Verilog HDL Coding Style Proposal

N&C FAE Team

Coding Style Goal

- Logic Function Precision and Reliability
- Design for Rapid Simulation
- Best Trade-off between Circuit Size and Performance
- Good Readability and Migration
- Good Reusability

Agenda

- NAME CONVENTION
- BASIC CODING PRACTICE
- CLOCK AND RESETS RULE
- DESIGN FOR SYNTHESIS
- BASIC PARTITION RULE

NAME CONVENTION

BASIC_NAME_CONVENTION

- One design module per file and use module name as file name

Label	**G_1_1**
Description	**A file must not contain more than one design module. Everything contained in a design module must be completely contained in a single module/endmodule construct. It simplifies design modification.**
Type	**Block-level**
Recommend	**Medium**

BASIC_NAME_CONVENTION

- Use allowable character set as Verilog HDL code items name and start with a letter

Label	G_1_2
Description	Name must be composed of alphanumeric characters or underscores [A-Z, a-z, 0-9,_]. But consecutive underscores are not allowed because double underscores will not work with hardware emulation.
Type	Block-level
Recommend	Medium


Good coding style
Counter, wr_addr_1, data_in_dly1


Bad coding style
9_data, wr__addr, _func_write

BASIC_NAME_CONVENTION

- Use lowercase letters for all signal reg, net and port names and all the names must be meaningful

Label	**G_1_3**
Description	**Verilog is case sensitive (for example, counter, current_stat, wr_gray_adr) .**
Type	**Block-level**
Recommend	**Low**

Good coding style

```
reg  [7:0] counter     ;
wire [3:0] wr_addr_1 ;
reg        data_in_dly1
```

bad coding style

```
reg     [7:0] a_c          ;
input         DATA         ;
```

BASIC_NAME_CONVENTION

- Use uppercase letters for all parameter names and maximum length no more than 20

Label	**G_1_4**
Description	**If your design uses several parameters, use short but descriptive names. During elaboration, the synthesis tool concatenates the module's name, parameter names and parameter values to form the design unit name. Thus, lengthy parameter names can cause excessively long design unit names. parameter**
Type	**Block-level**
Recommend	**Medium**

BASIC_NAME_CONVENTION

- Do not use long name – Maximum name no more than 20 characters

Label	**G_1_5**
Description	**Make your design readable and understanding.**
Type	**Block-level**
Recommend	**Medium**

BASIC_NAME_CONVENTION

- Clock name should be clk or prefixed with clk

Label	G_1_6
Description	Use the name clk for the clock signal. If there is more than one clock in the design, use clk as the prefix for all clock signals (for example, clk1, clk2, or clk_pci).
Type	Block-level
Recommend	Medium

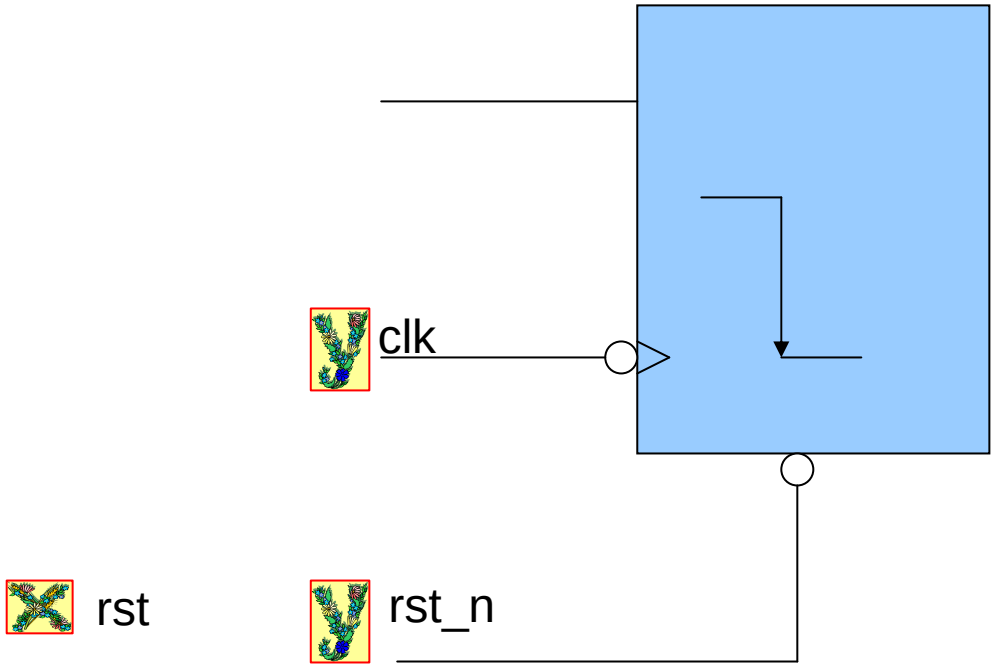
BASIC_NAME_CONVENTION

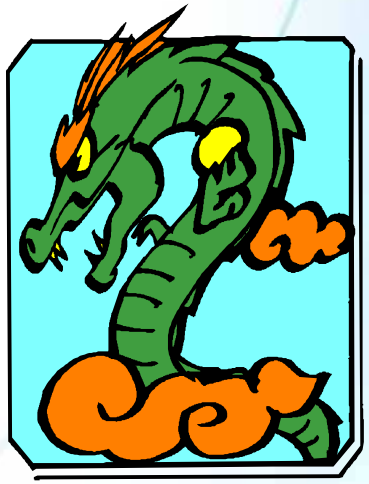
- Active high reset signal should be prefixed with rst; Active low reset should be named rst_n.

Label	**G_1_7**
Description	**Make reset signal name meaningful.**
Type	**Block-level**
Recommend	**Medium**

BASIC_NAME_CONVENTION

- Clock and Reset Signal Name Examples.





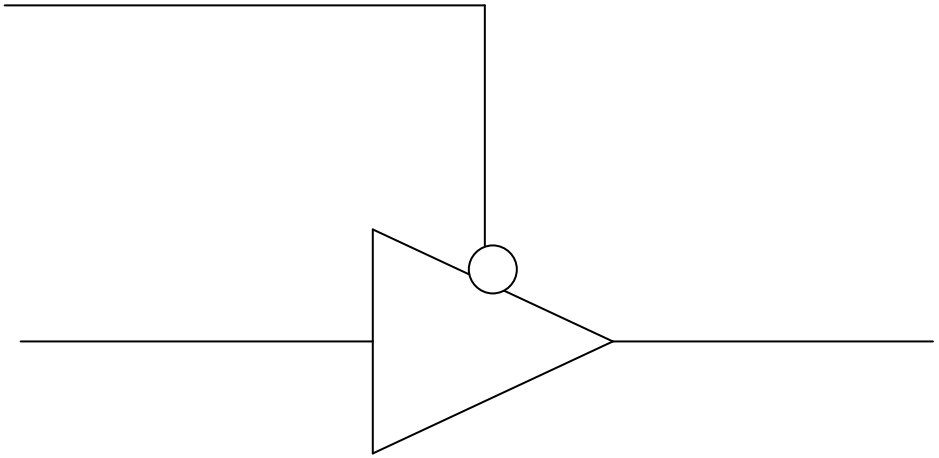
BASIC_NAME_CONVENTION

- Three state signal name should have suffix _z

Label	**G_1_8**
Description	**Make tri-state signal name meaningful.**
Type	**Block-level**
Recommend	**Low**

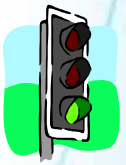
BASIC_NAME_CONVENTION

- Three state signal name should have suffix _z



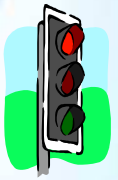
dout_z

Dout_z



Dout

Dout_n



BASIC_NAME_CONVENTION

- Always use descending range for multi-bit signals and ports and start with 0.

Label	**G_1_9**
Description	**Although the choice is somewhat arbitrary, it is recommended that you use [y:x] where y is greater than x and x=0. This recommendation is primary intended to establish a standard and thus achieve some consistency across multiple designs and design team-members.**
Type	**Block-level**
Recommend	**High**

BASIC_NAME_CONVENTION

- Signal Names Examples

- reg a[0:7]; Violation G_1_9
- reg a[8:1]; Violation G_1_9
- reg a[7:0];
- parameter INPUT_DATA_WIDTH_FOR_DESCRIPTION=16;
- parameter width_in = 16; Violation G_1_4
- parameter Width_In = 16; Violation G_1_4
- parameter WIDTH_IN = 16;
- input reset;
- There is control character inside a name string. Violation G_1_2

BASIC_NAME_CONVENTION

- Instance names should begin with ‘U_’

Label	**G_1_9**
Description	**In a multilayered design hierarchy, keep the labels short and meaningful. Long process and instance labels can cause excessively long path names in the design hierarchy.**
Type	**Block-level**
Recommend	**Medium**

BASIC_NAME_CONVENTION

- Instance names should begin with ‘U_’

```
my_count #(20, 5, 7)
        U_my_count   (

        U3   (

        CNT(

        V3   (
                .clk40      (clk40        ),
                .aclr       (reset        ),
                .clk_en     (1’b1         ),
                .cnt_en     (1’b1         ),
                .sclr       (frm_st       ),
                .c          (tslot_cnt_c),
                .q          (tslot_cnt    )
            );
```

BASIC_NAME_CONVENTION

- Verilog and VHDL reserved words can not be used as identifiers

Label	**G_1_10**
Description	**Do not use VHDL or Verilog reserved words for names of any elements in your RTL source files. Because macro designs must be translatable from VHDL to Verilog and from Verilog to VHDL, it is important not to use VHDL reserved words in Verilog code.**
Type	**Block-level**
Recommend	**High**

BASIC CODING PRACTICE

BASIC_CODING_PRACTICES

- Verilog file must have file header.

Label	R_2_1
Description	We recommend that the file header should include such fields: Filename, Author, Description, Date and Modification. All the header information make the source file reusable and maintainable.
Type	Block-level
Recommend	Medium

BASIC_CODING_PRACTICES

- Recommended File Header Example

```
/*----------------------------------------------------------------------------------
  This confidential and proprietary software may be only used as authorized
  by a licensing agreement from XXX Inc.
  (C) COPYRIGHT 2007 XXX INC. ALL RIGHTS RESERVED

  Filename    :    hdlc_top.v
  Author      :    Santos Chen
  Data        :    2007-03-08
  Version     :    0.1
  Description :    This file has the module of HDLC top.

  Modification History:
  Data         By              Version          Change Description
  ================================================================
  08/09/06    Frack Chen    0.1                 Original
  08/15/06    Peter Liu                         Add configuration ports
  ....
  ----------------------------------------------------------------------------------*/
```

BASIC_CODING_PRACTICES

- Function and task must have a header comment

Label	**R_2_2**
Description	**We recommend that each function or task should have a header comment to describe the function name, port and briefing. It makes the function or task reusable and maintainable.**
Type	**Block-level**
Recommend	**Low**

BASIC_CODING_PRACTICES

- Ports must be declared in two kinds of order - input inout output and function classification

Label	**R_2_3**
Description	**Declare ports in a logical order, and keep this order consistent throughout the design. We recommend that top level entity port declaration uses function classification and submodule uses input-inout-output sequence.**
Type	**Block-level**
Recommend	**Medium**

BASIC_CODING_PRACTICES

- Use a separate line for each HDL statement

Label	R_2_4
Description	Although Verilog allows more than one statement per line, the code is more readable and maintainable if each statement or command is on a separate line. Note that the check reports one error at a time per line.
Type	Block-level
Recommend	High

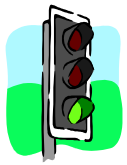
BASIC_CODING_PRACTICES

- Declare one port per line explicitly

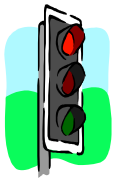
Label	**R_2_5**
Description	**Declare one port per line (preferably with a comment following it on the same line).**
Type	**Block-level**
Recommend	**Medium**

BASIC_CODING_PRACTICES

- One Verilog Statement per line



```
input     a;
input     b;
input     c;
```



```
input     a,b;
input     a,
          b;
```

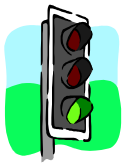
BASIC_CODING_PRACTICES

- Use named association when instantiating design units

Label	**R_2_6**
Description	**Connection by position is dangerous and can generate errors that are hard to debug when the instantiated cell port map changes. Connecting cells by name ensure that the connection is always valid even when there are changes in the instantiated cell's mapping.**
Type	**Block-level**
Recommend	**High**

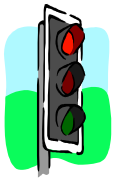
BASIC_CODING_PRACTICES

- Use named association when instantiating design units



Name_based

```
Instance    U_instance(
             .cs_n     (cs_n     ),
             .datain   (din      ),
             .dataout (dout     )
            );
```



Order_based

```
Instance    U_instance(
             cs_n      ,
             din       ,
             dout
            );
```

BASIC_CODING_PRACTICES

- Use four space code indentation

Label	R_2_7
Description	A constant indentation of four spaces must be used for code alignment. Do not use tab stop. Use space and empty lines to increase the readability of the code. The tab key of the text editor may be mapped to insert space and represent differently from one system to another.
Type	Block-level
Recommend	Low

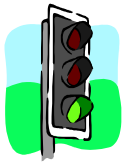
BASIC_CODING_PRACTICES

- One Verilog Statement per line

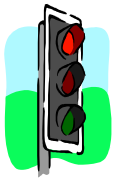
Label	R_2_8
Description	One line must not contain more than one statement. Do not concatenate multiple semicolon separated Verilog statement on the same line. It improves the code readability and easier be parsed with a design tool.
Type	Block-level
Recommend	High

BASIC_CODING_PRACTICES

- One Verilog Statement per line



```
upper_en  = (p5type & xadr1[0])  ;
lapper_en = (p5type & !xadr1[0]) ;
```



```
upper_en = (p5type & xadr1[0]);lapper_en = (p5type & !xadr1[0]);
```

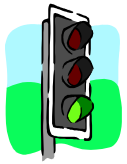
BASIC_CODING_PRACTICES

- Operand sizes should match

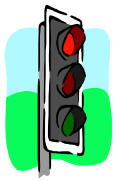
Label	R_2_9
Description	The operands of an operation are not recommended to differ in size. With different operand sizes the operand is not explicitly defined, but depends on how Verilog resolves the size differences. Verilog allows this since it is not a highly typed language.
Type	Block-level
Recommend	Medium

BASIC_CODING_PRACTICES

- One Verilog Statement per line



```
wire [63:0] bus_signal;
assign bus_signal = 64'b1;
```



```
wire [63:0] bus_signal;
assign bus_signal = 1;
```

BASIC_CODING_PRACTICES

- Do not assign signals to x

Label	R_2_10
Description	Signals must not be assigned to x. Known legal signal values must be assigned to all signals. Avoid x propagation through the circuit because there is not x-state existing in actual circuit. That will make subsequent logic confused.
Type	Block-level
Recommend	Medium

CLOCK AND RESETS RULE

CLOCK_AND_RESETS

- Avoid using both positive-edge and negative-edge triggered flip-flops in your design

Label	G_3_1
Description	The duty cycle of the clock becomes a critical issue in timing analysis, in addition to the clock frequency itself. Most scan-based testing methodologies require separate handling of positive- and negative-edge triggered flops. The compile tools will think the positive-edge and negative-edge clock as separated clocks.
Type	Chip-level
Recommend	Medium

CLOCK_AND_RESETS

- Buffers should not be explicitly added to clock path

Label	G_3_2
Description	Clock buffers are normally inserted after synthesis as part of the physical design. In synthesizable RTL code, clock nets are normally considered ideal nets, with no delays. During place and route, the clock tree insertion tool inserts the appropriate structure for creating as close to an ideal, balanced clock distribution network as possible. We recommend that insert the buffer by compile tools.
Type	Block-level
Recommend	Medium

CLOCK_AND_RESETS

- Gated clocks are not allowed in the design

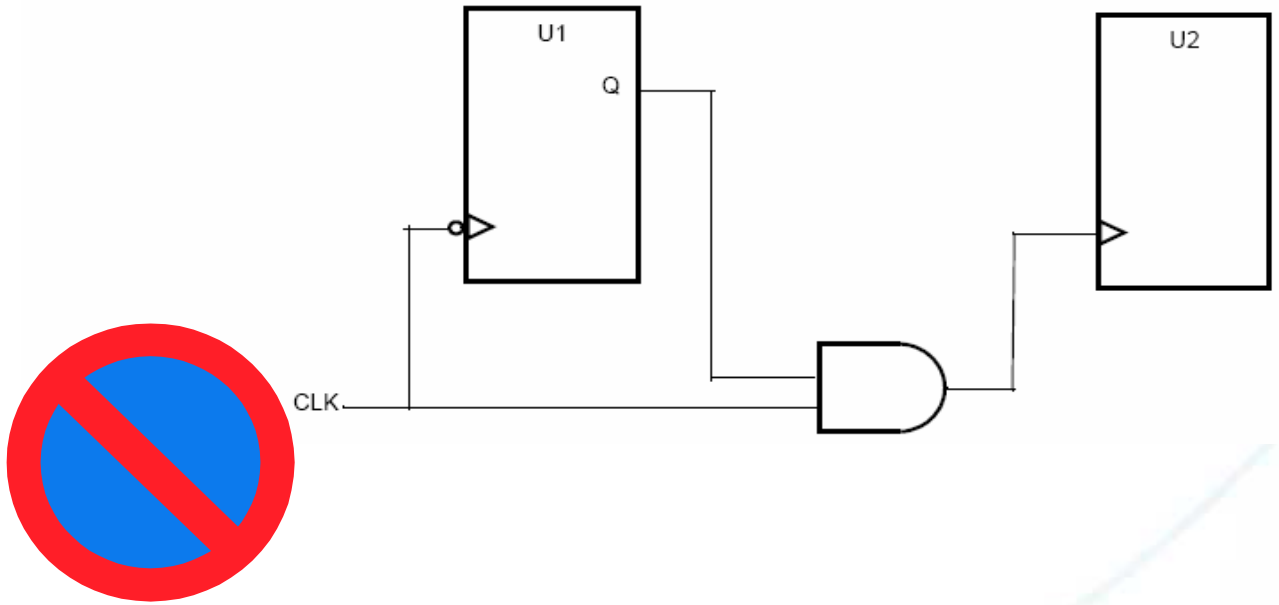
Label	G_3_3
Description	Improper timing of a gated clock can generate a false clock or glitch, causing a flip-flop to clock in the wrong data. The skew of different local clocks can cause hold time violations and gated clocks also cause limited testability. Although gated clocks are required, for many low-powered designs, they should not be coded in the RTL for a macro. If individual flip-flops need to be gated within a design, the clock gating should be inserted by a tool so that the RTL remains technology portable.
Type	Block-level
Recommend	Medium

CLOCK_AND_RESETS

- Gated clocks are not allowed in the design

The following is an example of an invalid circuit. It will have limited testability and skew problems because of the gated clock.

Note that U2 cannot be clocked during scan-in, test, or scan-out, and cannot be made part of the scan chain.



CLOCK_AND_RESETS

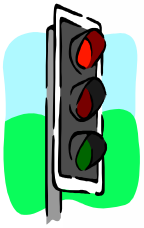
- Clock should be visible from top unit

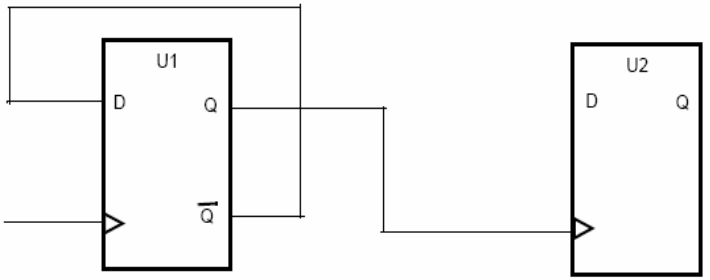
Label	G_3_4
Description	**All clocks have to be declared as input ports.** **Internally generated clocks cause limited testability and also make it more difficult to constrain the design for synthesis. We recommend that designer can use block PLL/DLL to generate the internal clocks which can be distinguished by compile tool better.**
Type	Chip-level
Recommend	Medium

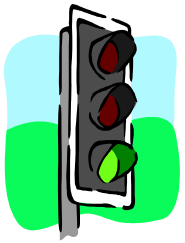
CLOCK_AND_RESETS

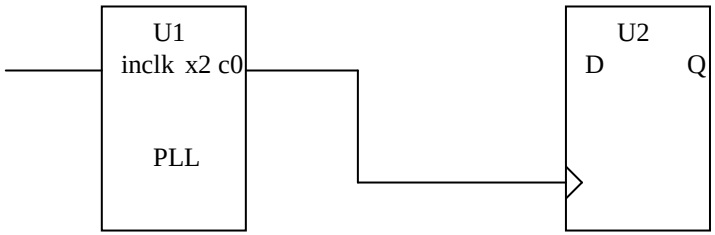
- Clock should be visible from top unit

The following is an example of an invalid circuit that has an internally generated clock









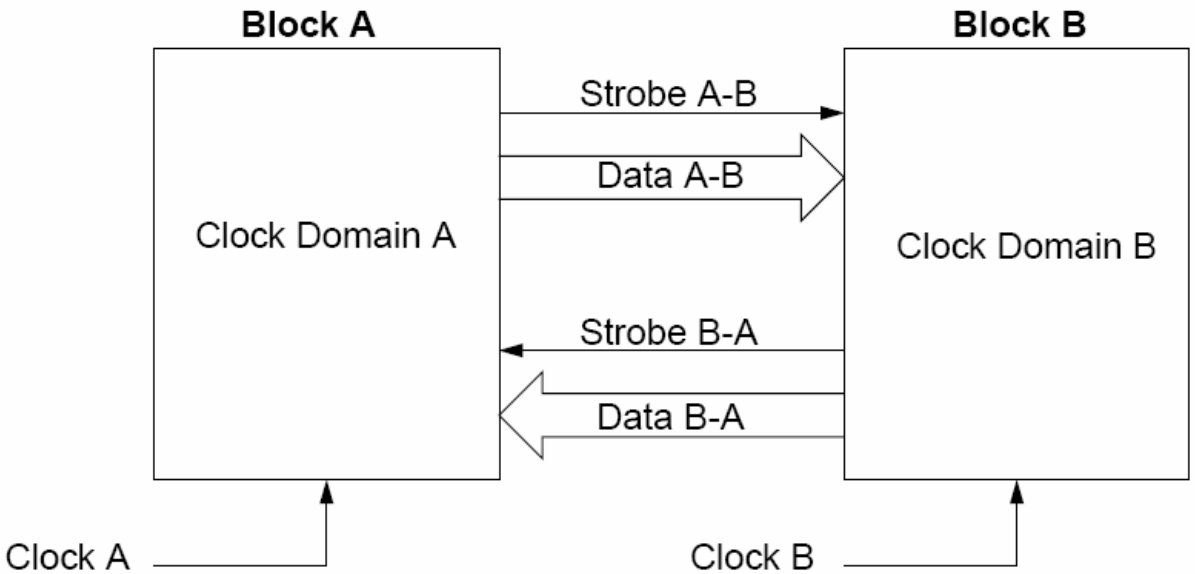
CLOCK_AND_RESETS

- Use synchronous methodology to cross clock domain

Label	G_3_5
Description	**Several synchronous method can help designer to cross data between different clock domain. The 2-register synchronizer is the most popular method. You can use handshake to guarantee the sent data can be sampled. The FIFO can be used as interface for multiple data bits crossing. The key point is to make the sampled data stable.**
Type	Block-level
Recommend	High

CLOCK_AND_RESETS

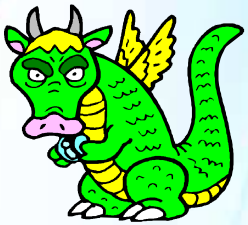
- Use synchronous methodology to cross clock domain



CLOCK_AND_RESETS

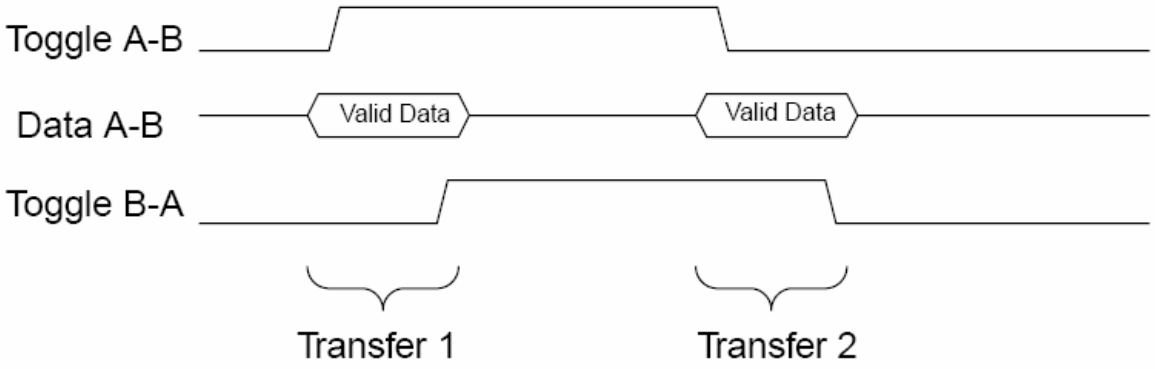
- Use synchronous methodology to cross clock domain - Solution

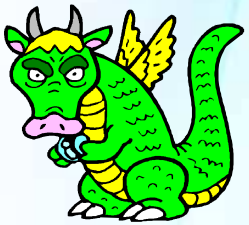
```
always @(posedge clka or negedge rst_n)
   if(rst_n == 1'b0)
      data_a <= 1'b0  ;
   else
      data_a <= data ;

always @(posedge clkb or negedge rst_n)
   if(rst_n == 1'b0) begin
      data_dly1 <= 1'b0;
      data_b    <= 1'b0;
   end
   else begin
      data_dly1 <= data_a    ;
      data_b    <= data_dly1;
   end
```



CLOCK_AND_RESETS

- Use synchronous methodology to cross clock domain – Example 2





CLOCK_AND_RESETS

- Apply approximate reset methodology in design

Label	G_3_6
Description	There are two kinds of reset method Synchronous reset asynchronous resets
Type	Block-level
Recommend	Medium

CLOCK_AND_RESETS

- Apply approximate reset methodology in design
  - Advantages of synchronous resets
    - Synchronous reset logic will be synthesized to smaller flip-flops, particularly if the reset if gated with the logic generating the d-input.
    - Synchronous resets generally insure that the circuit is 100% synchronous.
    - In some designs, the reset must be generated by a set of internal conditions. A synchronous reset is recommended for these types of designs because it will filter the logic equation glitches between clocks.

CLOCK_AND_RESETS

- Use synchronous methodology to cross clock domain
  - Disadvantages of synchronous resets
    - Not all ASIC libraries have flip-flops with built-in synchronous resets. However since synchronous reset is just another data input, you don't really need a special flop. The reset logic can easily be synthesized outside the flop itself.
    - Synchronous resets may need a pulse stretcher to guarantee a reset pulse width wide enough to ensure reset is present during an active edge of the clock.
    - Require a clock in order to reset the circuit.

CLOCK_AND_RESETS

- Use synchronous methodology to cross clock domain
  - Advantages of asynchronous resets
    - The biggest advantage favoring asynchronous resets is that the circuit can be reset with or without a clock present.
  - Disadvantages of asynchronous resets
    - The biggest problem is if the asynchronous reset is released at or near the active clock edge of a flip-flop, the output of the flip-flop could go to metastability.
    - noise or glitches sensitive .

CLOCK_AND_RESETS

- Use synchronous methodology to cross clock domain
  - Remove disadvantage for both asynchronous resets and synchronous resets
    - Asynchronous reset，synchronous release

CLOCK_AND_RESETS

- Use synchronous methodology to cross clock domain – coding example

```
Always @(posedge clk or negedge rst_n)
   if(rst_n == 1'b0) begin
      dff           <= 1'b0;
      rst_n_syn <= 1'b1;
   end
   else begin
      dff           <= 1'b1;
      rst_n_syn <= dff  ;
   end
```

CLOCK_AND_RESETS

- Avoid internally generated reset/set/load

Label	G_3_7
Description	Avoid internally generated, conditional resets, sets, and loads if possible. Generally, all the registers in the macro should be reset, set, or loaded at the same time. This approach makes analysis and design much simpler and easier. During test stages, all flip-flops must be controllable. If the reset, set, or load is internally generated and cannot be controlled at the boundary of the chip, either the test coverage will decrease or the test time will increase (as the number of test patterns will increase).
Type	Block-level
Recommend	Medium

DESIGN FOR SYNTHESIS

DESIGN_FOR_SYNTHESIS

- The always keyword must be followed by an event list @(...) in a sequential block

Label	G_4_1
Description	Registers (flip-flops) are the preferred mechanisms for sequential logic. To maintain consistency and ensure correct synthesis, use the following template to infer technology-independent registers. Use the design's reset signal to initialize registered signals. Do not initialize the signal in the declaration; this can cause mismatches between pre- and post-synthesis simulation.
Type	Block-level
Recommend	Medium

DESIGN_FOR_SYNTHESIS

- Use 'if(<%context> == 'b0)' or 'if(<%context> == 'b1)' for synchronous reset/set/load expressions : <%context>

Label	**G_4_2**
Description	**Please refer to G_4_1.**
Type	**Block-level**
Recommend	**Medium**

DESIGN_FOR_SYNTHESIS

- Use 'if(<%context> == 'b1)' for rising edge asynchronous reset/set/load expressions

Label	G_4_3
Description	Please refer to G_4_1.
Type	Block-level
Recommend	Medium

DESIGN_FOR_SYNTHESIS

- Use 'if(<%context> == 'b0)' for falling edge asynchronous reset/set/load expressions

Label	**G_4_4**
Description	**Please refer to G_4_1.**
Type	**Block-level**
Recommend	**Medium**

DESIGN_FOR_SYNTHESIS

- Do not use initial constructs to initialize signals

Label	G_4_5
Description	Initial constructs are ignored during synthesis. Using initial constructs to initialize signals instead of using a resets in the always constructs generates flip-flops without resets. Initial constructs also cause synthesis/simulation mismatches.
Type	Block-level
Recommend	Medium

DESIGN_FOR_SYNTHESIS

- There should be exactly one clock signal in the sensitivity list of a sequential block

Label	G_4_6
Description	Using multiple clocks in the sensitivity list can produce non-synthesizable code and generate local meta-stability issues. It is better to use only one clock signal and one reset signal per process/always construct.
Type	Block-level
Recommend	High

DESIGN_FOR_SYNTHESIS

- There should be at most one asynchronous reset/set/load signal in a sequential block

Label	**G_4_7**
Description	**Please refer to G_4_1.**
Type	**Block-level**
Recommend	**Medium**

DESIGN_FOR_SYNTHESIS

- An asynchronous reset/set/load signal should be preceded by the keyword 'posedge' or 'negedge' in the sensitivity list of a sequential block

Label	G_4_8
Description	Please refer to G_4_1.
Type	Block-level
Recommend	High

DESIGN_FOR_SYNTHESIS

- There should be at most one synchronous reset/set/load signal in a sequential block

Label	**G_4_9**
Description	**Please refer to G_4_1.**
Type	**Block-level**
Recommend	**Medium**

DESIGN_FOR_SYNTHESIS

- Always block with event and level expression detected in sensitivity list. This block is not synthesizable.

Label	**G_4_10**
Description	**Please refer to G_4_1.**
Type	**Block-level**
Recommend	**High**

DESIGN_FOR_SYNTHESIS

- A Verilog template for sequential processes

```
// Process with synchronous reset
always @(posedge clk)
  begin : EX20A_PROC
          if (reset == 1'b1)
          begin
          . . .
          end
          else
          begin
          . . .
          end
  end // EX20A_PROC
```

```
// Process with asynchronous reset
always @(posedge clk or posedge rst_a)
  begin : EX20B_PROC
          if (reset == 1'b1)
          begin
          . . .
          end
          else
          begin
          . . .
          end
  end // Ex20b_proc
```

DESIGN_FOR_SYNTHESIS

- Avoid inferring latches in design

Label	G_4_11
Description	Avoid using latches in your design. All latches must be instantiated, and you must provide documentation that lists each latch and describes any special timing requirements that result from the latch.
Type	Block-level
Recommend	Medium

DESIGN_FOR_SYNTHESIS

- Avoid inferring latches in design

Latches inferred because of missing s output assignments for the 2'b00 and 2'b01 conditions and a missing 2'b11 condition

```
always @ (d)
begin
  case (d)
    2'b00: z <= 1'b1;
    2'b01: z <= 1'b0;
    2'b10: z <= 1'b1; s <= 1'b1;
  endcase
end
```

DESIGN_FOR_SYNTHESIS

- Avoid inferring latches in design
  - Avoiding inferred latches by using any of the following coding techniques:
    - Assign default values at the beginning of a process
    - Assign outputs for all input conditions
    - Use else (instead of elsif) for the final priority branch

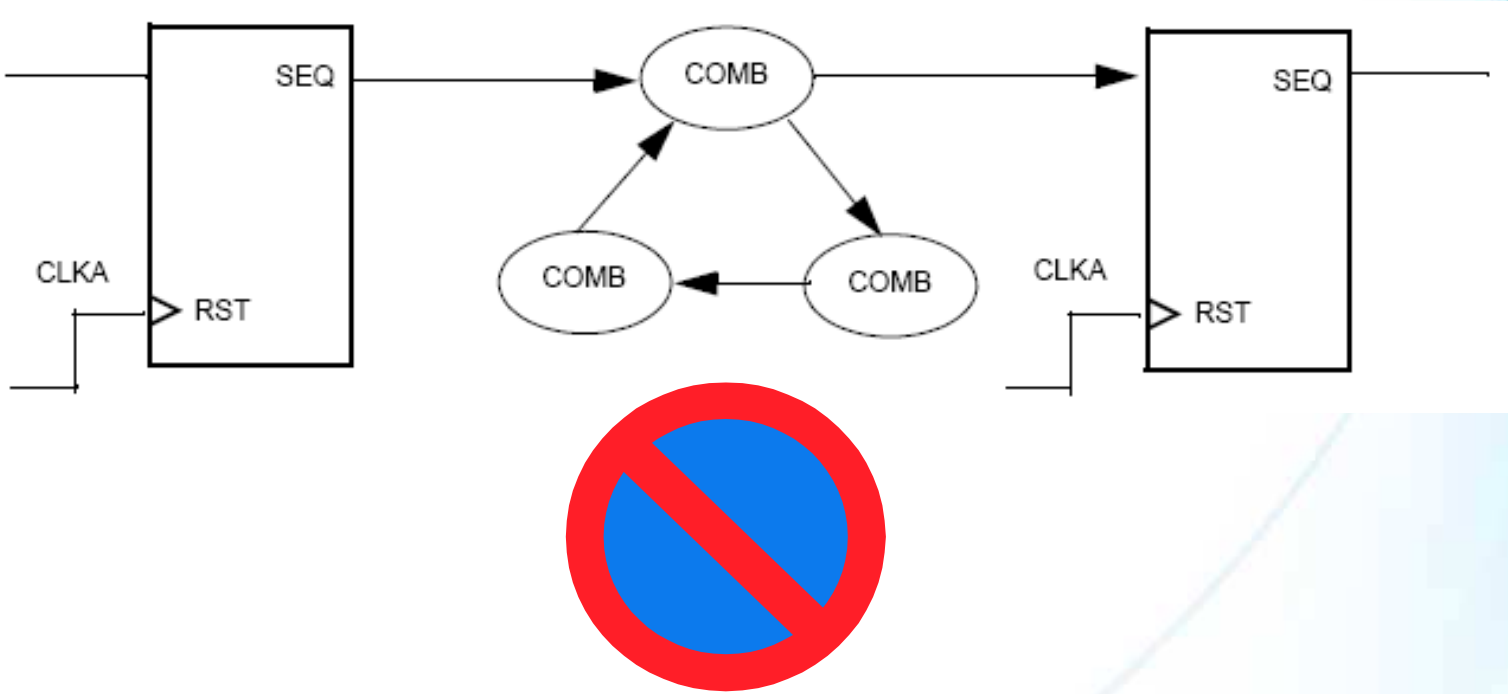
DESIGN_FOR_SYNTHESIS

- Avoid asynchronous feedback loops

Label	G_4_12
Description	Asynchronous feedback loops generate problems in different steps of the design flow. They increase simulation runtime and are broken during synthesis/timing analysis. Asynchronous feedback loops are also timing-dependent and can be difficult to analyze/debug.
Type	Chip-level
Recommend	Medium

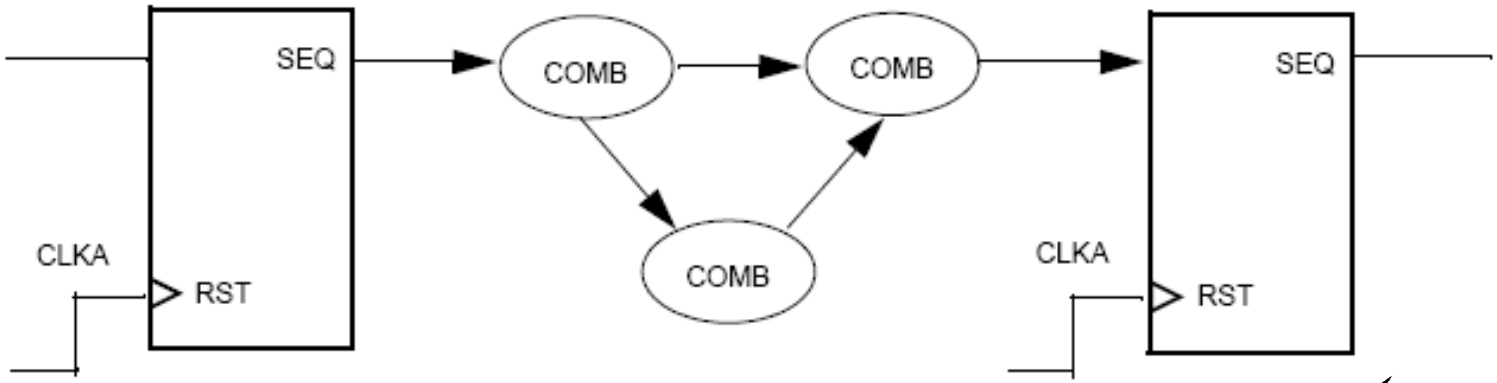
DESIGN_FOR_SYNTHESIS

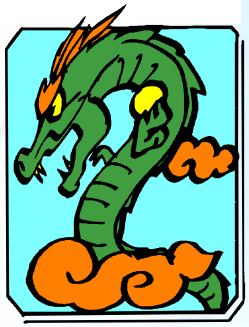
- Avoid asynchronous feedback loops
  - **Invalid: Combinational processes are looped – Bad Design**



DESIGN_FOR_SYNTHESIS

- Avoid asynchronous feedback loops
  - **Valid: Combinational processes are not looped – Good**





DESIGN_FOR_SYNTHESIS

- Redundant signals in sensitivity list

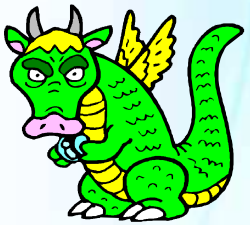
Label	G_4_13
Description	Include a complete sensitivity list in each of your Verilog always blocks and make sure your sensitivity lists contain only necessary signals. Adding unnecessary signals to the sensitivity list slows down simulation.
Type	Block-level
Recommend	High

DESIGN_FOR_SYNTHESIS

- Redundant signals in sensitivity list
  - **For combinational blocks (blocks that contain no registers or latches), the sensitivity list must include every signal that is read by the process. In general, this means every signal that appears on the right side of an assign (=) statement or in a conditional expression.**

Valid Verilog Code:

```
always @(a or inc_dec)
begin : COMBINATIONAL_PROC
  if (inc_dec == 0)
    sum = a + 1;
  else
    sum = a - 1;
end  // COMBINATIONAL_PROC
```

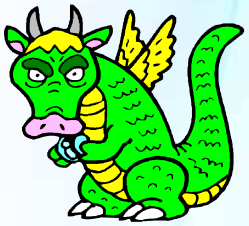


DESIGN_FOR_SYNTHESIS

- Redundant signals in sensitivity list
  - **For sequential blocks, the sensitivity list must include the clock signal that is read by the process, as shown in the following example. If the sequential process block also uses a reset signal, include the reset signal in the sensitivity list.**

Valid Verilog Code:

```
always @(posedge clk)
begin : SEQUENTIAL_PROC
  q <= d;
end  // SEQUENTIAL_PROC
```



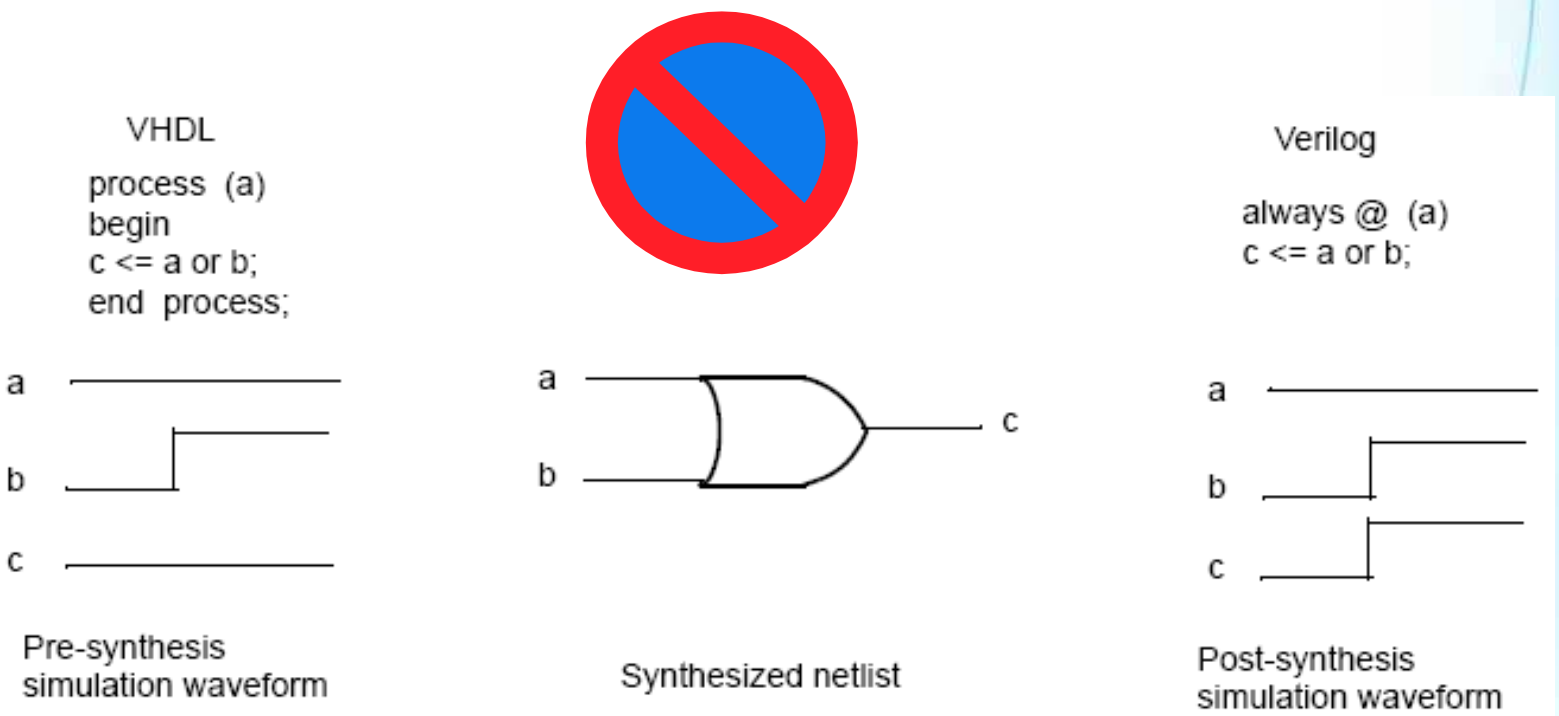
DESIGN_FOR_SYNTHESIS

- Missing signals in sensitivity list

Label	G_4_14
Description	Include a complete sensitivity list in each of your Verilog always blocks and make sure your sensitivity lists contain only necessary signals. Adding unnecessary signals to the sensitivity list slows down simulation. If you do not use a complete sensitivity list, the behavior of the pre-synthesis design may differ from that of the post-synthesis netlist, as illustrated in the figure.
Type	Block-level
Recommend	High

DESIGN_FOR_SYNTHESIS

- Missing signals in sensitivity list
  - **Invalid examples generate simulation mismatches because of incomplete sensitivity. Pre-synthesis and Post-synthesis simulation mismatch.**



DESIGN_FOR_SYNTHESIS

- Use non-blocking assignments in sequential always blocks

Label	G_4_15
Description	None.
Type	Block-level
Recommend	High

DESIGN_FOR_SYNTHESIS

- Use blocking assignments in combinational always blocks

Label	G_4_16
Description	None
Type	Block-level
Recommend	High

DESIGN_FOR_SYNTHESIS

- Code state machines description with two or there blocks

Label	**G_4_17**
Description	**Coding state machines in two blocks (one combinatorial process dedicated to the generation of the next state and one sequential process for the state register) makes the code easier to understand and debug.**
Type	**Block-level**
Recommend	**Medium**

DESIGN_FOR_SYNTHESIS

- Use parameter statements to define the state vector of a state machine

Label	**G_4_18**
Description	**Using parameter statements makes the code more readable and easier to understand/debug than using literals.**
Type	**Block-level**
Recommend	**Low**

DESIGN_FOR_SYNTHESIS

- In state machine, keep FSM logic and non-FSM logic apart

Label	**G_4_19**
Description	**Keep finite state machine (FSM) logic and non-FSM logic in separate modules.**
Type	**Block-level**
Recommend	**Medium**

DESIGN_FOR_SYNTHESIS

- Assign a default state to the state machine

Label	G_4_20
Description	Assign a default state for the state machine. This is useful to implement graceful entry into the idle state if no other state is initiated. For Verilog, assign a “default” state as shown in the following examples.
Type	Block-level
Recommend	Medium

DESIGN_FOR_SYNTHESIS

- Code state machines description

```
module fsm(clock, rst_n, condition, result);
input clock, rst_n, condition;
output result;

reg current_state, next_state;
reg result;

parameter STATE_0 = 0,
          STATE_1 = 1;

always @ ( posedge clock or negedge rst_n) begin
 if (rst_n == 1'b0)
  result <= 1'b0;
 else begin
  case(current_state)
   STATE_0 : result <= 1'b0;
   STATE_1 : result <= 1'b1;
   default : result <= 1'b0;
  endcase
 end
end
```

```
always @ ( posedge clock or negedge rst_n) begin
 if (rst_n == 1'b0)
  current_state <= STATE_0;
 else
  current_state <= next_state;
end

// combinational process calculates next state
always @ (current_state or condition) begin
 case(current_state)
  STATE_0 : begin
   if (condition) next_state = STATE_1;
   else  next_state = STATE_0;
  end
  STATE_1 : begin
   if (condition) next_state = STATE_0;
   else  next_state = STATE_1;
  end
  default : next_state = STATE_0;
 endcase
end

endmodule
```

BASIC PARTITION RULE

BASIC_PARTITION_RULE

- Locate related combinational logic in a single module

Label	G_5_2
Description	Keep related combinational logic together in the same module. Synthesis tools have more flexibility in optimizing a design when related combinational logic is located in the same module. This is because synthesis tools cannot move logic across hierarchical boundaries during default compile operations.
Type	Chip-level
Recommend	Medium

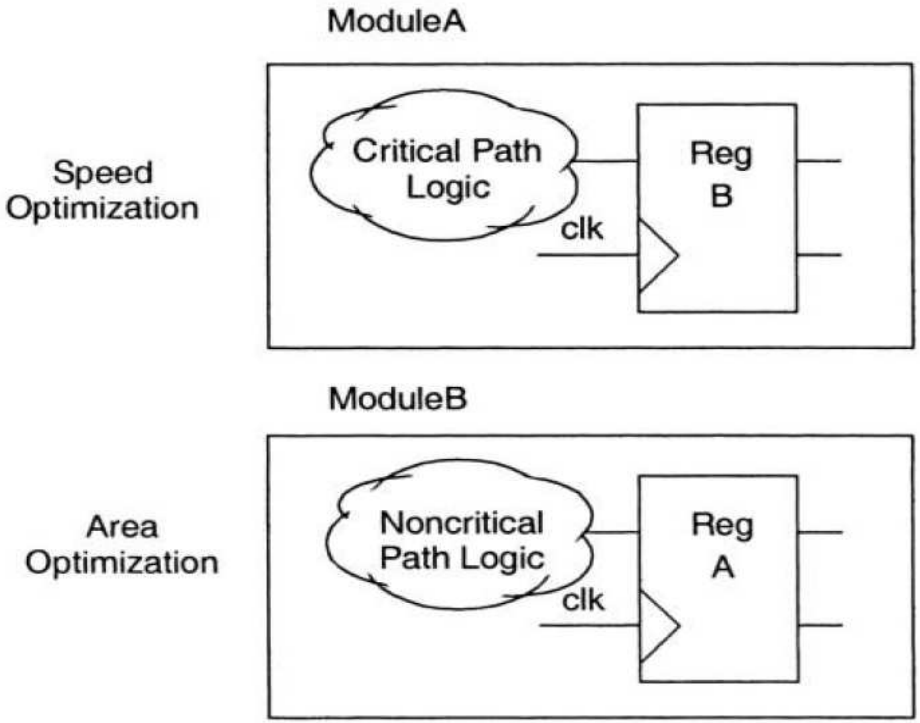
BASIC_PARTITION_RULE

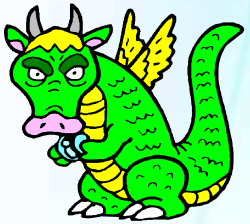
- Separate Modules That Have Different Design Goals

Label	**G_5_3**
Description	**Separate Modules That Have Different Design Goals.**
Type	**Chip-level**
Recommend	**Medium**

BASIC_PARTITION_RULE

- Separate Modules That Have Different Design Goals (cont.)





BASIC_PARTITION_RULE

- Avoid using asynchronous logic

Label	**G_5_4**
Description	**Avoid asynchronous logic. Asynchronous logic is more difficult to design correctly and verify. Correct timing and functionality may be technology dependent, which limits the portability of the design.**
Type	**Chip-level**
Recommend	**Low**

BASIC_PARTITION_RULE

- Partition asynchronous logic from synchronous logic

Label	**G_5_5**
Description	**If asynchronous logic is required in the design, partition the asynchronous logic in a separate module from the synchronous logic. Isolating the asynchronous logic in a separate module makes code inspection much easier. Asynchronous logic needs to be reviewed carefully to verify its functionality and timing.**
Type	**Chip-level**
Recommend	**Medium**

BASIC_PARTITION_RULE

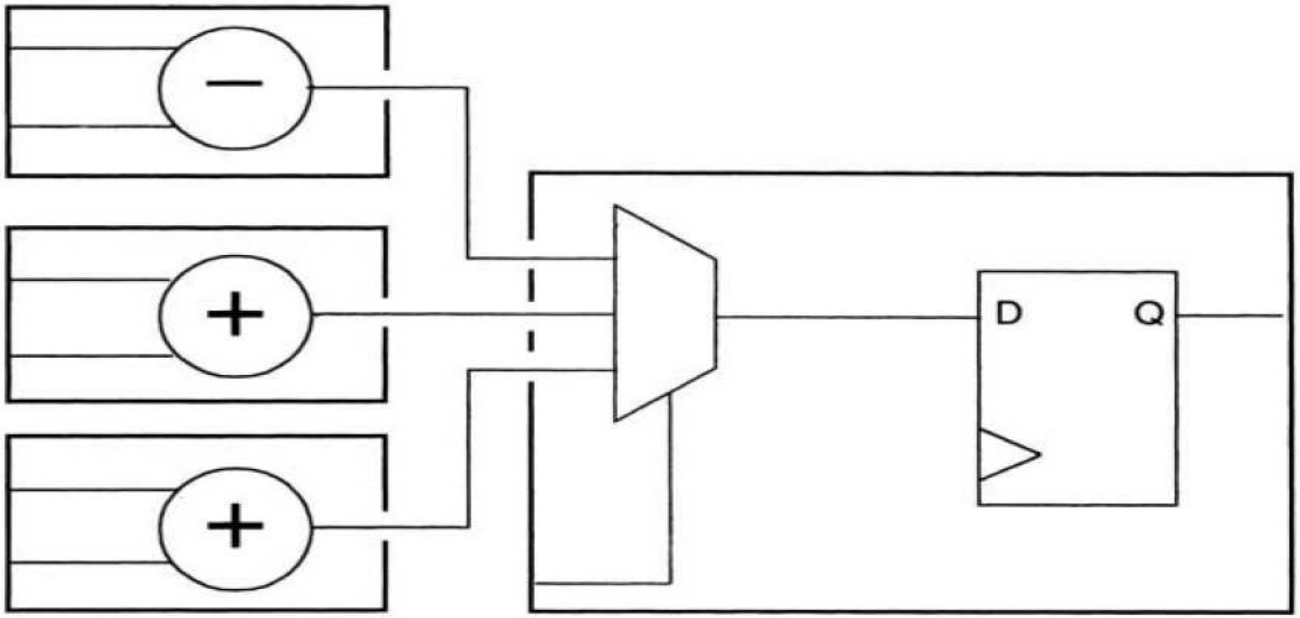
- Arithmetic operators: Merging resources

Label	G_5_6
Description	All relevant resources need to be in the same level of hierarchy. For synthesis tools to consider resource sharing, all relevant resources need to be in the same level of hierarchy; that is, within the same module.
Type	Block-level
Recommend	Low

BASIC_PARTITION_RULE

- **Arithmetic Operators: Merging Resources (cont.)**

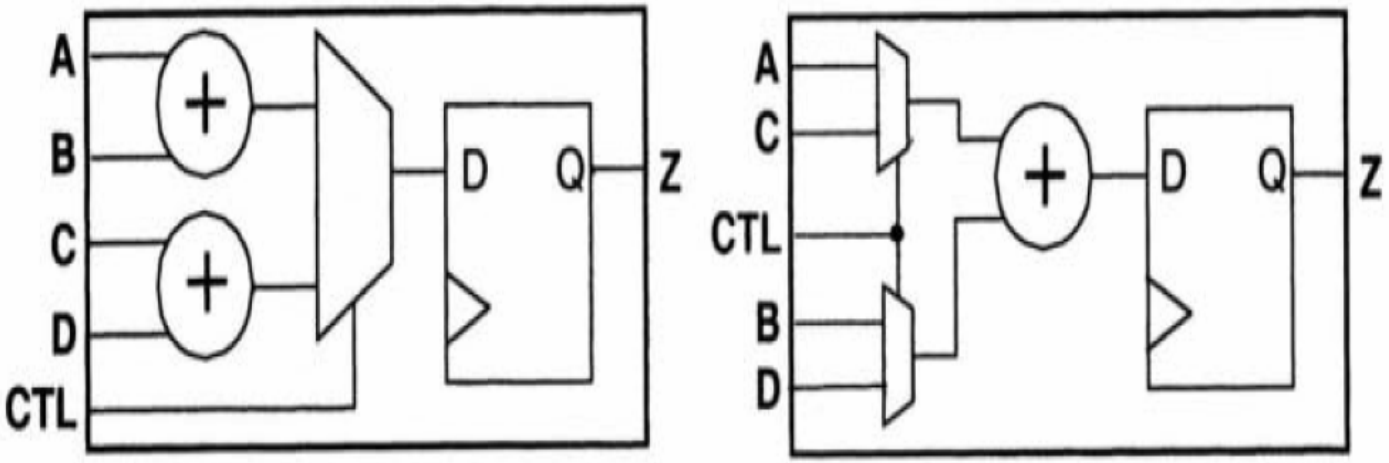
Poor Partitioning



BASIC_PARTITION_RULE

- **Arithmetic Operators: Merging Resources (cont.)**

Good Partitioning



BASIC_PARTITION_RULE

- Avoid timing exceptions

Label	G_5_7
Description	Avoid multi-cycle paths and other timing exceptions in your design. Timing exceptions are problematic because: They are difficult to analyze correctly and lend themselves to human error. They must be marked as exceptions to all of the design tools, each of which may have its own format and limitations for specifying exceptions.
Type	Chip-level
Recommend	Low

BASIC_PARTITION_RULE

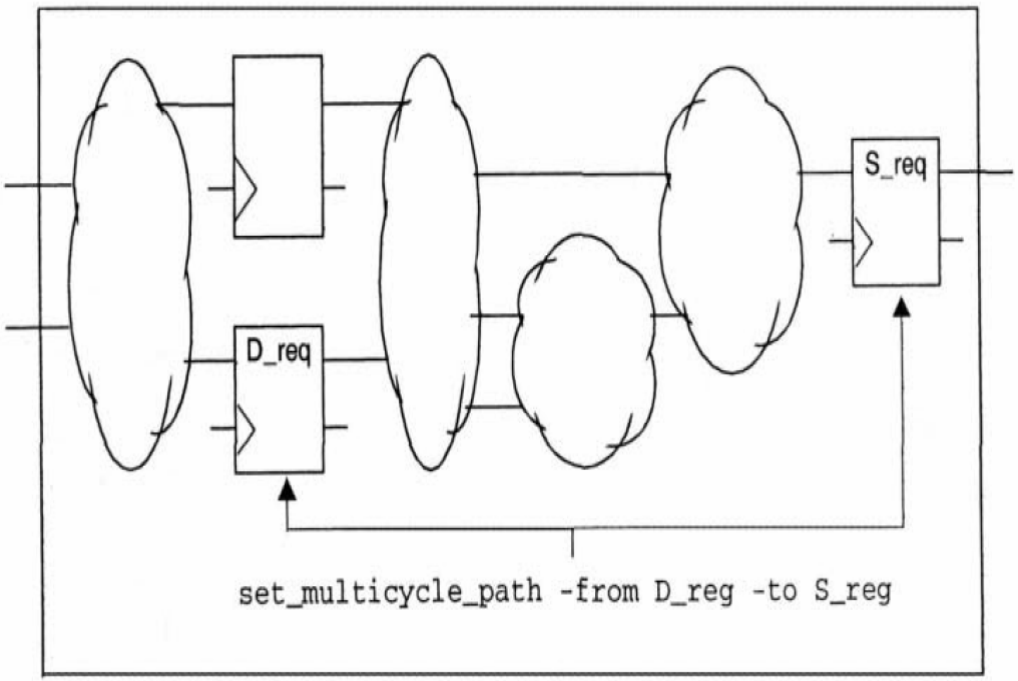
- Avoid false paths exceptions

Label	G_5_8
Description	Avoid false paths in your design. False paths are paths that static timing analysis identifies as failing timing, but that the designer knows are not actually failing. False paths are a problem because they require the designer to ignore a warning message from the timing analysis tool. If there are many false paths in a design, it is easy for the designer accidently to ignore valid warning message about actual failing paths. If it is necessary to use false paths, follow the multicycle path guidelines in this section.
Type	Chip-level
Recommend	Low

BASIC_PARTITION_RULE

- Avoid timing exceptions

Good example



BASIC_PARTITION_RULE

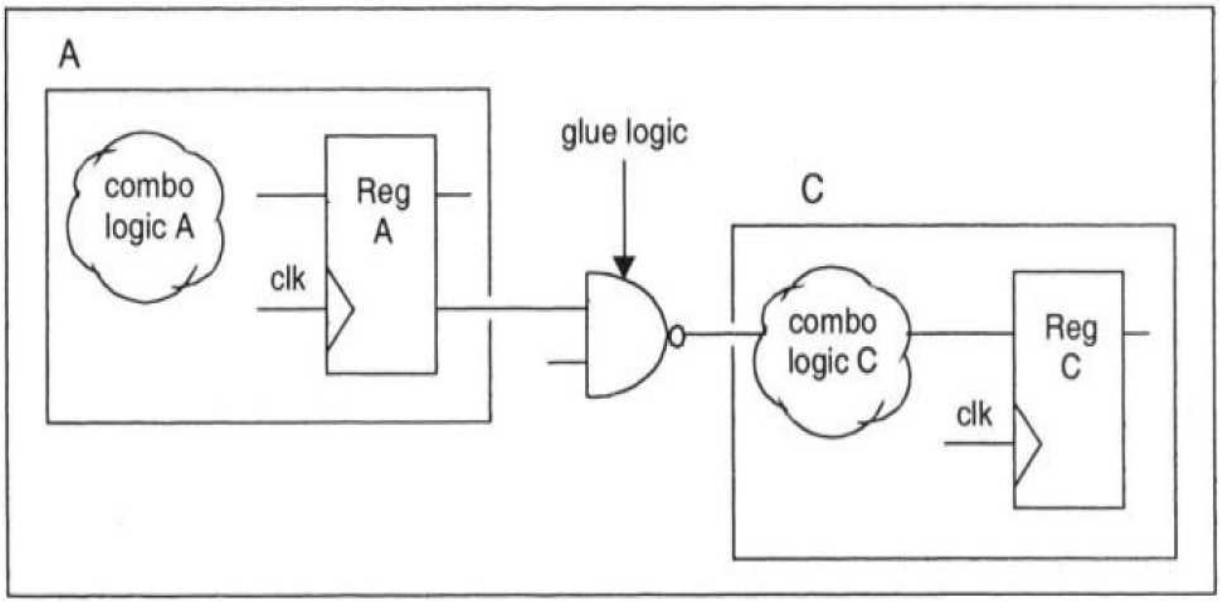
- Eliminate glue logic at the top level

Label	G_5_9
Description	Do not instantiate gate-level logic at the top level of the macro hierarchy. A design hierarchy should contain gates only at leaf levels of the hierarchy tree.
Type	Chip-level
Recommend	Medium

BASIC_PARTITION_RULE

- **Eliminate Glue Logic at the Top Level (cont.)**

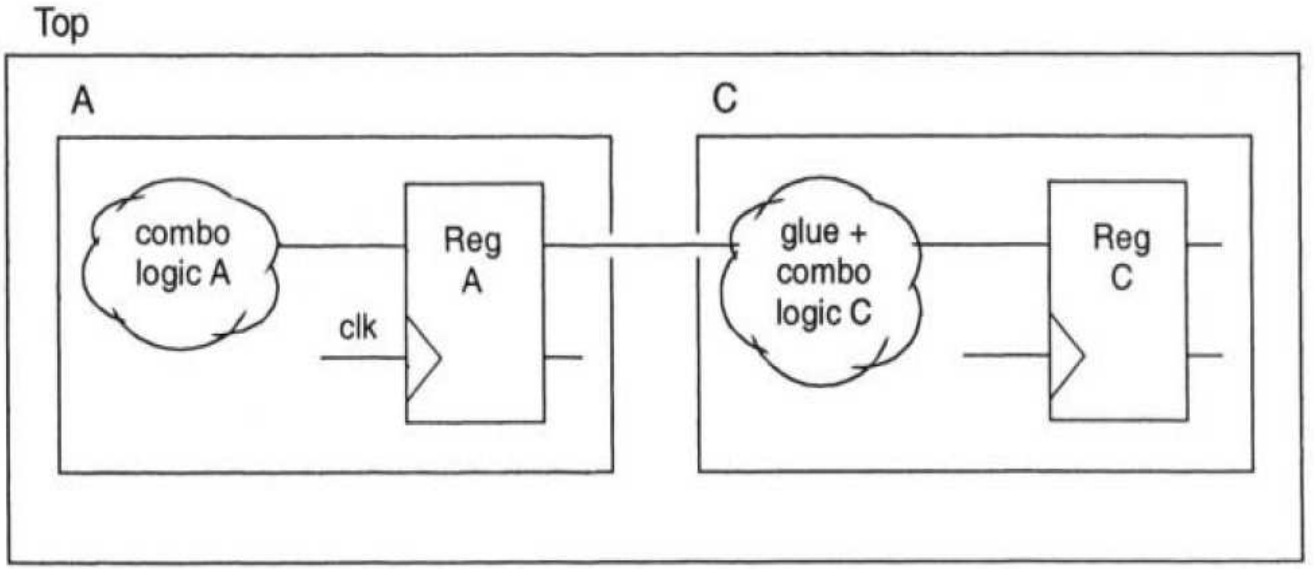
Poor partition



BASIC_PARTITION_RULE

- Eliminate glue logic at the top level (cont.)

Good partition



BASIC_PARTITION_RULE

- Separate the data path section from the controller

Label	**G_5_10**
Description	**Separate the data path section from the controller. Description styles are different for the data path section and the controller. Different synthesis methods can be chosen for the data path section and the controller.**
Type	**Black-level**
Recommend	**Low**

Coding Clock Enables Correctly

- LAB-wide clock enables allow clock gating at LAB level
  - All logic in LAB driven by clock is affected
- Shutting off LAB-wide clock enable reduces LAB power consumption
  - Global clock unaffected
- Clock enables automatically promoted to LAB-wide controls if coded correctly
  - Enable should be a single signal, not an equation
  - Follow control signal priority
    - aclr, pre, aload, ena, sclr, sload (Highest → Lowest)

Coding Enables Examples (VHDL)

```
enable <= (a XOR b) AND c;

PROCESS (aclr,clk)
BEGIN
   If aclr = '1' THEN
      reg <= (OTHERS => '0');
   ELSIF rising_edge (clk) THEN
      IF enable = '1' THEN
         reg <= new_value;
      END IF;
   END IF;
END PROCESS;
```

If equation is needed to evaluate clock enable, use separate statement

Coding Enables Examples (Verilog)

```
assign enable = (a ^ b) & c;

always @ (posedge clk or posedge aclr)
begin
   if (aclr)
      reg <= 0;
   else if (enable)
      reg <= new_value;
end;
```

If equation is needed to evaluate clock enable, use separate statement