these state machines if the state variables do not have explicit encoding in the HDL. If a state machine is coded using onehot encoding, a new section in the map report (**<project>.map.rpt**) shows the encoding information. See example.<br>0: Default. Disabled.<br>1: Enabled. |
| --fanout-limit | 0 to *n* | If something is high fanout, the tool duplicates the fanout source.<br>0: Default. Disable.<br>*n*: Indicate the fanout limit at which to begin duplication. |
| --hdl-compile-unit | 0, 1 | When considering multiple source files, resolve `define or parameters independently or across all files. This option only works with SystemVerilog files.<br>0: Across all files.<br>1: Default. Independently. |
| --infer-clk-enable | 0, 1, 2, 3, 4 | Infer flip-flop clock enables from control logic. See examples.<br>0: disable.<br>1, 2, 3, 4: Effort levels. |

| Name | Choices | Description |
|---|---|---|
| --infer-sync-set-reset | 0, 1 | Infer synchronous set/reset signals.<br>0: Disable.<br>1: Default. Enable. |
| --max_ram | -1, 0, *n* | -1: Default. There is no limit to the number of RAM blocks to infer.<br>0: Disable.<br>*n*: Any integer. |
| --max_mult | -1, 0, *n* | -1: Default. There is no limit to the number of multipliers to infer.<br>0: Disable.<br>*n*: Any integer. |
| --min-sr-fanout | 0, *n* | Infer the flipflop's synchronous set/reset signal from control logic if the set/reset signal fanout is greater than *n*. This option is useful if the design has a lot of small fanout set/reset signals that may create routing congestion.<br>0: Default. Disable.<br>*n*: Signal fanout. |
| --min-ce-fanout | 0, *n* | Infer the flipflops clock enable from c ontrol logic if the clock enable signal fanout is grester than *n*.<br>0: Default. Disable.<br>*n*: Signal fanout. |
| --mode | speed, area, area2 | speed: Default. Optimizes for fastest $f_{MAX}$.<br>area: Optimizes for smallest area.<br>area2: Uses techniques that help to optimize large multiplexer trees. |
| --mult-auto-pipeline | 0, 1, -1 | Performs automatic pipelining for wide multipliers to increase performance at the cost of extra latency. The software inserts pipeline registers at the output of partial multiplies and partial sums. For Titanium FPGAs, these pipeline registers are packed into the DSP48 as P and W registers. Additional registers are inserted at the input and output of the multiplier to balance latency issues caused by the insertion of the previous registers.<br>0: Default. Disabled.<br>1: Insert registers after partial multiplies creating 1 extra cycle of latency.<br>-1: Insert registers after partial multiplies and sums creating 2 or more extra cycles of latency depending on the width of the multiplier. For example, the software adds 4 latency cycles for a 32 x 32 multiplier. |
| --mult-decomp-retime | 0, 1 | Perform retiming after decomposition of a wide multiplier to improve performance.<br>0: Default. Disable.<br>1: Enable. |
| --operator-sharing | 0, 1 | Extract shared operators<br>0: Default. Disable<br>1: Enable |
| --optimize-adder-tree | 0, 1 | Optimize skewed adder trees<br>0: Default. Disable<br>1: Enable |

| Name | Choices | Description |
|---|---|---|
| --optimize-zero-init-rom | 0, 1 | Opitmize ROMs that are initialized to zero.<br>0: Disable<br>1: Default. Enable |
| --retiming | 0, 1 | Perform retiming optimization. Software moves registers to balance the combinational delay path.<br>0: Disable.<br>1: Default. Enable. |
| --seq_opt | 0, 1 | Turn on sequential optimization. This option can reduce LUT usage but may impact $f_{MAX}$.<br>0: Disable.<br>1: Default. Enable. |
| --seq-opt-sync-only | 0, 1 | Sequential synthesis only considers synchronous reset flipflops.<br>0: Default. Consider all flipflops.<br>1: Consider synchronous flipflops only. |
| --use-logic-for-small-mem | 0 to *n* | Set the size limit of small RAM blocks implemented in LEs. The number is the maximum number of LEs used.<br>0: Disable.<br>64: Default. |
| --use-logic-for-small-rom | 0 to *n* | Set the size limit of small ROM blocks implemented in LEs. The number is the maximum number of LEs used.<br>0: Disable.<br>64: Default. |

*Table 5: Synthesis Options (Titanium)*

| Name | Choices | Description |
|---|---|---|
| --dsp-input-regs-packing | 0, 1 | Allow packing of DSP input registers.<br>0: Disable.<br>1: Default. Enable. |
| --dsp-output-regs-packing | 0, 1 | Allow packing of DSP output registers.<br>0: Disable.<br>1: Default. Enable. |
| --dsp-mac-packing | 0, 1 | Allow multiplier packing, the software packs adder pairs using multipliers (Trion) or DSP Blocks.<br>0: Disable.<br>1: Default. Enable. |
| --insert-carry-skip | 0, 1 | Enable carry-skip optimization for long adders. This option can be useful for designs that have long carry chains. It implements the carry chain with carry skip instead of ripple carry, which can improve performance at the cost of increased area by splitting the carry chains into shorter ones.<br>0: Default. Disabled.<br>1: Enable. |
| --pack-luts-to-comb4 | 0, 1, 2 | Pack compatible LUTs into COMB4 primitives.<br>0: Default. Disable<br>1: Effort level 1<br>2: Effort level 2 |

*Table 6: Synthesis Options (Trion)*

| Name | Choices | Description |
|---|---|---|
| --mult-input-regs-packing | 0, 1 | Allow packing of multiplier input registers.<br>0: Disable.<br>1: Default. Enable. |
| --mult-output-regs-packing | 0, 1 | Allow packing of multiplier output registers.<br>0: Disable.<br>1: Default. Enable. |

# Example: --infer-clk-enable

The **--infer-clk-enable** synthesis option infers the flip-flop clock enable signal from control logic. This option has three effort levels, or you can disable it.

For example, if you choose effort level 1, this code:

```
always @(posedge clk) begin
  if (e1 | e2) q <= d;
end
```

infers `or (e1,e2)` as the CE pin of a flop with `d` in the D pin and `q` in the Q pin.

In a more complex case, this code:

```
always @(posedge clk) begin
  if (e1) q <= d1;
  else (e2) q <= d2;
end
```

does not result in an inferred clock enable.

With the effort level 3 option, which is the default, the synthesis tool traces multiplexer connections to look for a loop back to the Q pin of the flip-flop. If the tracing is successful, the software extracts the condition pins of the multiplexer path to form the clock enable signal. So in our complex example previously, the software infers `or (e1, e2)` inferred as the clock enable. The level 3 option reduces the number of LUTs by about 4-5% compared to the level 1 option.

# Example: --create-onehot-fsms Option

The following code snippet shows an example of the FSM encoding section in the **map.rpt** file.

```
### ### Finite State Machine Report (begin) ### ### ###
Module apb3_slave
-----------------
Recognized state machine 'busState' with states :
    "00"   :    IDLE
    "01"   :    SETUP
    "10"   :    ACCESS

Module axi4_slave
-----------------
Recognized state machine 'busState' with states :
    "000"   :    IDLE
    "001"   :    PRE_WR
    "100"   :    PRE_RD
    "010"   :    WR
    "011"   :    WR_RESP
    "101"   :    RD
```

# Example: --allow-const-ram-index

The following code snippet demonstrates code that can benefit from the **--allow-const-ram-index** option. Lines 13, 15, 23, and 25 use a constant index to address the first dimension of the memory:

```
subtype t_dim1 is std_logic_vector(7 downto 0);
type t_dim1_vector is array(natural range <>) of t_dim1;
subtype t_dim2 is t_dim1_vector(0 to 511);
type t_dim3_vector is array(natural range <>) of t_dim2;
subtype t_dim3 is t_dim3_vector(0 to 3);

signal RAM : t_dim3 := (others => (others => (others => '0')));
...
RAM3Proc_t : process(Clk)
begin
if(rising_edge(Clk)) then
  if(WriteEn = '1') then
    RAM(3)(to_integer(unsigned(Addr))) <= WriteData3_t;
  end if;
  ReadData3_t <= RAM(3)(to_integer(unsigned(Addr)));
end if;
end process RAM3Proc_t;

RAM2Proc_t : process(Clk)
begin
if(rising_edge(Clk)) then
  if(WriteEn = '1') then
    RAM(2)(to_integer(unsigned(Addr))) <= WriteData2_t;
  end if;
  ReadData2_t <= RAM(2)(to_integer(unsigned(Addr)));
end if;
end process RAM2Proc_t;

...
```

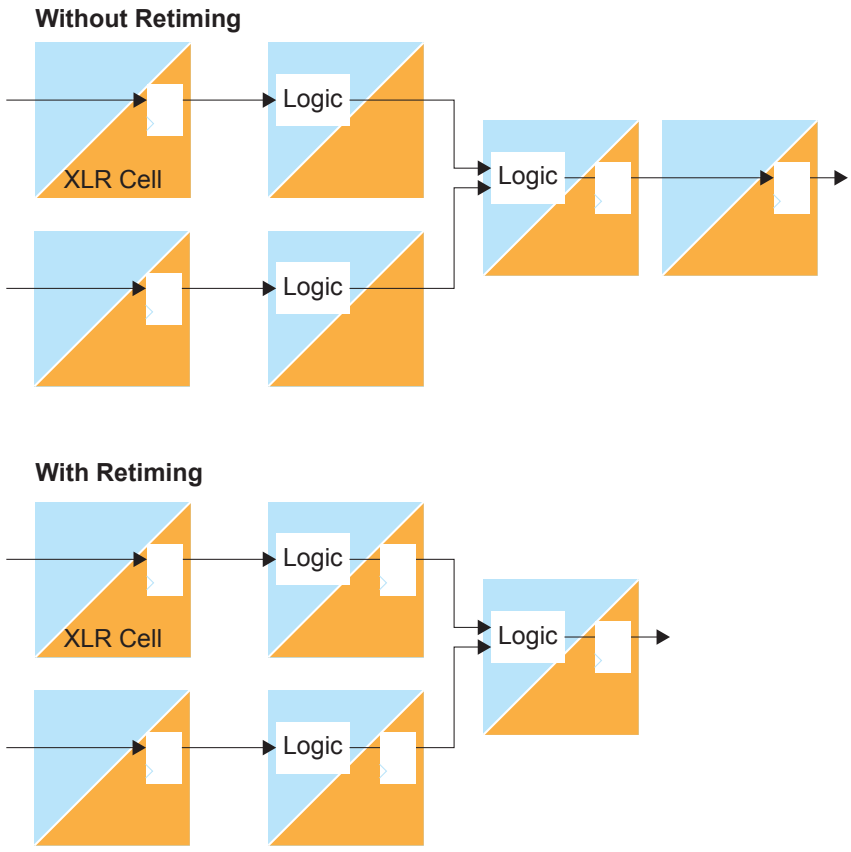
Setting the **--allow-const-ram-index** option to 1 (enable) instructs the synthesis tool to infer the code as a RAM block.

## Retiming

The Efinity® software includes an option for retiming. You enable it by setting the **--retiming** option to **1** in **Project Editor > Synthesis tab**.

When this option is turned on, the software moves registers forward or backward to improve the design's performance. Because the XLR cell comprises both logic and routing, the software can efficiently relocate registers with fine granularity.

*Figure 15: Retiming with XLR Cells*




# Synthesis Pragmas

The Efinity software supports these synthesis pragmas. Put the pragma in a comment, preceded by the `synthesis` keyword.

### synthesis on, synthesis off

This attribute directs synthesis to compile or skip a section of the RTL

### full_case

This attribute directs synthesis to interpret `case` statements as full_case (similar to the Synopsys `full_case` pragma). If you use a `full_case` pragma, synthesis assumes that the listed cases are the *only* possible conditions. All other input combinations are *don't care* and, therefore, synthesis does not generate logic for them

In the following example, the `full_case` pragma directs synthesis to treat the condition `sel = 2'b11` as don't care. The net effect is that the logic used to implement this case statement is simpler.

```
always @(a or b or c or sel) // synthesis full_case
  case (sel)
    2'b00: y = a;
    2'b01: y = b;
    2'b10: y = c;
  endcase
```

## parallel_case

This attribute directs synthesis to interpret `case` statements as parallel_case (similar to the Synopsys `parallel_case` pragma). If you use the `parallel_case` pragma, synthesis does *not* assume that the `case` conditions are mutually exclusive, that is, they can be true at the same time.

In the following example, if `sel = 3'b111`, all three of the conditions match, and `a`, `b`, and `c` are assigned to `1'b1`.

```
always @(sel)
   begin
      {a, b, c} = 3'b0;
      casez (sel)                  // synthesis parallel_case
         3'b1??: a = 1'b1;
         3'b?1?: b = 1'b1;
         3'b??1: c = 1'b1;
      endcase
   end
```

# Synthesis Attributes

You use synthesis attributes to guide the Efinity® software to perform (or not perform) certain actions during the synthesis step. The following sections describe the attributes the software supports.



## async_reg

This attribute applies to register outputs; it designates registers as synchronizers.

When `async_reg` is `true`, synthesis does not perform optimization to reduce, merge, or duplicate these registers. During place and route, the software keeps these registers close together to improve synchronization between asynchronous clock domains.

*Verilog HDL:*

```
(* async_reg = "true" *) reg [1:0] x;
```

*VHDL:*

```
attribute async_reg: boolean;
attribute async_reg of x : signal is true;
```

## syn_extract_enable

You use this signal attribute on register outputs. To use this attribute, set it to (`false`, `no`, or `0`). During synthesis, the software will not infer an active clock enable (except when there is a gated clock) on registers with this attribute set.

*Verilog HDL:*

```
(* syn_extract_enable="false" *) reg [3:0] cnt;
```

*VHDL:*

```
attribute syn_extract_enable: boolean;
attribute syn_extract_enable of cnt : signal is false;
```

## syn_keep

This attribute applies to signals or wires. It is similar to `syn_preserve` except it keeps more than just the signal itself. When it is set to `true`, `yes`, or `1`, the synthesis tool keeps the driver and the loads of the signal through optimization (synthesis does not minimize or remove them).

*Verilog HDL:*

```
(* syn_keep = "true" *) wire x;
```

*VHDL:*

```
attribute syn_keep: boolean;
attribute syn_keep of x : signal is true;
```

**Note:** A signal with `syn_keep` usually has it's name preserved through synthesis flow. However, if the signal is connected directly to a top-level port, the name in the **map.v** netlist may be changed to that of the top-level port name.

## syn_preserve

This attribute applies to signals. When it is set to `true`, `yes`, or `1`, synthesis keeps the signal through optimization, that is, synthesis does not minimize or remove the signal. This attribute can be helpful when you want to simulate or view a signal in the Debugger. Although the signal is kept, synthesis may still choose to implement downstream functions that depend on this signal independent of this preserved signal.

In the Efinity software v2022.2 and higher, the `syn_preserve` attribute is supported on a user hierarchy instance. The effect is equivalent to tagging all boundary signals of the instance with `syn_preserve`.

*Verilog HDL:*

```
(* syn_preserve = "true" *) wire x;
```

*VHDL:*

```
attribute syn_preserve: boolean;
attribute syn_preserve of x : signal is true;
```

**Note:** A signal with `syn_preserve` usually has it's name preserved through synthesis flow. However, if the signal is connected directly to a top-level port, the name in the **map.v** netlist may be changed to that of the top-level port name.

## syn_ramdecomp

This attribute applies to a RAM or ROM signal and controls how synthesis decomposes the RAM or ROM.

- When this attribute is not set, synthesis always chooses data-width decomposition for better performance.
- When set to `area`. synthesis decomposes the RAM or ROM for minimum area (least number of RAM block primitives), but it favors data-width decomposition.
- When set to `power`, synthesis decomposes the RAM or ROM for minimum area, but it favors address decomposition.

*Verilog HDL:*

```
(* syn_ramdecomp = "area" *) reg [DWIDTH-1:0]          mem [DEPTH-1:0];
(* syn_ramdecomp = "power" *) reg [DWIDTH-1:0]        mem [DEPTH-1:0];
```

*VHDL:*

```
attribute syn_ramdecomp: string;
attribute syn_ramdecomp of mem : signal is "power";
```

**Note:** You can use the Block RAM Resource Estimator to explore the number of blocks synthesis will use for various settings. Refer to Estimating Block RAM Resources on page 17 for details.

## syn_ramstyle

You apply this attribute to RAM signals:

- The `block_ram` value assigns the signals to block RAM.
- The `registers` value assigns the signals to registers.

*Verilog HDL:*

```
(* syn_ramstyle = "block_ram" *) reg [DWIDTH-1:0] mem [DEPTH-1:0];
(* syn_ramstyle = "registers" *) reg [DWIDTH-1:0] mem [DEPTH-1:0];
```

*VHDL:*

```
attribute syn_ramstyle: string;
attribute syn_ramstyle of mem : signal is "block_ram";
```

## syn_romstyle

You apply this attribute to ROM signals:

- The `block_rom` value assigns the signals to block ROM.
- The `logic` value assigns the signals to logic.

*Verilog HDL:*

```
(* syn_romstyle = "block_rom" *) reg [DWIDTH-1:0]     mem [DEPTH-1:0];
(* syn_romstyle = "logic" *) reg [DWIDTH-1:0]        mem [DEPTH-1:0];
```

*VHDL:*

```
attribute syn_romstyle: string;
attribute syn_romstyle of mem : signal is "block_rom";
```

## skip_ram_init

This attribute applies to an instantiated block RAM instance. When set to `1` or `true`, synthesis instructs the bitstream generation module to skip including default RAM initialization values to reduce the bitstream size. When the FPGA is configured, the RAM content is not initialized and must be written before being read.

*Verilog HDL:*

```
(*skip_ram_init = 'true'*) EFX_RAM10 dut (...);
```

*VHDL:*

```
attribute skip_ram_init : boolean;
attribute skip_ram_init of u0 : label is true;
```

## syn_srlstyle

This attribute, when applied to a register signal, directs the synthesis inference step to choose between shift register (`srl`), simple register (`registers`), or first register + shift register (`reg_srl`).

**Note:** Shift registers are only available in the Titanium family; therefore, you should only use this attribute when targeting Titanium FPGAs.

*Verilog HDL:*

```
(* syn_srlstyle = "registers" *) reg [WIDTH-1:0]   d;
```

*VHDL:*

```
attribute syn_srlstyle: string;
attribute syn_srlstyle of d : signal is "registers";
```

## translate_on, translate_off

You use these directives to tell the synthesis tool to ignore the code within them. You should use these together; a `translate_off` should have a corresponding `translate_on`.

For example, the FPGA's flipflop powers up to a `0` value. For simulation, you need to initialize the registers to `0` for the simulation to match this behavior. Using this directive allows the synthesis tool to demonstrate the FPGA's default behavior.

*Verilog HDL:*

```
module in_shifter #(parameter N=2)
(
input data,
input clk,
output reg [N-1:0] out
);

   reg [N-1:0]      shift_reg;

   // synthesis translate_off
   initial begin
       shift_reg = 0;
       out = 0;
   end
   // synthesis translate_on

   always @(posedge clk)
     begin
         shift_reg <= {shift_reg[N-2:0],data};
         out <= shift_reg;
     end
endmodule // in_shifter
```

## syn_use_dsp

You use this attribute on multiplier output signals.

- When set to `true`, `yes`, or `1`, the synthesis tool implements the multiply function using hard multipliers (that is, the EFX_MULT primitive).
- When set to `false`, `no`, or `0`, the synthesis tool generates adders and logic for the multiply function.

In Titanium FPGAs, applying this attribute to the output of an adder that is driven on one side by a multiplier tells synthesis to try to pack the adder into the DSP Block by:

- Using an extra DSP Block to feed the other operand through the cascin/cascout path.
- If possible, pack a feedback loop through the N-SEL path so that the multiply-accumulate is implemented in a single DSP Block.

*Verilog HDL:*

```
(* syn_use_dsp = "yes" *) signed [27:0] x;
```

*VHDL:*

```
attribute syn_use_dsp: boolean;
attribute syn_use_dsp of x : signal is true;
```

# Using VHDL Libraries

In the Efinity® software v2020.2 and higher, you can use VHDL libraries to organize and reference commonly used packages and entities.

## Create a Library

To create a library for your project:

1. Open the Project Editor.
2. Click the **Design** tab.
3. Add the design file(s) that have the packages you want to use. You can add multiple files.
4. Double click the cell under **Library**.
5. In the drop-down menu, choose **Add New**.
6. Enter the library name and click **OK**.

> **Note:** In VHDL, the **work** library refers to the current library in the design. When assigning a library name to a VHDL design file, you are encouraged not to use the word **work** as the library name only (instead use a variable like name, example: **my_work**). Doing so will cause an error in synthesis. Leave it blank (or default) if the file is part of the current library in the design project.

7. (Optional) If you add more than one library file to your project, double-click in the **Library** cell for each file and either choose the library name or add a new one.

Library names are saved across projects.

## Add a File to a Library

You add a file to a library in the **Project Editor > Design** tab. Double-click the Library cell for the file and choose the name from the drop-down list.

## Reference a Library

You use the `library` and `use` VHDL language constructs to reference your new library. The following simple code example shows a new library file for the package `mylibrary`:

**Example: mylibrary.vhd**

```
--! Use standard library
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package mylibrary is
 --! factor width
 constant DF_WIDTH : integer := 12;
end package mylibrary;
```

After you add this file to your project and create a library for it, you can refer to the file in your code:

**Example: Referring to the Package**

```
--! Use standard library
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

--! Use costum library
library mylibrary;
use mylibrary.mylibrary.all;
```

```
--! Multiplier entity brief description

--! Detailed description of this
--! multiplier design element.
entity multiplier is
    port (
        a    : in  signed (DF_WIDTH-1 downto 0);     --! Multiplier first factor
        b    : in  signed (DF_WIDTH-1 downto 0);     --! Multiplier second factor
        result     : out signed (2*DF_WIDTH-1 downto 0)    --! Multiplier result
    );
end entity;
```

### Reference Trion and Titanium Primitive Libraries

The Efinity® software includes VHDL libraries for Trion and Titanium primitives. You use the `library` and `use` VHDL language constructs to reference these libraries:

```
library efxphysicallib;
use efxphysicallib.efxcomponents.all;
```

**Learn more:** The following documents provide example code for these libraries:

Quantum Titanium Primitives User Guide
Quantum Trion Primitives User Guide

## Referencing Efinix VHDL Libraries

The Efinity® software includes VHDL libraries for Trion and Titanium primitives. You use the `library` and `use` VHDL language constructs to reference these libraries:

```
library efxphysicallib;
use efxphysicallib.efxcomponents.all;
```

The following code shows how to reference the Efinix library:

**Example: Referring to the Efinix Library**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library efxphysicallib;
use efxphysicallib.efxcomponents.all;

entity LUT4_VHDL is
   port
   (
      din : in std_logic_vector(3 downto 0);
      dout : out std_logic
   );
end entity LUT4_VHDL;

architecture Behavioral of LUT4_VHDL is
begin

 EFX_LUT4_inst : EFX_LUT4
   generic map (
      LUTMASK => x"8888"
   )
   port map (
      I0 => din(0),
      I1 => din(1),
      I2 => din(2),
      I3 => din(3),
      O => dout
   );

end architecture Behavioral;
```

# VHDL 2008 Support

The Efinity® software supports the synthesis subset of the VHDL 2008 standard. The following sections highlight support for new features from an Efinity® perspective. This list is not exhaustive.

## Relational Operators (9.2.1)

The software supports these relational operators.

| Operator | Description | Usage | Type Returned |
|---|---|---|---|
| = | a is equal to b | a = b | Boolean |
| /= | a not equal to b | a /= b | Boolean |
| < | a less than b | a < b | Boolean |
| <= | a is less than or equal to b | a <= b | Boolean |
| > | a is greater than b | a > b | Boolean |
| >= | a is greater than or equal to b | a >= b | Boolean |
| ?= | a is equal to b | a ?= b | Bit or `std_logic` |
| ?/= | a is not equal to b | a ?/= b | Bit or `std_logic` |
| ?< | a is less than b | a ?< b | Bit or `std_logic` |
| ?<= | a is less than or equal to b | a ?<= b | Bit or `std_logic` |
| ?> | a is greater than b | a ?> b | Bit or `std_logic` |
| ?>= | a is greater than or equal to b | a ?>= b | Bit or `std_logic` |

In this example, the output is 1 if A(14) is equal 1 and B(14) is equal to 1; otherwise the output is 0.

```
output <= A(14) ?= '1' and B(14);-
```

## Condition Operator (9.2.9)

The software supports the condition operator, as shown in the following code example.

```
process(ALL)    -- simplify sensitivity list VHDL 2008; ·
Begin
 if A(0) then  -- converts std_logic (A(0)) 1,H to boolean  True, other will be False
  C(0)<='1'
 else
  C(0)<='0';
 end if;
end process;

C(1) <= '1' when A(1) else '0';
```

## Vector Aggregates (9.3.3)

In VHDL 2008, you can use slices in an array aggregation, and you can use aggregates as targets.

```
c(15 downto 8) <= (others =>'1')                    when A(8) else   --means "11111111"
                  ('1','1','0','1', others =>'0')   when A(9) else   --means "1101000"
                  (11=>'1', others =>'0')           when A(10) else  --means "00001000"
                  ("1101" , others =>'0')        ;                  --means "1101000"

(C(39),C(40))<= A(1 downto 0);            -- C(39) becomes A(1) , C(40) becomes A(0)
```

## Conditional and Sequential Statements (10.5.3, 10.5.4)

VHDL 2008 supports conditional and selected sequential statements.

```
DFFwithReset_inst: Process(All)
begin
    if clk'event and clk='1' then
        c(26)<= '0' when reset else A(11);  -- c(26) is Q and A(11) is D
    end if;
end process;
```

## Case Statements with Don't Care (10.9)

VHDL 2008 permits a don't care, -, in a `case?` statement.

```
process(ALL)
begin
    case? A(3 downto 0) is
        when "1---" => c(24) <= '1';
        when "01--" => c(25) <= '1';
        when others => null;
    end case?;
end process;
```

## Sensitivity List (11.3)

You use sensitivity lists to specify a set of signals on which to act. To simplify the sensitivity list and be more comparable with simulation, you can add the keyword `ALL` to the sensitivity list. The `ALL` setting adds all signals to the sensitivity list.

```
process(ALL)
```

# Generate Statements (11.8)

In VHDL 2008 you can use `case` statements in generate statements, and `else/elsif` in `if` statements.

```
generate_test: case sel generate -- sel must be a constant
                  when "00" =>
                     comp1: entity work.test1(behave)
                            port map(B(13),A(13),C(27));
                  when "01" =>
                     comp1: entity work.test2(behave)
                            port map(B(13),A(13),C(27));
                  when others =>
end generate;
```

# Expressions in Port Maps (11.8)

VHDL 2008 allows expressions in port maps.

```
inst1 : entity work.test1(behave)
        port map (A=>(B(13) AND A(13)), B=> B(15),C=>C(41));
```

# Enhanced String Literals (15.8)

VHDL-2008 enhances bit string literals:

- They may have an explicit width
- They may be declared as signed or unsigned
- They may include meta-values ('U', 'X', etc.)

```
sig(5 downto 0) <= 6x"0F"  when A(2)        else   --means  6 bit value "001111"
                   6x"0F"  when A(3)        else   --means "001111"
                   6Sx"F"  when A(5)        else   --means "111111"     -- sign extension
                   6Ux"F"  when A(6)        else   --means "001111"     -- zero extention
                   6SB"11" when A(7)        else   --means "111111"     -- binary format
                   6uO"7";                         --means "000111"     -- octal format
```

# Block Comments (15.9)

VHDL you can use single-line comment or delimited comments.

- *Single-line*—These comments begin with two adjacent hyphens `--`, and the comment extends to the end of the line.
- *Delimited*—These commnents consist of text surrounded with delimiters. The comment begins with `/*` and continues until `*/`

```
Begin
 if A(0) then  -- here is a single line comment
  C(0)<='1'
 else
  C(0)<='0';
 end if;
end process;
/* here is a delimited comment
   that spans two lines */
```

## Fixed-Point Handling (16.10)

VHDL 2008 has fixed-point handling.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.fixed_pkg.all;

Entity test is
Port (
       s1   : in ufixed(21 downto -2);
       s2   : in ufixed(20 downto -3);
       sum  : out ufixed(22 downto -3);
       prod : out ufixed(42 downto -5)
);
end test;

Architecture behave of test is
Begin
sum <= s1 +S2;
prod <= S1*S2;
end behave;
```

## Minimum() and Maximum() Functions (16.3)

The software supports the new functions `minimum()` and `maximum()`, which were added in VHDL 2008.

```
min_val <= minimum(a,b) -- if a > b, b is returned
max_val <= maximum(a,b) -- if a > b, a is returned
```

# Where to Learn More

The Efinity® software includes documentation as PDF user guides and on-line HTML help. This documentation is provided with the software. You can also access the latest versions of PDF documentation in the Support Center:

- **Efinity Software User Guide**
- **Efinity Synthesis User Guide**
- **Efinity Timing Closure User Guide**
- **Efinity Software Installation User Guide**
- **Efinity Trion Tutorial**
- **Efinity Debugger Tutorial**
- **Titanium Interfaces User Guide**
- **Trion Interfaces User Guide**
- **Efinity Interface Designer Python API**
- **Quantum® Trion Primitives User Guide**
- **Quantum® Titanium Primitives User Guide**

In addition to documentation, Efinix field application engineers have created a series of videos to help you learn about aspects of the software. You can view these videos in the Support Center.

# Revision History

*Table 7: Revision History*

| Date | Version | Description |
|---|---|---|
| December 2023 | 3.8 | Added the `--use-logic-for-small-mem`, `--use-logic-for-small-rom`, and `--mult-auto-pipeline` synthesis options. (DOC-1484)<br>Corrected quotation marks in the synthesis attribute examples (now using straight quotes instead of curly). (DOC-1420)<br>Added topic (**Referencing Efinix VHDL Libraries** on page 34) describing which Efinix VHDL libraries to reference when using Efinix primitives. (DOC-1455) |
| June 2023 | 3.7 | Added `syn_keep` synthesis attribute.<br>Added `--mult-decomp-retime`, `--optimize-zero-init-rom`, and `--insert-carry-skip` synthesis options.<br>Provided an example for the `--allow-const-ram-index` synthesis option. |
| May 2023 | 3.6 | Added description about possibility of bit-blasting certain RAM operations but at the cost of FPGA resource. (DOC-1231) |
| December 2022 | 3.5 | Described synthesis rules for inferring output registers. (DOC-1020)<br>Updated the synthesis options. (DOC-1020)<br>The syn_preserve attribute is not supported on a user hierarchy instance. (DOC-1020)<br>Added synthesis pragmas. (DOC-1020) |
| November 2022 | 3.4 | Added note in "Using VHDL Libraries" about the work library. (DOC-957) |
| August 2022 | 3.3 | Added new synthesis options. (DOC-870) |
| December 2021 | 3.2 | Added new synthesis options.<br>Added more detail about the Synthesis project settings. (SYN-549) |
| October 2021 | 3.1 | The default for the -blast_const_operand_adders and -seq_opt synthesis options is 1 for Efinity v2021.1 and higher. (DOC-481) |
| June 2021 | 3.0 | Updated for Efinity software v2021.1.<br>Added support for Titanium FPGAs.<br>Described how to infer Titanium DSP Blocks, shift registers, and RAM.<br>Added Titanium synthesis options.<br>Added topic on retiming.<br>Added the syn_srlstyle attribute.<br>Added the area2 value for the mode synthesis option. |
| January 2021 | 2.1 | Added information on RAM inferencing.<br>Described the Block RAM Resource Estimator. |
| December 2020 | 2.0 | Described new support for VHDL libraries.<br>Added async_reg, skip_ram_init, syn_srlstyle, syn_ramdecomp, and syn_srlstyl synthesis attributes. |
| June 2020 | 1.0 | Initial release. |