# Topic 4
# RTL Coding Guidelines

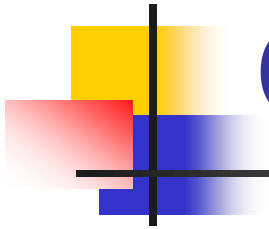
*Offers a collection of coding rules and guidelines to help develop readable, modifiable, and reusable HDL code and also help to achieve optimal results in synthesis and simulation*



# Outline

- Overview of the coding guidelines
- Basic coding practices
- Coding for portability
- Guidelines for clocks and resets
- Coding for synthesis
- Partitioning for synthesis
- Designing with memories





# Outline

- Overview of the coding guidelines
- Basic coding practices
- Coding for portability
- Guidelines for clocks and resets
- Coding for synthesis
- Partitioning for synthesis
- Designing with memories



# Overview of coding guidelines

- *Recommendation*
  - Develop RTL code simple and regular
    - Easier to design, code, verify, and synthesize
  - Consistent coding style, naming conventions and structure
  - Using a regular partition scheme
    - All module outputs registered and with modules of the same size
  - Easy to understand
    - Comments
    - Meaningful names
    - Constants or parameters instead of hard-coded numbers



# Outline

- Overview of the coding guidelines
- Basic coding practices
- Coding for portability
- Guidelines for clocks and resets
- Coding for synthesis
- Partitioning for synthesis
- Designing with memories



# General naming convention (1)

- Lowercase letters for all signals, variables, and port names
  - e.g. wire *clk, rst*;
- Uppercase letters for names of constants and user-defined types
  - e.g. `define MY_BUS_LENGTH 32
- Meaningful names
  - For a RAM address bus, *ram_addr* instead of *ra*

# General naming convention (2)

- Short but descriptive
  - During elaboration, the synthesis tool concatenates the names
- *clk* for clock signal
  - More than one clock? *-> clk1, clk2 or clk_interface …*
- Active low signals
  - Postfix with '*_n*'
- *rst* for reset signals
  - If active low -> *rst_n*



# General naming convention (3)

- When describing multi-bit buses, use a consistent ordering of bits
  - For Verilog: use (x:0) or (0:x)
- The same or similar names for ports and signals that are connected
  - (e.g. *a*=>*a* or *a*=>*a_int*)
- Case insensitive naming
- Don't use HDL reserved words



# General naming convention (4)

- Other naming conventions
  - **_r*: output of a register
    - count_r
  - **_a*: asynchronous signal
    - addr_strob_a
  - **_z*: tristate internal signal
  - **_pn*: signal used in the $n^{th}$ phase
    - enable_p2
  - **_nxt*: data before being registered into a register with the same name



# Include header in source files

- *Rule*
  - Include a header at the top of every source file
    - Filename
    - Author
    - Description of function and list of key features of the module
    - Date
    - Modification history
      - Date
      - Name of modifier
      - Description of change



# A sample HDL file header

- Example

```
/ *This confidential and proprietary
  *software…
  * © copyright 1996 Synopsys inc.
  *File        : Dwpci_core.v
  *Author    : Jeff Hackett
  *Date        : mm/dd/yy
  *Version   : 0.1
  *Abstract : …
  *Modification History : date, by who, version,
                          change description… */
```



# Use comments

- *Rule*
  - Use comments appropriately to explain all processes, functions, task and declarations of types and subtypes
- *Guideline*
  - Use comments to explain ports, signals, and variables, or group of signals or variables
  - Should be placed logically, near the code that describe
  - Should be brief, concise, and explanatory

# Keep commands on separate lines

- *Rule*
  - Use a separate line for each HDL statement
    - Readable
    - Maintainable



# Line length

- *Guideline*
  - Keep the line length to 72 characters or less
    - Provides a margin that enhances the readability of the code
    - Using carriage returns to divide the lines that exceed 72 characters

always @(*a* or *b* or *c* or *d* or *e* or *f* or *g* or

*h* or *i*)



# Indentation

- *Rule*
  - Use indentation to improve the readability of continued code lines and nested loop
- *Guidelines*
  - Use indentation of 2 spaces (4 spaces is recommended)
  - Avoid use tabs

***example***

```
if(a)
  if(b)
    if(c)
      ...
```



# Do not use HDL reserved words

- *Rule*
  - Do not use HDL reserved words for names of any elements in the HDL source files
    - Do not use VHDL reserved words in Verilog code
    - Do not use Verilog reserved words in VHDL code



# Port ordering

- *Rule*
  - Declare ports in a logical order, and keep this order consistent throughout the design
- *Guidelines*
  - One port per line, with a comment following it (preferably on the same line)
  - Use comments to describe groups of ports

# Port ordering

- *Guideline*
  - Declare the ports in the following order

*Inputs:*

*Clocks*
*Resets*
*Enables*
*Other control signals*
*Data and address lines*

*Outputs:*

*Clocks*
*Resets*
*Enables*
*Other control signals*
*Data*

***example***

```
module my_module(clk, rst, …);
    {      // Inputs:
       clk, // comment for each
       rst, // …
       …    // Outputs:
       …
    }
```



# Port maps and generic maps

- *Rule*
  - Always use explicit mapping for ports and generics, using named association rather than positional association
- *Example*

```
my_module U_my_module(
   .clk(clk),
   .rst(rst),
   …
);
```



# Use functions

- *Guidelines*
  - Use functions instead of repeating the same sections of codes
  - Generalizing the function to make it reusable
  - Use comments to explain the function
- *Example*

```
function [`BUS_WIDTH-1:0] convert_address;
        input input_address, offset;
        begin
                // … function body
        end
endfunction //convert_address
```



# Use loops and arrays

- *Guideline*
  - Use loops and arrays for improved readability of the source code
    - For example, describing a shift register, PN-sequence generator, or Johnson counter
- *Guideline*
  - Arrays are significantly faster to simulate than for-loops
    - Using vector operations on arrays rather than for-loops whenever possible



# Use loops and arrays

- Using loop to increase readability
  - Example

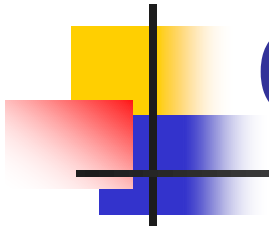
```
module my_module( ... );
...
reg [31:0] reg_file[15:0];
integer tmp;

always @(posedge clk or posedge rst)
If(rst)
for(tmp=0; tmp<16; tmp++)
        reg_file[tmp] <= 32'd0;
...
```

# Use meaningful labels

- *Rule*
  - Label each process block with a  meaningful name
- *Guideline*
  - Label each process block<name>_PROC
- *Rule*
  - Label each instance with a meaningful name
- *Guideline*
  - Label each instance U_<name>
- *Rule*
  - Do not duplicate any signal, variable, or entity names





# Outline

- Overview of the coding guidelines
- Basic coding practices
- Coding for portability
- Guidelines for clocks and resets
- Coding for synthesis
- Partitioning for synthesis
- Designing with memories



# Coding for portability

- To Create code that is
  - Technology-independent
  - Compatible with various simulation tools
  - Easily translatable between Verilog and VHDL
- Constants instead of hard-coded value
  - `define MY_BUS_SIZE 8
  - reg [`MY_BUS_SIZE-1:0] *my_out_bus* ;
- Keep the `define statements for a design in a single separate file and name the file
  - *DesignName*_params.v

# Do not use hard-coded numeric values

- *Guideline*
  - Do not use hard-coded numeric values in your design
- Examples

***Poor coding example***

```
wire [7:0] my_in_bus;
reg [7:0] my_out_bus;
```

***Recommended coding example***

```
`define MY_BUS_SIZE 8

wire [`MY_BUS_SIZE-1:0] my_in_bus;
reg [[`MY_BUS_SIZE-1 :0] my_out_bus;
```



# Include files

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



# Technology Independence

- Avoid embedding dc_shell scripts
  - To prevent from hidden commands in the code
  - Exception: the synthesis directives to turn synthesis on and off must be embedded in the code in the appropriate place
- Use *DesignWare* Foundation Libraries
- Avoid instantiating gates
- If you must use technology-specific gates, then isolate these gates in a separate module

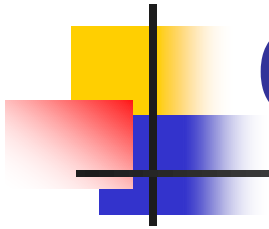
# *DesignWare* Foundation Libraries

- Arithmetic components
  - Adders
  - Multipliers
  - Comparators
  - Incrementers and decrementers
  - Sin, Cos
  - Modulus, divide
  - Square root
  - Arithmetic and barrier shifters
- Sequential components
  - FIFO's and FIFO controllers
  - ECC
  - CRC
  - JTAG components and ASIC debugger

# Technology Independence

- *Guideline→The GTECH library*
  - If you must instantiate a gate, use Synopsys library, GTECH.
    - Contain the following components:
      - AND , OR, and NOR gates (2,3,4,5, and 8)
      - 1-bit adders and half adders
      - 2-of-3 majority
      - Multiplexors
      - Flip-flops
      - Latches
      - Multiple-level logic gates, such as AND-NOT, AND-OR, AND-OR-INVERT





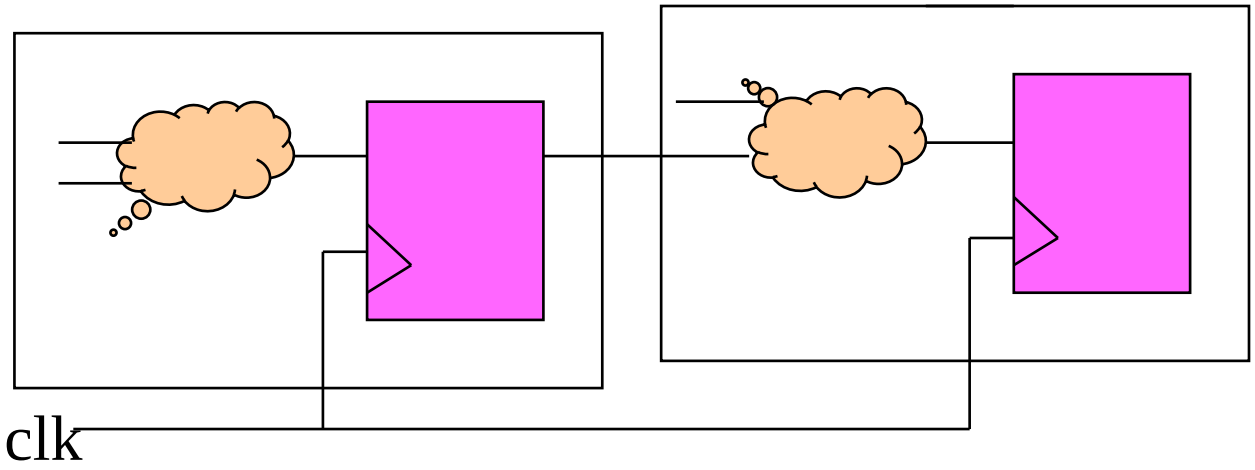
# Outline

- Overview of the coding guidelines
- Basic coding practices
- Coding for portability
- Guidelines for clocks and resets
- Coding for synthesis
- Partitioning for synthesis
- Designing with memories



# Ideal clocking

- Simple clocking structure
  - A single global clock
  - Positive edge-triggered flops as the only sequential devices

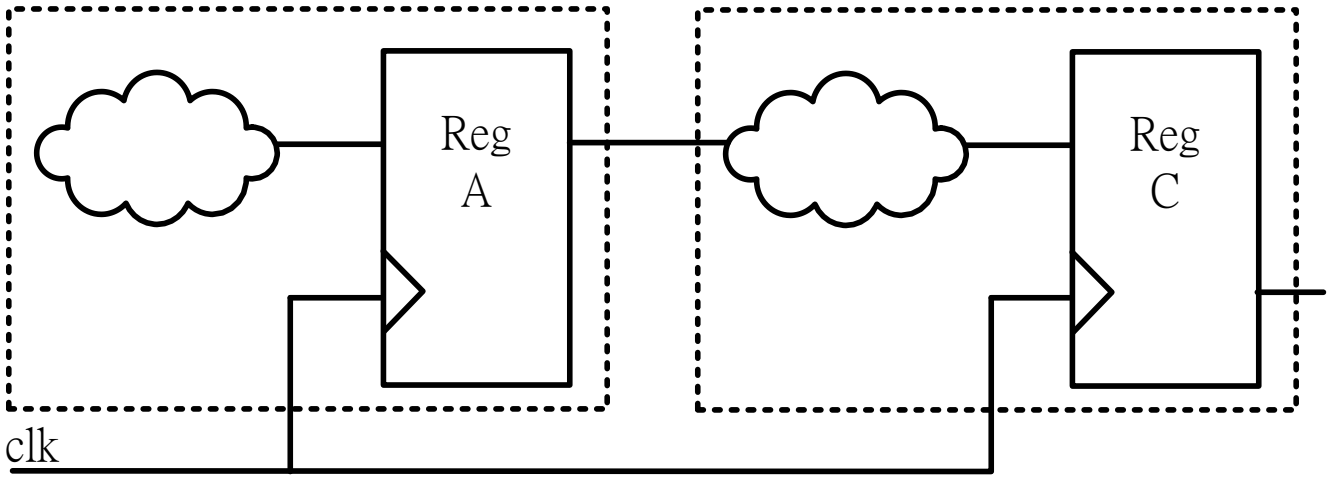


# Avoiding mixed clock edges

- *Impacts*
  - Duty cycle of the clock become a critical issue in timing analysis
  - Most scan-based testing methodologies require separate handling of positive and negative-edge triggered flops
- If you *must use both clock edges*, you must
  - Model the worst case duty cycle accurately
  - Document the assumed duty cycle
  - If you must use many both edge-triggered FFs, separate them into different modules

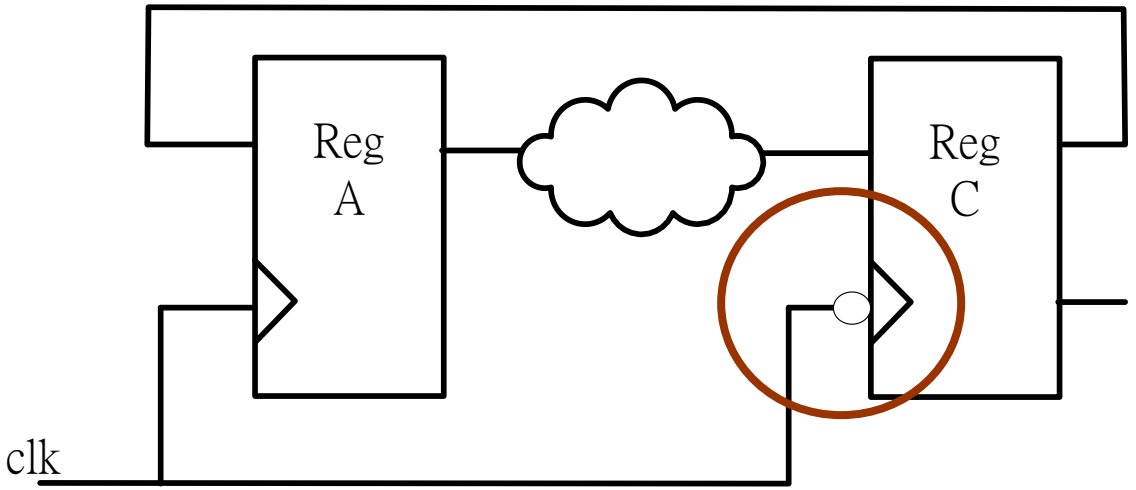
# Examples of clocking structures

- Ideal clocking structure



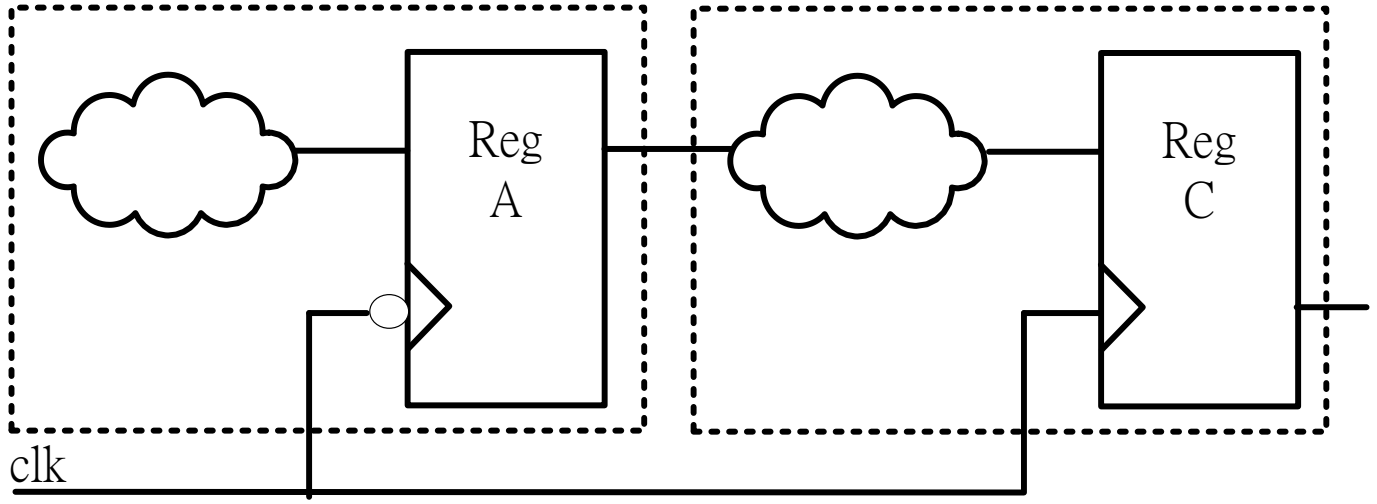
# Examples of clocking structures

- Bad example
  - Using mixed clock edges



# Examples of clocking structures

- Better example
  - Negative-edge and positive-edge flip-flops are separated



# Avoid clock buffers

- *Guideline*
  - Avoid hand instantiating clock buffers in RTL code
  - They are normally inserted after synthesis as part of the physical design
  - During P&R, use clock tree insertion tool

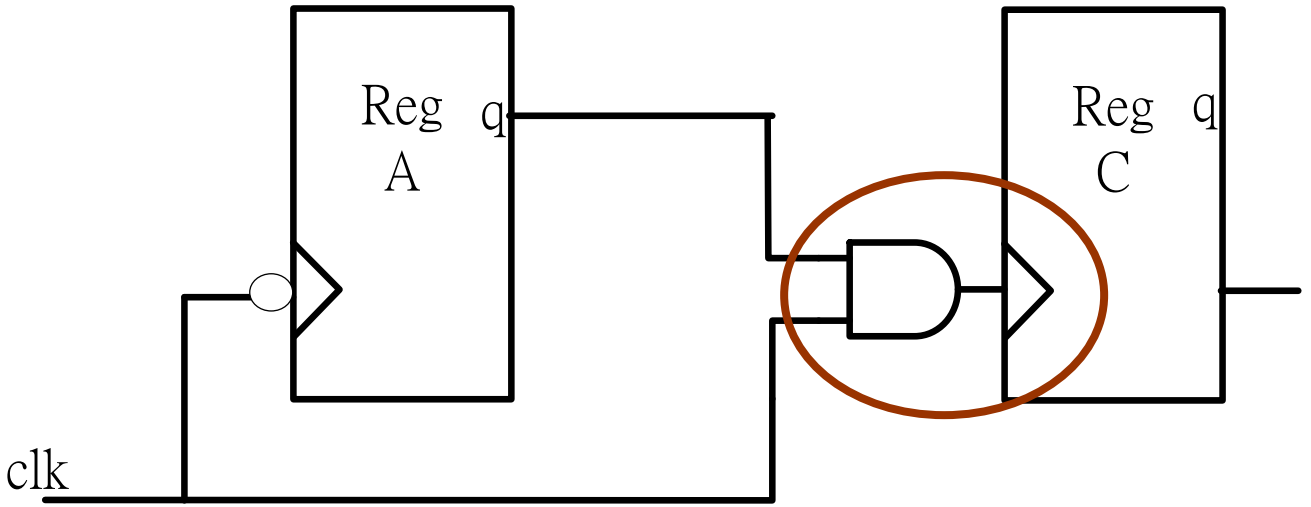


# Avoid gated clocks

- *Guideline*
  - Avoid coding gated clocks in your RTL.
    - Tend to be technology specific and timing dependent
    - Generate a false clock or glitch
    - The skew of different local clocks can cause hold time violations
    - Also cause limited testability, because can't make the scan chain
  - If you must use it
    - keep the clock and/or reset generation circuitry as a separate module at the top level of the design

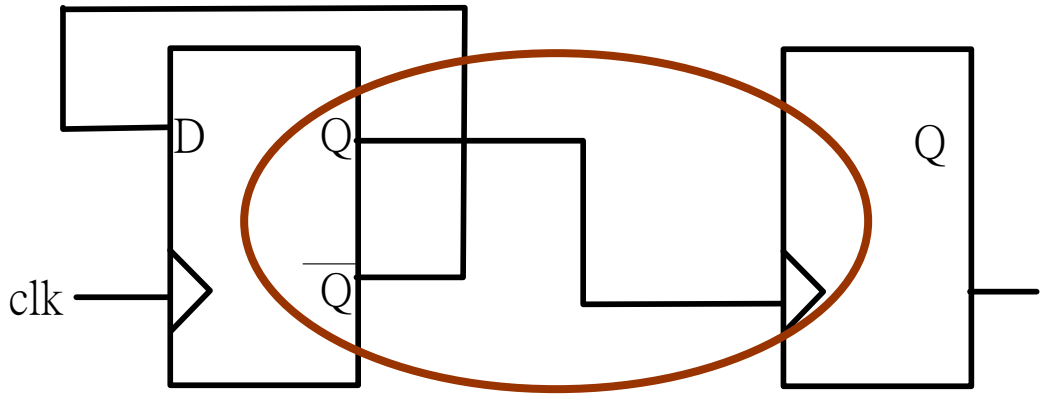
# Avoid gated clocks

- Bad example
  - Limited testability and skew problems because of gated clock



# Avoid internally generated clocks

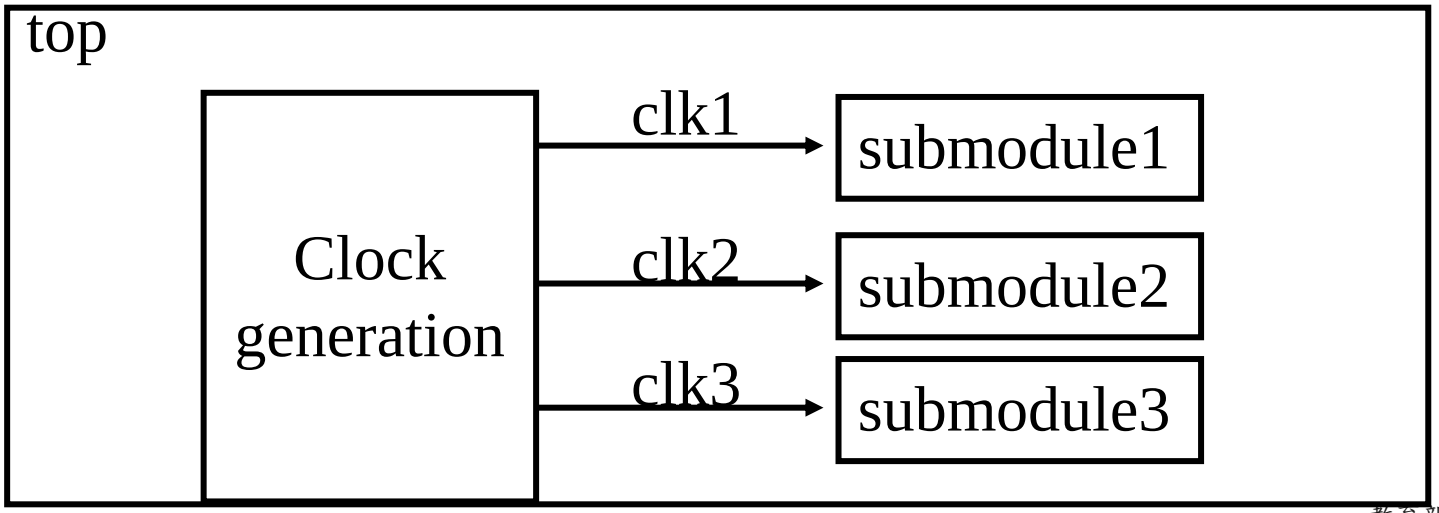
- *Guideline*
  - Avoid internally generated clocks in your design
    - Because logic driven by the internally generated clock cannot be made part of a scan chain
    - Also make it more difficult to constrain the design for synthesis




***Bad example : internally generated clock***

# Gated clocks and low power designs

- *Guideline*
  - If you must use a gated clock, or an internally generated clock or reset, keep the clock generation circuitry as a separate module at the top level of the design
  - Partition the design so that all the logic in a single module uses a single clock and a single reset



# Avoid internally generated resets

- *Guideline*
  - Avoid internally generated, conditional resets if possible
  - Generally, all the registers in the macro should be reset at the same time
  - This approach makes analysis and design much simpler and easier
- *Guideline*
  - If a conditional reset is required, create a separate signal for the reset signal, and isolate the conditional reset logic in a separate module

# Outline

- Overview of the coding guidelines
- Basic coding practices
- Coding for portability
- Guidelines for clocks and resets
- Coding for synthesis
- Partitioning for synthesis
- Designing with memories



# Coding for synthesis

- Guidelines address the synthesis issues
  - Infer registers
  - Avoid latches
  - If you must use a latch
  - Avoid combinational feedback
  - Specify complete sensitivity lists
  - Blocking and non-blocking assignments
  - Case statements versus if-then-else statements
  - Coding state machines

***To achieve best compilation time and synthesis results***



# Infer Register

- *Guideline*
  - Flip-flops are the preferred mechanism for sequential logic
  - Use *reset signal* to initialize registered signals instead of use an initial statement

***example***

```
always @ (posedge clk)
  begin
    if (reset==1'b1)
          begin
            ...
          end
    else
          begin
            ...
          end
  end
```



# Avoid latches

- *Rule*
  - Avoid using any latches in your design
- *Guideline*
  - you can avoid inferred latches bye using any of the following coding techniques
    - Assign default values at the beginning of a process
    - Assign outputs for all input conditions



# Examples of using latches

- Poor coding style
  - Latches inferred because of missing assignments and missing 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
```

# Examples of using latches

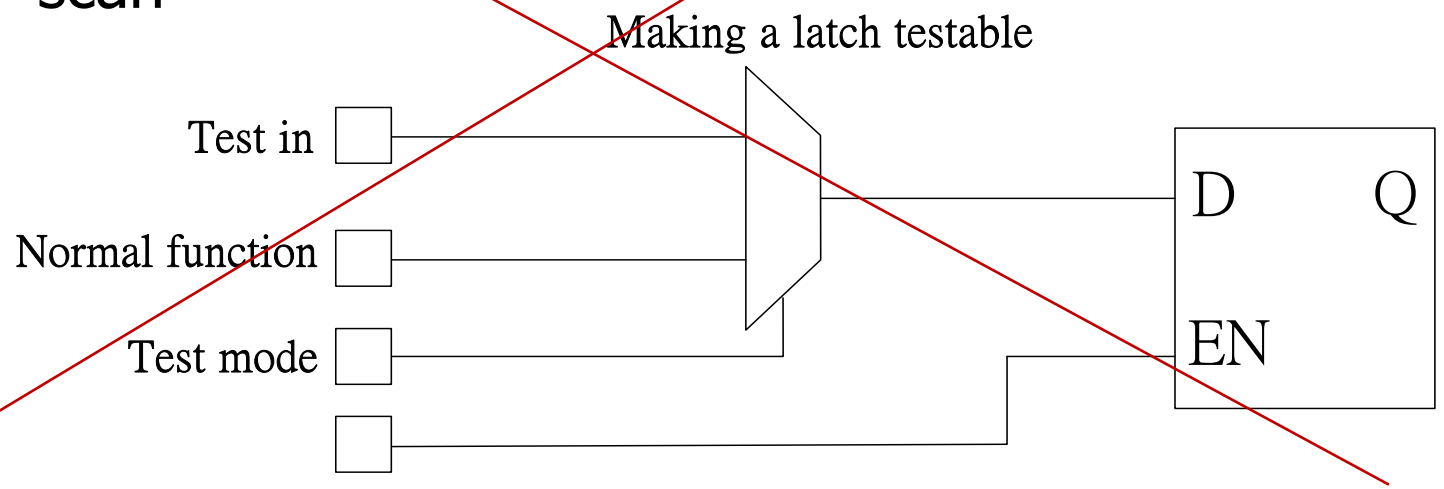
- Recommended coding style

```
always @ (d)
  begin
        case (d)
          2'b00: z<=1'b1; s<=1'b0;
          2'b01: z<=1'b0; s<=1'b0;
          2'b10: z<=1'b1; s<=1'b1;
          2'b11: z<=1'b0; s<=1'b0;
        endcase
  end
```



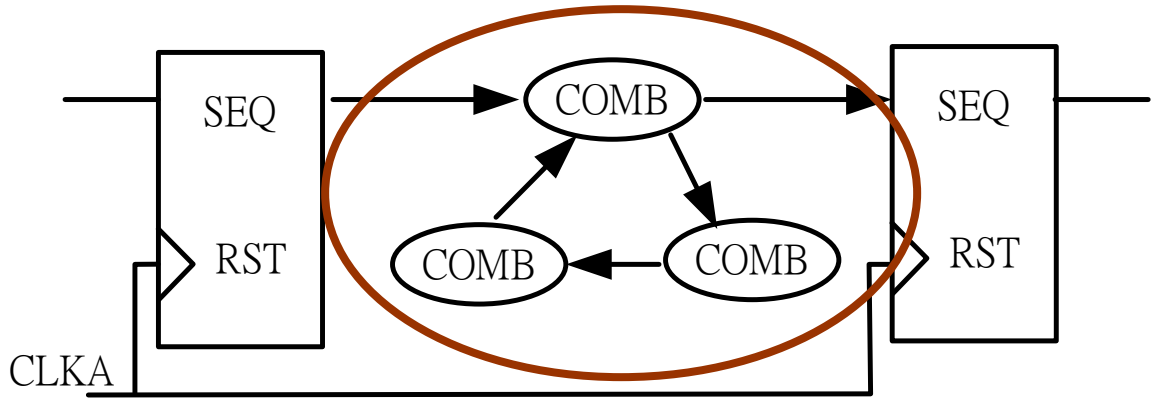
# If you must use a latch

- To achieve testability
  - Use a MUX to provide either the normal function or the input from an I/O pad as data to the MUX
  - The MUX was selected by the mode pin used to enable scan



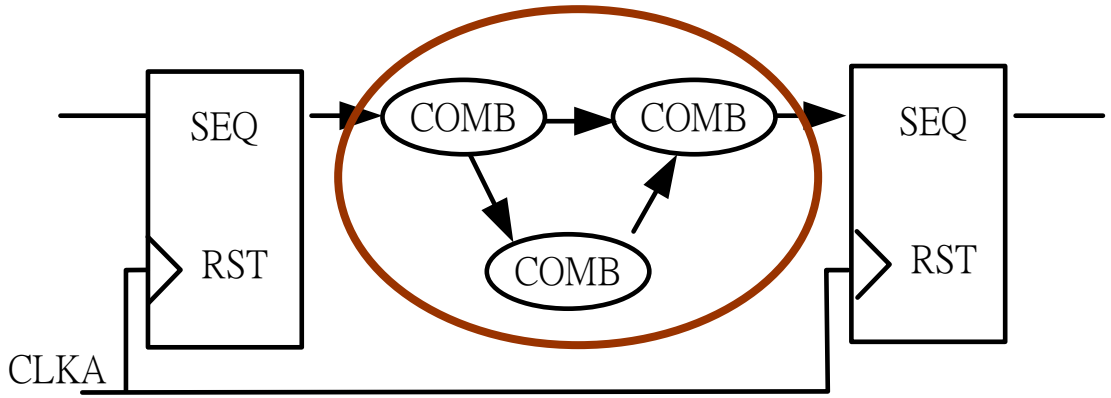
# Avoid combinational feedback

- *Guideline*
  - Avoid combinational feedback
    - The looping of combinational processes
- Bad example
  - Combinational processes are looped



# Avoid combinational feedback

- Good example
  - Combinational processes are not looped



# Specify complete sensitivity list

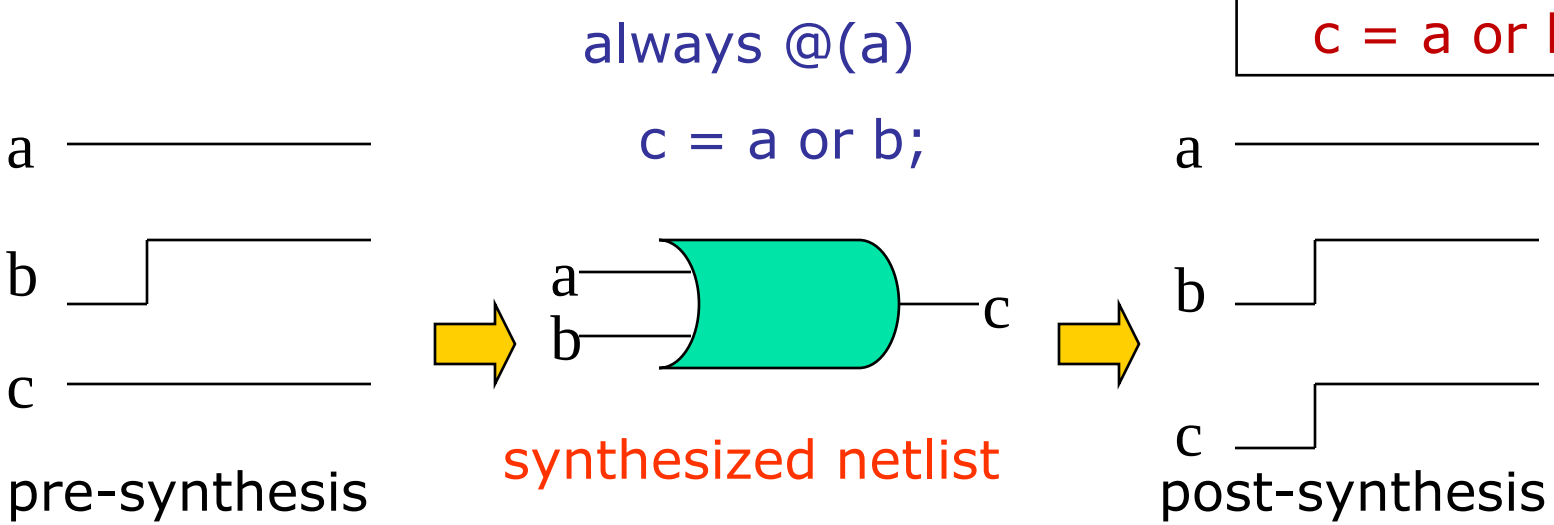
- *Rule*
  - Include a *complete sensitivity* list in each of your *always* blocks
  - Combinational blocks
    - The sensitivity list must include every signal that appears on the right side of an assign (<=) statement or in a conditional expression.
  - Sequential blocks
    - The sensitivity list must include the clock signal that is read by the process.
- *Guideline*
  - Avoid unnecessary signals in list to prevent from slowing down the simulation speed



# Specify complete sensitivity list

- Specify complete sensitivity list avoid difference between pre-synthesis and post-synthesis netlist in combinational blocks

```
always @(a or b)
  c = a or b;
```



# Examples of specifying sensitivity list

- Good coding style
  - Sensitivity list for combinational process block

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



# Examples of specifying sensitivity list

- Good coding style
  - Sensitivity list in a sequential process block

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

```
always @ (posedge clk or negedge rst)
  begin
        if (rst==0)
          q<=0;
        else
          q<=d;
  end
```



# Blocking and non-blocking assignments

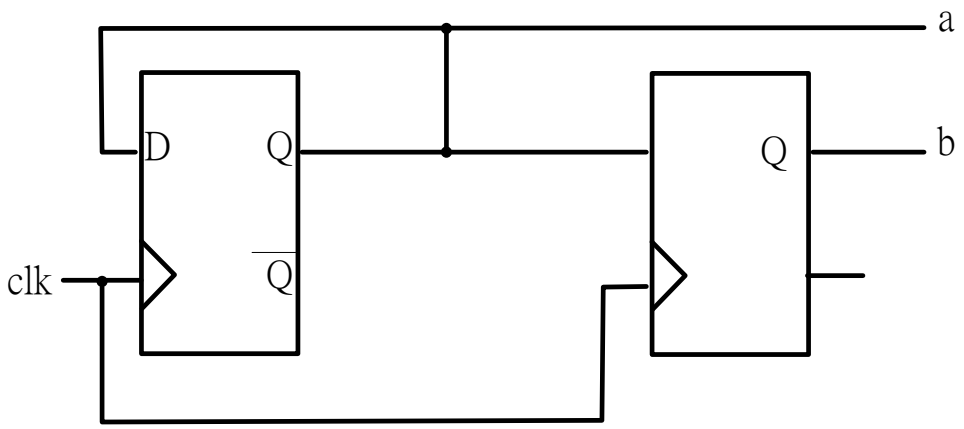
- *Rule*
  - When writing synthesizable code, always use nonblocking assignments in *always @ (posedge clk) blocks*
  - Otherwise, the simulation behavior of the RTL and gate-level designs may differ



# Examples of blocking and non-blocking assignments

- Poor coding style
  - Verilog blocking assignment

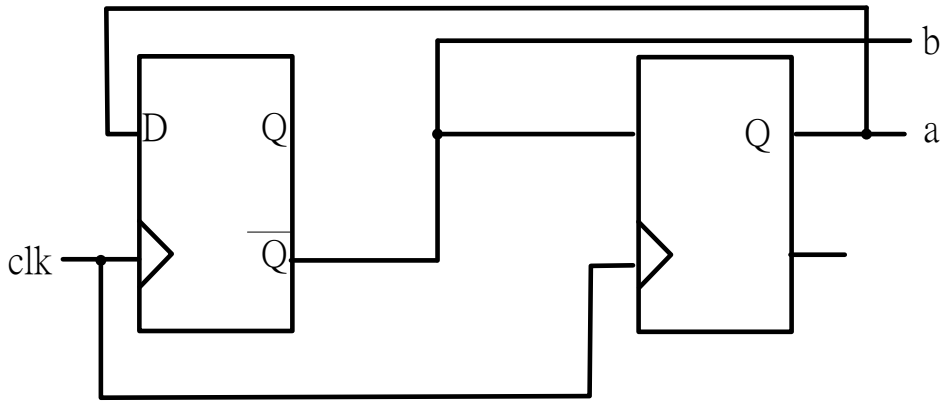
```
always @ (posedge clk)
  begin
        b=a;
        a=b;
  end
```



# Examples of blocking and non-blocking assignments

- Recommended coding style
  - Verilog non-blocking assignment

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

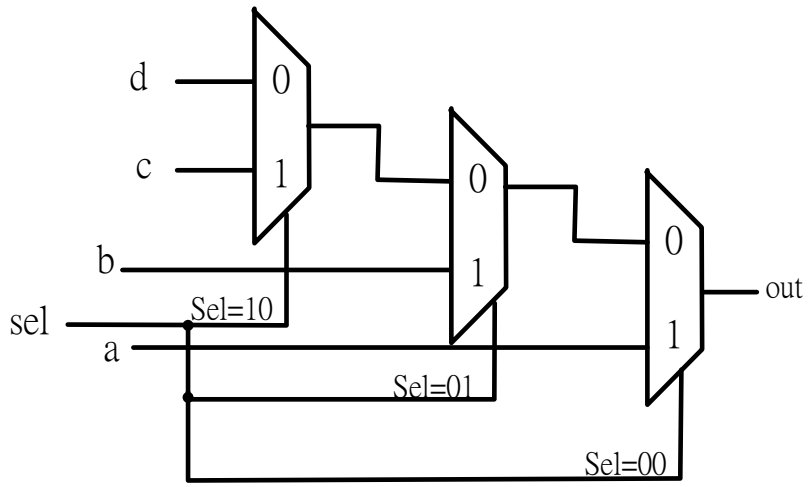


# Case versus if-then-else

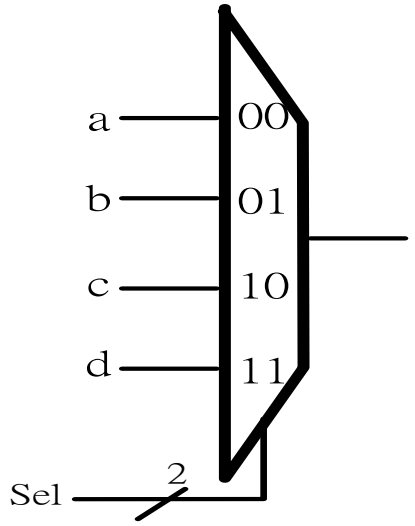
- *Guideline*
  - The multiplexer is a faster circuit. Therefore. If the priority-encoding structure is not required, we recommend using the case statement rather than an if-then-else statement.
- *Synthesis results*
  - Case statements infer single level MUX
    - For large multiplexers, case is preferred because faster in cycle-based simulator
  - If-then-else infer priority encoder
    - useful if you have a late arriving signal
  - Conditional assignment infers MUX
    - assign *z1=(sel_a)? a: b;*

# Case versus if-then-else

- Circuits built from if-then-else statement




- Circuits built from case statement



# Coding state machines

- *Guideline*
  - Separate the state machine HDL description into tow processes, one for the combinational logic and one for the sequential logic
- *Guideline*
  - In verilog, use `define statements to define the state vector
- *Guideline*
  - Keep FSM logic and non-FSM logic in separate modules
- *Guideline*
  - Assign a default state for the state machine



# Examples of coding state machines

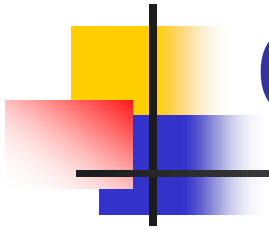
- Poor coding style

```
always @(posedge clk)
   a <= b+c;
```

- Recommended coding style

```
always @(b or c)
   a_nst = b+c;
always @(posedge clk)
   a <= a_nst;
```





# Outline

- Overview of the coding guidelines
- Basic coding practices
- Coding for portability
- Guidelines for clocks and resets
- Coding for synthesis
- Partitioning for synthesis
- Designing with memories



# Partitioning for synthesis

- Good synthesis partitioning in your design provides several *advantages*
  - Better synthesis results
  - Faster compile runtimes
  - Ability to use simpler synthesis strategies to meet timing



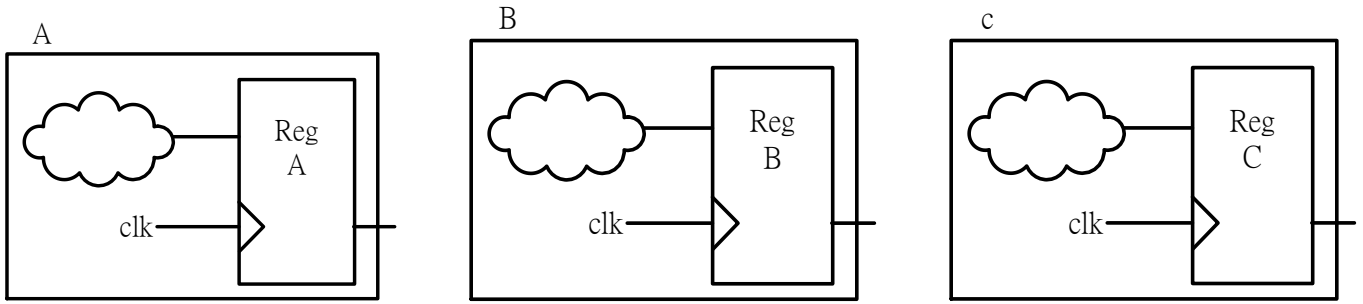
# Partition for Synthesis

- Good partition provides advantages
  - Better synthesis results
  - Faster compile runtimes
  - Ability to use simpler synthesis strategies to meet timing



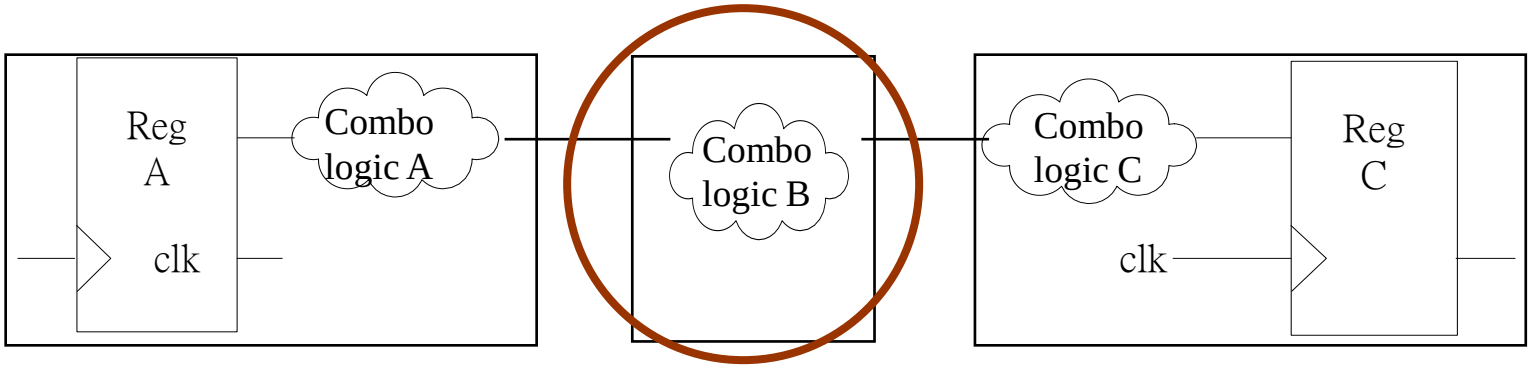
# Register all outputs

- *Guideline*
  - For each block of a hierarchical design, register all output signals from the block
    - Output drive strengths equal to the drive strength of the average flip-flop
    - Input delays predictable
- Good examples



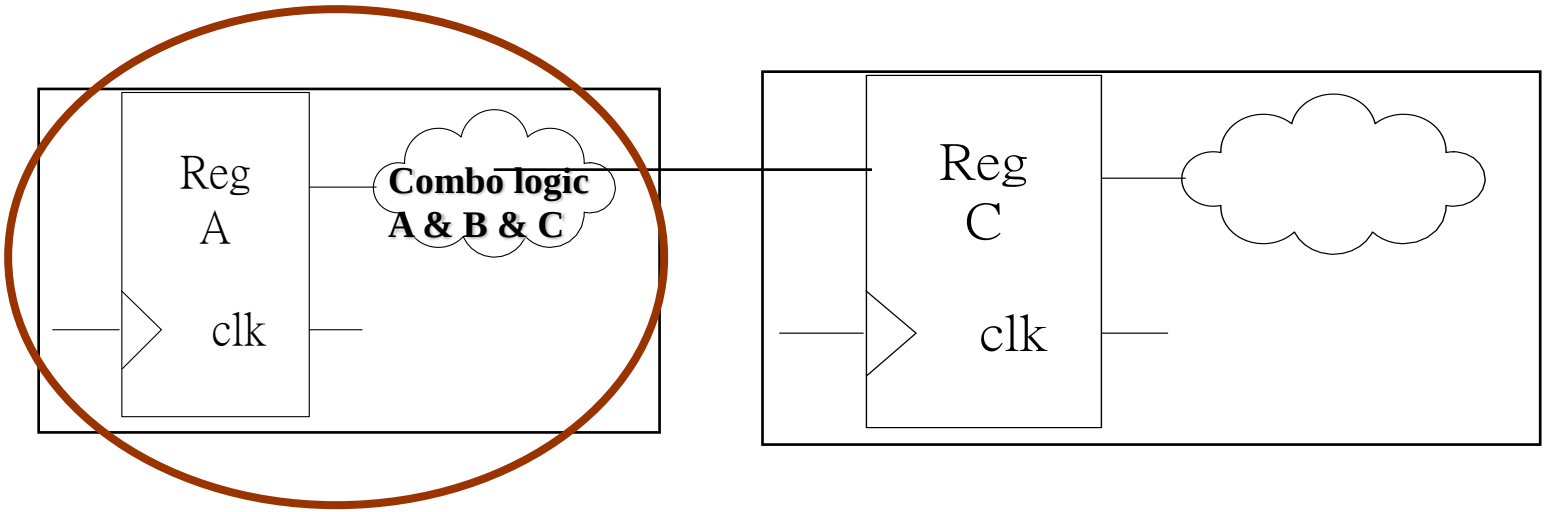
# Locate related combinational logic in a signal module

- *Guideline*
  - Keep related combinational logic together in the same module
- Bad example
  - Combinational logic split between modules



# Locate related combinational logic in a signal module

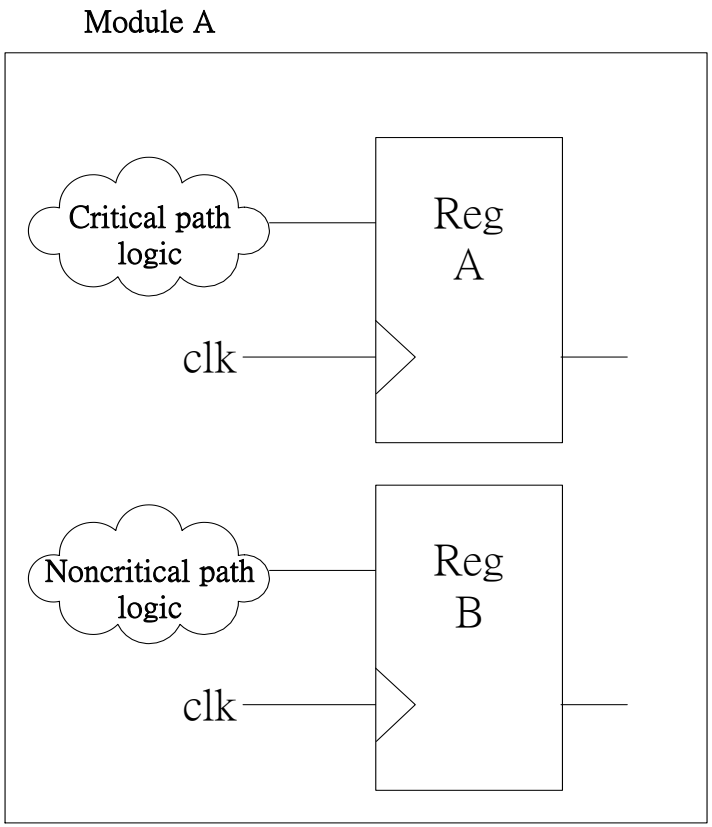
- Good example
  - Combinational logic grouped into same module



# Separate modules that have different design goals

- *Guideline*
  - Keep critical path logic in a separate module from non-critical path logic
    - Can optimize the critical path logic for speed
    - Optimize the non-critical path logic for area

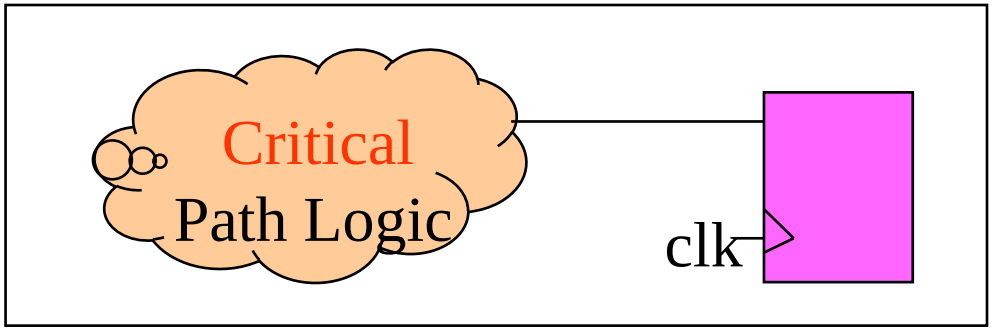
**Bad example**



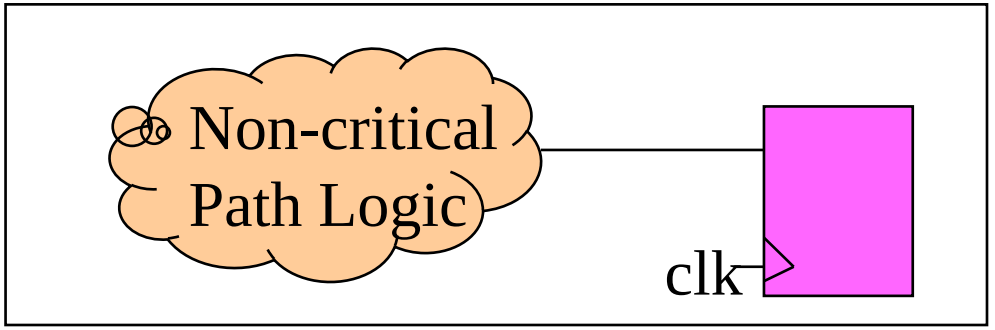
# Separate modules that have different design goals

- Good example

Speed Optimization




Area Optimization



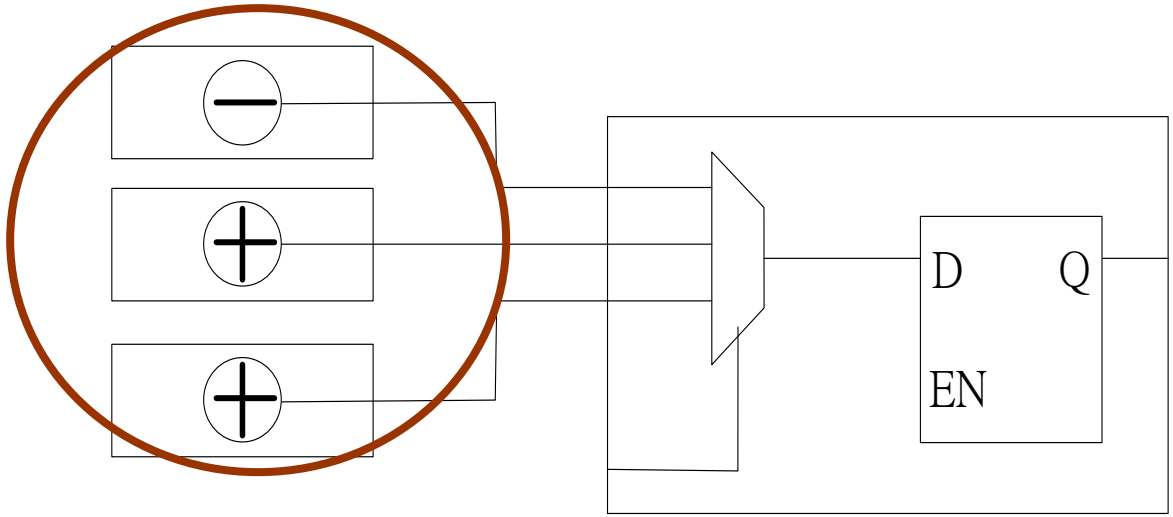
# Asynchronous logic

- *Guideline*
  - Avoid asynchronous logic
- *Guideline*
  - If asynchronous logic is required, partition the asynchronous logic in a separate module form the synchronous logic



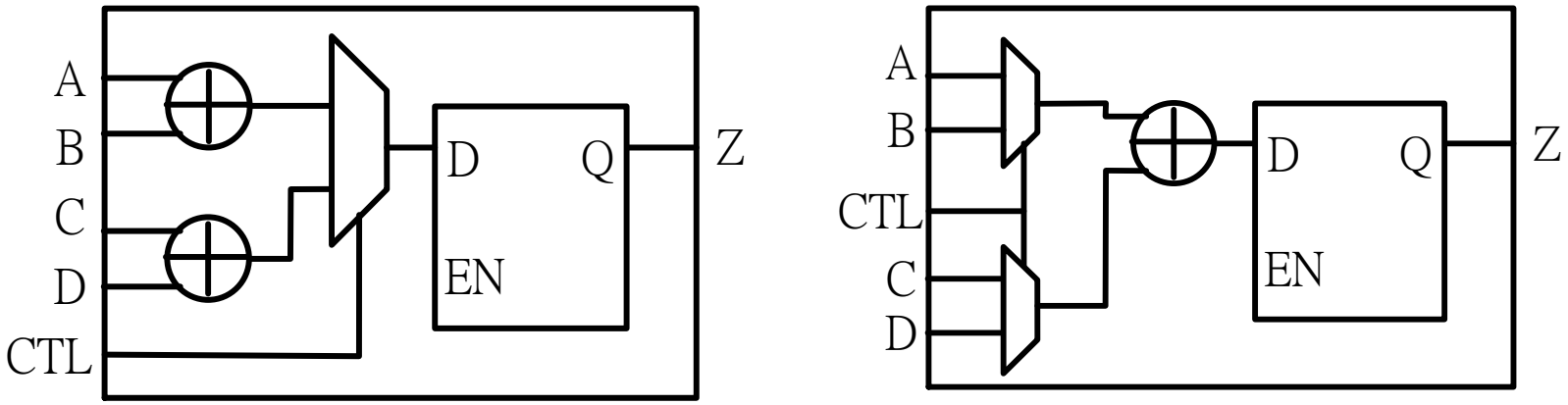
# Merging resources

- Poor partitioning
  - Resources area separated by hierarchical boundaries



# Merging resources

- Good partitioning
  - Adders in the same hierarchy



# Partitioning for synthesis runtime

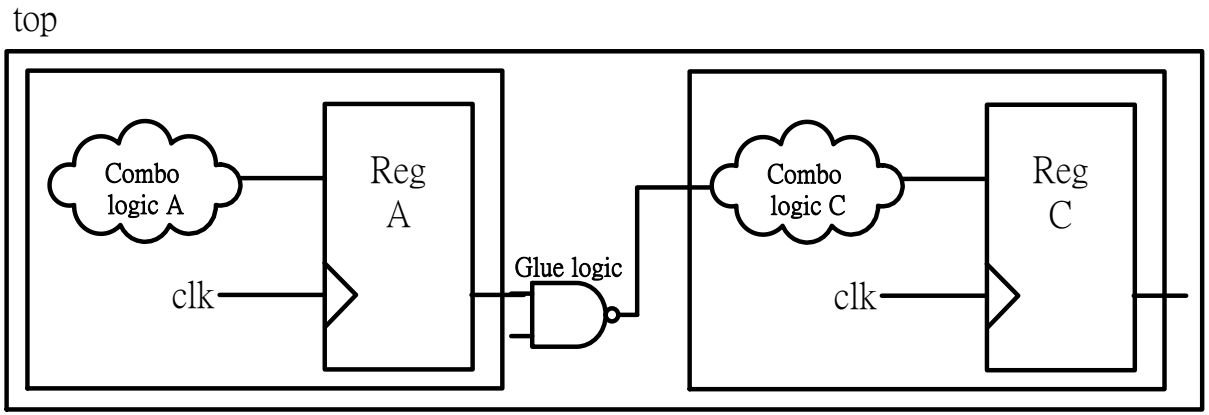
- Good timing budgets and appropriate constraints
  - Have a larger impact on synthesis runtime than circuit size
- A key technique for reducing runtimes
  - To develop accurate timing budgets early in the design phase and design the macro to meet these budgets
- Most important considerations in partition
  - Logic function, design goals and timing and area requirements
- Grouping related functions together

# Avoid point-to-point exceptions and false paths

- *Guideline*
  - Avoid multicycle paths in your design
- *Guideline*
  - If you muse use a multicycle path in your design, keep point-to-point exceptions within a single module, and comment them well in your RTL code.
- *Guideline*
  - Avoid false paths in your design

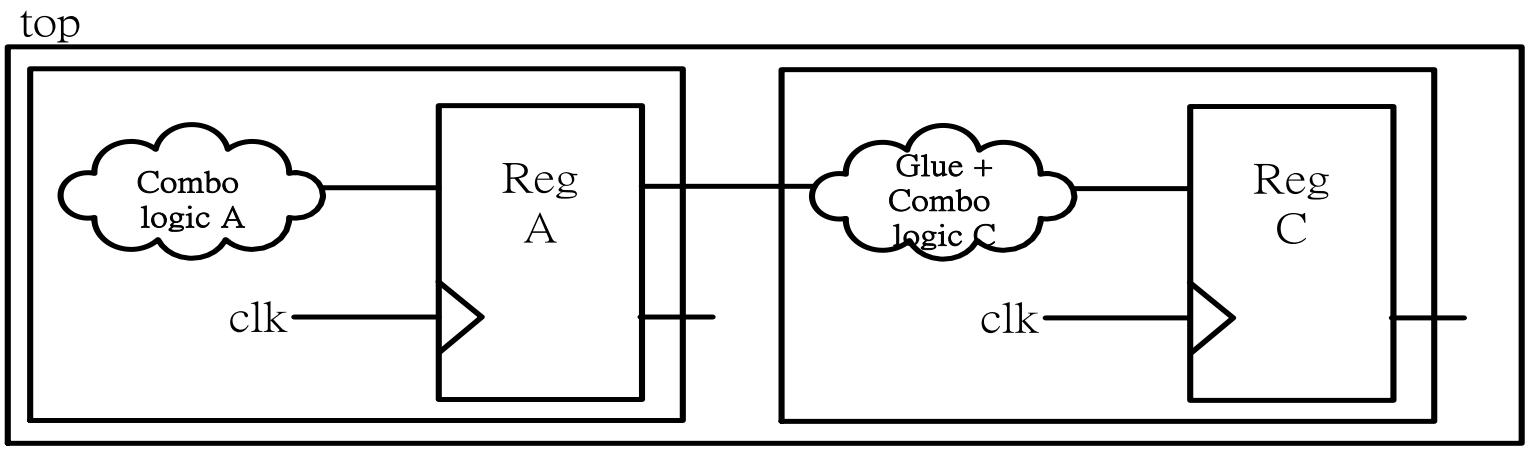
# Eliminate glue logic at the top level

- *Guideline*
  - Do not instantiate gate-level logic at the top level of the design hierarchy
- Bad example



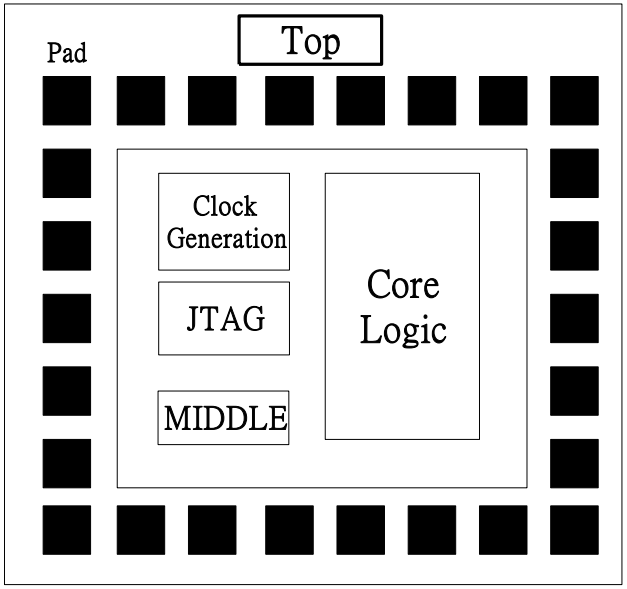
# Eliminate glue logic at the top level

- Good example



# Chip-level partitioning

- *Guideline*
  - Make sure that only the top level of the design contains an I/O pad ring



# Outline

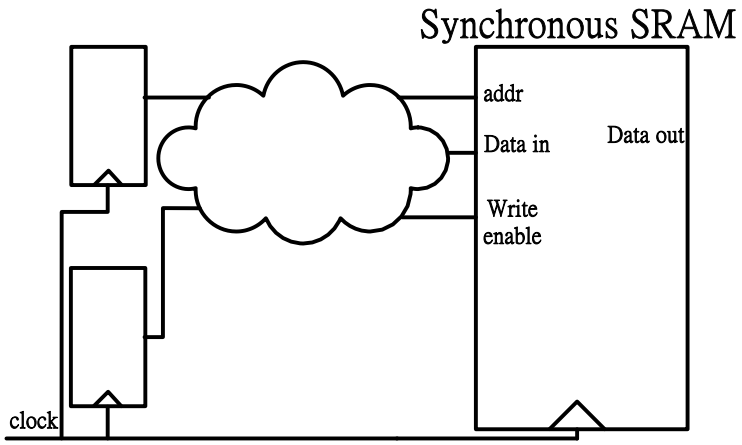
- Overview of the coding guidelines
- Basic coding practices
- Coding for portability
- Guidelines for clocks and resets
- Coding for synthesis
- Partitioning for synthesis
- Designing with memories



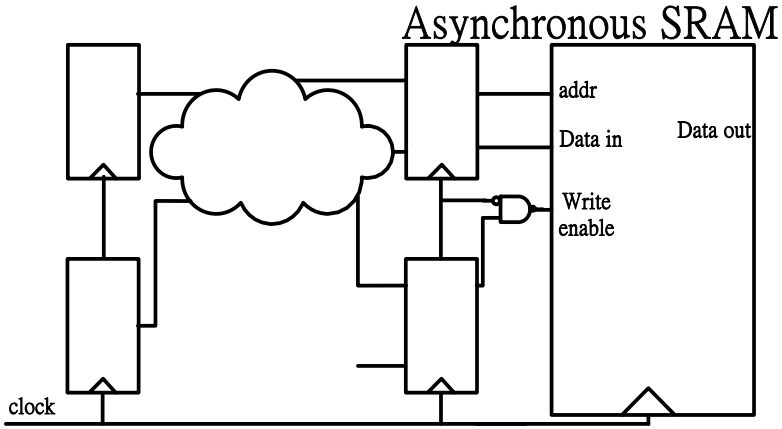
# Designing with memories

- *Guideline*
  - Partition the address and data registers and the write enable logic in a separate module

***Synchronous memory interface***

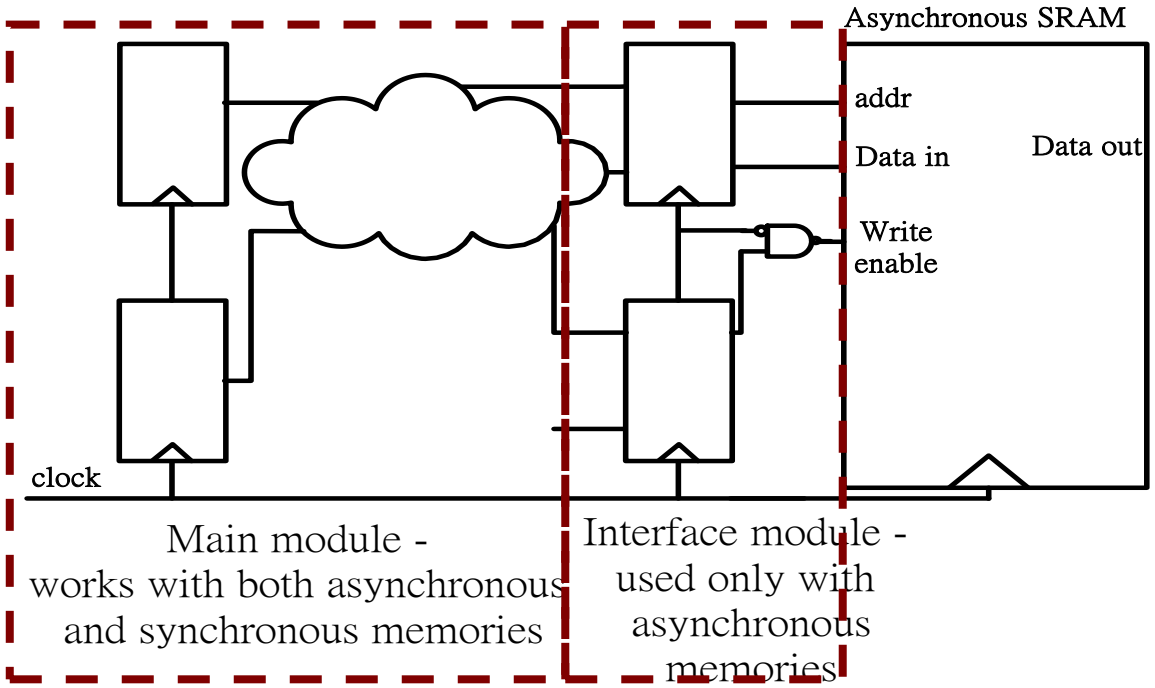



***Asynchronous memory interface***



# Designing with memories

- Example
  - Partition memory control logic separately



# 期末考範圍

- 4.dataflow
  - p. 116 Excercises:  1, 2, 3
- 5.behavioral
  - p. 167 Exercise p.167
  - 2, 3, 4, 7, 8, 10, 11, 12, 14, 15, 16
- 6.Logic Synthesis(14章)
  - p. 340 Exercise: 1, 2, 3, 4, 5