N26F300 VLSI SYSTEM DESIGN (GRADUATE LEVEL)

Fall 2010

Synthesis for Verilog (II)

Outline

- Synthesis of Sequential Logic
- Coding Guidelines

Synthesis of Sequential Logic

Synthesis of Sequential Logic with Latches

- Accidentally

  A synthesis tool unexpectedly inferred latch-based logic from a Verilog statements that are intended to be combinational logic.

- Intentionally

  A synthesis tool did what the designers want from a Verilog statements.

When Combinational Becomes Sequential?

- A feedback-free netlist of combinational primitives ➔ NO latch
- A set of feedback-free continuous assignments ➔ NO latch
- A continuous assignment using a conditional operator with feedback ➔ latch

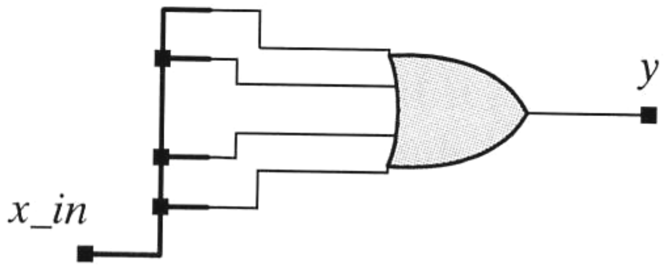
Intentional Synthesis of Latches

- Incompletely specified assignment
  - In level-sensitive cyclic behavior
  - In conditional branch (if or case)

Clean Combinational Logic

```
module or4_behav (y, x_in);
  parameter     word_length = 4;
  output                    y;
  input       [word_length - 1: 0]    x_in;
  reg                    y;
  integer                  k;

  always @  x_in
    begin: check_for_1
      y = 0;
      for (k = 0; k <= word_length -1; k = k+1)
        if (x_in[k] == 1) begin
            y = 1;
            disable check_for_1;
        end
    end
endmodule
```

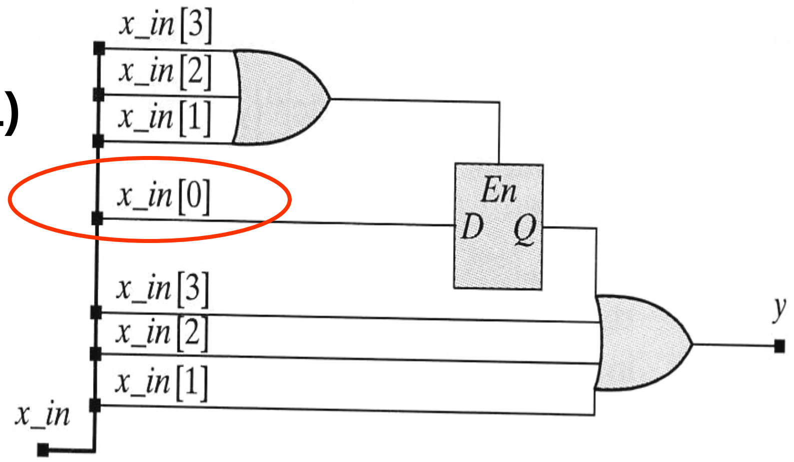
Event Control Sensitive to x_in[3:0]



Output NOT Completely Specified by ALL Inputs

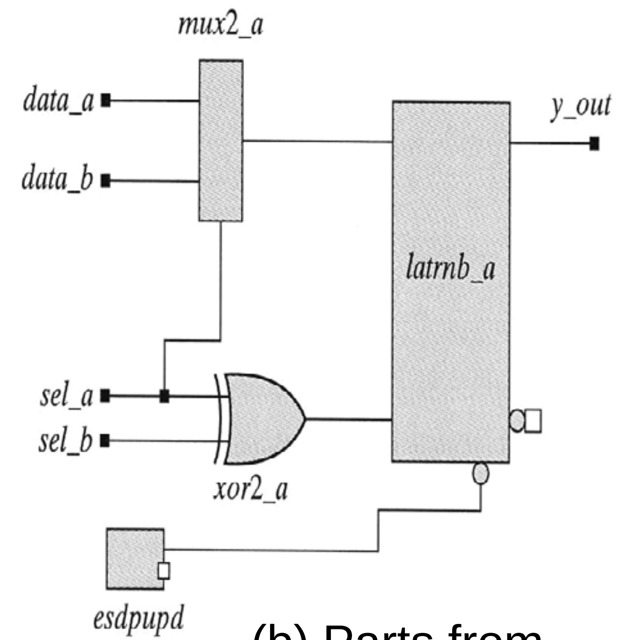
```
module or4_behav_latch (y, x_in);
  parameter     word_length = 4;
  output              y;
  input       [word_length - 1: 0]    x_in;
  reg                y;
  integer             k;

  always @  (x_in[3:1])
    begin: check_for_1
      y = 0;
      for (k = 0; k <= word_length -1; k = k+1)
        if (x_in[k] == 1)
          begin
            y = 1;
            disable check_for_1;
          end
    end
endmodule
```

Event Control Sensitive Only to x_in[3:1]

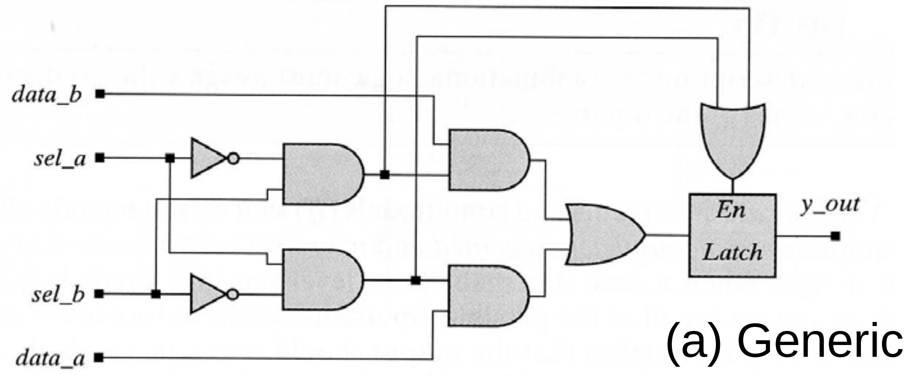


Incompletely Specified Case

```
module mux_latch (y_out, sel_a, sel_b, data_a,
data_b);
  output      y_out;
  input       sel_a, sel_b, data_a, data_b;
  reg         y_out;

  always @ ( sel_a or sel_b or data_a or data_b)
  case ({sel_a, sel_b})
    2'b10: y_out = data_a;
    2'b01: y_out = data_b;
  endcase
endmodule
```



(b) Parts from cell library,i.e., after technology mapping

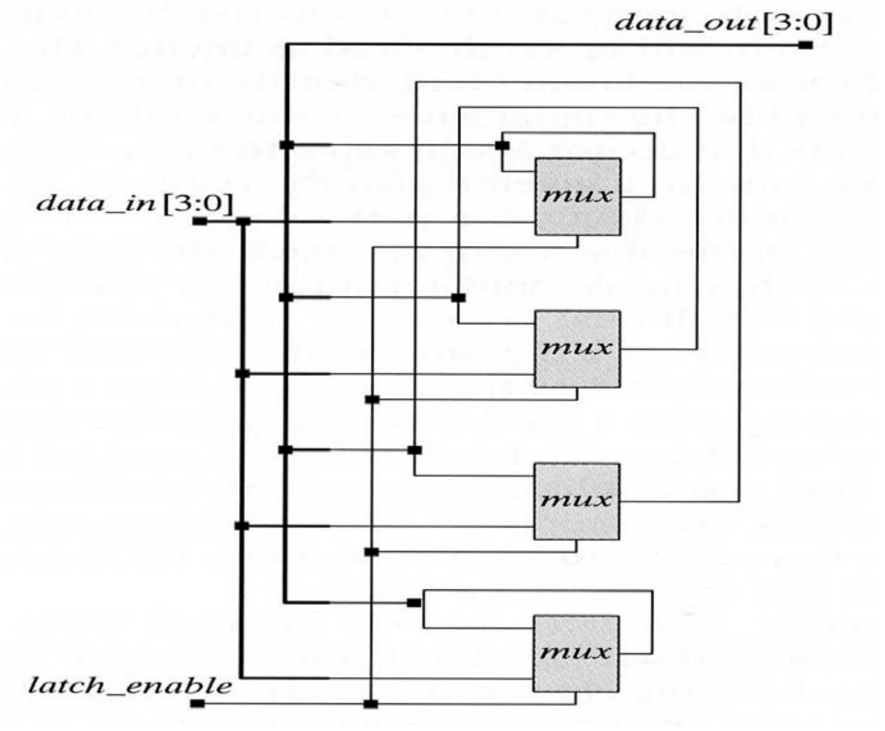


(a) Generic parts

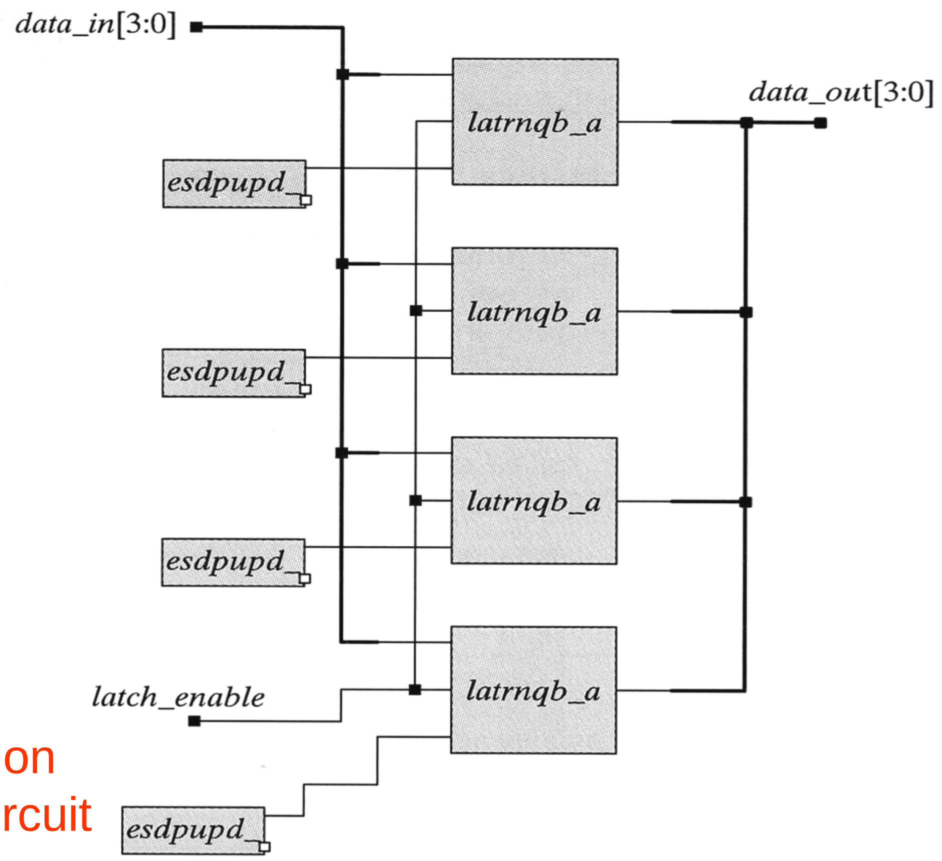
MUXES vs LATCHES

- If a ***case*** statement has a default assignment with feedback
- Or If an ***if*** statement in a level-sensitive behavior assign a value to itself ➔ a **mux**

```
module latch_if1(data_out, data_in,
latch_enable);
  output      [3: 0]  data_out;
  input       [3: 0]  data_in;
  input               latch_enable;
  reg         [3: 0]  data_out;

  always @  (latch_enable or data_in)
    if (latch_enable) data_out = data_in;
      else data_out = data_out;
endmodule
```



```
module latch_if2 (data_out, data_in,
latch_enable);
  output   [3: 0]  data_out;
  input     [3: 0]  data_in;
  input               latch_enable;
  reg        [3: 0]  data_out;

  always @  (latch_enable or data_in)
    if (latch_enable) data_out = data_in;
    // Incompletely specified
endmodule
```

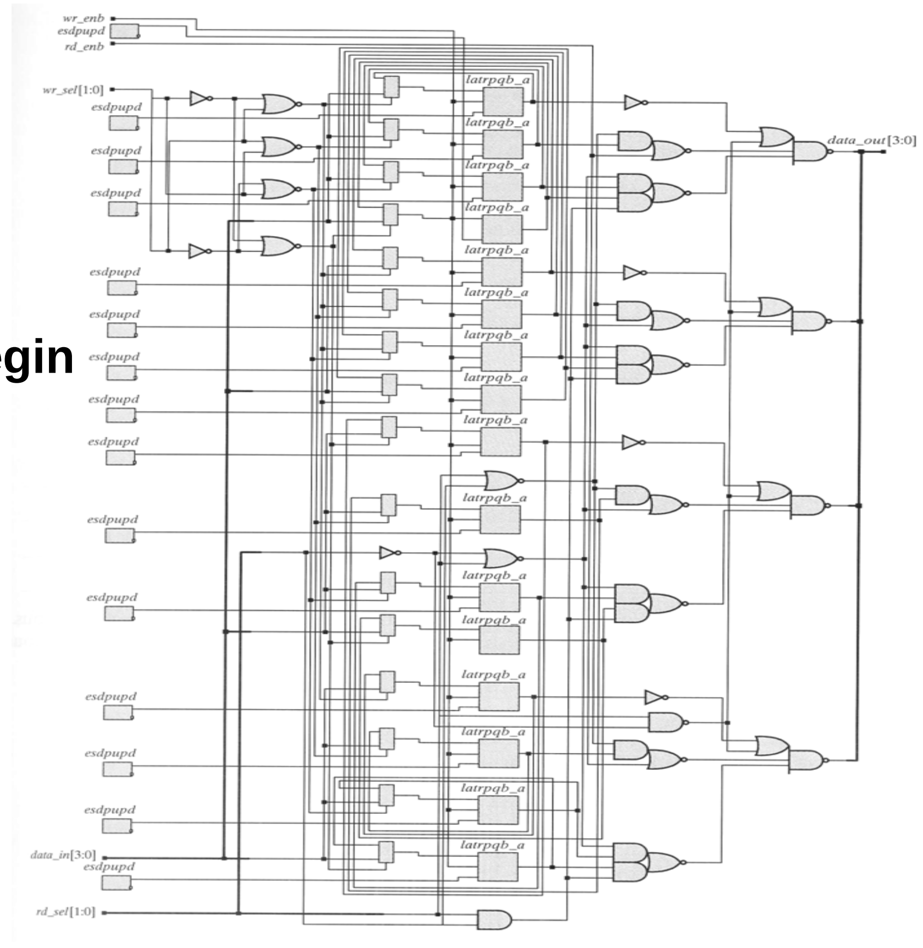


A slight change in the behavioral description

➔ Change structure of the synthesized circuit

A latch file

```
module sn54170 (data_out, datajn, wr_sel, rd_sel,
wr_enb, rd_enb);
output        [3:0] data_out;
input                   wr_enb, rd_enb;
input         [1:0]   wr_sel, rd_sel;
input         [3:0]   datajn;
reg           [3:0]    latched_data     [3:0];
always @ (wr_enb or wr_sel or datajn) begin
   if(!wr_enb)
           latched_data[wr_sel] = datajn;
   end

assign data_out = (rd_enb) ? 4'b1111 :
latched_data[rd_sel];

endmodule
```



Synthesis of Sequential Logic with Flip-Flops

- Flip-flops are synthesized only from edge-sensitive cyclic behaviors, but not every register variable that is assigned value in an edge-sensitive block is synthesized to a flip-flop.

- A register variable in an edge-sensitive block will be synthesized as a flip-flop if
  - it is referenced outside the scope of the block, or
  - it is referenced within the block before it is assigned value, or
  - it is assigned value in only some of the branches of the activity within the block.

Synthesis of Sequential Logic with Flip-Flops

- For an incomplete conditional statement (i.e., an if … else statement or a case statement), if the block is edge-sensitive, it will be synthesized to a logic that implements a “clock enable”

- If the event-control expression is sensitive to the edge of more than one signal, an if statement must be the first statement in the block.

Referenced Outside the Scope of the Block

```
module empty_circuit (D_in, elk);
input  D_in;
input   clk;
reg     D_out;
always @ (posedge clk) begin D_out < = D_in; end
endmodule
```

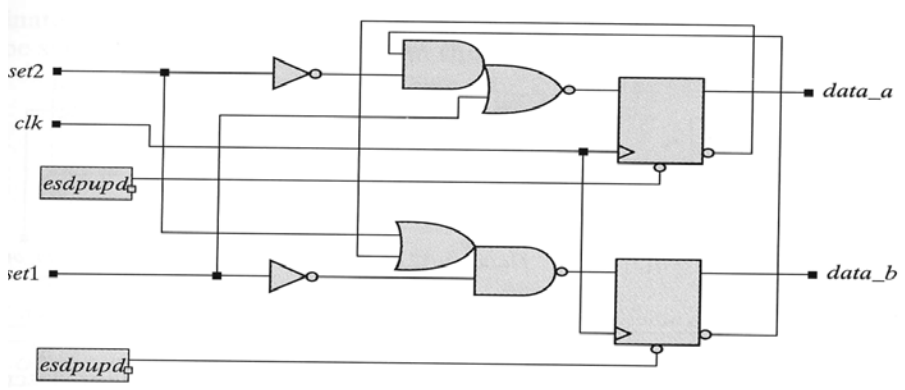
D_out is not referenced outside the scope ➔ eliminate D_out

```
module circuit_FF (D_out, D_in, elk);
Output D_out;
input  D_in;
input   clk;
reg     D_out;
always @ (posedge clk) begin D_out < = D_in; end
endmodule
```

D_out is referenced outside the scope ➔ synthesized as a FF.

Referenced Before Values Assigned

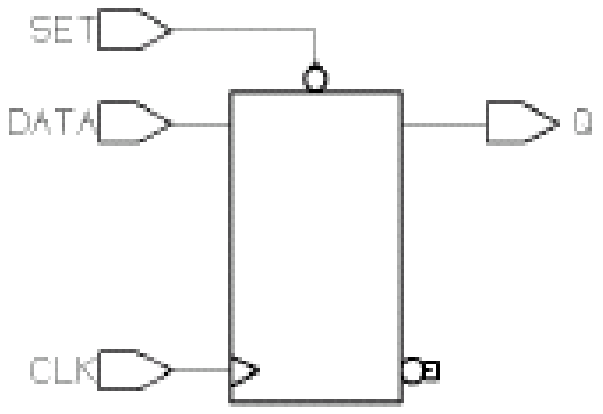
```
module swap_synch (set1, set2, clk, data_a, data_b);
  output       data_a, data_b;
  input        clk, set1, set2, swap;
  reg      data_a, data_b;

  always @  (posedge clk)
    begin
      if (set1) begin data_a <= 1; data_b <= 0; end else
        if (set2) begin data_a <= 0; data_b <= 1; end else
          else
            begin
              data_b <= data_a;
              data_a <= data_b;
            end
    end
endmodule
```



D Flip-Flop With Asynchronous Set

- When inferring a D flip-flop with an asynchronous set or reset, include edge expressions for the clock and the asynchronous signals in the sensitivity list of the always block.

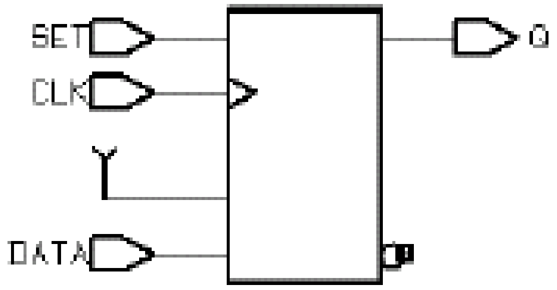
```
module dff_async_set (DATA, CLK, SET, Q);
    input DATA, CLK, SET;
    output Q;
    reg Q;
    always @(posedge CLK or negedge SET)
        if (~SET)
            Q <= 1'b1;
        else
            Q <= DATA;
    endmodule
```



D Flip-Flop With Synchronous Set

- When inferring a D flip-flop with an synchronous set or reset, include only the clock in the sensitivity list of the always block.

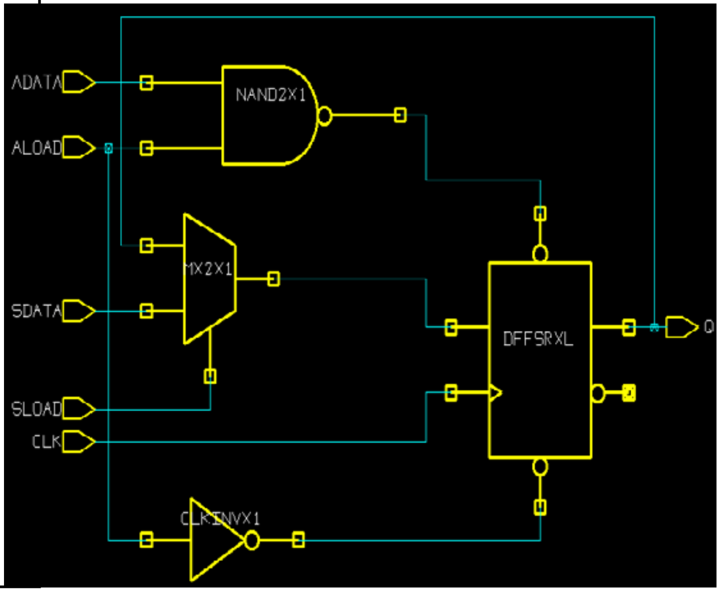
```
module dff_sync_set (DATA, CLK, SET, Q);
      input DATA, CLK, SET;
      output Q;
      reg Q;
      //synopsys sync_set_reset "SET"
      always @(posedge CLK)
            if (SET)
                  Q <= 1'b1;
            else
                  Q <= DATA;
endmodule
```



D Flip-Flop With Syn. and Asyn. Load Controls

- D flip-flops can have asynchronous or synchronous controls.

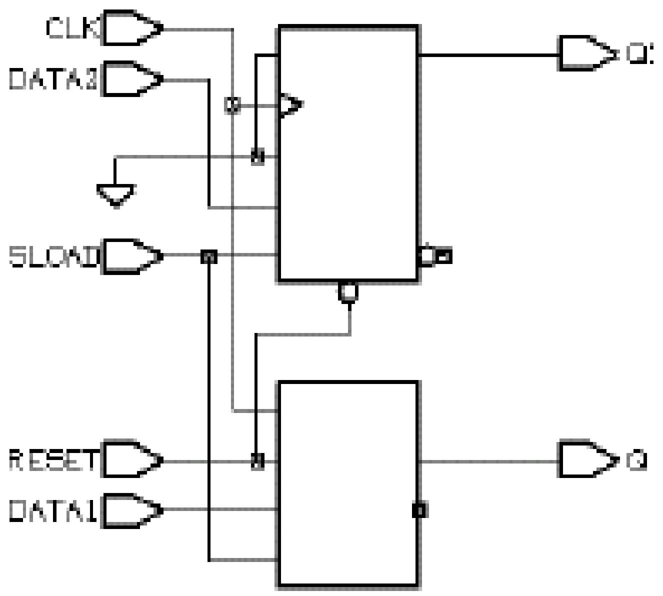
```
module dff_a_s_load (ALOAD, SLOAD, ADATA,
SDATA, CLK, Q);
input ALOAD, ADATA, SLOAD, SDATA, CLK;
output Q;
reg Q;
always @ (posedge CLK or posedge ALOAD)
      if (ALOAD)
            Q <= ADATA;
      else if (SLOAD)
            Q <= SDATA;
endmodule
```



Multiple Flip-Flops With Asyn. and Syn. Controls

- If a signal is synchronous in one block but asynchronous in another block,

```
//synopsys sync_set_reset "RESET"
always @(posedge CLK)
      begin : infer_sync
      if (~RESET)
            Q1 <= 1'b0;
      else if (SLOAD)
            Q1 <= DATA1;      // note: else hold Q
      end
always @(posedge CLK or negedge RESET)
      begin: infer_async
      if (~RESET)
            Q2 <= 1'b0;
      else if (SLOAD)
            Q2 <= DATA2;
      end
```



Limitations of D Flip-Flop Inference

- An if statement must occur at the top level of the always block.

```
always @(posedge clk or posedge reset)
     begin
     q = d;
     if (reset)
          . . .
end
```

```
Error: The statements in this ’always’ block are outside the
scope of the synthesis policy (%s). Only an ’if’ statement
is allowed at the top level in this ’always’ block. (ELAB-302)
```

Writing Efficient Circuit Description

- Restructure a design that makes repeated use of several large components, to minimize the number of instantiations.
- In a design that needs some, but not all, of its variables or signals stored during operation, minimize the number of latches or flip-flops required.
- Consider collapsing hierarchy for more-efficient synthesis for small circuit, but use hierarchy for large circuit.

Minimizing Registers (1/2)

- In an **always** block that is triggered by a clock edge, every variable that has a value assigned has its value held in a flip-flop.

```
always @(posedge clock) begin
      if (reset)
         count = 0; // reg [2:0] count;
      else
         count = count + 1;
         and_bits = & count;
         or_bits = | count;
         xor_bits = ^ count;
   end
endmodule
```

- Use six flip-flops; however, and_bits, or_bits, and xor_bits depend solely on the value of count.

Minimizing Registers (2/2)

- Assign the outputs from within an asynchronous **always** block to avoid implying extra registers

```
always @(posedge clock) begin
//synchronous
      if (reset)
            count = 0;
      else
            count = count + 1;
end
always @(count) begin
//asynchronous
      and_bits = & count;
      or_bits = | count;
      xor_bits = ^ count;
end
```

Separating Sequential and combinational Ckts

- Make a separate *always* block within edge trigger (synchronous) and no edge trigger (asynchronous)

```
always @(posedge clk or negedge reset) // state vector flip-flops
(sequential)
     if (!reset)
          current_state = 0;
     else
          current_state = next_state;

always @(in1 or in2 or current_state)
                                       // output and state vector decode
(combinational)
     case (current_state)
          0: begin
               next_state = 1;
               out = 1'b0;
               end
          1:  begin
               next_state = 1'b2;
               out = in1;
               end
          2:     .....
     endcase
```

Why cannot synthesize?(1/2)

- Verilog code is not pure RTL code.
  - Cannot use “#” delay command

```
Ex:always@(posedge clk) begin
       #20 a=c;
       end
```

# delay cannot be synthesis to gate

  - Cannot use initial block in your design.

```
Ex:   module count(clk,c);
       initial begin
           c=0;
      end
```

Initial block only can be used in testfixture

- Cannot use the same reg in different always blocks.

```
Ex:  always@(posedge clk) begin
            a=c;
     end
     always@(posedge clk or negedge reset) begin
            a=d;        end
```

One value cannot be driven by 2 conditions

Why cannot synthesize?(2/2)

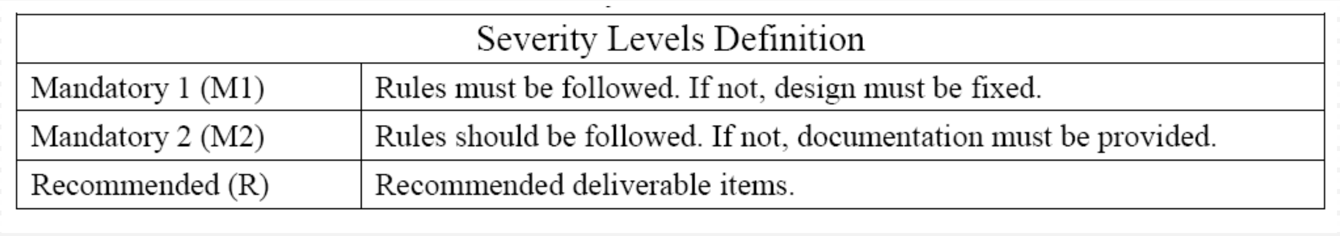
- Avoid using "**task**" command.

Try to use pure "always block" and "wire" and "assign" command.

- **Note:**

The warning message like "… cannot be synthesized" will appear when reading a verilog code that cannot be synthesized in Synopsys. So please carefully check the messages when warned during synthesis.

Coding Guideline



Severity Levels Definition	
Mandatory 1 (M1)	Rules must be followed. If not, design must be fixed.
Mandatory 2 (M2)	Rules should be followed. If not, documentation must be provided.
Recommended (R)	Recommended deliverable items.

Ref : Reuse Methodology Manual (RMM) Keating & Bricaud, 3rd ed., 2002.

Basic Coding Practices

- Naming convention
  - Lower case for signals, variables, ports
  - Upper case for constants (define directive)
  - Meaningful names

    Ex: `ram_addr` for RAM address, `sys_bus` for system bus.
  - Short but descriptive parameters (for synthesis)
  - Prefix `clk` for clocks, prefix `rst` for reset.
  - `*_x` for active low, e.g., rst_n, act_b.
  - Use `[x:0]` rather than `[0:x]`
  - Use `*_r` for output of a register, e.g., count_r
  - Use `*_z` for tristate signal

- Do not use HDL reserved words.

Basic Coding Practices (cont.)

- State variables: `<name>.cs` for current state, `<name>.ns` for next state
- Use informational header for each source file:
  - Legal information (confidentiality, copyright, restriction, etc.), filename, author, date, version history, main contains.
- Use concise but explanatory comments where appropriately.
- Avoid multiple statements in one line.
- Use indentation to improve readability.
  - Use indentation of 2 spaces; do not use tab
- Port (core I/O) ordering
  - One port per line; a comment followed each port declaration
  - Follow the following order:
    - ➢ clocks, resets, enables, other control signals, data & Address lines

Basic Coding Practices (cont.)

- Port mapping
  - Use explicit mapping
  - Use named association rather than positional association

Ex:

```
 TagRam TagRam (
.Address     (PAddress[`INDEX]),
.TagIN        (PAddress[`TAG]),
.Tagout      (TagRAMTag[`TAG]),
.Write         (Write),
.Clk            (Clk)
);
```

Basic Coding Practices (cont.)

- Use functions wherever possible (for reusability and avoid repetition).
- Use vector operations (array) rather than loops for faster simulation.

Ex:

```
Poor:     for (i = 1, i < k, i ++)
              c_vec[i] = a_vec[i] ^ b_vec[i];

Better:   c_vec = a_vec ^ b_vec;
```

Clocks & Resets

- Preferably use a single global clock and positive edge-triggered flip-flops only.
- Avoid both positive and negative edge-triggered flip-flops.
- Separate positive-edge and negative-edge triggered flip-flops into different modules **--- make scan design easier.**
- Avoid clock buffers **--- leave it to clock insertion tool.**
- Avoid gated clock **--- to avoid false clock or glitch, and improve** testability.
- Avoid internally generated clocks **--- for testability.**
- If internally generated clocks are necessary, separate it in a top-level module.
- Avoid internally generated resets.

Coding for Synthesis

- Registers (flip-flops) are preferred.
- Latch should be avoided.
- Use design tools to check for latches.
- Poor coding may infer latches:
  - Missing `else` statement in `if-then-else` structures.
  - Missing assignments or conditions in `case` structures.
- To avoid the undesired inferred latches:
  - Assign default values at the beginning of a process.
  - Assign outputs for all input conditions.
  - Use default statements for case structures.
  - Always consider the else case in a if-then-else structure.
- If a latch must be used, well document it and prepare to make it testable via a mux.

Coding for Synthesis (cont.)

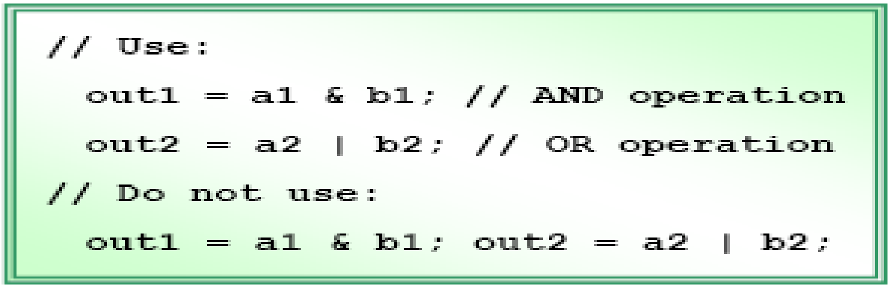
- Avoid combinational loops.
- Specify complete sensitivity lists.
- Always use nonblocking assignments in `always @(posedge clk)` blocks.
- Use `case` instead of nested `if-then-else` statements.
- Separate FSM and non-FSM logic in different modules.
- Keep late-arriving signals with critical timing closest to the output of a module, e.g., earlier in if-then-else structures.
- Do not use delay constants in RTL code to be synthesized.

Partitioning for Synthesis

- Partitioning can result in better synthesis results, faster compile and simulation time, and enable simpler synthesis strategy to meet timing requirement.
- Locate related combinational logic in a single module
- Separate modules that have different design goals.
- Avoid asynchronous design except reset.
- If asynchronous is required, put it in a separate module.
- Put relevant resources to be shared in the same module.

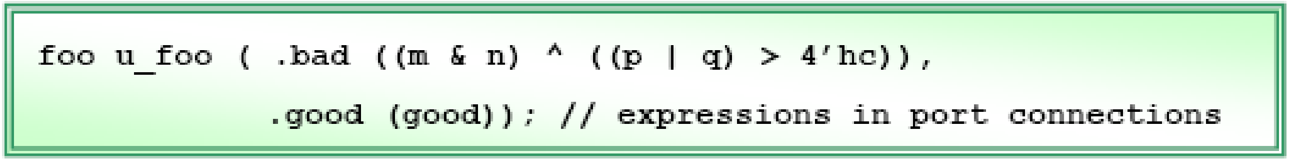
Coding Style (1/2)

- [M1] Use a separate line for each HDL statement



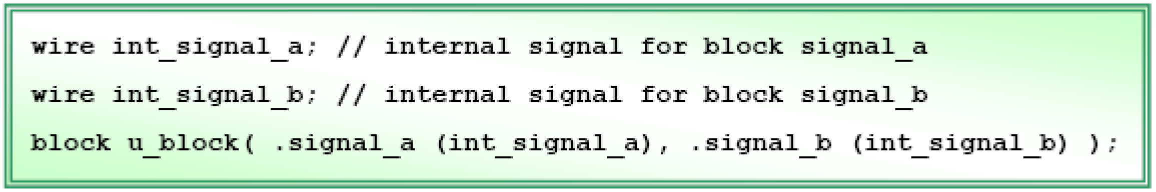
```
// Use:
  out1 = a1 & b1; // AND operation
  out2 = a2 | b2; // OR operation
// Do not use:
  out1 = a1 & b1; out2 = a2 | b2;
```

- [M1] Avoid expression in port connections



```
foo u_foo ( .bad ((m & n) ^ ((p | q) > 4'hc)),
            .good (good)); // expressions in port connections
```

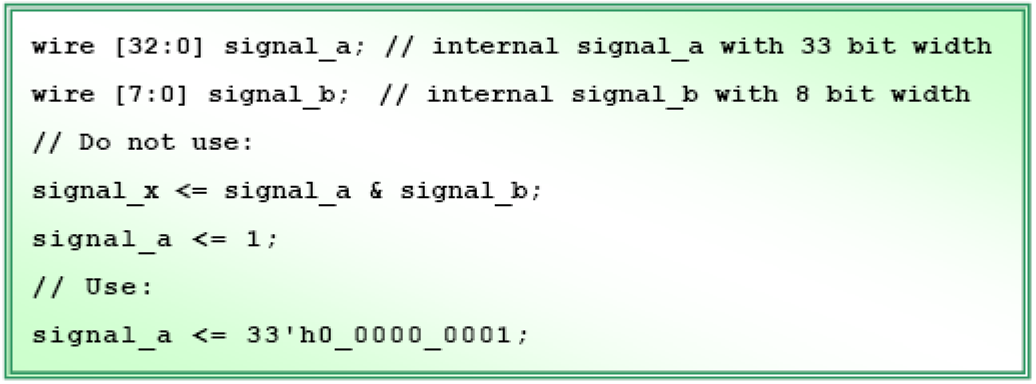
- [M1] Wire must be explicitly declared



```
wire int_signal_a; // internal signal for block signal_a
wire int_signal_b; // internal signal for block signal_b
block u_block( .signal_a (int_signal_a), .signal_b (int_signal_b) );
```

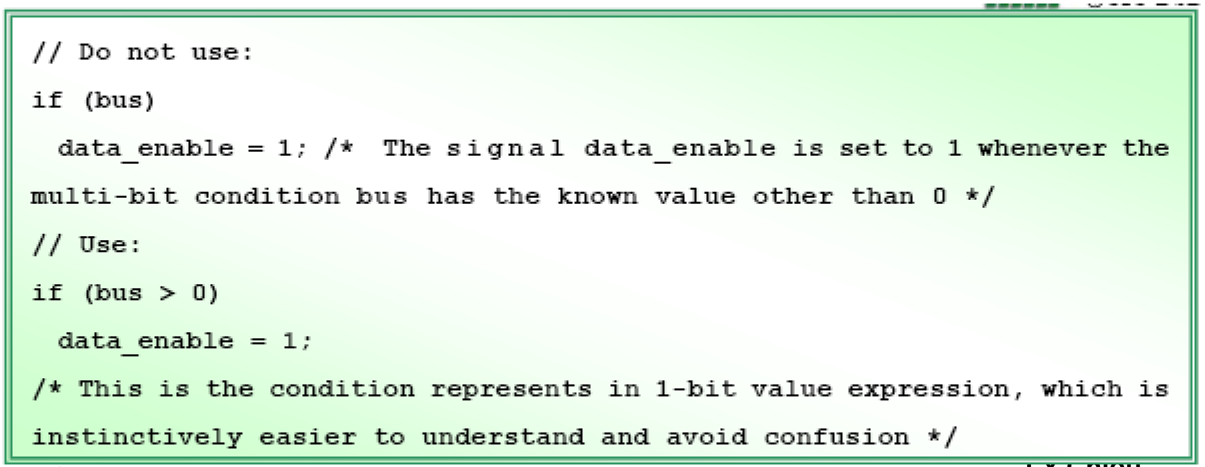
Coding Style (2/2)

- [M1] Operand size must match



```
wire [32:0] signal_a; // internal signal_a with 33 bit width
wire [7:0] signal_b;  // internal signal_b with 8 bit width
// Do not use:
signal_x <= signal_a & signal_b;
signal_a <= 1;
// Use:
signal_a <= 33'h0_0000_0001;
```

- [R] Expression in condition should be 1- bit value



```
// Do not use:
if (bus)
  data_enable = 1; /*  The signal data_enable is set to 1 whenever the
multi-bit condition bus has the known value other than 0 */
// Use:
if (bus > 0)
  data_enable = 1;
/* This is the condition represents in 1-bit value expression, which is
instinctively easier to understand and avoid confusion */
```

Coding for Synthesis (1/3)

- **[M1] Use only synthesizable statement**
  - NOT waveform statements (initial)
  - NOT system task and function ($display, $monitor, ..)
  - NOT wait statement and # delay statement
  - NOT real, time, and event
  - ONLY one clock per always sensitivity list
  - ONLY loop with a static range

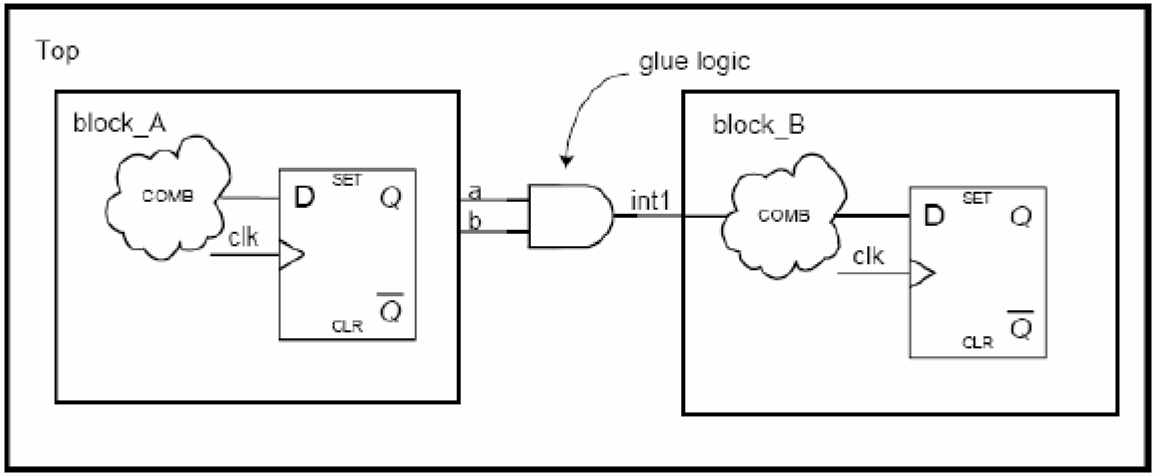
Coding for Synthesis (2/3)

- [M2] NO full_case and parallel_case directive statements
- [M2] Avoid unexpected latch inference
- [M1] NO User-Defined-Primitive (UDP)
- [M1] All unused module inputs must be driven

  Either by some other signal or by a fixed logic zero or one

Coding for Synthesis (3/3)

- [M2] Avoid top-level glue logic



Coding for Simulation

- **[M1] Use non-blocking assignments in sequential always block**
- **[M1] Use blocking assignments in combinational always block**

- **[M1] Avoid missing sensitivity lists in combinational always block**
- **[M1] Avoid redundant sensitivity lists**

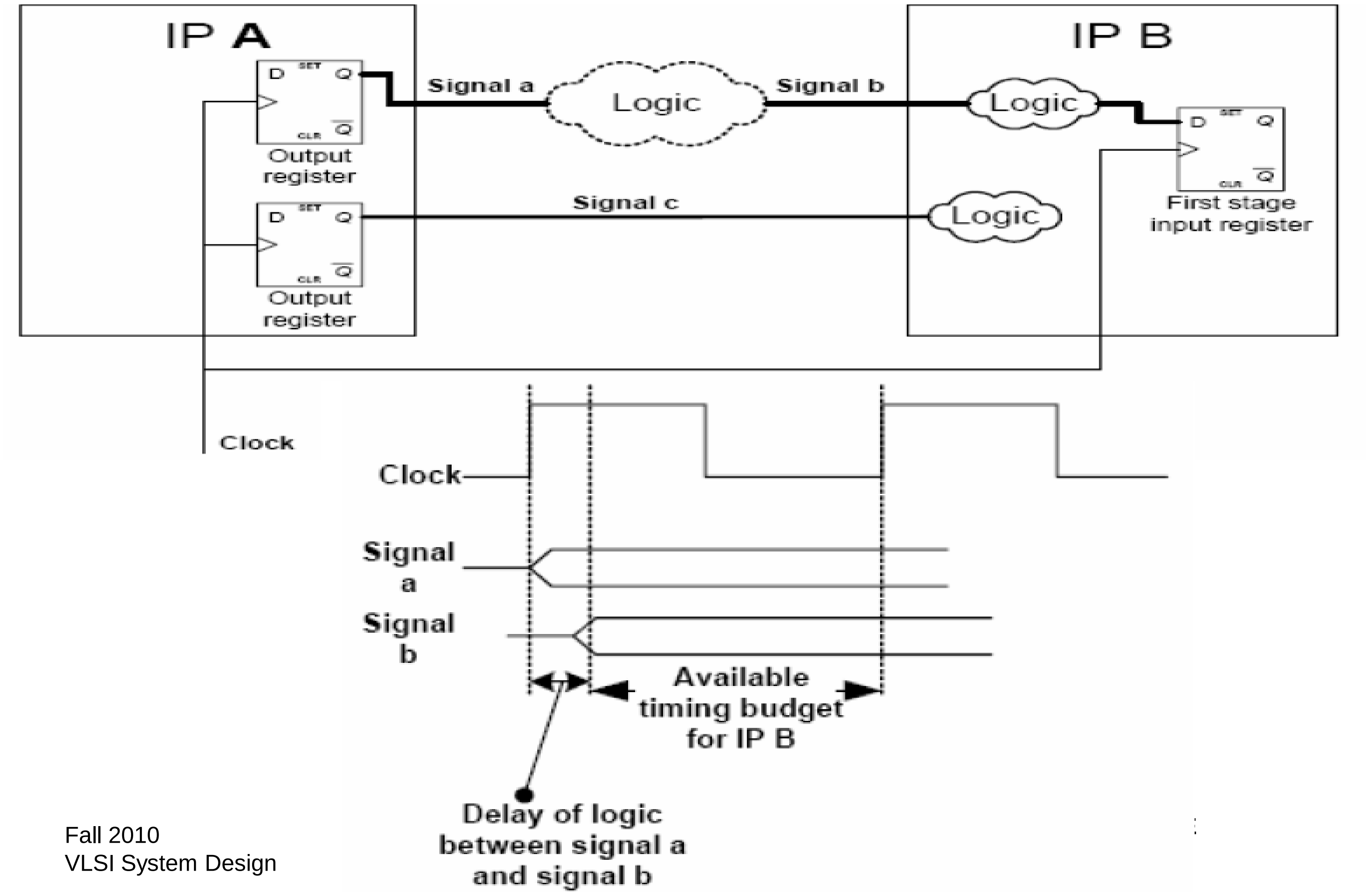
  The presence of a signal in the sensitivity list that is not used in the process may cause unneeded evaluations in simulation and result in longer simulation times.

- **[M1] Do not assign signals don't care (x) value**

  If there exists **x** value in a control path, it may potentially cause the chain of unknown state for all related circuits in simulation. Furthermore, it is extremely possible cause the problem of pre-simulation and post-simulation mismatch.

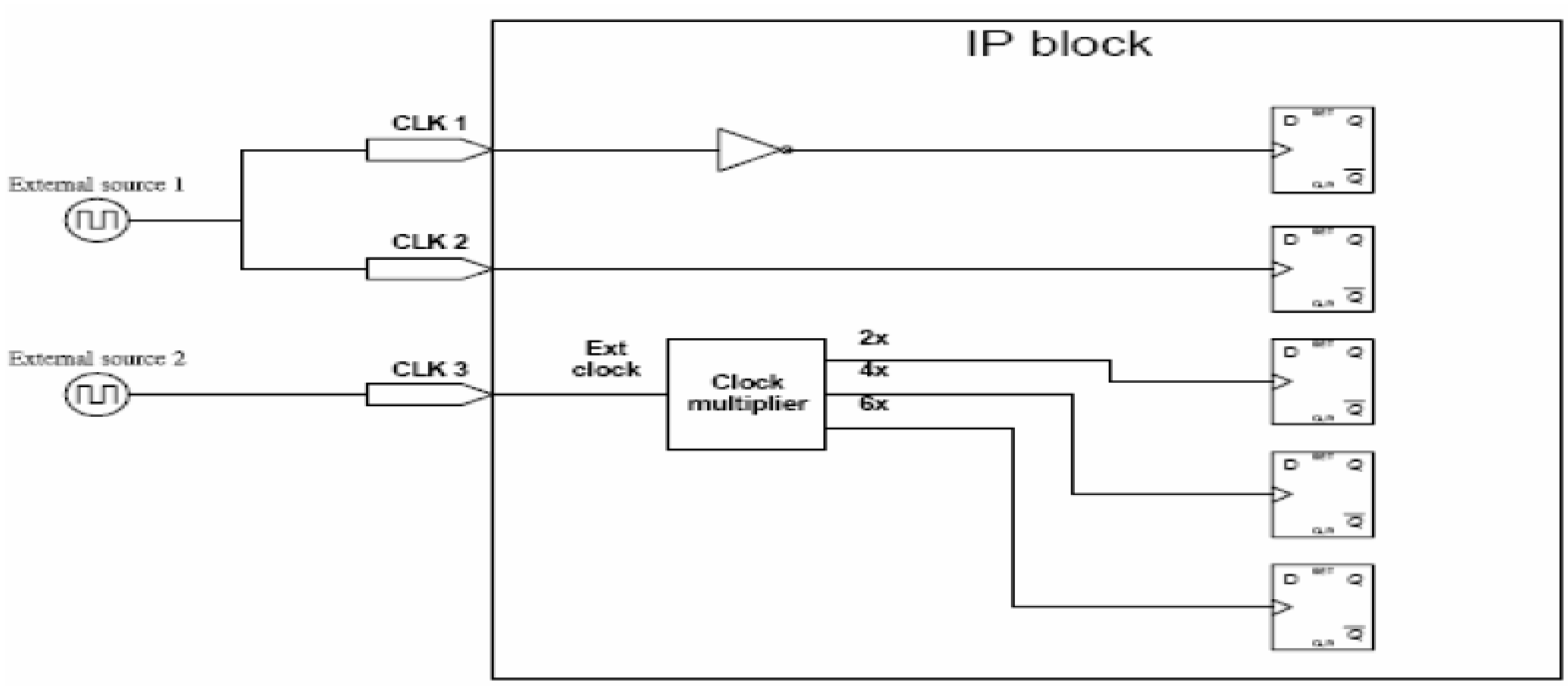
Design Style (1/2)

- [M2] Output signals must be registered



Design Style (2/2)

- [R] Registers in an IP's interface should be clocked in a single edge of the same clock
- [M1] Clock strategy must be documented


IP block
CLK 1
External source 1
CLK 2
External source 2
CLK 3
Ext clock
Clock multiplier
2x
4x
6x
D
Q