Verilog Coding Style

Optimize the Right Thing

Good Comments Improve Maintainability

T.Y. Hsu

Coding Guidelines

- **Reasons for coding guidelines**
  - **Enhance readability & understandability**
  - **Maintainable designs “in the large”**
  - **Avoid common pitfalls**
  - **Learning what you “should do” vs. what you “can do”**
- **Guidelines vary - choose a set and be consistent**

Suggested Coding Style (1/2)

Write one module per file, and name the file the same as the module. Break larger designs into modules on meaningful boundaries.

- Always use formal port mapping of sub-modules.
- Use parameters for commonly used constants.
- Be careful to create correct sensitivity lists.

Suggested Coding Style (2/2)

Don’t ever just sit down and “code”. Think about what hardware you want to build, how to describe it, and how you should test it.

- You are not writing a computer program, you are describing hardware… *Verilog isn’t “C”* *!*
- Only you know what is in your head. If you need help from others, you need to be able to explain your design -- either verbally, or by detailed comments in your code.

General Coding Guidelines

- Create a separate Verilog file for each module. Name each file `<module_name>.v`.
- Limit lines to 80 characters or less.
- Avoid tabs when indenting.
- Use 4 spaces for each level of indentation.
- Use a standard header in each Verilog file
- Use a uniform naming convention
- Use consistent ordering for bit vectors `[high:low]`
- Include Comments
- Use symbolic constants with `parameter` and/or `` `define ``
- Collect symbolic constants into “include files”
- Use consistent order in port lists (out / inout / in / control)

General Coding Guidelines

- Create a separate Verilog file for each module. Name each file `<module_name>.v`.
- Limit lines to 80 characters or less.
- Avoid tabs when indenting.
- Use 4 spaces for each level of indentation.

```
module counter(clk, rst, ld, d, q_r);
    input        clk;
    input        rst;
    input  [3:0] d;
    output [3:0] q_r;

    always @(posedge clk)
        begin
            . . .
        end
endmodule
```

General Coding Guidelines

- **Use a uniform naming convention**
  - **Clock signals: `clk` or `clk1`, `clk2`, …**
  - **Reset/Preset signbal: `rst, reset, preset, prst`**
  - **Other signals - use "postfix" to indicate type:**

Extension	Meaning
`_r`	Output of a register (e.g., `count_r`)
`_a`	Asynchronous signal (e.g. `req_a`)
`_nxt`	Data before being stored in a register with the same name (e.g. `count_nxt`)
`_n`	Active low signal
`_z`	Tristate signal

- **Use consistent ordering for bit vectors `[high:low]`**
  - `input [3:0] q_r;`
  - `input [3:0] data;`
  - `wire [0:3] bug; <- Error !!`

General Coding Guidelines

- Include Comments
  - Document ports, wires
  - Explain non-obvious code
- Use symbolic constants with `parameter` and/or `` `define ``
- Collect symbolic constants into “include files”
  - `` `include “system_decls.v” ``

General Coding Guidelines

- **Use consistent order in port lists** **(out / inout / in / control) or ……**
  - Control:
    - Clocks
    - Resets
    - Enables
    - Other control signals
  - Input:
    - Data
  - Outputs:
    - Clocks
    - Resets
    - Enables
    - Other control signals
    - Data
  - Bidirectional (inout) signals

Do you use a Asynchronous or Synchronous Reset?

- It is not recommended that you use a asynchronous reset unless it is absolutely necessary.
- It slows down your code
- If your design is large you can't guarantee that all parts of it will end up in the correct state afterwards

Why is it slow?

```
//Asynchronous Reset
always@ (posedge clk or negedge rst)
begin
        if(~rst)
                value = 0;
        else
                value = value +5;
end
```

Max Clock Freq: 206.4 Mhz

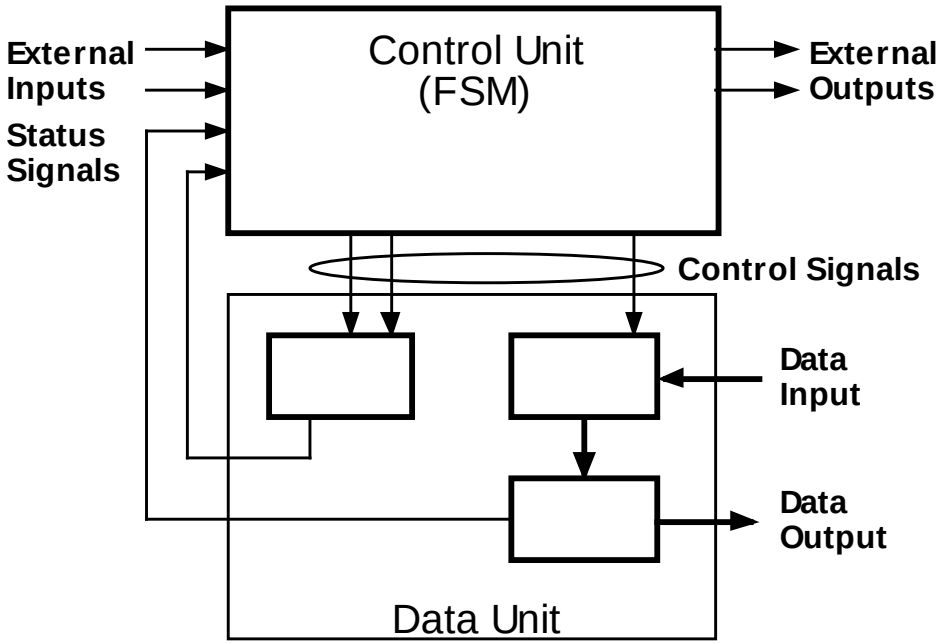
```
//Synchronous Reset
always@ (posedge clk)
begin
        if(~rst)
                value = 0;
        else
                value = value +5;
end
```

Max Clock Freq: 450.0Mhz

It simply takes more logic

The Datapath-Controller Abstraction

- **Key idea: break up design into two parts:**
  - **Datapath- components that manipulate data**
  - **Controller - FSM that controls datapath modules**



Ex: Accuracy & Efficiency

```
module orNand1(y, en, a, b, c, d );
      input en, a, b, c, d;
      output y;
      reg y;

      always @ (en or a or b or c or d )
         y = ~( en & (a | b) & (c | d) );
endmodule
```

Lose efficient

```
module orNand1(y, en, a, b, c, d );
      input en, a, b, c, d;
      output y;
      reg y;

      always @ ( en )
         if ( en )
               y = ~((a|b) & (c|d));
         else
               y = 1;
endmodule
```

Hint: “en” is a control signal in the design

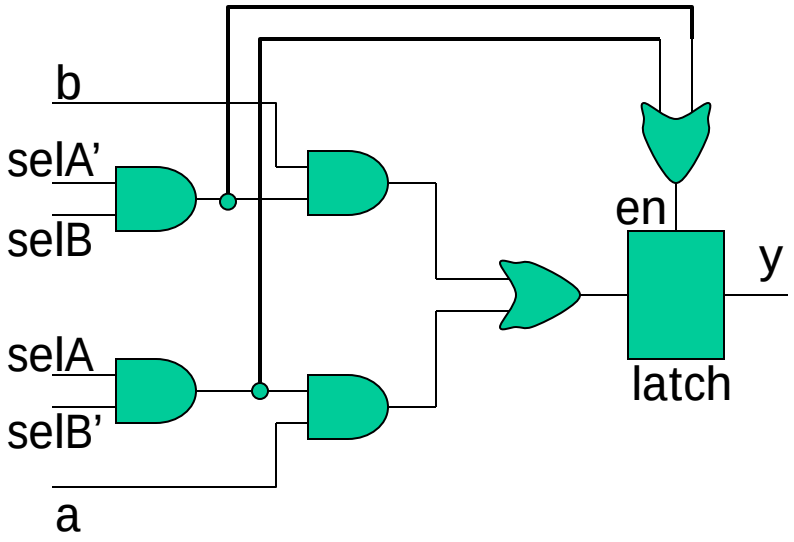
Coding Guidelines Module Instantiation and Hierarchy

- Use long form for module connections
- Avoid mixing “module instantiations” and “procedural code”
  - Ex:

```
module f_adder( ….. );

    h_adder  add1( …..);

    always @(posedge clock)
      begin


      end
endmodule
```

Ex: Unwanted Latch

```
module myMux( y, selA, selB, a, b );
    input selA, selB, a, b;
    output y;
    reg y;

    always @ ( selA or selB or a or b )
        case ( {selA, selB} )
            2'b10: y = a;
            2'b01: y = b;
        endcase
endmodule
```



Miss to use a full case

Coding for Synthesis

Non-Synthesizable Verilog

- Certain high-level behavioral constructs
  - Special equality and inequality (===, !==)
  - while, wait, forever
  - Initial
  - Delay specifications
  - System calls (e.g., $time, $display)
  - force … release
  - blocking and non-blocking in same always block
  - Variable for loops

Bad Always Blocks

```
always @(posedge clk) begin
    if (state > 2'b00) begin
        data = mem[count];
        count <= count + 1;
    end
end
```

```
always @(posedge clk) begin
    count = count + 1;
    output1 = mem[count];
    count = count + 1;
    output2 = mem[count];
end
```

What is wrong with each code segment?

```
always @(posedge clk) begin
    count <= count + 1;
    output1 <= mem[count];
    count <= count + 1;
    output2 <= mem[count];
end
```

Bad Always Blocks

```
always @(count) begin
    count = count + 1;
end
```

What is wrong with each code segment?

```
always @(state) begin
    if (state == 2'b01)
        output = 1'b1;
end
always @(state, a) begin
    if (state == 2'b00)
        output = 1'b0;
end
always @(b) begin
    output = 1'b1;
end
```

Swapping Values

- **Two better options**

```
always @(left, right) begin
    newleft = right;
    newright = left;
end

always @(posedge clk) begin
    right <= newright;
    left <= newleft;
end
```

```
always @(posedge clk) begin
    left <= right;
    right <= left;
end
```

What Is A Register?

- **Written to on clock edge!**
- **“reg” doesn’t make it be a register!**

Sequential!

```
reg [3:0] myregister;
always @(posedge clk) begin
    myregister <= a + b;
end
```

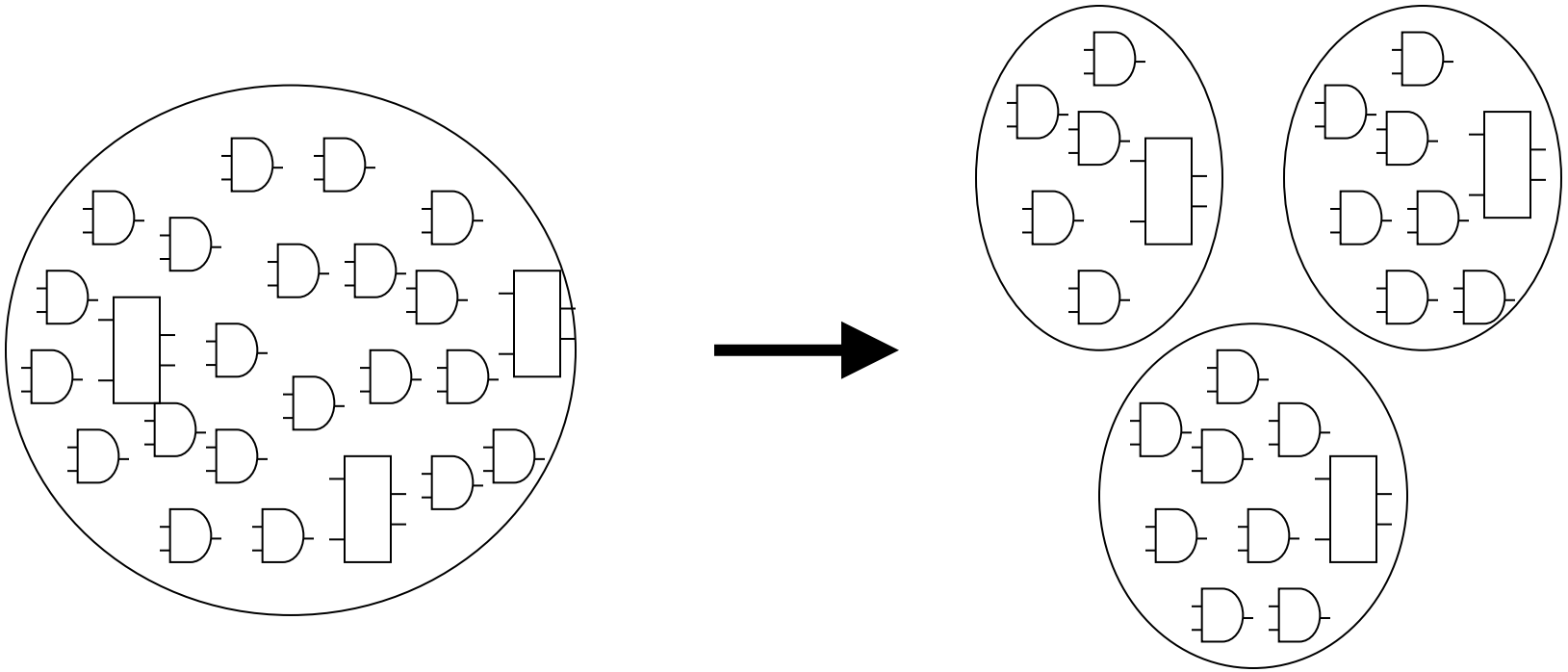
Combinational!

```
reg [3:0] notregister;
always @(a, b) begin
    notregister = a + b;
end
```

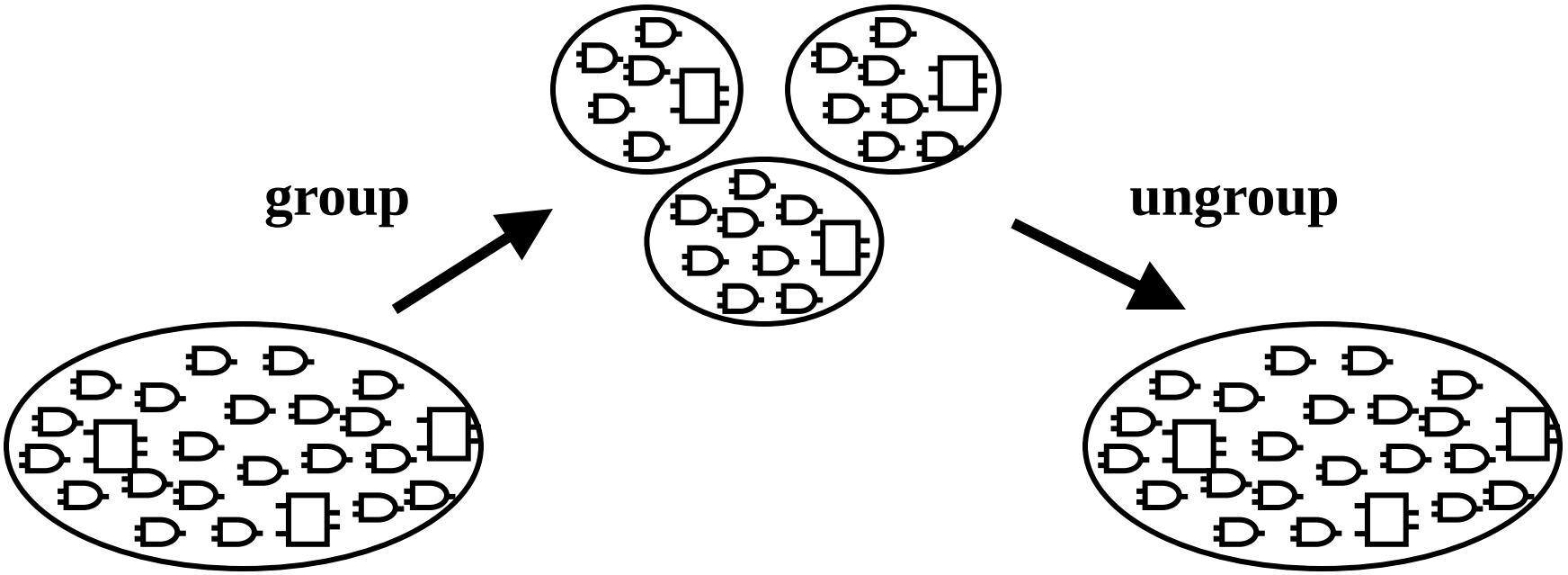
Partition for Synthesis

What is Partition ?

- **Partitioning is dividing a design into smaller parts.**



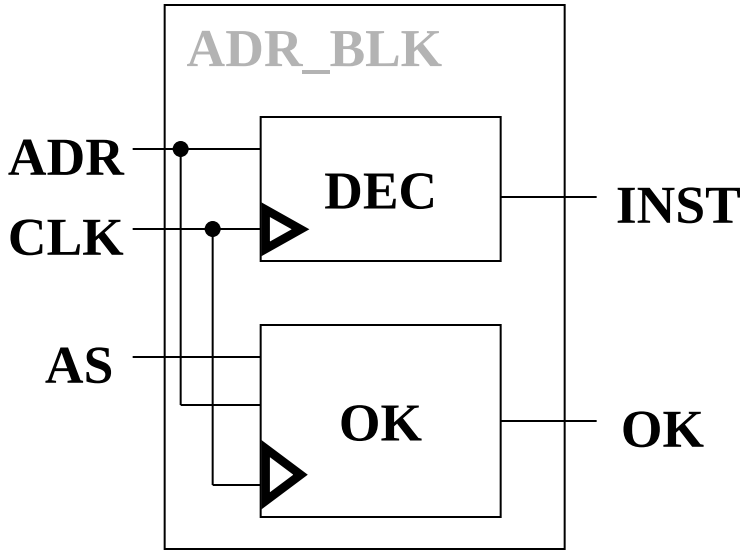
Partition within Design Compiler



- The group and ungroup commands manipulate hierarchy within Design Compiler

Partition within HDL Codes

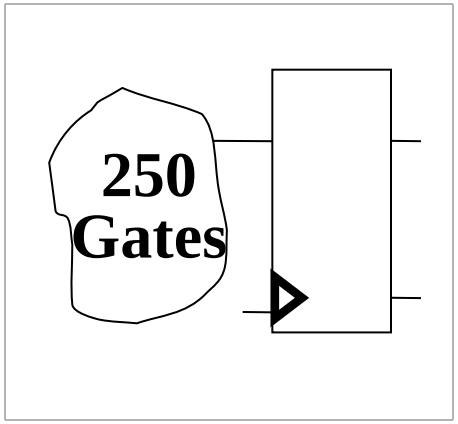
```
module ADR_BLK(…..
U1: DEC(ADR,CLK,INST)
U2: OK(ADR,CLK,AS,OK)
end module;
```



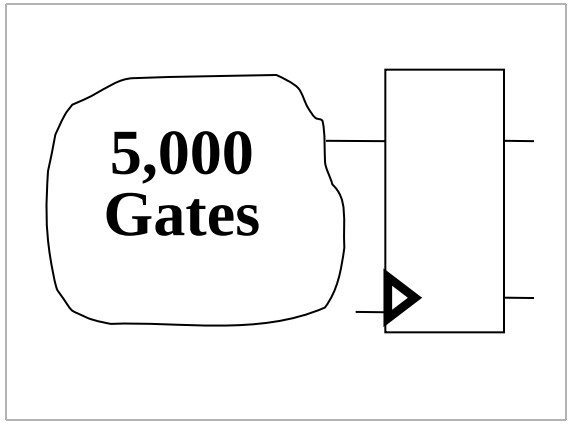
- Module statements create hierarchical design blocks
- Instantiating an entity creates a level of hierarchy
- Concurrent/continuous assignments and process/always @ statements do not create hierarchy

Rule: Maintain a Reasonable Block Size – (250-5000 gates)

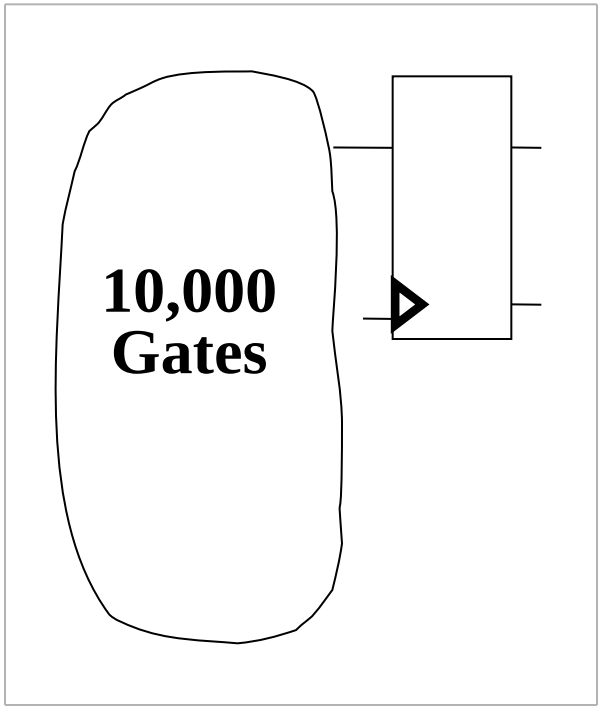
SMALL



BIG

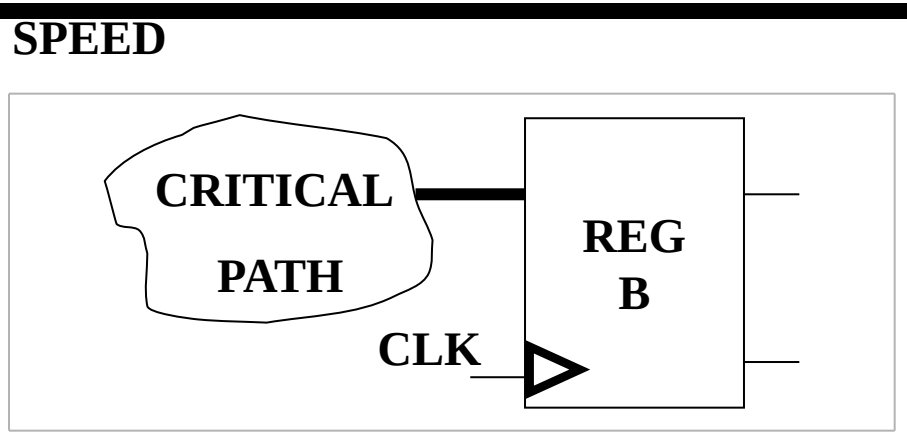


BIGGEST

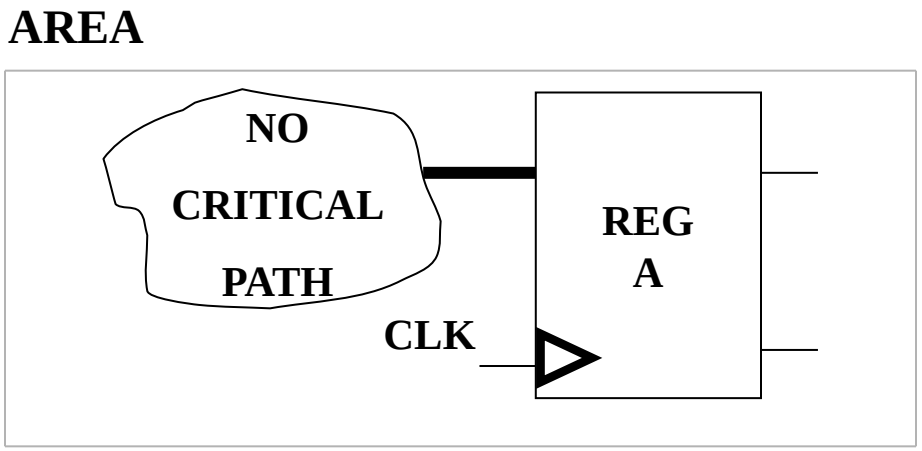


Rule: Separate Designs With Different Goals

Speed Optimized



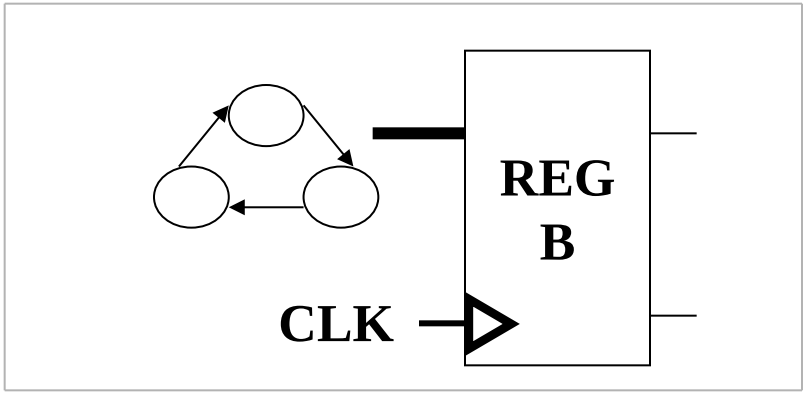
Area Optimized



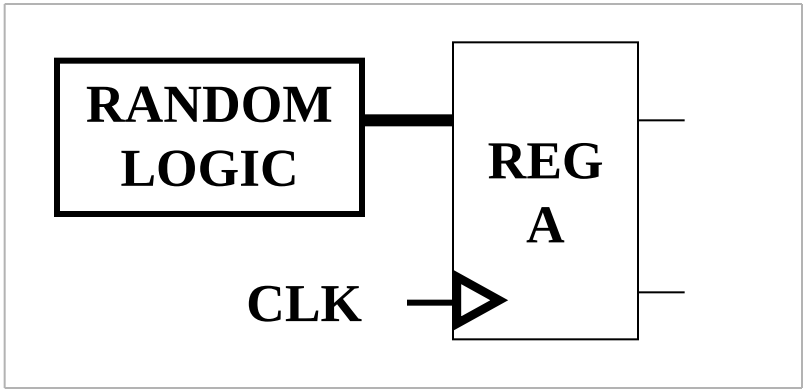
Rule: Isolate State Machines

FSM

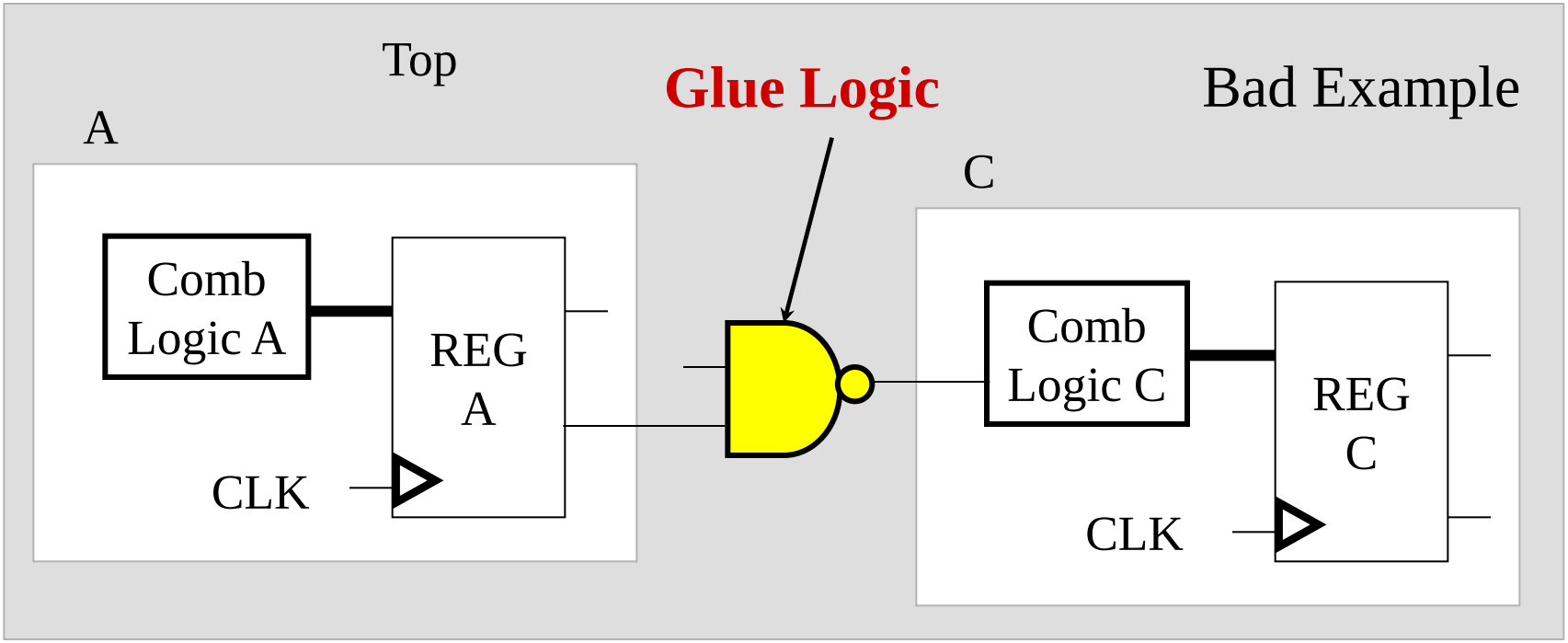
Use FSM Optimization Tool



Use Standard Compile Techniques

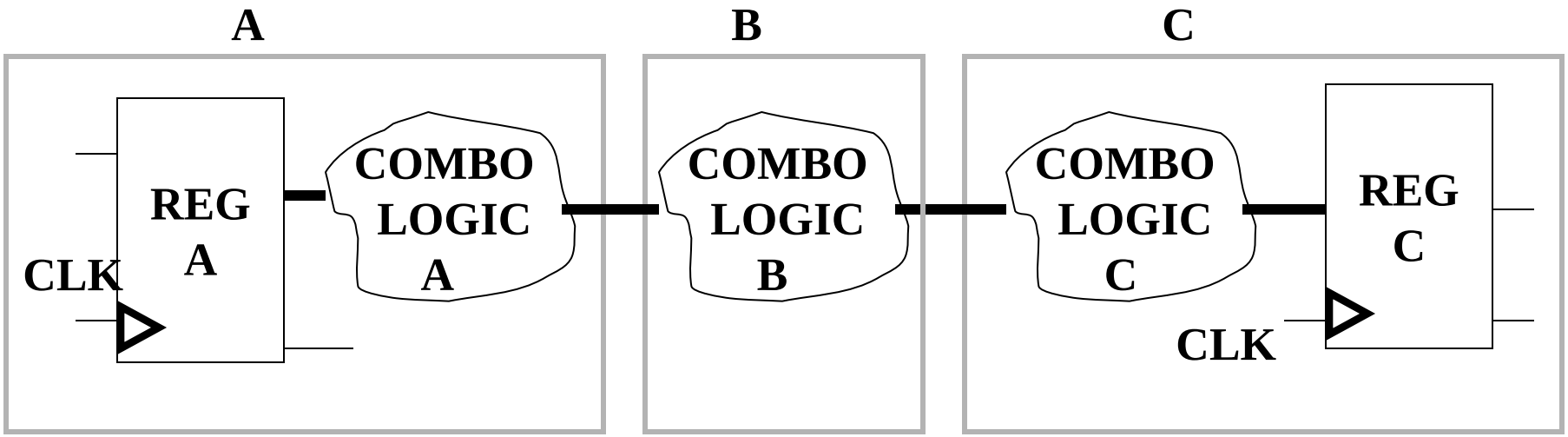


Rule: No Glue Logic Between Blocks - Bad Example



- A NAND bridges the two instantiated lower-level blocks.
- Glue logic can't be passed to the lower-level blocks.

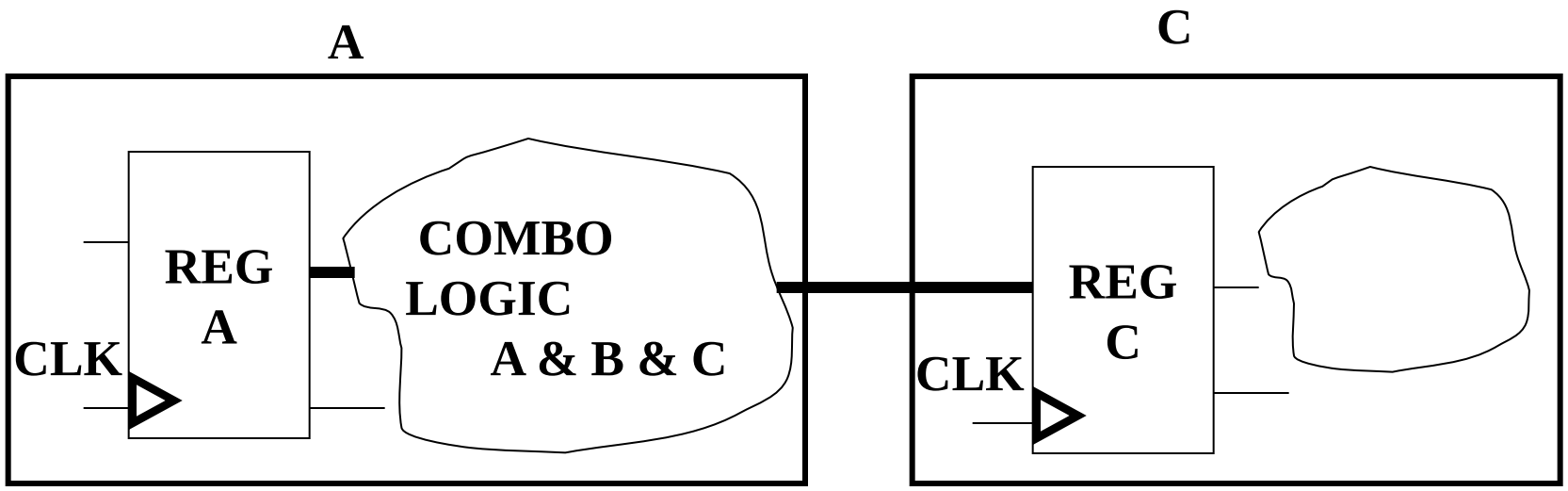
Rule: No Hierarchy in Combinational Paths – Bad



Bad Example

- **Optimization is limited because hierarchical boundaries prevent sharing of common terms.**

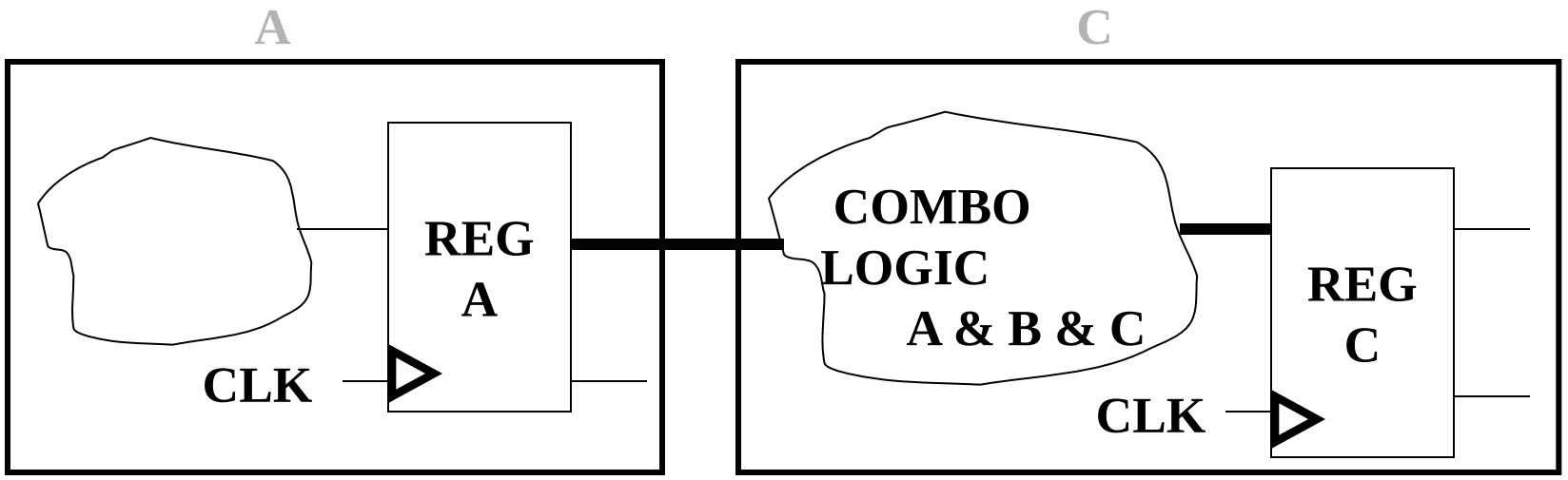
Rule: No Hierarchy in Combinational Paths – Better



Normal Example

- Related combinational logic is grouped into one block

Rule: No Hierarchy in Combinational Paths – Best



Best Example

- Combinational logic is grouped with the destination flip-flop.
- Allows for sequential mapping during optimization.

Summary: Partition Rules

- No hierarchy in combinational paths.
- Register all outputs.
- No glue logic between blocks.
- Separate designs with different goals.
- Isolate state machines.
- Maintain a reasonable block size.
- Separate core logic, pads, clocks and JTAG.