Verilog RTL Coding Guideline

June 2009

Ando KI

Contents

Purposes of coding guidelines

- HDL code should be
  - readable,
  - modifiable, and
  - reusable.
- As a result, the HDL code is optimal result in synthesis and simulation.

Principles of coding guidelines

- To develop RTL code that is simple and regular.
  - Simple and regular structures are easier to design, code, verify, and synthesis than more complex designs.
- General recommendations
  - Use simple construct, basic types, and simple clock schemes.
  - Use a consistent coding style, consistent naming conventions, and a consistent structure for processes and state machines.
  - Use a regular partitioning scheme, with all module outputs registered and with modules roughly of the same size.
  - Make the RTL code easy to understand, by using comments, meaningful names and constants or parameters instead of hard-coded numbers.

Categories of coding guidelines

- Basic coding practices
- Coding for portability
- Guidelines for clocks and resets
- Coding for synthesis
- Partitioning for synthesis

Basic coding practices

- Addressing basic coding practices, focusing on lexical conventions and basic RTL constructs.

- General naming conventions
- Including headers in source files
- Use comments
- Keep commands on separate lines
- Line length
- Indentation
- Do not use HDL reserved words
- Port ordering
- Port maps and generic maps
- Use functions

General naming conventions (1/2)

- Rule: develop a naming convention for the design, which must be documented and used consistently throughout the design.
- Guidelines
  - Use lowercase letters for all signal names, variable names, and port names.
  - Use uppercase letters for names of constants and user-defined types.
  - Use meaningful names for signals, ports, functions, and parameters.
  - Use short name, but descriptive names for parameters.
  - Use ‘clk’ as the prefix for all clock signals.
  - Use ‘_n’ (or ‘_b’) as the postfix for all active all signals.
  - Use ‘rst’ for reset signals. If it is active low, then use ‘rst_n’.
  - Use a consistent ordering of bits for multi-bit buses.
  - Use the same name or similar names for ports and signals that are connected.

General naming conventions (2/2)

- Signal name conventions

convention	Use
*_b	Active low signal (rst_b)
*_r	Output of a register (count_r)
*_a	Asynchronous signal (addr_strobe_a)
*_pn	Signal used in the nth phase (enable_p2)
*_nxt	Data before being registered into a register with the same name
*_z	Trisate internal signal
clk*	Use ‘clk’ as a prefix for clock signals
rst*	Use ‘rst’ as a prefix for reset signals

Include headers in source files

- Rule: include a header file at the top of every source file, including scripts.
- Syntax of header
  - Filename
  - Author
  - Description of functions
  - List of key features
  - Date the file was created
  - Modification history including date, name of modifier, and description of the change

```
/************************************************
 * Copyright (c) 2005 by Ando Ki.
 * All right reserved.
 *
 * http://www.dynalith.com
 ************************************************
 * File: euclide.v
 * Author: Ando Ki
 * Abstract: Euclide GCD (greatest common divisor)
 *           computes the greatest common divisor
 *           of two non-negative integers.
 * Date: 2005.09.30
 * Revision history:
 *      2005.10.01  Ando Ki   0.2  refinement
 *      2005.09.20  Ando Ki   0.1  original
 ************************************************
 */

module GCD ( clk, reset, a, b, valid, c, done);
  ...
  ...
endmodule
```


Use comments

- Rule: use comments appropriately to explain all processes, functions, and declarations of types and subtypes.
- Guidelines
  - Use comments to explain ports, signals, and variables, or groups of signals or variables.
  - Comments should be brief, concise, and explanatory.
  - Comments should be placed logically, near the code that they describe.

Keep commands on separate lines

- Rule: use a separate line for each HDL statement.
  - In order to be more readable and maintainable.

Line length

- Rule: keep the line length to 72 character or less.

Indentation

- Rule: use indentation to improve the readability of continued code lines and nested loops.
- Guideline
  - Use indentation of 2 spaces.
    - Large indentation restricts line length when there are several levels of nesting.
  - Avoid using tabs.
    - Differences in editors and user setups make the positioning of tabs unpredictable and can corrupt the intended indentation.

Do not use HDL reserved words

- Rule: do not use HDL (Verilog and VHDL) reserved words for names of any elements in RTL source files.
  - Macro designs must be translatable from Verilog to VHDL and vice verse.

Port ordering

- Rule: declare ports in a logical order, and keep this order consistent throughout the design.
- Guideline
  - Declare one port per line, with a comment following it.
  - Declare the ports in the following order.
    - inputs
      - Clocks, resets, enables, controls, data and address
    - Outputs
      - Clocks, resets, enables, controls, data

Port maps and generic maps

- Guideline: always use explicit mapping for ports and generics, using named association rather than positional association.
- Guideline
  - Leave a blank line between the input and output ports to improve readability.

```
//-------------------------------------------------------
gcd Ugcd (
        .clk    (clk),
        .reset(reset),
        .A      (A),
        .B      (B),
        .valid  (valid),
        .C      (C),
        .done   (done)
 );
```

Use functions

- Guideline
  - Use functions when possible, instead of repeating the same sections of code.

```
//-----------------------------------------------------
function [`WIDTH-1:0] conv_addr;
   input  [`WIDTH-1:0] input_addr;
   input                offset;
   begin
      ...
      ...
   end
endfunction
```

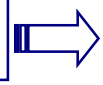

Coding for portability

- Create code that is technology-independent, compatible with various simulation tools, and easily translatable from Verilog to VHDL (or from VHDL to Verilog).
- Do not use hard-coded numeric values
- Include files
- Avoid embedding dc_shell scripts
- Use technology-independent libraries

Do not use hard-coded numeric values

- Guideline
  - Do not use hard-coded numeric values.
    - Use constants instead of hard-coded values

```
//--------------------------------
 wire [7:0] my_in_bus;
 reg  [15:0] my_reg;
```



```
`define MY_BUS_SIZE 8
`define MY_REG_SIZE 16
parameter MY_BUS_SIZR = `MY_BUS_SIZE
parameter MY_REG_SIZE = `MY_REG_SIZE
//-----------------------------------------
  wire [MY_BUS_SIZE-1:0] my_in_bus;
  reg  [MY_REG_SIZE-1:0] my_reg;
```


Include files

- Guideline
  - Keep the `define statements for a design in a single separate file and name the file DesignName_params.v.

Avoid embedding dc_shell scripts

- Guideline
  - Avoid using dc_shell script in the source code in order to portability reason.

Use technology-independent libraries

- Guideline
  - Use the DesignWare Foundatioin Library to maintain technolgy independence.
  - Avoid instantiating gates in the design
    - Gate-level design is hard to read, and thus difficult to maintain and reuse.
    - Technology-specific gates make the design non-portable.
  - Use GTECH library, if gate must be used.
    - GTECH library (Synopsys generic technology library – one of technology-independent library) contains technology independent logical components
      - AND, OR, NOR, flip-flops, ..
- DesignWare Foundation Library includes the followings
  - Adders, multipliers, comparators, incrementers, and decrementers.
  - Sine, cos, modulus, divide, square root, arithmetic and barrel shifters.
- DesignWare library provides improved timing performance over the equivalent internal Design Compiler architectures.
- DesignWare library components are all high-performance designs that are portable across processes.

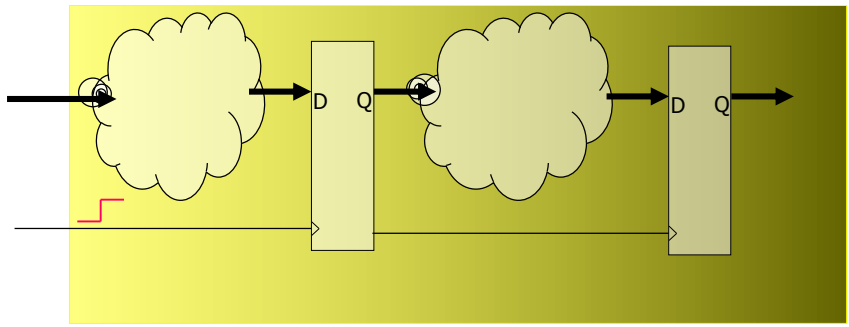
Guidelines for clocks and resets

- A simple clocking structure is easier to understand, analyze, and maintain.
- In addition to this, a simple clocking structure helps to produce better synthesis results.

- Avoid mixed clock edges
- Avoid clock buffers
- Avoid gated clocks
- Avoid internally generated clocks
- Gated clocks and low power designs
- Avoid internally generated resets

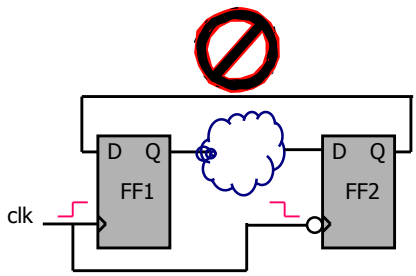
Ideal clocking structure

- The preferred clocking structure is
  - A single global clock, and
  - Positive edge-triggered flops as the only sequential devices.



Avoid mixed clock edges (1/2)

- Guideline
  - Avoid using both positive-edge and negative-edge triggered flip-flops.

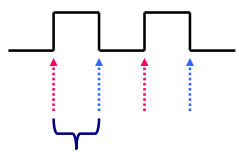


Bad example: mixed clock edges

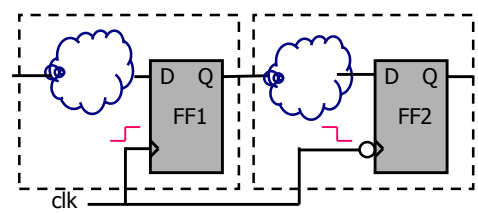
- Mixed clock edges can be used for very aggressive timing goals, such as DDR (double data rate).
- The cares must be taken.
  - The duty cycle is critical, since it is a critical issue in timing analysis in addition to frequency.
  - Scan-based testing requires separate handing of positive and negative triggered flops.

Avoid mixed clock edges (1/2)

- Rule: be sure to model the worst case duty cycle of the clock accurately in synthesis and timing analysis, if both edges are necessary.
  - 50% duty cycle is optimistic
    - Giving signals half the clock cycle to propagate from one register to the next.



- Rule: be sure to document the assumed duty cycle
- Guideline
  - It may be useful to separate positive & negative edge flip-flops into different modules.



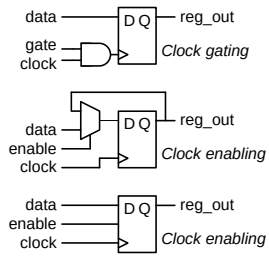
Better example: separated modules

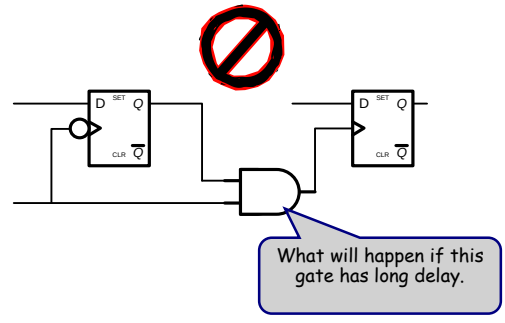
Avoid clock buffers

- Guideline
  - Avoid hand instantiating clock buffers in RTL code.
    - Clock buffers are morally inserted after synthesis as port 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.

Avoid gated clocks

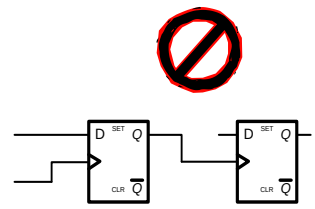
- Guideline
  - Avoid coding gated clocks in RTL, since clock gating circuits tend to be technology specific and timing dependent.
- Improper timing of a gate clock can cause
  - A false clock or glitch,
  - Set-up and hold time violation.





Avoid internally generated clocks

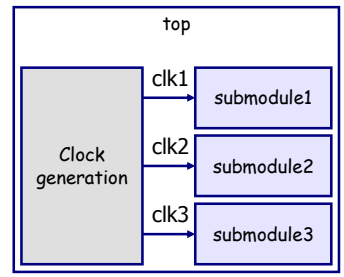
- Guideline
  - Avoid using internally generated clocks.
  - It cause limited testability since logic driven by the internally generated clock cannot be made part of a scan chain.



Bad example: internally generated clock

Gated clocks and low power designs

- Guideline
  - Keep the clock and/or reset generation circuitry (for a gate clock, an internally generated clock or reset) as a separate module at the top level.
  - Partition the design so that all the logic in a single module uses a single clock and a single reset.
- A gated clock should never occur within a macro.
- Isolating clock and reset generation logic in a separate module enables to use the standard timing analysis and scan insertion techniques.



Avoid internally generated resets

- Guideline
  - Avoid internally generated, conditional reset if possible.
    - Generally, all the registers in the macro should be reset at the same time.
  - If a conditional reset is required, create a separate signal for the reset signal, and isolate the conditional reset logic in a separate module.

```
module poor_style (clk, rst, a, b);
  input clk;
  input rst;
  input a, b;
  reg [7:0] my_reg;
  always @ (posedge clk or rst or a or b) begin
       if ((rst or a or b) == 1) my_reg = 8'b0;
       else begin
          ...
       end
  end
endmodule
```

```
module better_style (rst_out, rst_in, a, b);
  output rest_out;
  input rst_in;
  input a, b;
  wire rst_out = rst_in | a | b;
endmodule
module better_style (clk, rst);
  input clk, rst;
  reg [7:0] my_reg;
  always @ (posedge clk or rst) begin
       if (rst == 1) my_reg = 8'b0;
       else begin
          ...
       end
  end
endmodule
```


Coding for synthesis

- In order to create code that achieves the best compile times and synthesis results.
  - Testability
  - Performance
  - Simplification of static timing analysis
  - Gate-level circuit behavior that matches that of the original RTL code
- Infer registers
- Avoid latches
- Avoid combinational feedback
- Specify complete sensitivity lists
- Blocking and non-blocking assignments
- Case statements v.s. if-then-else statements
- Coding state machines

Infer registers

- Guideline
  - Registers (flip-flops) are the preferred mechanism for sequential logic.
  - Use designer's reset signal to initialize registered signals.
    - Do not use initial statement to initialize the signal, since it is not synthesizable construct.
      - These can cause mismatches between pre-synthesis and port-synthesis simulation.

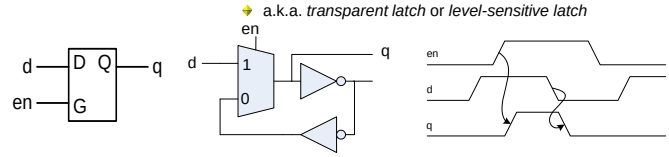
```
// process with synchronous reset
always @ (posedge clk)
  begin : INFER_REG_PROC
    if (rst == 1'b1) begin
      … something …
    end else begin
      … something else …
    end
  end // INFER_REG_PROC
```

```
// process with asynchronous reset
always @ (posedge clk or negedge rst_n)
  begin : INFER_REG_PROC
    if (rst_n == 1'b0) begin
      … reset behavior
    end else begin
      … normal behavior
    end
  end // INFER_REG_PROC
```

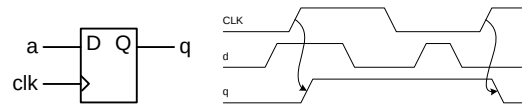

Avoid latches (1/2)

- Rule: avoid using any latches

- Latch
  - When CLK = 1, latch is *transparent*
    - D flows through to Q like a buffer
  - When CLK = 0, the latch is *opaque*
    - Q holds its old value independent of D
  - a.k.a. *transparent latch* or *level-sensitive latch*



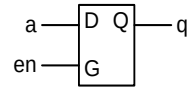
- Flip-flop
  - When CLK rises, D is copied to Q
  - At all other times, Q holds its value
  - a.k.a. *positive edge-triggered flip-flop*, *master-slave flip-flop*



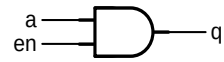
Avoid latches (2/2)

- What makes unintended latch inference.
  - Missing assignment
  - Missing condition
- How to avoid the unintended latch inference.
  - Avoid latch by fully assigning outputs for all input conditions using else and default.

```
// process causes latch inference
always @ (a or en)   if (en==1'b1) q = a;
 // its semantics imply that q must hold
 // its old value when en is low.
```

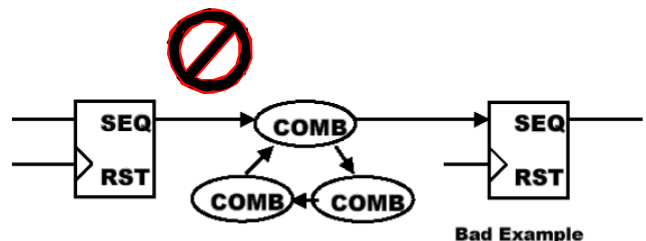


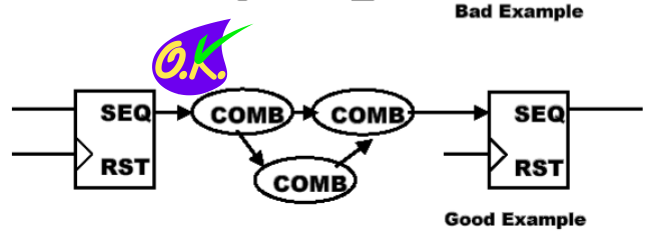

```
// process without latch inference
always @ (a or en)   if (en==1'b1) q = a; else q = 1'b0;
```



Avoid combinational feedback

- Guideline
  - Avoid combinational feedback.





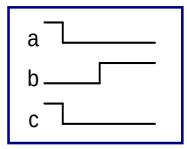
Specify complete sensitivity lists (1/2)

- Rule: include a complete sensitivity list in each of always blocks.
  - If not, the behavior of the pre-synthesis design may differ from that of the post-synthesis netlist.
- Guideline
  - For combinational blocks (blocks with no registers or latches), the sensitivity list must include every signal that is read by the process.
  - For sequential blocks, the sensitivity list must include the clock signal that is read by the process.
  - Make sure the sensitivity lists contains only necessary signals.
    - Adding unnecessary signals to the sensitivity list slows down simulation.

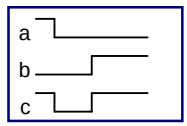
Specify complete sensitivity lists (2/2)

```
// process with incomplete sensitivity list
always @ (a)   c = a or b;
```

Synthesis netlist

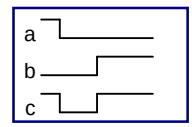


Pre-synthesis simulation waveform



Post-synthesis simulation waveform

```
// process with complete sensitivity list
always @ (a or b)   c = a or b;
```



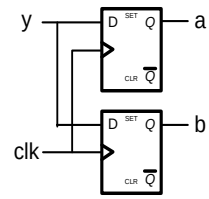
Blocking & non-blocking assignments (1/3)

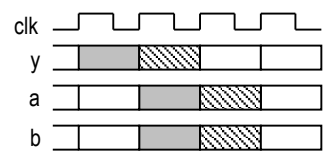
- Blocking assignments execute in sequential order,
- While non-blocking assignments execute concurrently.
- Rule: use non-blocking assignment (<=) in always blocks.

Blocking & non-blocking assignments (2/3)

Blocking assignments

```
always @
(posedge clk)
begin
   a = y;
   b = a;
end
```

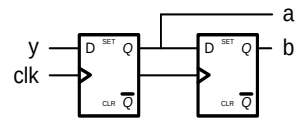


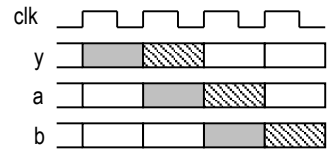


Blocking & non-blocking assignments (2/3)

Non-blocking assignments

```
always @
(posedge clk)
begin
   a <= y;
   b <= a;
end
```






```
always @
(posedge clk)
begin
   b = a;
   a = y;
end
```

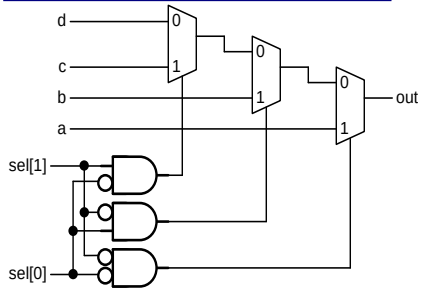

Case statements v.s. if-then-else statements (1/2)

- A case statement infers a single-level multiplexer,
- While an if-then-else statement infers a priority-encoded, cascaded combination of multiplexers.

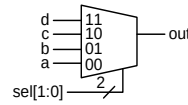
- Guideline
  - The multiplexer is a faster circuit.

Case statements v.s. if-then-else statements (2/2)

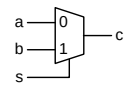
```
always @ (sel or a or b or c or d)
  if (sel == 2'b00) out = a;
  else if (sel == 2'b01) out = b;
  else if (sel == 2'b10) out = d;
  else out = d;
```




```
always @ (sel or a or b or c or d)
  case (sel)
   2'b00:   out = a;
   2'b01:   out = b;
   2'b10:   out = c;
   default: out = d;
  endcase
```

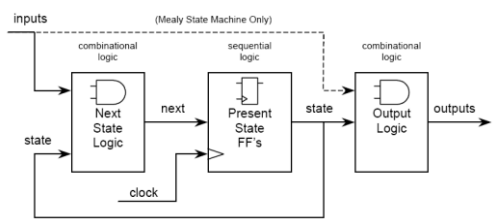



```
assign c = (s) ? b : a;
```



Coding state machines

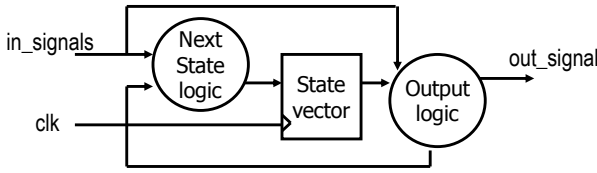
- Guideline
  - Separate the state machine HDL description into two processes, one for the combinational logic and one for the sequential logic.
  - Use `define statements to define the state vector.
  - Keep FSM logic and non-FSM logic in separate modules.
  - Assign a default state for the state machine.

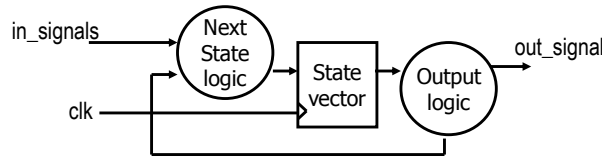



```
.....
 reg [1:0]  state, next_state; // FSM state
 //-----------------------------------------------
 parameter IDLE    = 2'h0;
 parameter FOUND   = 2'h1;
 parameter MODULUS = 2'h2;
 parameter DONE    = 2'h3;
 //-----------------------------------------------
always @ (state)
  case (state)
     IDLE:   begin
               ...
               next_state <= FOUND;
             end  end // IDLE
     FOUND:  begin
               ...
               next_state <= DONE;
             end // FOUND
     endcase
 always @ (posedge clk or negedge rst_b)
     if (rst_b == 1'b0) state <= IDLE;
     else state <= next_state;
endmodule
```

State machine

- Mealy state machine
  - The output logic is always a function of the current state (state vector) and the inputs.
- Moore state machine
  - The output logic is only a function of the current state, i.e. inputs are not included.



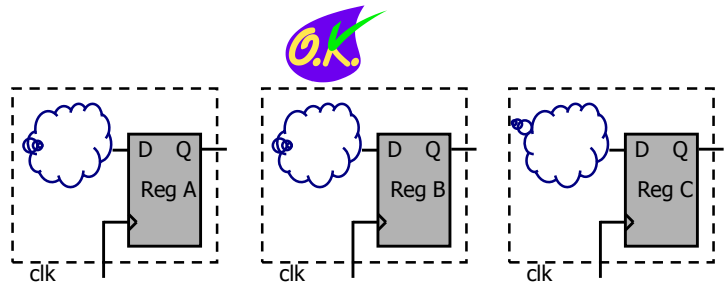


Partitioning for synthesis

- Register all outputs
- Locate related combinational logic in a single module
- Separate module that have different design goals
- Asynchronous logic
- Arithmetic operations: merging resources

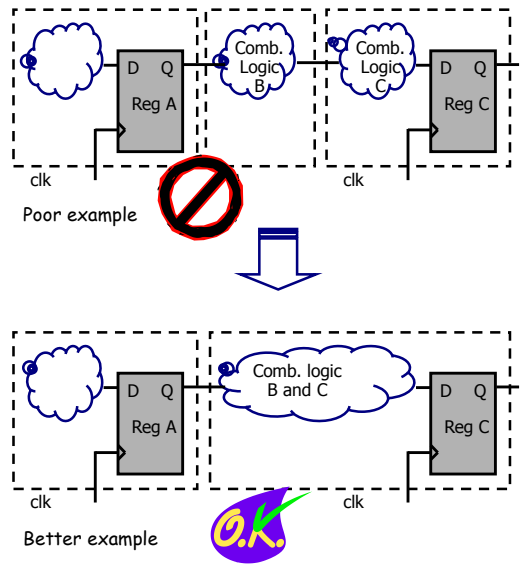
Register all outputs

- Guideline
  - For each block of a hierarchical design, register all output signals from the block.
    - Registering the output signals from each block simplifies the synthesis process because it makes output drive strengths and input delays predicable.



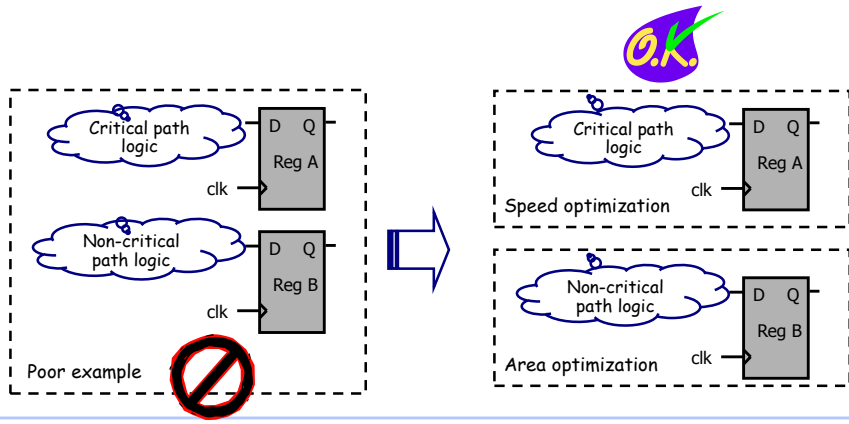
Locate related combinational logic in a single module

- Guideline
  - Keep related combinational logic together in the same module.
    - Synthesizer has more flexibility in optimizing a design when related combinational logic is located in the same module.
    - Keep related combinational logic in the same module also eases time budgeting and allows for faster simulation.



Separate modules that have different design goals

- Guideline
  - Keep critical path logic in a separate module from non-critical path logic.
    - Synthesizer can optimize the critical path logic for speed, while optimizing the non-critical path logic for area.

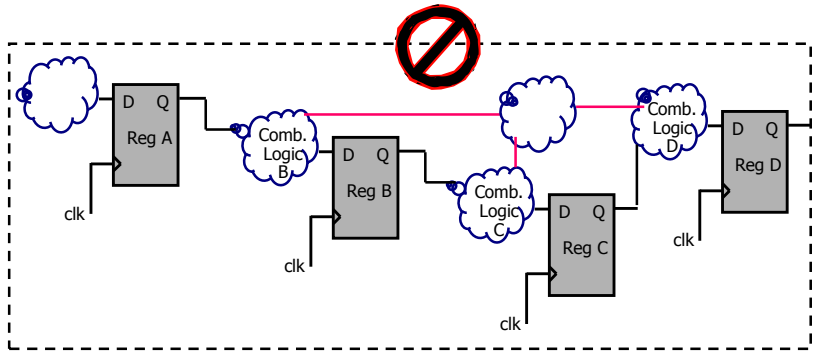


Avoid asynchronous logic

- Guideline
  - Avoid asynchronous logic.
  - If asynchronous logic is required, 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 need to be reviewed carefully to verify its functionality and timing.

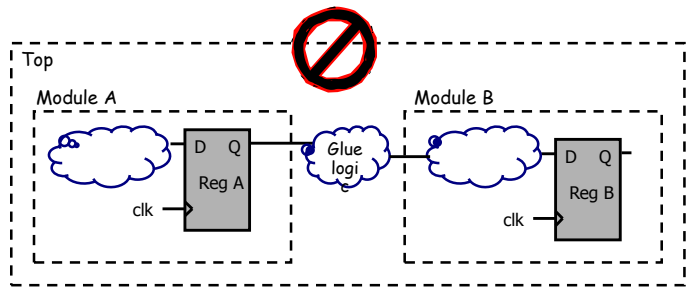
Avoid point-to-point exceptions

- Guideline
  - Avoid multi-cycle paths
    - Multi-cycle path is more difficult to analyze correctly.
  - If needed, keep point-to-point exceptions within a single module.

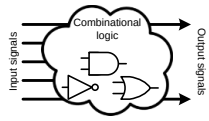


Avoid glue logic at the top level

- Guideline
  - Do not instantiate gate-level logic at the top level of the design hierarchy.
    - Let design hierarchy contain gates only at leaf levels of the hierarchy tree.

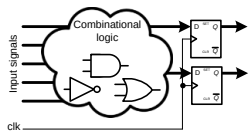


Coding




```
wire sig_in_a;
wire sig_in_b;
wire sig_out_c;
wire sig_out_d = sig_in_a & sig_in_b;
assign sig_out_c = sig_in_a | sig_in_b;
```

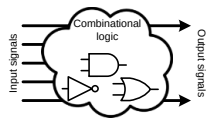
Combinational logic inference by continuous assignment (not wire and assign)




```
wire sig_in_a;
wire sig_in_b;
reg sig_out_c;
reg sig_out_d;

always @ (posedge clk) begin
   sig_out_d <= sig_in_a & sig_in_b;
   sig_out_c <= sig_in_a | sig_in_b;
end
```

Register or Flip-Flop inference by always block (note “posedge clk”)

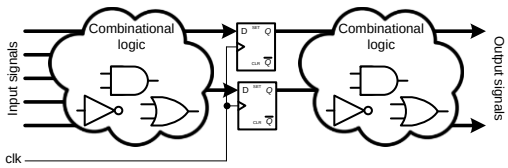



```
wire sig_in_a;
wire sig_in_b;
reg sig_out_c;
reg sig_out_d;

always @ ( * ) begin
   sig_out_d <= sig_in_a & sig_in_b;
   sig_out_c <= sig_in_a | sig_in_b;
end
```

Combinational logic inference by always block (note event list or sensitivity list)

Coding




```
wire sig_in_a;
wire sig_in_b;
wire sig_out_c;
wire sig_out_d;
reg sig_reg_x;
reg sig_reg_y

always @ (posedge clk) begin
   sig_reg_x <= sig_in_a & sig_in_b;
   sig_reg_y <= sig_in_a | sig_in_b;
end

assign sig_out_c = sig_reg_x & sig_reg_y;
assign sig_out_d = sig_reg_x | sig_reg_y;
```

Reading

- M. Keating and P. Bricaud, Reuse Methodology Manual, Chapter 5 RTL Coding Guidelines, KAP, 1999.