UltraFast Design Methodology Guide for FPGAs and SoCs (UG949)

RTL Coding Guidelines

RTL Coding Guidelines

You can create custom RTL to implement interconnect logic functions as well as functions without suitable IP. For optimal results, follow the coding guidelines in this section. For additional guidelines, see this link in the *Vivado Design Suite User Guide: Synthesis* (UG901).

Using Vivado Design Suite HDL Templates

Use the Vivado Design Suite Language Templates when creating RTL or instantiating AMD primitives. The Language Templates include recommended coding constructs for proper inference to the AMD device architecture. Using the Language Templates can ease the design process and lead to improved results.
To open the Language Templates from the Vivado IDE, select the Language Templates option in the Flow Navigator, and select the desired template.

Control Signals and Control Sets

A control set is the grouping of control signals (set/reset, clock enable and clock) that drives any given SRL, LUTRAM, or register. For any unique combination of control signals, a unique control set is formed. This is important, because registers within a 7 series slice all share common control signals, and thus, only registers with a common control set can be packed into the same slice. For example, if a register with a given control set has just one register as a load, the other seven registers in the slice it occupies will be unusable.
Designs with too many unique control sets might have many wasted resources as well as fewer options for placement, resulting in higher power and lower achievable clock frequency. Designs with fewer control sets have more options and flexibility in terms of placement, generally resulting in improved results.
In AMD UltraScale™ devices, there is more flexibility in control set mapping within a CLB. Resets that are undriven do not form part of the control set, because the tie off is generated locally within the slice. However, it is good practice to limit unique control sets to give maximum flexibility in placement of a group of logic.

Resets

Resets are one of the more common and important control signals to take into account and limit in your design. Resets can significantly impact your design's maximum clock frequency, area, and power.
Inferred synchronous code might result in resources such as:

- LUTs
- Registers
- SRLs
- Block or LUT memory
- DSP48 registers

The choice and use of resets can affect the selection of these components, resulting in less optimal resources for a given design. A misplaced reset on an array can mean the difference between inferring one block RAM, or inferring several thousand registers.
Asynchronous resets described at the input or output of a multiplier might result in registers placed in the slices rather than the DSP block. In such situations, additional logic resources are used, which negatively impacts the power consumption and design performance.

When and Where to Use a Reset

AMD devices have a dedicated global set/reset signal (GSR). This signal sets the initial value of all sequential cells in hardware at the end of device configuration.
If an initial state is not specified, sequential primitives are assigned a default value. In most cases, the default value is zero. Exceptions are the FDSE and FDPE primitives that default to a logic one. Every register will be at a known state at the end of configuration. Therefore, it is not necessary to code a global reset for the sole purpose of initializing a device on power up.
AMD highly recommends that you take special care in deciding when the design requires a reset, and when it does not. In many situations, resets might be required on the control path logic for proper operation. However, resets are generally less necessary on the data path logic. Limiting the use of resets:

- Limits the overall fanout of the reset net.
- Reduces the amount of interconnect necessary to route the reset.
- Simplifies the timing of the reset paths.
- Results in many cases in overall improvement in clock frequency, area, and power.

Recommended: Evaluate each synchronous block, and attempt to determine whether a reset is required for proper operation. Do not code the reset by default without ascertaining its real need.

Functional simulation should easily identify whether a reset is needed or not.
For logic in which no reset is coded, there is much greater flexibility in selecting the device resources to map the logic.
The synthesis tool can then pick the best resource for that code to arrive at a potentially superior result by considering, for example:

- Requested functionality
- Clock period requirements
- Available device resources
- Power

Synchronous Reset vs. Asynchronous Reset

If a reset is needed, AMD recommends using synchronous resets. Synchronous resets have the following advantages over asynchronous resets:

- Synchronous resets can directly map to more resource elements in the device architecture.
- Asynchronous resets impact the maximum clock frequency of the general logic structures. Because all AMD device general-purpose registers can program the set/reset as either asynchronous or synchronous, it might seem like there is no penalty in using asynchronous resets. If a global asynchronous reset is used, it does not increase the control sets. However, the need to route this reset signal to all register elements increases routing complexity.
- Asynchronous resets have a greater probability of corrupting memory contents of block RAMs, LUTRAMs, and SRLs during reset assertion. This is especially true for registers with asynchronous resets that drive the input pins of block RAMs, LUTRAMs, and SRLs.
- Synchronous resets offer more flexibility for control set remapping when higher density or fine tuned placement is needed. A synchronous reset can be remapped to the data path of the register if an incompatible reset is found in the more optimally placed slice. This can reduce routing resource utilization and increase placement density where needed to allow proper fitting and improved achievable clock frequency.
- Some resources such as the DSP48 and block RAM have only synchronous resets for the register elements within the block. When asynchronous resets are used on register elements associated with these elements, those registers may not be inferred directly into those blocks without impacting functionality.

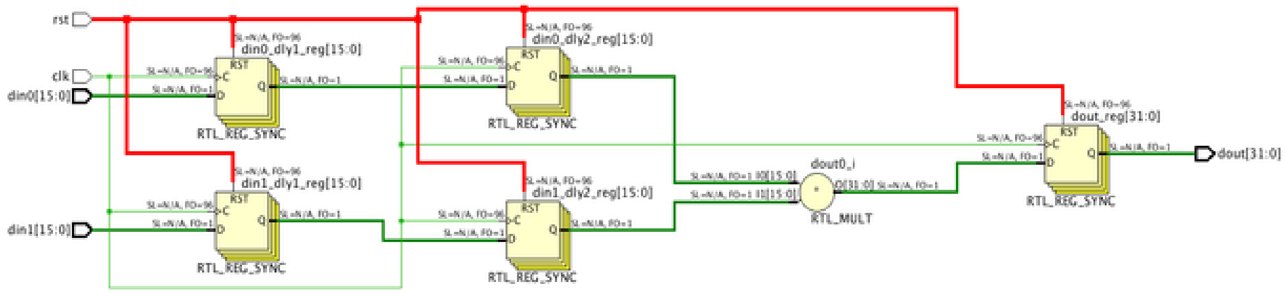
Following are additional considerations:

- The clock works as a filter for small reset glitches for synchronous resets. However, if these glitches occur near the active clock edge, the flip-flop might become metastable.
- Synchronous resets might need to stretch the pulse width to ensure that the reset signal pulse is wide enough for the reset to be present during an active edge of the clock.
- When using asynchronous resets, remember to synchronize the deassertion of the asynchronous reset. Although the relative timing between clock and reset can be ignored during reset assertion, the reset release must be synchronized to the clock. Avoiding the reset release edge synchronization can lead to metastability. During reset release, setup and hold timing conditions must be satisfied for the reset pin relative to the clock pin of a register. A violation of the setup and hold conditions for asynchronous reset (for example, reset recovery and removal timing) might cause the flip-flop to become metastable, causing design failure due to switching to an unknown state. Note that this situation is similar to the violation of setup and hold conditions for the flip-flop data pin.

Reset Coding Example: Multiplier with Synchronous Reset

To take advantage of the existing DSP primitive features, the following example shows a multiplier with synchronous reset.

Figure: Multiplier with Pipeline Registers (Synchronous Reset)



In this circuit, the DSP48 primitive is *inferred* with all pipeline registers packed within the DSP primitive (AREG/BREG=1, MREG=1, PREG=1).
The following figure shows the coding example for multiplier pipeline registers that use a synchronous reset.

Figure: Synchronous Reset Coding Example

```
always @ (posedge clk) begin
  if (rst) begin
    din0_dly1 <= 16'h0;
    din0_dly2 <= 16'h0;
    din1_dly1 <= 16'h0;
    din1_dly2 <= 16'h0;
    dout      <= 32'h0;
  end else begin
    din0_dly1 <= din0;
    din0_dly2 <= din0_dly1;
    din1_dly1 <= din1;
    din1_dly2 <= din1_dly1;
    dout      <= din0_dly2 * din1_dly2;
  end
end
```

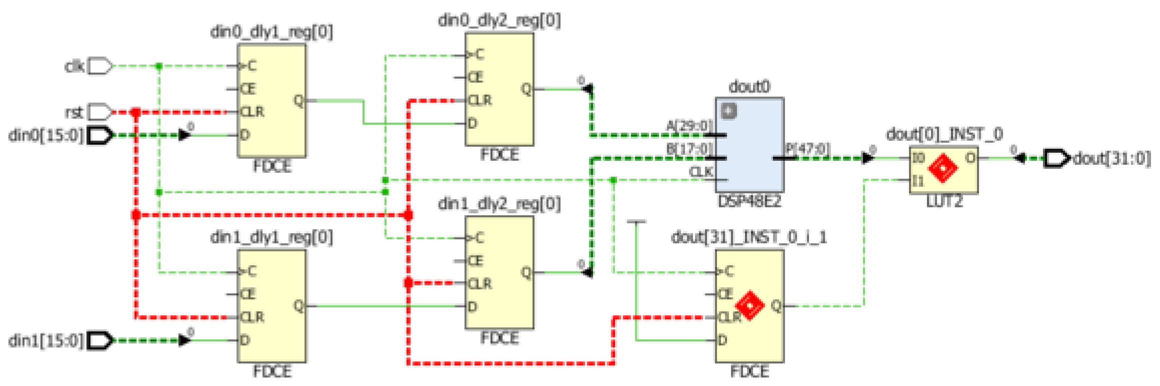
This coding example has the following advantages:

- Optimal resource usage
- Better performance and lower power
- Lower number of endpoints

Reset Coding Example: Multiplier with Asynchronous Reset

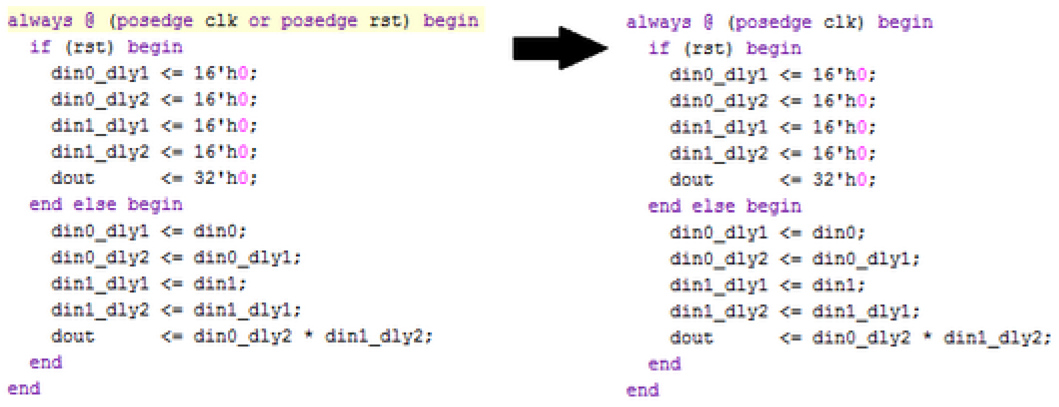
The following example illustrates the importance of using registers with synchronous resets for the logic targeting the dedicated DSP resources. The following figure shows a 16x16 bits DSP48-based multiplier using pipeline registers with asynchronous reset. Synthesis must use regular fabric registers for the input stages, as well as an external register and 32 LUT2s (red markers) to emulate the asynchronous reset on the DSP output (DSP48 P registers are enabled but not connected to reset). This costs an extra 65 registers and 32 LUTs, and the DSP48 results in the configuration: AREG/BREG=0, MREG=0, PREG=1.

Figure: Multiplier with Pipeline Registers Using Asynchronous Resets



By simply changing the reset definition as shown in the following figure, such that the multiplier pipeline registers use a synchronous reset, synthesis can take advantage of the DSP48 internal registers: AREG/BREG=1, MREG=1, PREG=1.

Figure: Changing Asynchronous Reset into Synchronous Reset on Multiplier



```
always @ (posedge clk or posedge rst) begin
  if (rst) begin
    din0_dly1 <= 16'h0;
    din0_dly2 <= 16'h0;
    din1_dly1 <= 16'h0;
    din1_dly2 <= 16'h0;
    dout      <= 32'h0;
  end else begin
    din0_dly1 <= din0;
    din0_dly2 <= din0_dly1;
    din1_dly1 <= din1;
    din1_dly2 <= din1_dly1;
    dout      <= din0_dly2 * din1_dly2;
  end
end
```

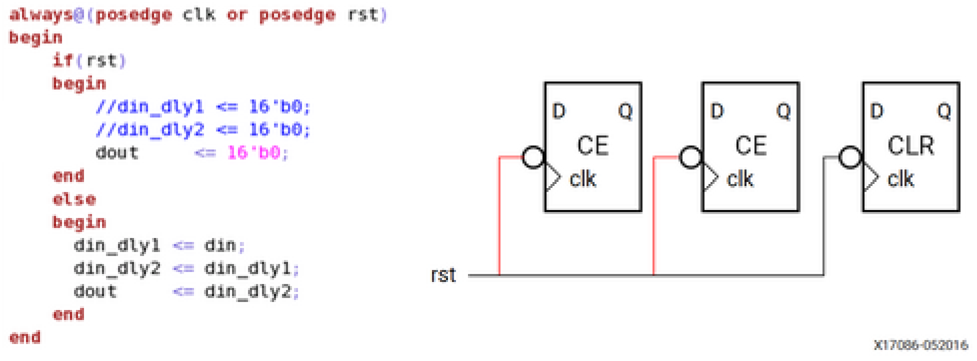
```
always @ (posedge clk) begin
  if (rst) begin
    din0_dly1 <= 16'h0;
    din0_dly2 <= 16'h0;
    din1_dly1 <= 16'h0;
    din1_dly2 <= 16'h0;
    dout      <= 32'h0;
  end else begin
    din0_dly1 <= din0;
    din0_dly2 <= din0_dly1;
    din1_dly1 <= din1;
    din1_dly2 <= din1_dly1;
    dout      <= din0_dly2 * din1_dly2;
  end
end
```

Due to saving fabric resources and taking advantage of all DSP48 internal registers, the design performance and power efficiency are optimal.

Issues When Trying to Eliminate Reset in HDL Code

When optimizing the code to eliminate reset, commenting out the conditions within the reset declaration does not create the desired structures and instead creates issues. For example, the following figure shows three pipeline stages with asynchronous reset used for each. If you attempt to eliminate the reset condition for two of the pipeline stages by commenting out the code with the reset condition, the asynchronous reset becomes enabled (inverted logic of `rst`).

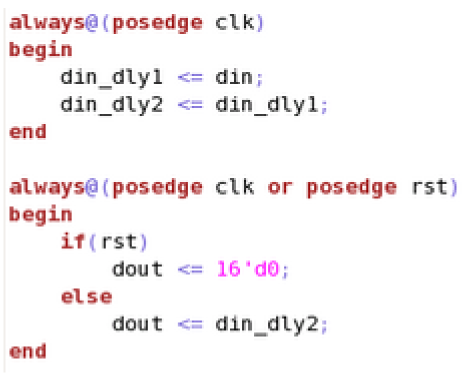
Figure: Commenting Out Code with Reset Conditions



```
always@(posedge clk or posedge rst)
begin
    if(rst)
    begin
        //din_dly1 <= 16'b0;
        //din_dly2 <= 16'b0;
        dout      <= 16'b0;
    end
    else
    begin
      din_dly1 <= din;
      din_dly2 <= din_dly1;
      dout     <= din_dly2;
    end
end
```

The optimal way to remove the resets is to create separate sequential logic procedures with one for reset conditions and the other for non-reset conditions, as shown in the following figure.

Figure: Separate Procedural Statements for Registers With and Without Reset



```
always@(posedge clk)
begin
    din_dly1 <= din;
    din_dly2 <= din_dly1;
end

always@(posedge clk or posedge rst)
begin
    if(rst)
        dout <= 16'd0;
    else
        dout <= din_dly2;
end
```

★ **Tip:**When using a reset, make sure that all registers in the procedural statement are reset.

Clock Enables

When used properly, clock enables can significantly reduce design power with little impact on area or maximum clock frequency. However, when clock enables are used improperly, they can lead to:

- Increased resource utilization
- Decreased placement density
- Increased power
- Reduced achievable clock frequency

In most cases, low fanout clock enables are the main contributor to the high number of control sets.

Creating Clock Enables

Clock enables are created when an incomplete conditional statement is coded into a synchronous block. A clock enable is inferred to retain the last value when the prior conditions are not met. When this is the desired functionality, it is valid to code in this manner. However, in some cases when the prior conditional values are not met, the output is a don't care. In that case, AMD recommends

closing off the conditional (that is, use an `else` clause), with a defined constant (that is, assign the signal to a one or a zero).
In most implementations, this does not result in added logic, and avoids the need for a clock enable. The exception to this rule is in the case of a large bus when inferring a clock enable in which the value is held can help in power reduction. The basic premise is that when small numbers of registers are inferred, a clock enable can be detrimental because it increases control set count. However, in larger groups, it can become more beneficial and is recommended.

Reset and Clock Enable Precedence

In AMD devices, all registers are built to have set/reset take precedence over clock enable, whether an asynchronous or synchronous set/reset is described. In order to obtain the most optimal result, AMD recommends that you always code the set/reset before the enable (if deemed necessary) in the `if/else` constructs within a synchronous block. Coding a clock enable first forces the reset into the data path and creates additional logic.

Related Information

Clocking Guidelines

Controlling Enable/Reset Extraction with Synthesis Attributes

You can force control set mapping by applying the DIRECT_RESET / DIRECT_ENABLE / EXTRACT_RESET / EXTRACT_ENABLE attributes as needed to handle the mapping of control sets for a given structure.
When the design includes a synchronous reset/enable, synthesis creates a logic cone mapped through the CE/R/S pins when the load is equal to or above the threshold set by the `-control_set_opt_threshold` synthesis switch, or creates a logic cone that maps through the D pin if below the threshold. The default thresholds are:

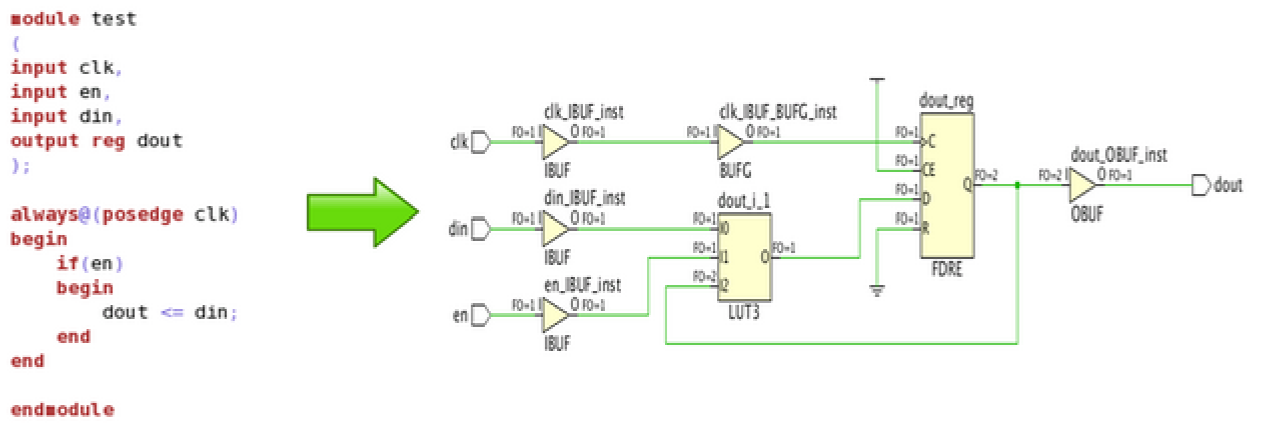
- 7 series devices: 4
- UltraScale devices: 2

Using DIRECT_ENABLE and DIRECT_RESET

To use control set mapping you can apply attributes to the nets connected to enable/reset signals, which will force synthesis to use the CE/R pin.

In the following figure, the enable signal (`en`) is only connected to one flip-flop. Therefore, the synthesis engine connected the en signal to the FDRE/D pin cone of logic. Note that the CE pin is tied to logic 1.

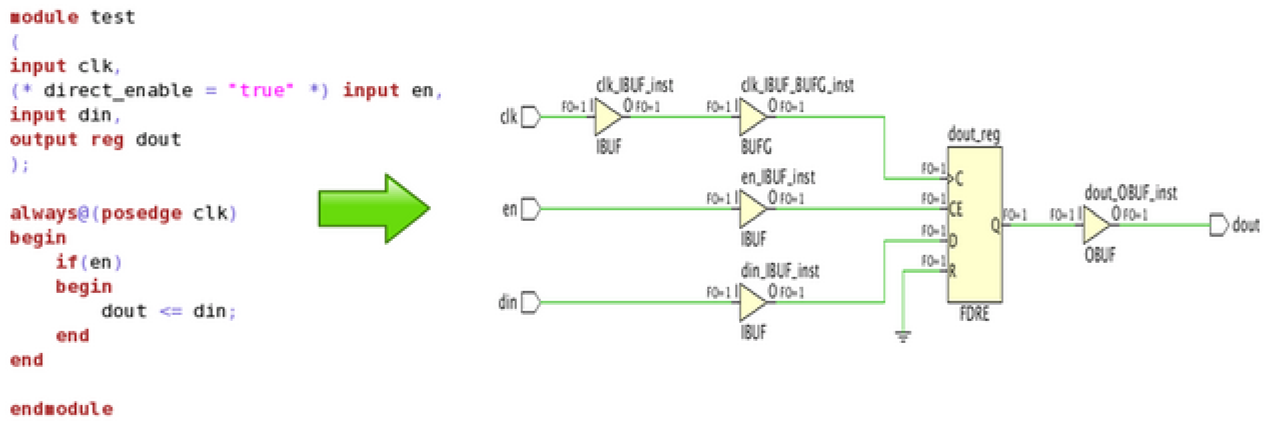
Figure: Clock Enable Implementation Using Datapath Logic



```
module test
(
input clk,
input en,
input din,
output reg dout
);

always@(posedge clk)
begin
    if(en)
    begin
        dout <= din;
    end
end

endmodule
```

To override this default behavior, you can use the DIRECT_ENABLE attribute. For example, the following figure shows how to connect the enable signal (`en`) to the CE pin of the register by adding the DIRECT_ENABLE attribute to the port/signal.

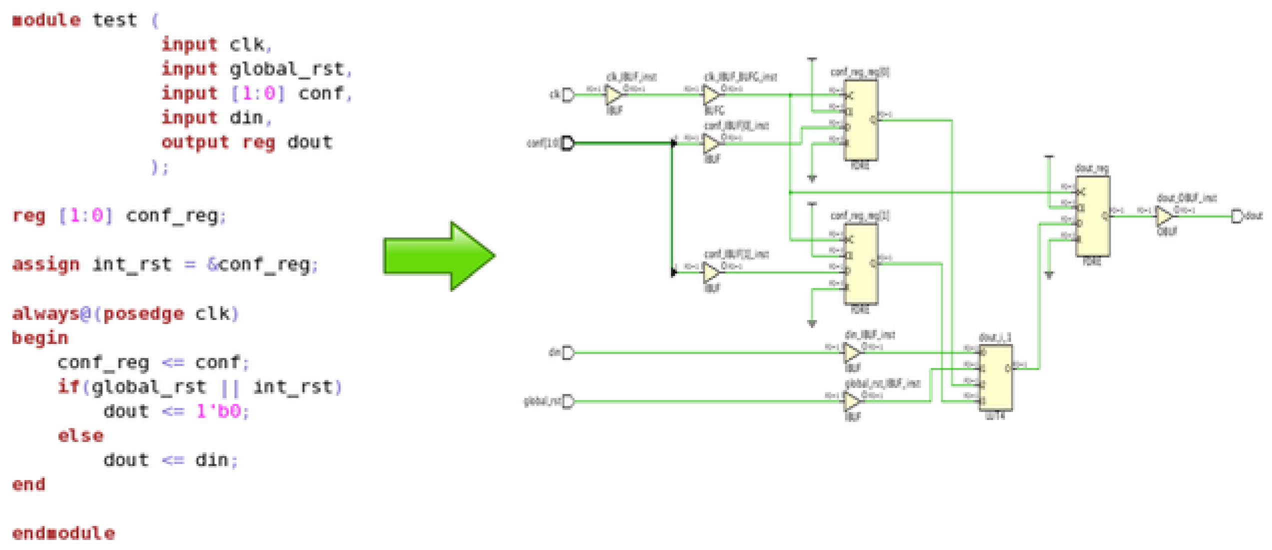
Figure: Dedicated Clock Enable Implementation Using direct_enable



```
module test
(
input clk,
(* direct_enable = "true" *) input en,
input din,
output reg dout
);

always@(posedge clk)
begin
    if(en)
    begin
        dout <= din;
    end
end

endmodule
```

The following figure shows RTL code in which either `global_rst` or `int_rst` can reset the register. By default, both are mapped to the reset pin cone of logic.

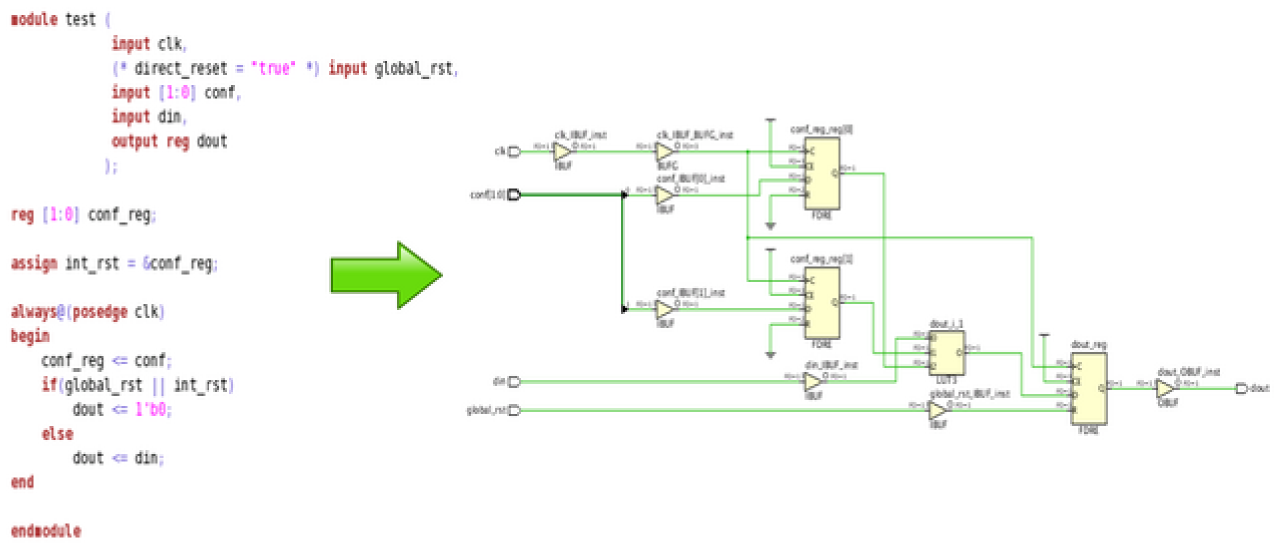
Figure: Multiple Reset Conditions Mapped Through Datapath Logic



```
module test (
                   input clk,
                   input global_rst,
                   input [1:0] conf,
                   input din,
                   output reg dout
                  );

reg [1:0] conf_reg;

assign int_rst = &conf_reg;

always@(posedge clk)
begin
      conf_reg <= conf;
      if(global_rst || int_rst)
            dout <= 1'b0;
      else
            dout <= din;
end

endmodule
```

You can use the DIRECT_RESET attribute to specify which reset signal to connect to the register reset pin. For example, the following figure shows how to use the DIRECT_RESET attribute to connect only the `global_rst` signal to the register FDRE/R pin and connect the `int_rst` signal to the FDRE/D cone of logic.

Figure: Dedicated Reset Pin Usage Using DIRECT_RESET Attribute



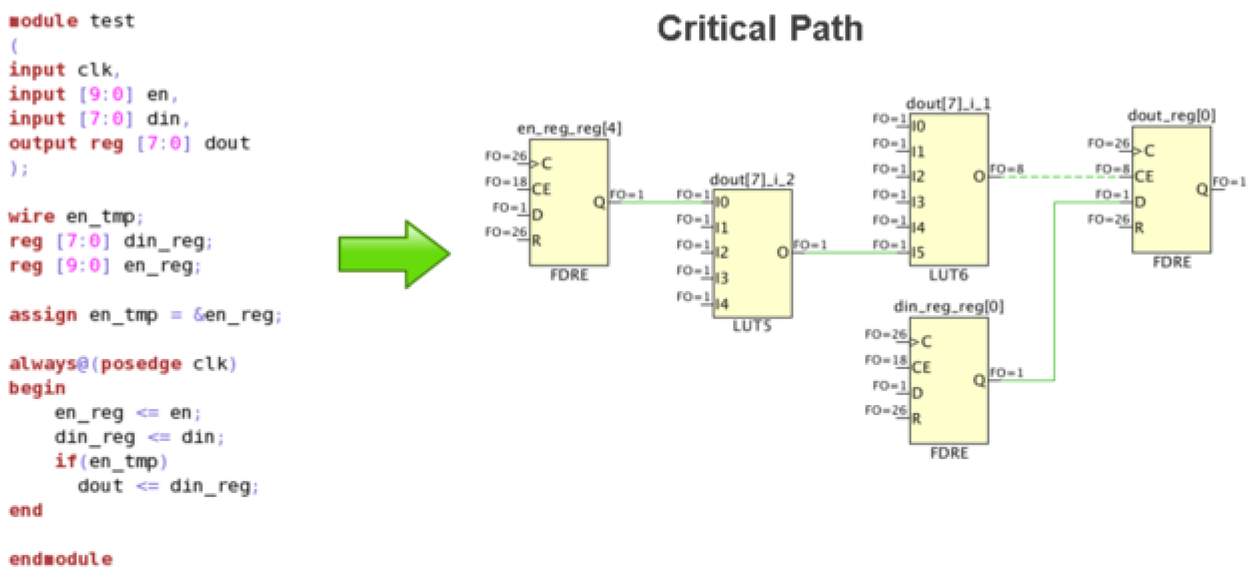
```
module test (
              input clk,
              (* direct_reset = "true" *) input global_rst,
              input [1:0] conf,
              input din,
              output reg dout
             );

reg [1:0] conf_reg;

assign int_rst = &conf_reg;

always@(posedge clk)
begin
    conf_reg <= conf;
    if(global_rst || int_rst)
        dout <= 1'b0;
    else
        dout <= din;
end

endmodule
```

Pushing the Logic from the Control Pin to the Data Pin

During analysis of critical paths, you might find multiple paths ending at control pins. You must analyze these paths to determine if there is a way to push the logic into the datapath without incurring penalties, such as extra logic levels. There is less delay in a path to the D pin than CE/R/S pins given the same levels of logic because there is a direct connection from the output of the last LUT to

the D input of the FF. The following coding examples show how to push the logic from the control pin to the data pin of a register.
In the following example, the enable pin of dout_reg[0] has two logic levels, and the data pin has zero logic levels. In this situation, you can improve timing by moving the enable logic to the D pin by setting the EXTRACT_ENABLE attribute to "no" on the dout register definition in the RTL file.

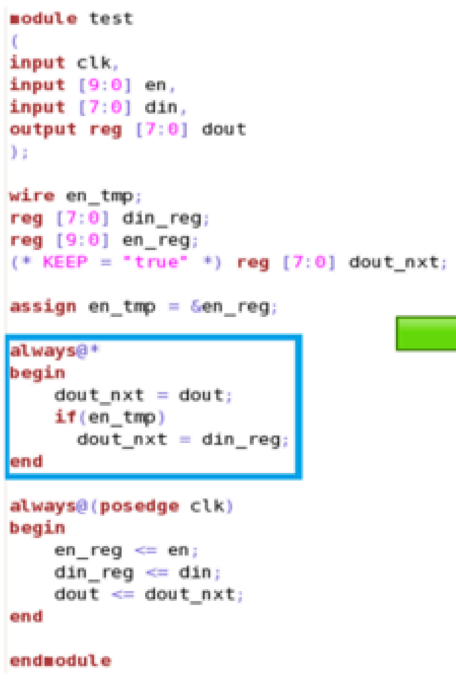
Figure: Critical Path Ending at Control Pin (Enable) of a Register



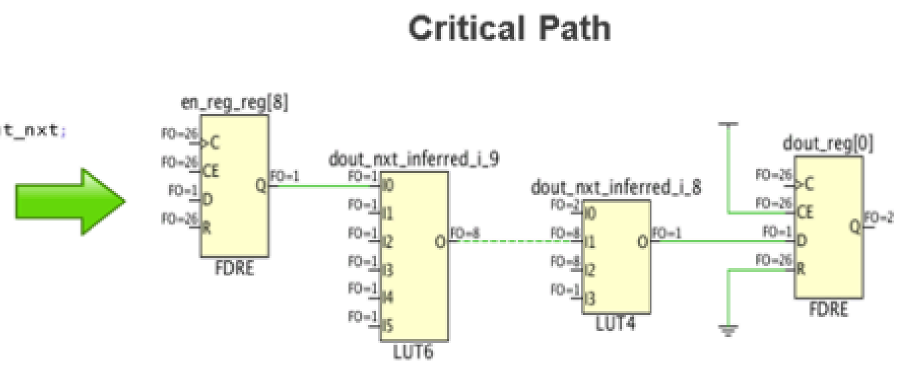
```
module test
(
input clk,
input [9:0] en,
input [7:0] din,
output reg [7:0] dout
);

wire en_tmp;
reg [7:0] din_reg;
reg [9:0] en_reg;

assign en_tmp = &en_reg;

always@(posedge clk)
begin
    en_reg <= en;
    din_reg <= din;
    if(en_tmp)
      dout <= din_reg;
end

endmodule
```

The following example shows how to separate the combinational and sequential logic and map the complete logic in to the datapath. This pushes the logic into the D pin, which still has two logic levels.
You can achieve the same structure by setting the EXTRACT_ENABLE attribute to "no." For more information on the EXTRACT_ENABLE attribute, see this link in the *Vivado Design Suite User Guide: Synthesis* (UG901).

Figure: Critical Path Ending at Data Pin of a Register (Disabling Enable Extraction)



```
module test
(
input clk,
input [9:0] en,
input [7:0] din,
output reg [7:0] dout
);

wire en_tmp;
reg [7:0] din_reg;
reg [9:0] en_reg;
(* KEEP = "true" *) reg [7:0] dout_nxt;

assign en_tmp = &en_reg;

always@*
begin
    dout_nxt = dout;
    if(en_tmp)
      dout_nxt = din_reg;
end

always@(posedge clk)
begin
    en_reg <= en;
    din_reg <= din;
    dout <= dout_nxt;
end

endmodule
```



Tips for Control Signals

- Check whether a global reset is really needed.
- Avoid asynchronous control signals.
- Keep clock, enable, and reset polarities consistent.
- Do not code a set and reset into the same register element.
- If an asynchronous reset is absolutely needed, remember to synchronize its deassertion.

Know What You Infer

Your code finally has to map onto the resources present on the device. Make an effort to understand the key arithmetic, storage, and logic elements in the architecture you are targeting. Then, as you code the functionality of the design, anticipate the hardware resources to which the code will map. Understanding this mapping gives you an early insight into any potential problem.
The following examples demonstrate how understanding the hardware resources and mapping can help make certain design decisions:

- For larger than 4-bit addition, subtraction, and add-sub, a carry chain is generally used and one LUT per 2-bit addition is used (that is, an 8-bit by 8-bit adder uses 8 LUTs and the associated carry chain). For ternary addition or in the case where the result of an adder is added to another value without the use of a register in between, one LUT per 3-bit addition is used (that is, an 8-bit by 8-bit by 8-bit addition also uses 8 LUTs and the associated carry chain).
  If more than one addition is needed, it may be advantageous to specify registers after every two levels of addition to cut device utilization in half by allowing a ternary implementation to be generated.
- In general, multiplication is targeted to DSP blocks. Signed bit widths less than 18x25 (18x27 in UltraScale devices) map into a single DSP Block. Multiplication requiring larger products might map into more than one DSP block. DSP blocks have pipelining resources inside them.
  Pipelining properly for logic inferred into the DSP block can greatly improve performance and power. When a multiplication is described, three levels of pipelining around it generates best setup, clock-to-out, and power characteristics. Extremely light pipelining (one-level or none) might lead to timing issues and increased power for those blocks, while the pipelining registers within the DSP lie unused.
- Two SRLs with depths of 16 bits or less can be mapped into a single LUT, and single SRLs up to 32 bits can also be mapped into a single LUT.
- For conditional code resulting in standard MUX components:
  - A 4-to-1 MUX can be implemented into a single LUT, resulting in one logic level.
  - An 8-to-1 MUX can be implemented into two LUTs and a MUXF7 component, still resulting in effectively one logic (LUT) level.
  - A 16-to-1 MUX can be implemented into four LUTs and a combination of MUXF7 and MUXF8 resources, still resulting in effectively one logic (LUT) level.

  A combination of LUTs, MUXF7, and MUXF8 within the same CLB/slice structure results in a very small combinational delay. Hence, these combinations are considered as equivalent to only one logic level. Understanding this code can lead to better resource management, and can help in better appreciating and controlling logic levels for the data paths.

For general logic, take into account the number of unique inputs for a given register. From that number, an estimation of LUTs and logic levels can be achieved. In general, 6 inputs or fewer always results in a single logic level.

Theoretically, two levels of logic can manage up to 36 inputs. However, for all practical purposes, you should assume that approximately 20 inputs is the maximum that can be managed with two levels of logic. In general, the larger the number of inputs and the more complex the logic equation, the more LUTs and logic levels are required.

!! Important: Check the availability of hardware resources and how efficiently they are being utilized early in the design cycle to enable easier modifications. This approach yields better results than waiting until late in the design cycle during timing closure.

Inferring RAM and ROM

RAM and ROM may be specified in multiple ways. Each has advantages and disadvantages.

- Inference
  Advantages:
  - Highly portable
  - Easy to read and understand
  - Self-documenting
  - Fast simulation

  Disadvantages:
  - Might not have access to all RAM configurations available
  - Might produce less optimal results

  Because inference usually gives good results, it is the recommended method, unless a given use is not supported, or it is not producing adequate results in achievable clock frequency, area, or power. In that case, explore other methods.
  When inferring RAM, AMD recommends that you use the HDL Templates provided in the Vivado tools. As mentioned earlier, using asynchronous reset impacts RAM inference, and should be avoided.
- Xilinx Parameterizable Macros (XPMs)
  Advantages:
  - Portable between AMD device families
  - Fast simulation
  - Support for asymmetric width
  - Predictable quality of results (QoR)

  Disadvantages:
  - Limited to supported XPM options

  XPMs are built on inference using fixed templates that you cannot modify. Therefore, they can guarantee QoR and can support features that standard inference does not. When standard inference does not support the features required, AMD recommends you use XPMs instead.

  Note: When you compile simulation libraries using `compile_simlib`, XPMs are automatically compiled. For more information, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900).

- Direct Instantiation of RAM Primitives
  Advantages:
  - Highest level control over implementation
  - Access to all capabilities of the block

  Disadvantages:
  - Less portable code
  - Wordier and more difficult to understand functionality and intent

- Core from IP catalog
  Advantages:
  - Generally more optimized result when using multiple components
  - Simple to specify and configure

  Disadvantages:
  - Less portable code
  - Core management

Related Information

Using Vivado Design Suite HDL Templates

Performance Considerations When Implementing RAM

To efficiently infer memory elements, consider these factors affecting performance:

- Using Dedicated Blocks or Distributed RAMs
  RAMs can be implemented in either the dedicated block RAM or within LUTs using distributed RAM. The choice not only impacts resource selection, but can also significantly impact the achievable clock frequency and power.
  In general, the required depth of the RAM is the first criterion. Memory arrays described up to 64 bits deep are generally implemented in LUTRAMs, where depths of 32 bits and less are mapped 2 bits per LUT and depths up to 64-bits can be mapped one bit per LUT. Deeper RAMs can also be implemented in LUTRAM depending on available resources and synthesis tool assignment.
  Memory arrays deeper than 256 bits are generally implemented in block memory. AMD devices have the flexibility to map such structures in different width and depth combinations. Familiarize yourself with these configurations to understand the number and structure of block RAMs used for larger memory array declarations in the code.
- Using the Output Pipeline Register
  Using an output register is required for designs operating at higher clock frequency, and is recommended for all designs to ease timing closure. This improves the clock to output timing of the block RAM. Additionally, a second output register is beneficial, as slice output registers have faster clock to out timing than a block RAM register. Having both registers has a total read latency of 3. When inferring these registers, they should be in the same level of hierarchy as the RAM array. This allows the tools to merge the block RAM output register into the primitive.
- Using the Input Pipeline Register
  When RAM arrays are large and mapped across many primitives, they can span a considerable area of the die. This can lead to timing closure issues on address and control lines. Consider adding an extra register after the generation of these signals and before the RAMs. To further improve timing, use `phys_opt_design` later in the flow to replicate this register. Registers without logic on the input will replicate more easily.

Scenarios Preventing Block RAM Output Register Inference

AMD recommends that the memory and the output registers are all inferred in a single level of hierarchy, because this is the easiest method to ensure inference is as intended. There are two scenarios that will infer a block RAM output register. The first one is when an extra register exists on the output, and the second is when the read address register is retimed across the memory array. This can only happen using single port RAM. This is illustrated below:

Figure: RAM with Extra Read Register for Block RAM Output Register Inference

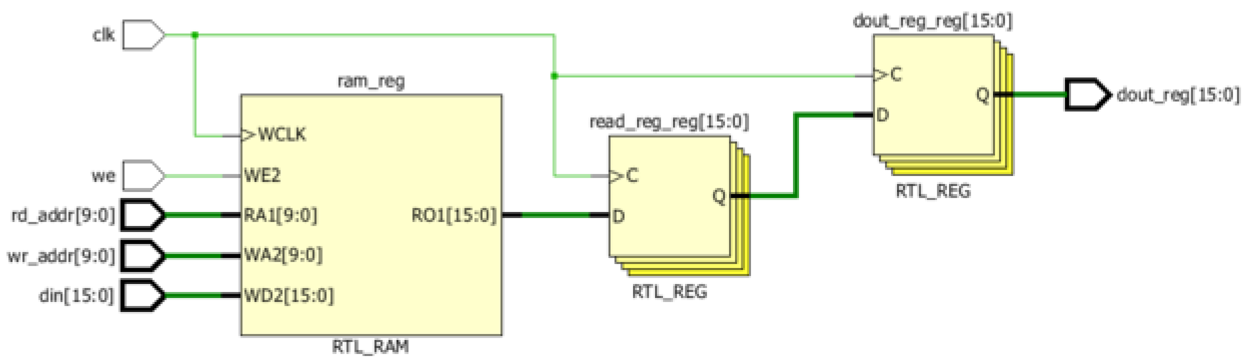
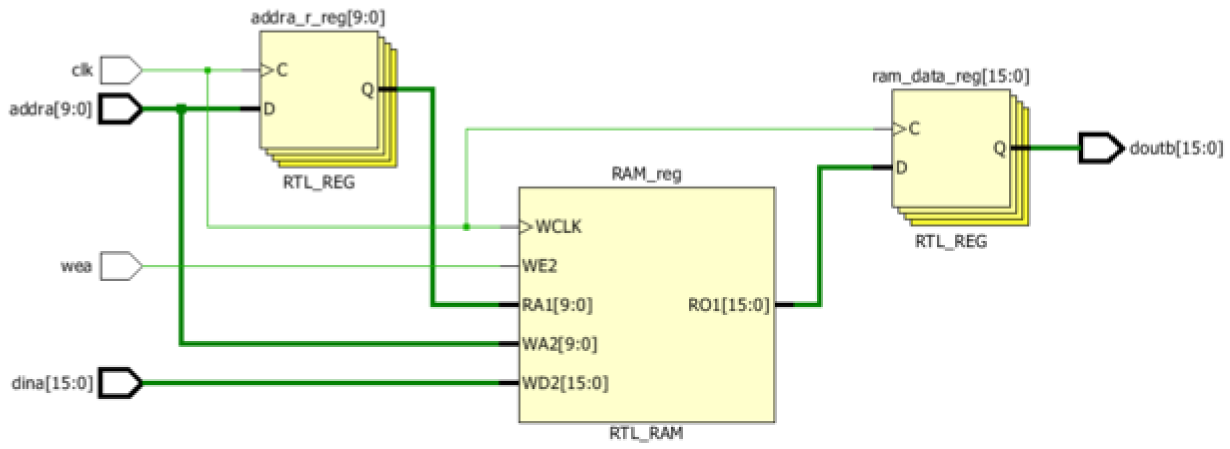


Figure: View of RAM Before Address Register Retiming

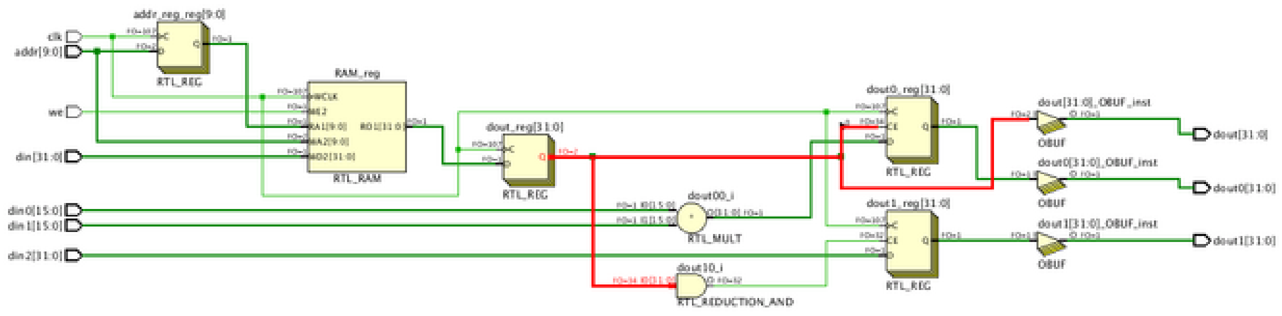


Certain deviations from these examples can prevent the inference of the output register.

Checking for Multi-Fanout on the Output of Read Data Registers

The fanout of the data output bits from the memory array must be 1 for the second register to be absorbed by the RAM primitive. This is illustrated in the following figure.

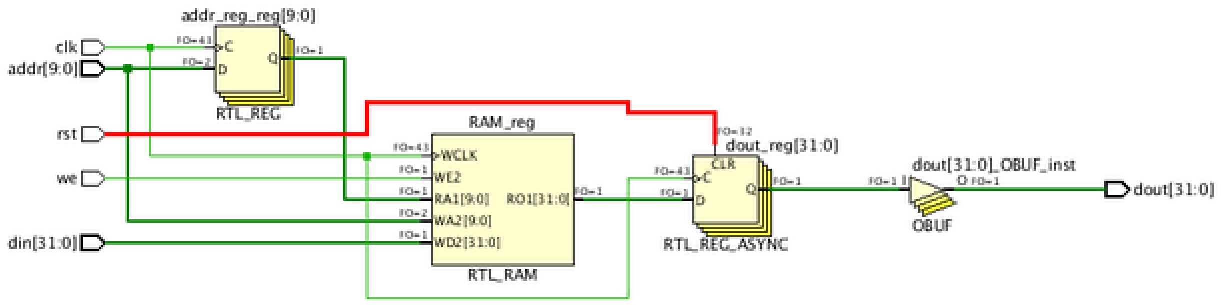
Figure: Multiple Fanout Preventing Block RAM Output Register Inference



Checking for Reset Signals on the Address/Read Data Registers

Memory arrays should not be reset. Only the output of the RAM can tolerate a reset. The reset must be synchronous in order for inference of the output register into the RAM primitive. An asynchronous reset will cause the register to not be inferred into the RAM primitive. Additionally, the output signal can only be reset to 0.
The following figure highlights an example of what to avoid to ensure correct inference of RAMs and output registers.

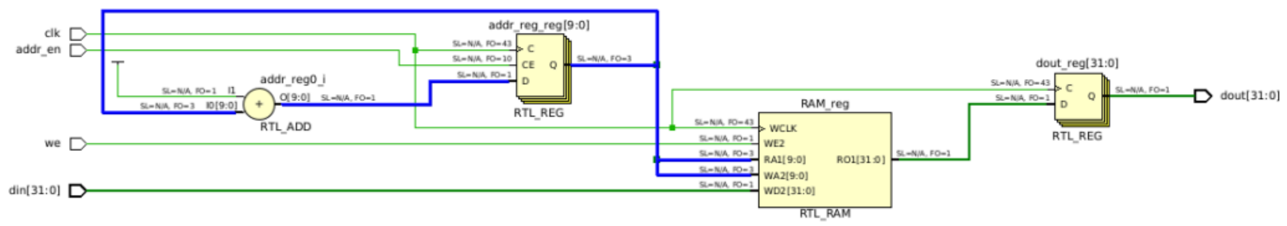
Figure: Checking for Reset On Address/Read Data Registers



Checking for Feedback Structures in Registers

Make sure that registers do not have feedback logic, because this prevents register optimizations. In the following example, the address register drives both the RAM and the adder, which means that the register cannot be packed into a block RAM. The resulting circuit is a block RAM in which the dout register is packed into the RAM to make the RAM fully synchronous. However, the RAM does not use the output registers (DOA_REG and DOB_REG will be set to '0'), which is inefficient.

Figure: Check for the Presence of Feedback on Registers Around the RAM Block



Mapping Memories to UltraRAM Blocks

UltraRAM is a 4Kx72 memory block with two ports using a single clock. This primitive is only available in certain AMD UltraScale+™ devices. In these devices, UltraRAM is included in addition to block RAM resources.
UltraRAM can be used in your design using one of the following methods:

- Rely on synthesis to infer UltraRAMs by setting the `ram_style = "ultra"` attribute on a memory declaration in HDL.
- Instantiate AMD XPM_MEMORY primitives.
- Instantiate UltraRAM UNISIM primitives.

The following code example shows the instantiation of XPM memory and is available in the HDL Language templates. Highlighted parameters `MEMORY_PRIMITIVE` and `READ_LATENCY` are the key parameters to infer memory as UltraRAM for optimal resource mapping efficiency.

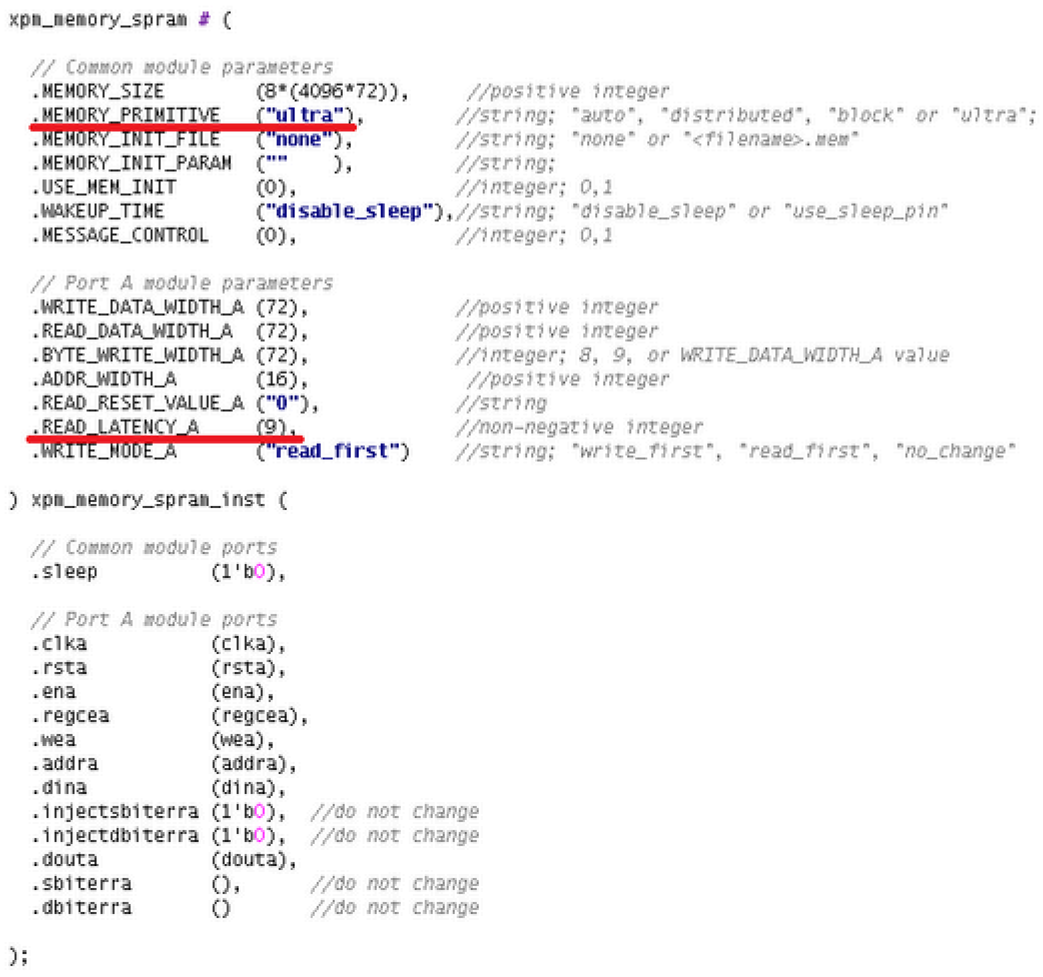
- `MEMORY_PRIMITIVE = "ultra"` specifies the memory is to be inferred as UltraRAM.
- `READ_LATENCY` defines the number of pipeline registers present on the output of the memory.

Larger memories are mapped to an UltraRAM matrix consisting of multiple UltraRAM cells configured as row x column structures.
A matrix can be created with single or multiple columns based on the depth.
The current default threshold for UltraRAM column height is 8 and it can be controlled with the attribute CASCADE_HEIGHT.

The difference between single column and multiple column UltraRAM matrix is as follows:

- Single column UltraRAM matrix uses the built-in hardware cascade without fabric logic.
- Multiple column UltraRAM matrix uses built-in hardware cascade within each column, plus some fabric logic for connecting the columns. Extra pipelining can be required to maintain achievable clock frequency. This is controlled by increasing the read latency. The Vivado tools automatically pack these registers into UltraRAM as required.

Figure: Specifying UltraRAM in RTL Code (via XPM)



```
xpm_memory_spram # (

  // Common module parameters
  .MEMORY_SIZE         (8*(4096*72)),     //positive integer
  .MEMORY_PRIMITIVE    ("ultra"),        //string; "auto", "distributed", "block" or "ultra";
  .MEMORY_INIT_FILE    ("none"),         //string; "none" or "<filename>.mem"
  .MEMORY_INIT_PARAM   (""    ),         //string;
  .USE_MEM_INIT        (0),              //integer; 0,1
  .WAKEUP_TIME         ("disable_sleep"),//string; "disable_sleep" or "use_sleep_pin"
  .MESSAGE_CONTROL     (0),              //integer; 0,1

  // Port A module parameters
  .WRITE_DATA_WIDTH_A (72),              //positive integer
  .READ_DATA_WIDTH_A  (72),              //positive integer
  .BYTE_WRITE_WIDTH_A (72),              //integer; 8, 9, or WRITE_DATA_WIDTH_A value
  .ADDR_WIDTH_A       (16),               //positive integer
  .READ_RESET_VALUE_A ("0"),             //string
  .READ_LATENCY_A     (9),               //non-negative integer
  .WRITE_MODE_A       ("read_first")     //string; "write_first", "read_first", "no_change"

) xpm_memory_spram_inst (

  // Common module ports
  .sleep          (1'b0),

  // Port A module ports
  .clka           (clka),
  .rsta           (rsta),
  .ena            (ena),
  .regcea         (regcea),
  .wea            (wea),
  .addra          (addra),
  .dina           (dina),
  .injectsbiterra (1'b0),  //do not change
  .injectdbiterra (1'b0),  //do not change
  .douta          (douta),
  .sbiterra       (),      //do not change
  .dbiterra       ()       //do not change

);
```

The preceding example uses a 32 K x 72 memory configuration, which uses eight UltraRAMs. To increase the maximum clock frequency of the UltraRAM, more pipelining registers should be added to the cascade chain. This is achieved by increasing the read latency parameter value.
For more information on inferring UltraRAM in Vivado synthesis, see this link in the *Vivado Design Suite User Guide: Synthesis* (UG901).

Coding for Optimal DSP and Arithmetic Inference

The DSP blocks within the AMD devices can perform many different functions, including:

- Multiplication
- Addition and subtraction
- Comparators
- Counters
- General logic

The DSP blocks are highly pipelined blocks with multiple register stages allowing for high-speed operation while reducing the overall power footprint of the resource. AMD recommends that you fully pipeline the code intended to map into the DSP48, so that all pipeline stages are utilized. To allow the flexibility of use of this additional resource, a set condition cannot exist in the function for it to properly map to this resource.
DSP48 slice registers within AMD devices contain only resets, and not sets. Accordingly, unless necessary, do not code a set (value equals logic 1 upon an applied signal) around multipliers, adders, counters, or other logic that can be implemented within a DSP48 slice. Additionally, avoid asynchronous resets, since the DSP slice only supports synchronous reset operations. Code resulting in sets or asynchronous resets may produce suboptimal results in terms of area, performance, or power.
Many DSP designs are well-suited for the AMD architecture. To obtain best use of the architecture, you must be familiar with the underlying features and capabilities so that design entry code can take advantage of these resources.
The DSP48 blocks use a signed arithmetic implementation. AMD recommends code using signed values in the HDL source to best match the resource capabilities and, in general, obtain the most efficient mapping. If unsigned bus values are used in the code, the synthesis tools may still be able to use this resource, but might not obtain the full bit precision of the component due to the unsigned-to-signed conversion.
If the target design is expected to contain a large number of adders, AMD recommends that you evaluate the design to make greater use of the DSP48 slice pre-adders and post-adders. For example, with FIR filters, the adder cascade can be used to build a systolic filter rather than using multiple successive add functions (adder trees). If the filter is symmetric, you can evaluate using the dedicated pre-adder to further consolidate the function into

both fewer LUTs and flip-flops and also fewer DSP slices as well (in most cases, half the resources).

If adder trees are necessary, the 6-input LUT architecture can efficiently create ternary addition (A + B + C = D) using the same amount of resources as a simple 2-input addition. This can help save and conserve carry logic resources. In many cases, there is no need to use these techniques.

By knowing these capabilities, the proper trade-offs can be acknowledged up front and accounted for in the RTL code to allow for a smoother and more efficient implementation from the start. In most cases, AMD recommends inferring DSP resources.

For more information about the features and capabilities of the DSP48 slice, and how to best leverage this resource for your design needs, see the *7 Series DSP48E1 Slice User Guide* (UG479) and *UltraScale Architecture DSP Slice User Guide* (UG579).

Coding Shift Registers and Delay Lines

In general, a shift register is characterized by some or all of the following control and data signals:

- Clock
- Serial input
- Asynchronous set/reset
- Synchronous set/reset
- Synchronous/asynchronous parallel load
- Clock enable
- Serial or parallel output

AMD devices contain dedicated SRL16 and SRL32 resources (integrated in LUTs). These allow efficiently implemented shift registers without using flip-flop resources. However, these elements support only LEFT shift operations, and have a limited number of I/O signals:

- Clock
- Clock Enable
- Serial Data In
- Serial Data Out

In addition, SRLs have address inputs (A3, A2, A1, A0 inputs for SRL16) determining the length of the shift register. The shift register can be a fixed static length or can be dynamically adjusted. In dynamic mode, each time a new address is applied to the address pins, the new bit position value is available on the Q output after the time delay to access the LUT.

Synchronous and asynchronous set/reset control signals are not available in the SRL primitives. However, if your RTL code includes a reset, the AMD synthesis tool infers additional logic around the SRL to provide the reset functionality.

To obtain higher clock frequency when using SRLs, AMD recommends that you implement the last stage of the shift register in the dedicated slice register. Slice registers have a better clock-to-out time than SRLs. This allows additional slack for the paths sourced by the shift register logic. Synthesis tools automatically infer this register unless this resource is instantiated or the synthesis tool is prevented from inferring this type of register because of attributes or cross-hierarchy boundary optimization restrictions. To infer the extra register, register the dynamically delayed signal separately in the RTL.

AMD recommends that you use the HDL coding styles represented in the Vivado Design Suite HDL Templates.

When using registers to obtain placement flexibility in the chip, turn off SRL inference using the following attribute:

```
SHREG_EXTRACT = "no"
```

For more information about synthesis attributes and how to specify the attributes in the HDL code, see the *Vivado Design Suite User Guide: Synthesis* (UG901).

Initialization of All Inferred Registers, SRLs, and Memories

The GSR net initializes all registers to the specified initial value in the HDL code. If no initial value is supplied, the synthesis tool is at liberty to assign the initial state to either zero or one. Vivado synthesis generally defaults to zero with a few exceptions such as one-hot state machine encodings.

Any inferred SRL, memory, or other synchronous element can also have an initial state defined that will be programmed into the associated element upon configuration.

AMD highly recommends that you initialize all synchronous elements accordingly. Initialization of registers is completely inferable by all major device synthesis tools. This lessens the need to add a reset for the sole purpose of initialization, and makes the RTL code more closely match the implemented

design in functional simulation, as all synchronous elements start with a known value in the device after configuration.
Initial state of the registers and latches VHDL coding example one:

```
signal reg1 : std_logic := '0'; -- specifying register1 to start as a zero
signal reg2 : std_logic :=  '1' ; -- specifying register2 to start as a
one
signal reg3 : std_logic_vector(3 downto 0):="1011"; -- specifying INIT
value for
4-bit register
```

Initial state of the registers and latches Verilog coding example two:

```
reg register1 = 1' b0; // specifying regsiter1 to start as a zero
reg register2 = 1' b1; // specifying register2 to start as a one
reg [3:0] register3 = 4' b1011; //specifying INIT value for 4-bit register
```

Another possibility in Verilog is to use an `initial` statement:

```
reg [3:0] register3;
initial begin
    register3= 4' b1011;
end
```

Deciding When to Instantiate or Infer

AMD recommends that you have an RTL description of your design; and that you let the synthesis tool do the mapping of the code into the resources available in the device. In addition to making the code more portable, all inferred logic is visible to the synthesis tool, allowing the tool to perform optimizations between functions. These optimizations include logic replications; restructuring and merging; and retiming to balance logic delay between registers.

Synthesis Tool Optimization

When device library cells are instantiated, synthesis tools do not optimize them by default. Even when instructed to optimize the device library cells, synthesis tools generally cannot perform the same level of optimization as with the RTL. Therefore, synthesis tools typically only perform optimizations on the paths to and from these cells but not through the cells.

For example, if a SRL is instantiated and is part of a long path, this path might become a bottleneck. The SRL has a longer clock-to-out delay than a regular register. To preserve the area reduction provided by the SRL while improving its clock-to-out timing characteristics, a SRL of one delay less than the actual desired delay is created, with the last stage implemented in a regular flip-flop.

When Instantiation Is Desirable

Instantiation might be desirable when the synthesis tool mapping does not meet the timing, power, or area constraints; or when a particular feature within a device cannot be inferred.
With instantiation, you have total control over the synthesis tool. For example, to achieve a higher clock frequency, you can implement a comparator using only LUTs, instead of the combination of LUT and carry chain elements usually chosen by the synthesis tool for area saving reasons.
Sometimes instantiation can be the only way to make use of the complex resources available in the device. This can be due to:

- HDL Language Restrictions
  For example, it is not possible to describe double data rate (DDR) outputs in VHDL because it requires two separate processes to drive the same signal.
- Hardware Complexity
  It is easier to instantiate the I/O SerDes elements than to create synthesizable description.
- Synthesis Tools Inference Limitations
  For example, synthesis tools currently do not have the capability to infer the hard FIFOs from RTL descriptions. Therefore, you must instantiate them.
  If you decide to instantiate an AMD primitive, see the appropriate User Guide and Libraries Guide for the target architecture to fully understand the component functionality, configuration, and connectivity.
  In case of both inference as well as instantiation, AMD recommends that you use the instantiation and language templates from the Vivado Design Suite language templates.

Following are tips:

- Infer functionality whenever possible.
- When synthesized RTL code does not meet requirements, review the requirements before replacing the code with device library component instantiations.
- Consider the Vivado Design Suite language templates when writing common Verilog and VHDL behavioral constructs or if necessary instantiating the desired primitives.

Coding Styles to Improve Maximum Frequency

For high performance designs, the coding techniques discussed in this section can mitigate possible timing hazards.

High Fanouts in Critical Paths

High fanout nets are much easier to deal with early in the design process. What constitutes too high of a fanout is often dictated by the target clock frequency requirements and the construction of the paths. You can use the following techniques to address issues with high fanout nets.

Recommended: Identify high fanout nets using the `report_high_fanout_nets` Tcl command after synthesis. Monitor the impact of these nets on design timing closure as you progress through the implementation process.

Reduce Loads in Portions of the Design That Do Not Require It

For high fanout control signals, evaluate whether all coded portions of the design require that net. Reducing the number of loads can greatly reduce timing problems.

Replicate High Fanout Net Drivers

Register replication can increase the speed of critical paths by making copies of registers to reduce the fanout of a given signal. This gives the implementation tools more flexibility in placing and routing the different loads and associated logic. Synthesis tools use this technique extensively.
Most synthesis tools use a fanout threshold limit to automatically determine whether to duplicate a register. Lowering this global threshold allows automatic

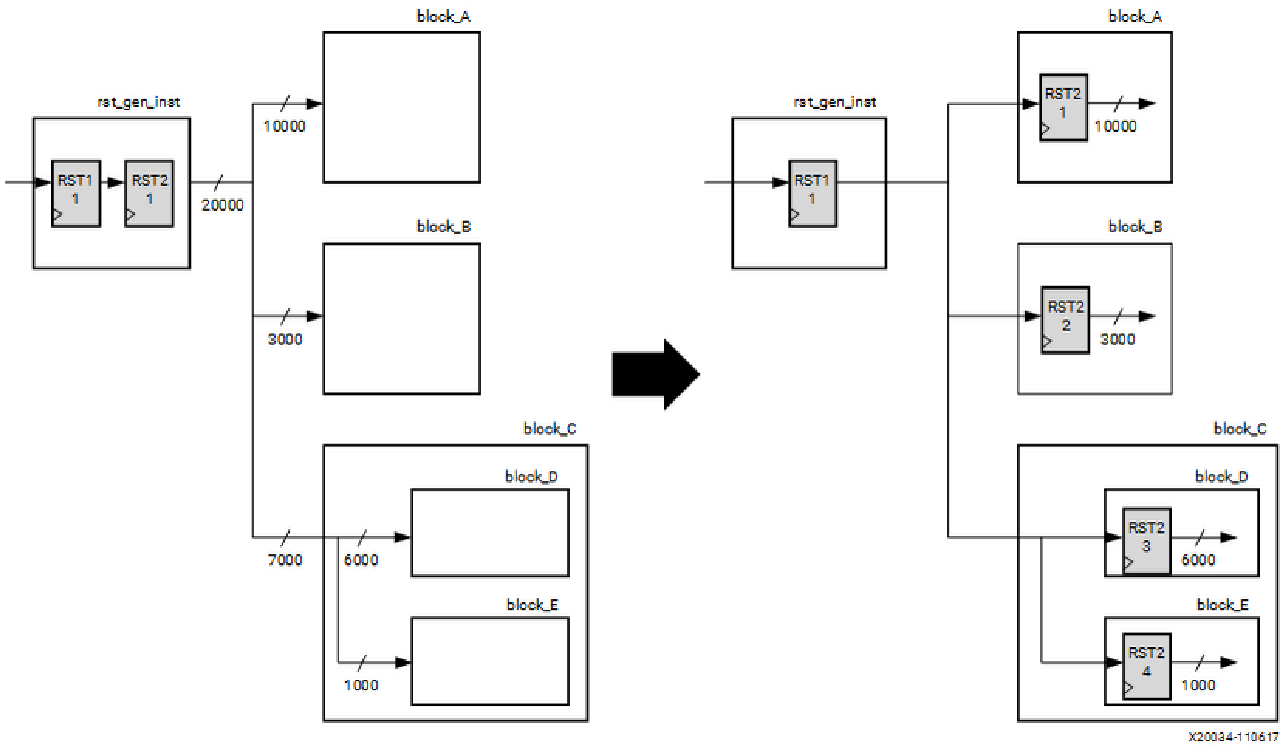
duplication of high fanout nets. However, it does not allow control over which registers are duplicated or how their loads are grouped. In addition, the global replication mechanism does not assess timing slack accurately, which can lead to unnecessary replicated cells, logic utilization increase, and potentially higher power consumption.

For high frequency designs, a better approach to reducing fanout is to use a balanced tree for the high fanout signals. Consider manually replicating registers based on the design hierarchy, because the cells included in a hierarchy are often placed together. For example, in the balanced reset tree shown in the following figure, the high fanout reset FF RST2 is replicated in RTL to balance the fanout across the different modules. If required, physical synthesis can perform further replication to improve WNS based on placement information.

★ **Tip:**To preserve the duplicate registers in synthesis, use a KEEP attribute instead of DONT_TOUCH. A DONT_TOUCH attribute prevents further optimization during physical optimization later in the implementation flow.

Note: If a LUT1 rather than a register is replicated, it indicates that an attribute or constraint is applied incorrectly.

Figure: High Fanout Reset Transformed to Balanced Reset Tree



Recommended: Using MAX_FANOUT attributes on global high fanout signals leads to suboptimal replication similar to when the global fanout limit is lowered in synthesis. For this reason, AMD recommends only using MAX_FANOUT inside the hierarchies on local signals with medium to low fanout.

Do not replicate registers used for synchronizing signals that cross clock domains. The presence of the ASYNC_REG attribute on these registers prevents the tool from replicating these registers. If the synchronizing chain has a very high fanout and replication must meet timing, add an extra register after the synchronization chain that does not have the ASYNC_REG constraint.

The following table provides guidelines on the number of fanouts that might be acceptable for your design.

Table: Fanout Guidelines for Medium Performance 7 Series Devices

Condition	Fanout > 5000	Fanout > 200	Fanout > 100
Low Frequency 1 to 125 MHZ	Few logic levels between synchronous logic <13 levels of logic at maximum frequency	N/A	N/A
Medium Frequency 125 to 250 MHz	If the design does not meet timing, you might need to reduce fanout and/or logic levels.	<6 levels of logic at maximum frequency. (Driver and load types impact performance.)	N/A
High Frequency > 250 MHz	Not recommended for most designs.	Small number of logic levels is typically necessary for higher speeds.	Advance pipelining methods required. Careful logic replication. Compact functions. Low logic levels required. (Driver and load types impact performance.)

★ **Tip:**If the timing reports indicate that high-fanout signals are limiting the design performance, consider replicating the signals using the implementation tool options, such as `opt_design -hier_fanout_limit`, `place_design`, and `phys_opt_design`.

★ **Tip:**When replicating registers, consider using a naming convention for the registers, such as `<original_name>_a`, `<original_name>_b`, etc., to make it easier to understand intent of the replication and easier to maintain the RTL code.

Pipelining Considerations

Another way to increase performance is to restructure long datapaths with several levels of logic and distribute them over multiple clock cycles. This method allows for a faster clock cycle and increased data throughput at the expense of latency and pipeline overhead logic management.
Because devices contain many registers, the additional registers and overhead logic are usually not an issue. However, the datapath spans multiple cycles, and you must make special considerations for the rest of the design to account for the added path latency.

Consider Pipelining for SSI Devices

When designing high performance register-to-register connections for SLR boundary crossings, the appropriate pipelining must be described in the HDL code and controlled at synthesis. This ensures that the shift register LUT (SRL) inference and other optimizations do not occur in the logic path that must cross an SLR boundary. Modifying the code in this manner along with appropriate use of Pblocks defines where the SLR boundary crossing occurs.

Consider Pipelining Up Front

Considering pipelining up front rather than later on can improve timing closure. Adding pipelining at a later stage to certain paths often propagates latency differences across the circuit. This can make one seemingly small change require a major redesign of portions of the code.
Identifying pipelining opportunities early in the design can often significantly improve timing closure, implementation runtime (due to easier-to-solve timing problems), and device power (due to reduced switching of logic).

Check Inferred Logic

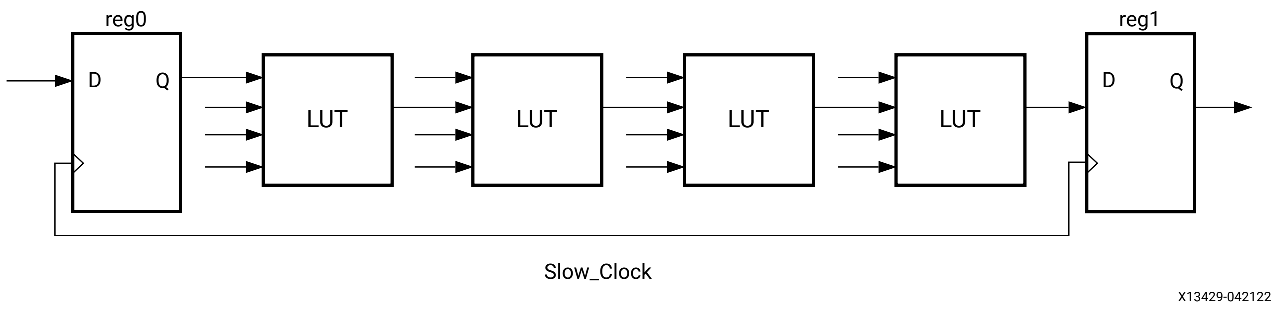
As you code your design, be aware of the logic being inferred. Monitor the following conditions for additional pipelining considerations:

- Cones of logic with large fanin
  For example, code that requires large buses or several combinational signals to compute an output.
- Blocks with restricted placement or slow clock-to-out or large setup requirements
  For example, block RAMs without output registers or arithmetic code that is not appropriately pipelined.
- Forced placement that causes long routes
  For example, a pinout that forces a route across the chip might require pipelining to allow for high-speed operation.
- Logic comprised of large XOR functions
  Large XOR functions often have high switch rates that can generate large dynamic power dissipation. Pipelining these functions can reduce switching, which positively impacts power consumption of the described circuit.

In the following figure the clock speed is limited by the following:

- Clock-to out-time of the source flip-flop
- Logic delay through four levels of logic
- Routing associated with the four function generators
- Setup time of the destination register

Figure: Before Pipelining

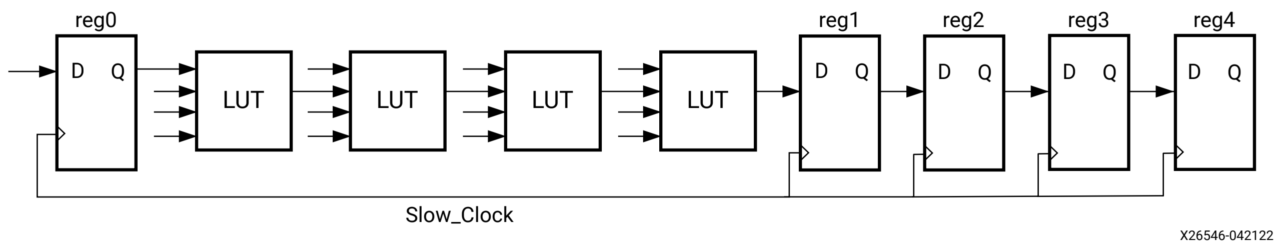


Use one of the following methods to ensure that your design uses pipeline registers correctly:

- In your RTL code, add the registers before or after the logic to be retimed, preferably within the hierarchy.
- Use the Vivado synthesis global retiming or BLOCK_SYNTH.RETIMING option, which analyzes the timing of a path and moves the registers to improve timing, if possible.
- Alternatively, for more control, use the `retiming_forward` and `retiming_backward` synthesis attributes. You can add these attributes on specific registers to force the tool to retime through combinational logic regardless of the timing score of the logic. For more information on these attributes, see the *Vivado Design Suite User Guide: Synthesis* (UG901).

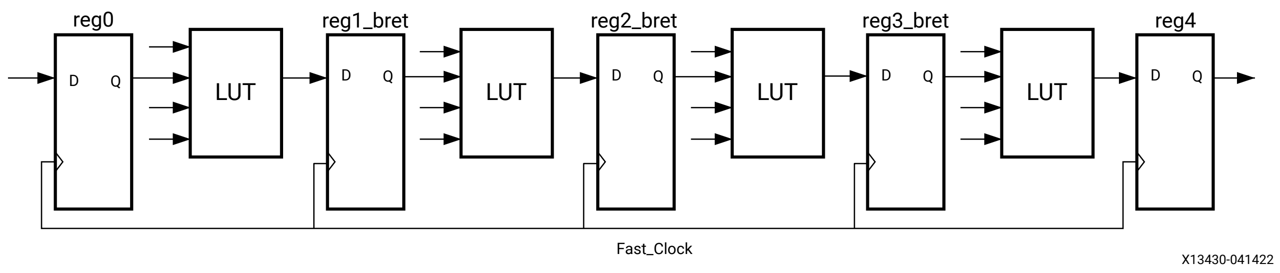
The following figure shows the pipelining after adding extra registers.

Figure: Pipelining After Adding Extra Registers



The following figure is an example of the same data path shown in the Before Pipelining diagram. Because the flip-flop is contained in the same slice as the function generator, the clock speed is limited by the clock-to-out time of the source flip-flop, the logic delay through one level of logic, one routing delay, and the setup time of the destination register. In this example, the system clock runs faster after pipelining and retiming than in the original design.

Figure: Pipelining After Retiming



Following is a code example that shows how to use the retiming attributes to force the specific retiming shown in the Pipelining After Retiming figure.

```
(* retiming_backward = 3 *) reg reg1;
(* retiming_backward = 2 *) reg reg2;
(* retiming_backward = 1 *) reg reg3;
```

Determine Whether Pipelining is Needed

A commonly used pipelining technique is to identify a large combinatorial logic path, break it into smaller paths, and introduce a register stage between these paths, ideally balancing each pipeline stage.
To determine whether a design requires pipelining, identify the frequency of the clocks and the amount of logic distributed across each of the clock groups. You can use the `report_design_analysis` Tcl command with the `-logic_level_distribution` option to determine the logic-level distribution for each of the clock groups.

★ **Tip:**The design analysis report also highlights the number of paths with zero logic levels, which you can use to determine where to make modifications in your code.

Balance Latency

To balance the latency by adding pipeline stages, add the stage to the control path and not the data path. The data path includes wider buses, which increases the number of flip-flop and register resources used.
For example, if you have a 128-bit data path, 2 stages of registers, and a requirement of 5 cycles of latency, inserting 3 register stages results in an extra 3 x 128 = 384 flip-flops. Alternatively, you can use registers to control logic to enable the data path. Use 5 stages of single-bit registers to control the enable signal of datapath flip-flops and multicycle path timing exceptions accordingly.

Note: This example is only possible for certain designs. For example, in cases where there is a fanout from the intermediate data path flip-flops, having only 2 stages does not work.

Recommended: The optimal LUT:FF ratio in a device is 1:1. Designs with significantly more FFs will increase unrelated logic packing into slices, which will increase routing complexity and can degrade QoR.

Balance Pipeline Depth and SRL Usage

When there are deep register pipelines, map as many registers as possible into the SRLs to avoid significant increases in register utilization. For example, a 9-deep pipeline for a data width of 32 results in 9 registers for each bit, which uses

32 x 9 = 288 registers. Mapping the same structure to SRLs uses 32 SRLs. Each SRL has address pins A4 through A0 connected to 5'b01000 to implement a depth of 9 stages.
There are multiple ways to infer SRLs during synthesis, including the following:

- SRL
- REG -> SRL
- SRL -> REG
- REG -> SRL -> REG

You can create these structures using the `srl_style` attribute in the RTL code as follows:

- `(* srl_style = "srl" *)`
- `(* srl_style = "reg_srl" *)`
- `(* srl_style = "srl_reg" *)`
- `(* srl_style = "reg_srl_reg" *)`

A common mistake is to use different enable/reset control signals in deeper pipeline stages. Following is an example of a reset used in a 9-deep pipeline stage with the reset connected to the third, fifth, and eighth pipeline stages. With this structure, the tools map the pipeline stages to registers only, because there is a reset pin on the SRL primitive.

```
FF->FF->FF(reset) -> FF->FF(reset)->FF->FF->FF(reset)->FF
```

To take advantage of SRL inference:

- Ensure there are no resets for the pipeline stages.
- Analyze whether the reset is really required.
- Use the reset on one flip-flop (for example, on the first or last stage of the pipeline).

Avoid Unnecessary Pipelining

For highly utilized designs, too much pipelining can lead to suboptimal results. For example, unnecessary pipeline stages increase the number of flip-flops and routing resources, which might limit the place and route algorithms if the utilization is high.

Note: If there are many paths with 0/1 levels of logic, check to make sure this is intentional.

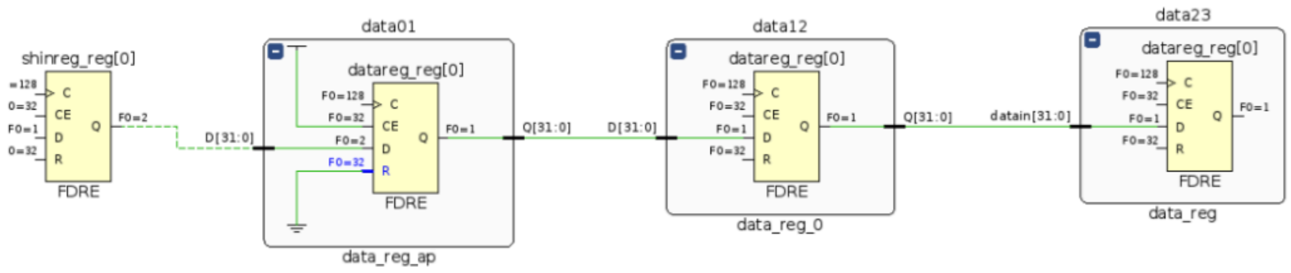
Consider Pipelining Macro Primitives

Based on the target architecture, dedicated primitives such as block RAMs and DSPs can work at over 500 MHz if enough pipelining is used. For high frequency designs, AMD recommends using all of the pipelines within these blocks.

Auto-Pipelining Considerations

The auto-pipelining feature allows the placer to determine the number of required pipeline stages and their optimal location, which helps timing closure across interface boundaries. You can enable this feature by setting up the auto-pipelining mode of the AXI Register Slice core or by applying the auto-pipelining HDL attribute or XDC constraints for data buses. Because the insertion is timing-driven, always be sure to apply proper timing constraints on the targeted paths. For more information, see this link in the *Vivado Design Suite User Guide: Implementation* (UG904).
The following example shows auto-pipelining applied on the interface between the module data01 and data12. The output from data01 consists of registers with no control sets.

Figure: Simple Data Flow Connections Between Modules



Following is the RTL code for this example. The `autopipeline_module` attribute is applied on the hierarchical module data01, and the `autopipeline_group/autopipeline_limit/autopipeline_include` attributes are applied on the nets directly driven by the Q pins of the registers.

```
data_reg_ap #( .C_DATA_WIDTH(C_DATA_WIDTH)) data01 (
.clk (clk),
.datain (shinreg),
.datareg (d1)
);

data_reg #( .C_DATA_WIDTH(C_DATA_WIDTH)) data12 (
.clk (clk),
.datain (d1),
.datareg (d2)
);

(* autopipeline_module="yes" *)
module data_reg_ap # (
parameter integer C_DATA_WIDTH = 32
)
( input wire clk,
input wire [C_DATA_WIDTH-1:0] datain,
(* autopipeline_group="fwd",autopipeline_limit=24 *)
output reg [C_DATA_WIDTH-1:0] datareg
);

always @(posedge clk) begin
datareg <= datain;
end
endmodule
```

Following are the XDC constraints for this example, which is an alternative approach to using attributes in the RTL code.

```
# It's suggested to add the USER_SLR_ASSIGNMENT property at the module
#level to ensure better logic clustering with its driver and load, see
UG912
#for more details on this property
set_property USER_SLR_ASSIGNMENT APSRC [get_cells data01]
set_property USER_SLR_ASSIGNMENT APDST [get_cells data12]

set_property AUTOPIPELINE_MODULE TRUE [get_cells data01]
set_property AUTOPIPELINE_GROUP WBUS [get_nets -of [get_pins -filter
REF_PIN_NAME==Q -of [get_cells data01/*]]]
set_property AUTOPIPELINE_LIMIT 10 [get_nets -of [get_pins -filter
REF_PIN_NAME==Q -of [get_cells data01/*]]]
```

Coding Styles to Improve Power

Gate Clock or Data Paths

Gating the clock or data paths is a common technique to stop transition when the results of these paths are not used. Gating a clock stops all driven synchronous loads and prevents data path signal switching and glitches from continuing to propagate.
Power optimization (`power_opt_design`) can automatically generate signal gating logic to reduce switching activity. However, you have information about the application, data flow, and dependencies that is not available to the tool, which only you can specify.

Maximize Gating Elements

Maximize the number of elements affected by the gating signal. For example, it is more power efficient to gate a clock domain at its driving source than to gate each load with a clock enable signal.

Use Clock Enable Pins of Dedicated Clock Buffers

When gating or multiplexing clocks to minimize activity or clock tree usage, use the clock enable ports of dedicated clock buffers. Inserting LUTs or using other methods to gate-off clock signals is not efficient for power and timing.

Use Case Block When Priority Encoder Not Needed

When a priority encoding is not needed, use a case block instead of an if-then-else block or ternary operator.
Inefficient coding example:

```
if (reg1)
    val = reg_in1;
else if (reg2)
      val = reg_in2;
else if (reg3)
      val = reg_in3;
else val = reg_in4;
```

Correct coding example:

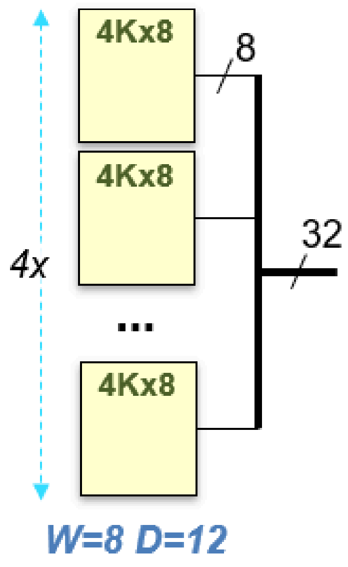
```
(* parallel_case *) casex ({reg1, reg2, reg3})
1xx: val = reg_in1 ;
01x: val = reg_in2 ;
001: val = reg_in3 ;
default: val = reg_in4 ;
endcase
```

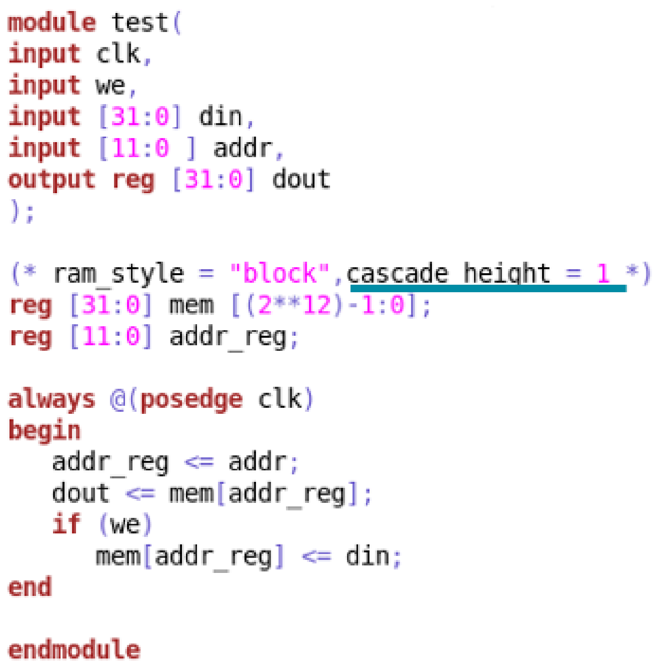
Performance/Power Trade-Off for Block RAMs

There are multiple ways of breaking a memory configuration to serve a particular requirement. The requirement for a particular design can be clock frequency, power, or a mixture of both.
The following example highlights the different structures that can be generated to achieve your requirements. Synthesis can limit the cascading of the block RAM for the clock frequency/power trade-off using the CASCADE_HEIGHT attribute. The usage and arguments for the attribute are described in the *Vivado Design Suite User Guide: Synthesis* (UG901).
The following figure shows an example of 4Kx32 memory configuration for achieving a higher clock frequency (timing).

Note: This example applies to UltraScale and UltraScale+ devices only.

Figure: RTL Representation of 4Kx32 Using 4Kx8 and CASCADE_HEIGHT=1

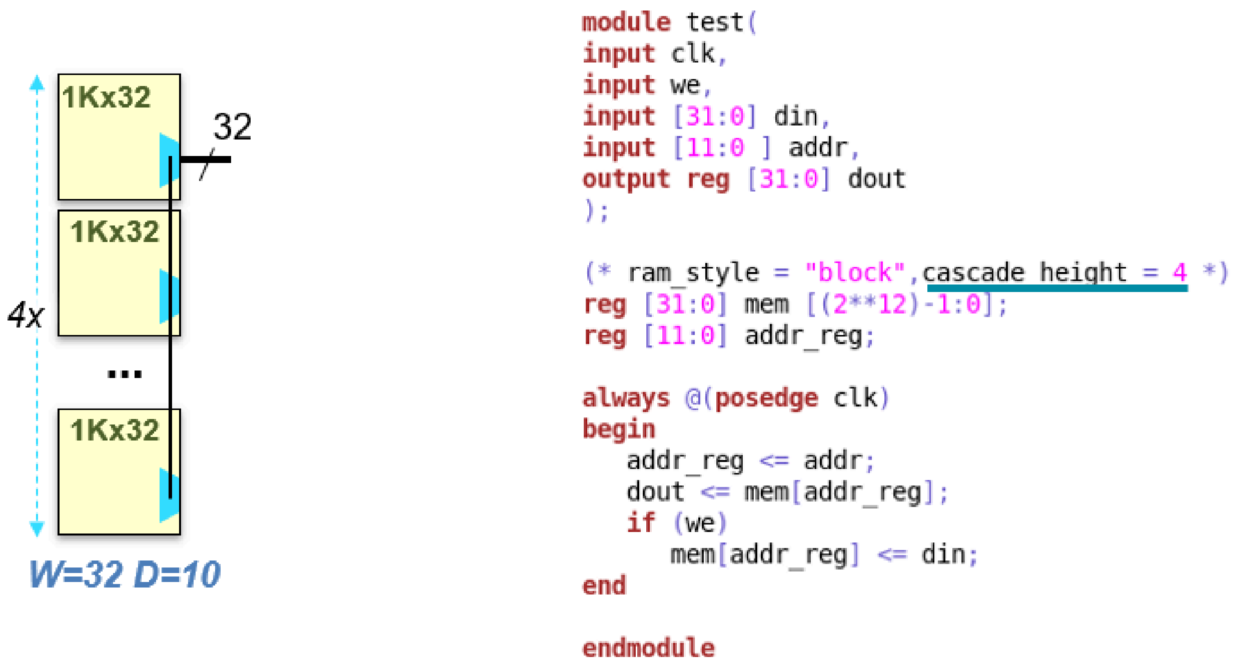




```
module test(
input clk,
input we,
input [31:0] din,
input [11:0 ] addr,
output reg [31:0] dout
);

(* ram_style = "block",cascade_height = 1 *)
reg [31:0] mem [(2**12)-1:0];
reg [11:0] addr_reg;

always @(posedge clk)
begin
   addr_reg <= addr;
   dout <= mem[addr_reg];
   if (we)
      mem[addr_reg] <= din;
end

endmodule
```

In this implementation, all block RAMs are always enabled (for each read or write) and consume more power.
The following figure shows an example of cascading all the block RAMs for low power.

Figure: RTL Representation of 4Kx32 Using 1Kx32 and CASCADE_HEIGHT=4

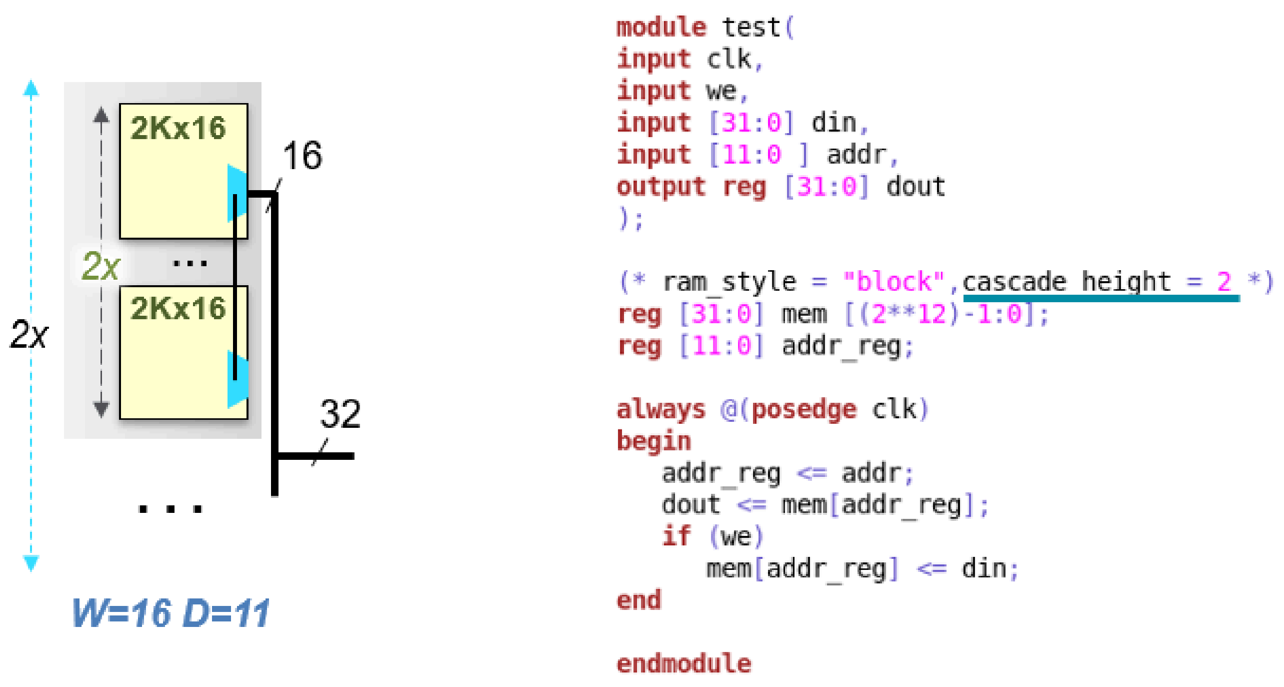


```
module test(
input clk,
input we,
input [31:0] din,
input [11:0 ] addr,
output reg [31:0] dout
);

(* ram_style = "block",cascade_height = 4 *)
reg [31:0] mem [(2**12)-1:0];
reg [11:0] addr_reg;

always @(posedge clk)
begin
   addr_reg <= addr;
   dout  <= mem[addr_reg];
   if (we)
      mem[addr_reg] <= din;
end

endmodule
```

In this implementation, because one block RAM at a time is selected (from each unit), the dynamic power contribution is almost half. Block RAMs have a dedicated cascade MUX and routing structure that allows the construction of wide, deep memories requiring more than one block RAM primitive to be built in a very power efficient configuration.
The following figure shows an example of how to limit the cascading and gain both power and clock frequency at the same time, often with no trade-off in performance.

Note: This example applies to UltraScale and UltraScale+ devices only.

Figure: RTL Representation of 4Kx32 Using 2Kx16 and CASCADE_HEIGHT=2



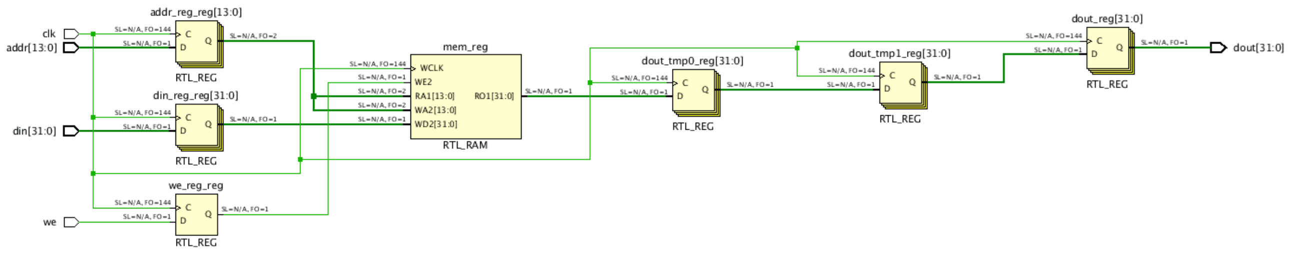
```
module test(
input clk,
input we,
input [31:0] din,
input [11:0 ] addr,
output reg [31:0] dout
);

(* ram_style = "block",cascade_height = 2 *)
reg [31:0] mem [(2**12)-1:0];
reg [11:0] addr_reg;

always @(posedge clk)
begin
   addr_reg <= addr;
   dout <= mem[addr_reg];
   if (we)
      mem[addr_reg] <= din;
end

endmodule
```

Because two block RAMs are selected at a time in this implementation, the dynamic power contribution is better than for the high clock frequency structure, but not as good as for the low power structure. The advantage with this structure compared to a low power structure is that it uses only two block RAMs in the cascaded path, which has impact on the target frequency when compared to four block RAMs in the critical path for the low power structure.

Decomposing Deeper Memory Configurations for Balanced Power and Clock Frequency

When working with deeper memory configurations, you can use the RAM_DECOMP synthesis attribute in the RTL to reduce power by improving memory composition. When the RAM_DECOMP attribute is applied to a memory array, the memory logic is mapped to a wider array of block RAM primitives. To balance power and clock frequency, you can control cascading using the CASCADE_HEIGHT attribute along with the RAM_DECOMP attribute. This approach requires more address decoding logic but helps to reduce the number of block RAMs that are enabled for each read operation, which helps to reduce power.
For example, the following figure shows a 32x16K memory configuration.

Figure: 32x16K Memory Configuration



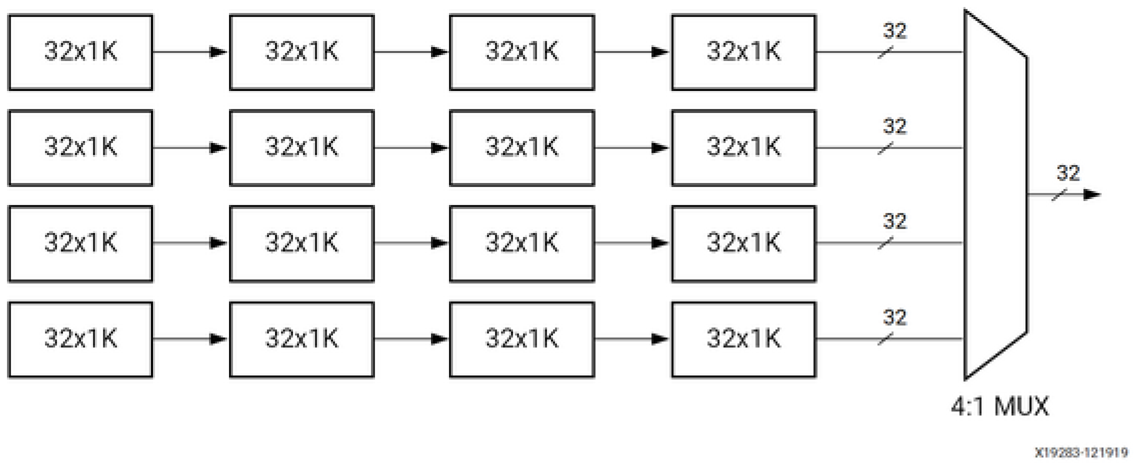
If you apply the following attributes:

```
ram_decomp = ″power″
cascade_height = 4
```

16 RAMB36E2 is inferred and the memory is decomposed as follows:

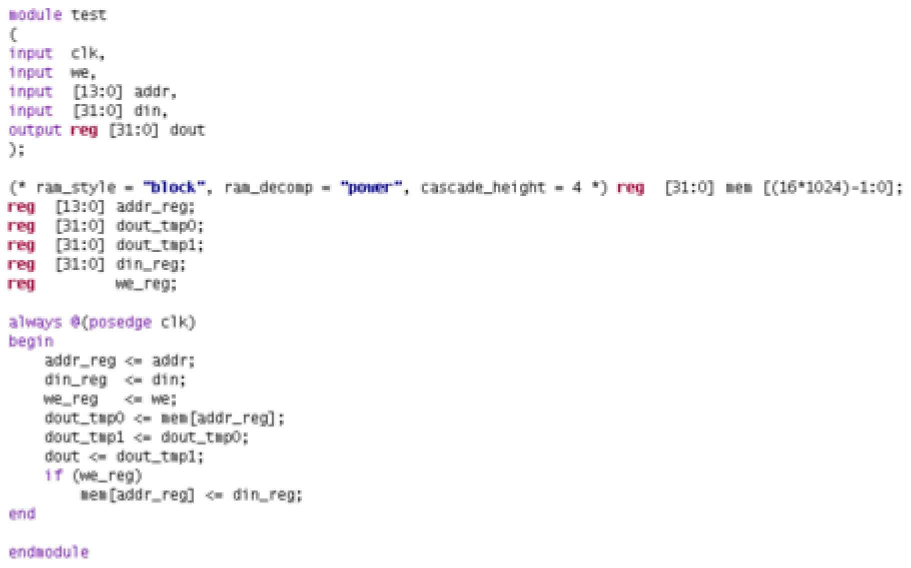
- The base primitive is 32x1K.
- 4 block RAMs are cascaded to create a 32x4K configuration.
- 4 parallel structures create a 16K deep memory.
- The outputs are multiplexed to generate the output data.

Figure: Generated Structure for 32x16K Memory Configuration Example Using CASCADE_HEIGHT and RAM_DECOMP Attributes



The following RTL code example shows the use of the CASCADE_HEIGHT and RAM_DECOMP attributes.

Figure: RTL Code for 32x16K Memory Configuration Using the CASCADE_HEIGHT and RAM_DECOMP Attributes

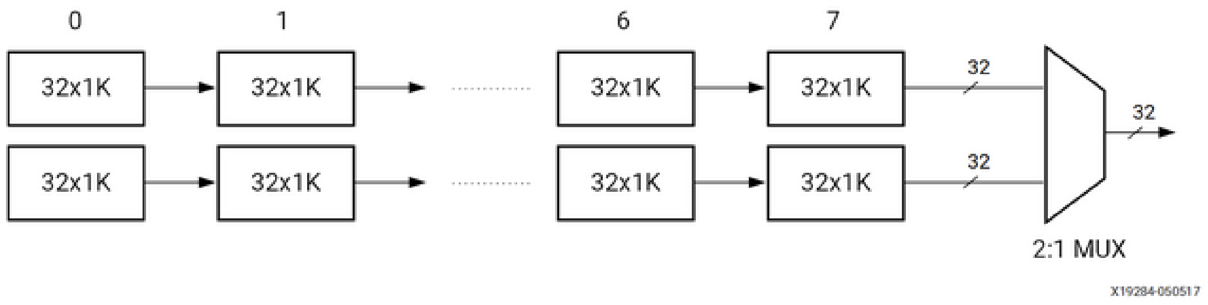


```
module test
(
input  clk,
input  we,
input  [13:0] addr,
input  [31:0] din,
output reg [31:0] dout
);

(* ram_style = "block", ram_decomp = "power", cascade_height = 4 *) reg  [31:0] mem [(16*1024)-1:0];
reg  [13:0] addr_reg;
reg  [31:0] dout_tmp0;
reg  [31:0] dout_tmp1;
reg  [31:0] din_reg;
reg         we_reg;

always @(posedge clk)
begin
    addr_reg <= addr;
    din_reg  <= din;
    we_reg   <= we;
    dout_tmp0 <= mem[addr_reg];
    dout_tmp1 <= dout_tmp0;
    dout <= dout_tmp1;
    if (we_reg)
        mem[addr_reg] <= din_reg;
end

endmodule
```

If you apply only the `ram_decomp = ″power″` attribute, 16 RAMB36E2 are inferred and the memory is decomposed as follows:

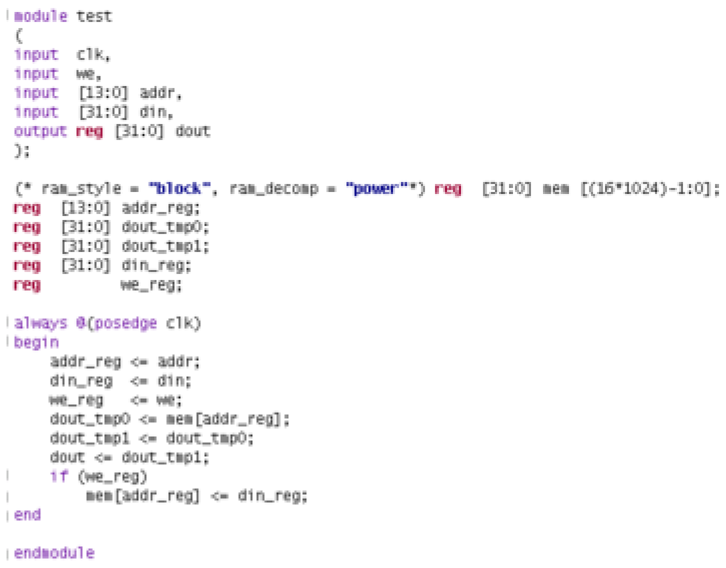
- The base primitive is 32x1K.
- 8 block RAMs are cascaded to create a 32x8K configuration.
- 2 parallel structures create a 16K deep memory.
- The outputs are multiplexed into a 2:1 MUX to generate the output data.

Figure: Generated Structure for 32x16K Memory Configuration Using the RAM_DECOMP Attribute



The following RTL code example shows the use of the RAM_DECOMP attribute.

Figure: RTL Code for 32x16K Memory Configuration Using the RAM_DECOMP Attribute



```
module test
(
input  clk,
input  we,
input  [13:0] addr,
input  [31:0] din,
output reg [31:0] dout
);

(* ram_style = "block", ram_decomp = "power"*) reg  [31:0] mem [(16*1024)-1:0];
reg  [13:0] addr_reg;
reg  [31:0] dout_tmp0;
reg  [31:0] dout_tmp1;
reg  [31:0] din_reg;
reg         we_reg;

always @(posedge clk)
begin
    addr_reg <= addr;
    din_reg  <= din;
    we_reg   <= we;
    dout_tmp0 <= mem[addr_reg];
    dout_tmp1 <= dout_tmp0;
    dout <= dout_tmp1;
    if (we_reg)
        mem[addr_reg] <= din_reg;
end

endmodule
```

If you use only the RAM_DECOMP attribute, the overall power savings is similar to using both the RAM_DECOMP and CASCADE_HEIGHT attributes together, because only one block RAM is active at a time. Creating a 4-deep cascaded block RAM chain is better for maximum clock frequency when compared to an 8-deep cascaded block RAM chain.
For more information, see this link in the *Vivado Design Suite User Guide: Synthesis* (UG901).

Running RTL DRCs

A set of RTL DRC rules identify potential coding issues with your HDL. You can perform these checks on the elaborated views, which you can open by clicking Open Elaborated Design in the Flow Navigator. You can run these DRC checks by selecting RTL Analysis > Report Methodology in the Flow Navigator or by executing `report_methodology` at the Tcl command prompt.