

111-1 DCLab

# Synthesizable Verilog Coding

**Presenter: Chung-Hsuan Yang**

**Advisor: Prof. Chia-Hsiang Yang**

*Graduate Institute of Electronics Engineering, National Taiwan University*



DCSL

# Outline

---

- Brief Introduction to Logic Synthesis
- Syntax for Synthesis
- Partition for Synthesis
- Circuit-Level Coding Skills
  - Translation between circuits and codes
  - Circuit Refining
- Check for Synthesizability

# Brief Introduction to Logic Synthesis (1/2)

---

- Process of converting a **high-level description of design** into an **optimized gate-level representation**.
- Logic synthesis uses **standard cell library**
  - Basic logic gates like **and**, **or**, and **nor**
  - Macro cells like adder, multiplexers, memory, and special flip-flops.
- Constraint-driven
  - Timing, area, testability, and power.

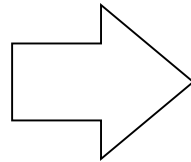
# Brief Introduction to Logic Synthesis (2/2)

- Synthesis Flow

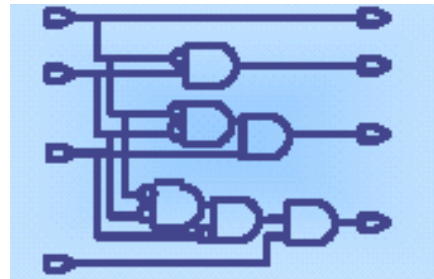
```
residue = 16'h0000;  
if ( high_bits == 2'b10)  
    residue = state_table[index];  
else state_table[index] =16'h0000;
```

**HDL Source  
(RTL)**

no timing info.

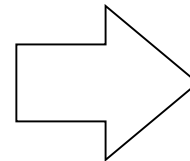


**Translate (HDL Compiler)**

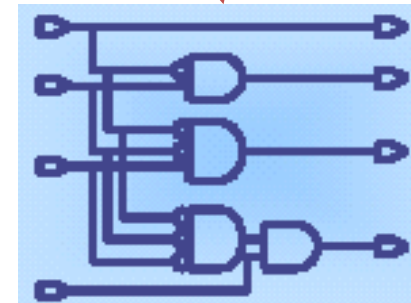


**Generic Boolean  
(GTECH)**

timing info.



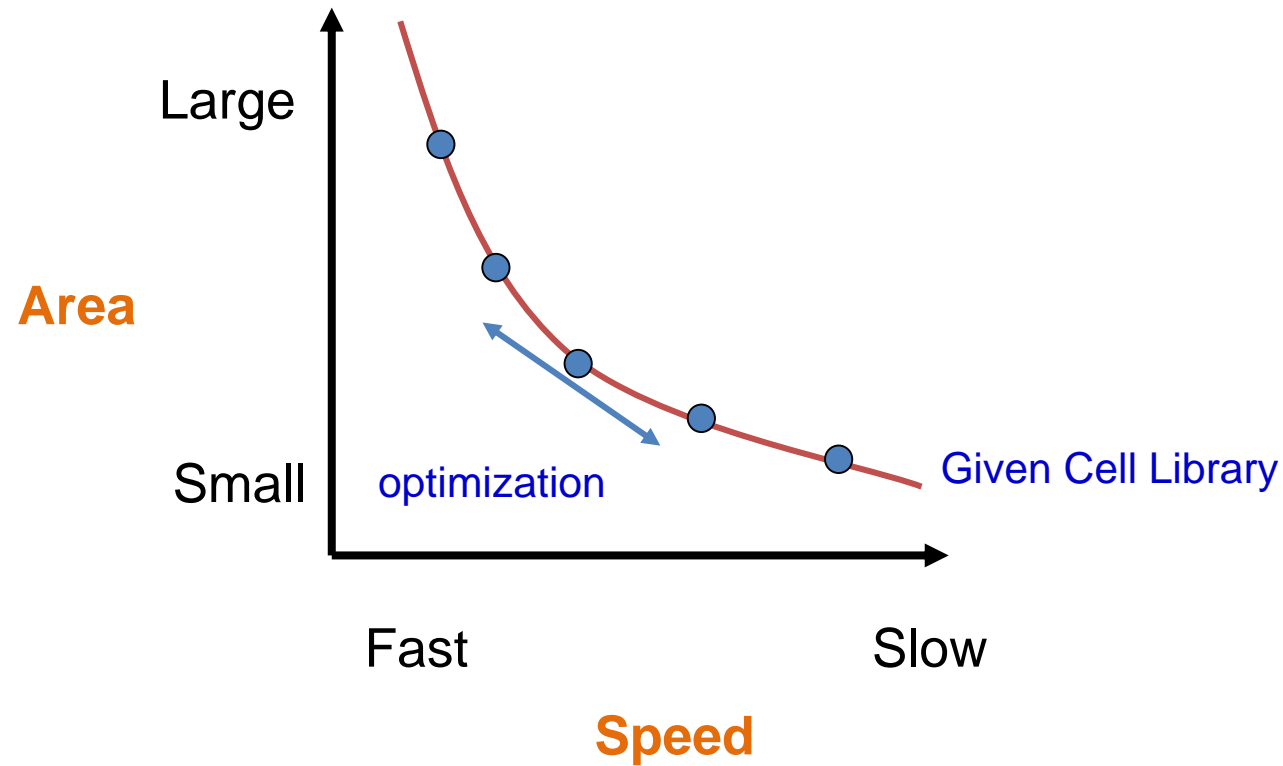
**Optimize + Map  
(Design Compiler)**



**Target Technology**

# Trade-off between Timing & Area

- Given the same library and the same source code, you can only
  - Sacrifice area for higher speed
  - Sacrifice speed for lower area



# Translating Verilog to Logic Gates

---

- Parts of the language easy to translate
  - Structural descriptions with primitive gates
    - Already a netlist
  - Continuous assignment
    - Expressions turn into little datapaths
- Behavioral statements
  - Can consist of synthesizable coding

# Outline

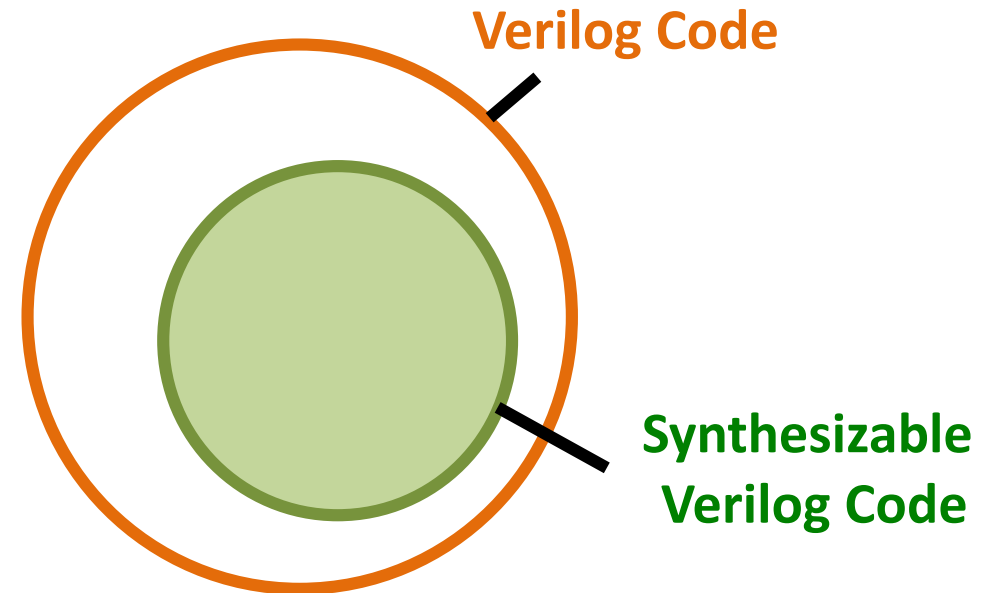
---

- Brief Introduction to Logic Synthesis
- **Syntax for Synthesis**
- Partition for Synthesis
- Circuit-Level Coding Skills
  - Translation between circuits and codes
  - Circuit Refining
- Check for Synthesizable

# Synthesizable Verilog Codes

---

- Verilog HDL is not only for synthesizable designs
- Not all kinds of Verilog constructs can be synthesized
- Only a subset of Verilog constructs can be synthesized and **codes containing only this subset is synthesizable**





# Not Supported Syntax

---

- **delay**
- **initial**
- repeat
- wait
- fork ... join
- event
- deassign
- force
- release
- primitive -- User defined primitive
- time
- triand, trior, tri1, tri0, trireg
- nmos, pmos, cmos, rnmos, rpmos, rcmos
- pullup, pulldown
- rtran, tranif0, tranif1, rtranif0, rtranif1
- **case identity (===)** and **not identity (!==)** operators

# Supported Verilog Basis

---

- Verilog basis
  - Parameter declarations
  - Wire, wand, wor declarations
  - Reg declarations
  - Input, output, inout declarations
  - Continuous assignments
  - Module instantiations
  - Gate instantiations
  - Always blocks
  - ~~– Task statements (partially synthesizable)~~
  - Function definitions (partially synthesizable)
  - For loop (partially synthesizable)

# Supported Verilog Primitives

---

- Synthesizable Verilog primitive cells
  - And, or, not, nand, nor, xor, xnor
  - Bufif0, bufif1, notif0, notif1

# Supported Verilog Operators

---

- Binary bit-wise ( $\sim$ ,  $\&$ ,  $|$ ,  $\wedge$ ,  $\sim\wedge$ )
- Unary reduction ( $\&$ ,  $\sim\&$ ,  $|$ ,  $\sim|$ ,  $\wedge$ ,  $\sim\wedge$ )
- Logical ( $!$ ,  $\&\&$ ,  $||$ )
- 2's complement arithmetic ( $+$ ,  $-$ ,  $*$ )
- Relational ( $>$ ,  $<$ ,  $>=$ ,  $<=$ )
- Equality ( $==$ ,  $!=$ )
- Logical shift ( $>>$ ,  $<<$ )
- Conditional ( $?:$ )

# Comparisons to X or Z

---

- A comparison to an X or Z is always evaluated to false.
  - May cause simulation vs. synthesis mismatch

```
module compare_x(A,B);  
input A;  
output B;  
reg B;  
always begin  
if (A== 1'bx)  
    B=0;  
else  
    B=1;  
end  
endmodule
```

**Warning:** Comparisons to a “don’t care” are treated as always being false in routine compare\_x line 7 in file “compare\_x.v” this may cause simulation to disagree with synthesis. (HDL-170)

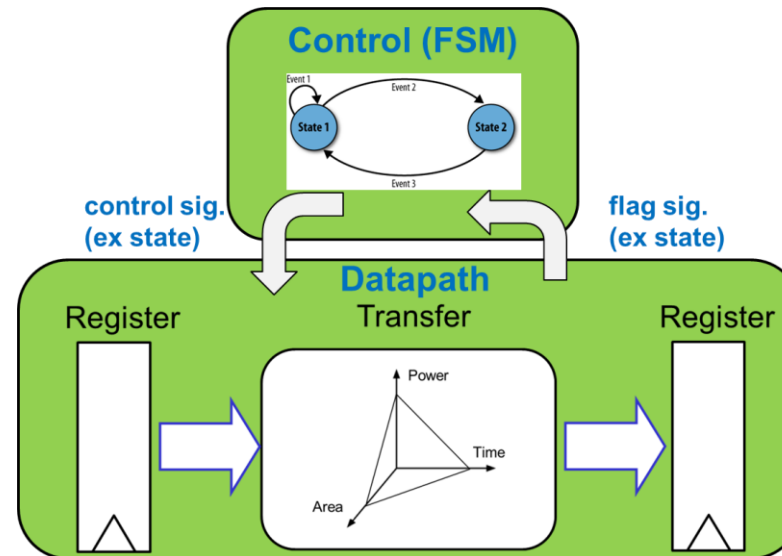
# Outline

---

- Brief Introduction to Logic Synthesis
- Syntax for Synthesis
- **Partition for Synthesis**
- Circuit-Level Coding Skills
  - Translation between circuits and codes
  - Circuit Refining
- Check for Synthesizable

# RTL Coding Cautions: Partitioning for Synthesis

- Separate combinational and sequential part
  - Logic Propagation / Flip-Flops
- Separate control and VLSI-design strategy
- Keep major blocks separate
- Register at hierarchical output, keep related combinational logic together at the same module
- Avoid asynchronous logic, false path, and multi-cycle path
- Avoid the glue logic



# Partition for Synthesis

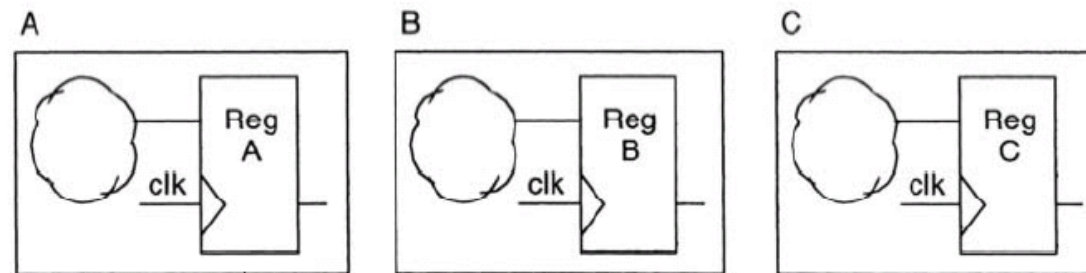
---

- Separate the design into two parts
  - Pure combinational: Logic Propagation
  - Pure Sequential: Flip-Flops
- Avoid misunderstanding by synthesis tools
- Easily tracing of next/current state values after synthesis
- (Refer to last week's slides for details)



# Register All Outputs

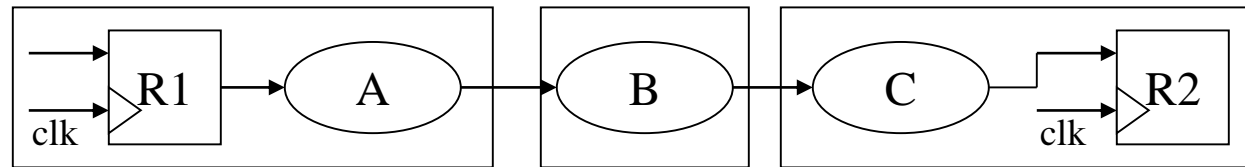
- For each subblock of a hierarchical macro design, **register all output signals** from the subblock.
  - All the inputs of each block arrive with the same relative delay
  - Output **drive strength** is equal to the drive strength of the average flip-flop



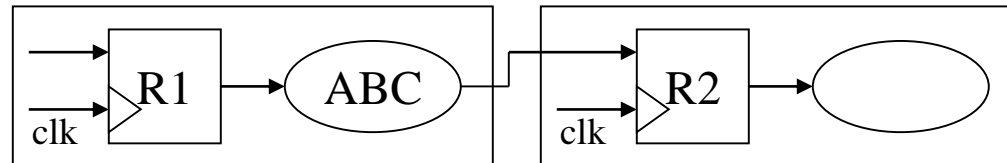
**Figure** Good example: All output signals are registered

# Locate Related Combinational Logic in a Single Module

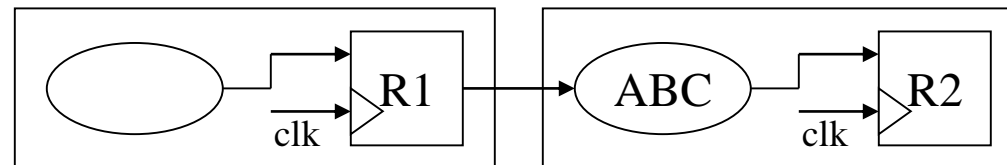
- Keep related combinational logic together in the same module
  - Synthesis tools **cannot optimize logic across hierarchical boundaries**



**Bad**



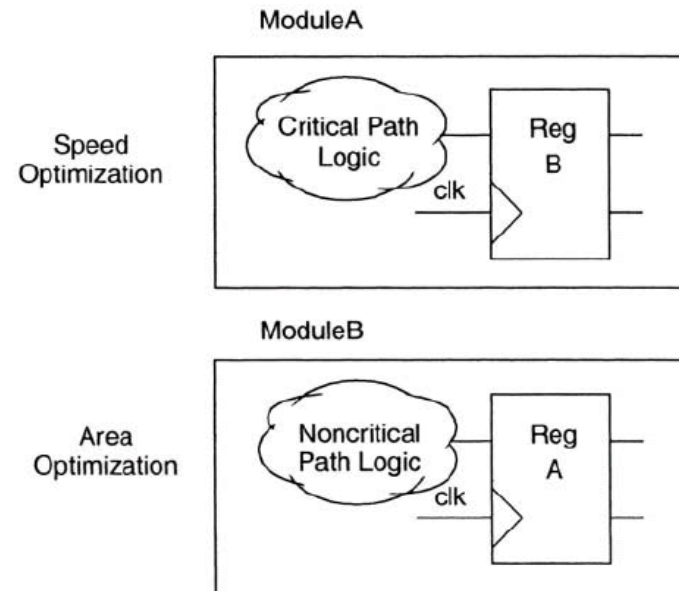
**Better**



**Best**

# Separate Modules with Different Design Goals

- Synthesis tools can perform **speed optimization** on the critical path logic, while performing **area optimization** on the noncritical path logic.



**Figure** Good example: Critical path logic and noncritical path logic grouped separately

# Outline

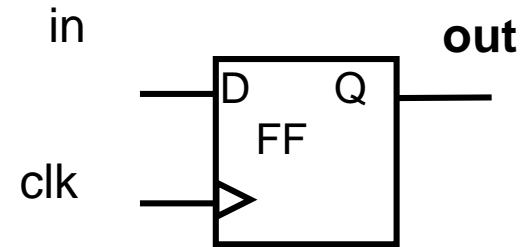
---

- Brief Introduction to Logic Synthesis
- Syntax for Synthesis
- Partition for Synthesis
- Circuit-Level Coding Skills
  - Translation between circuits and codes
  - Circuit Refining
- Check for Synthesizable

# Mapping of Sequential Circuits

- Pure sequential circuits can be mapped as flop-flops
- The name of a flip-flop is its output port

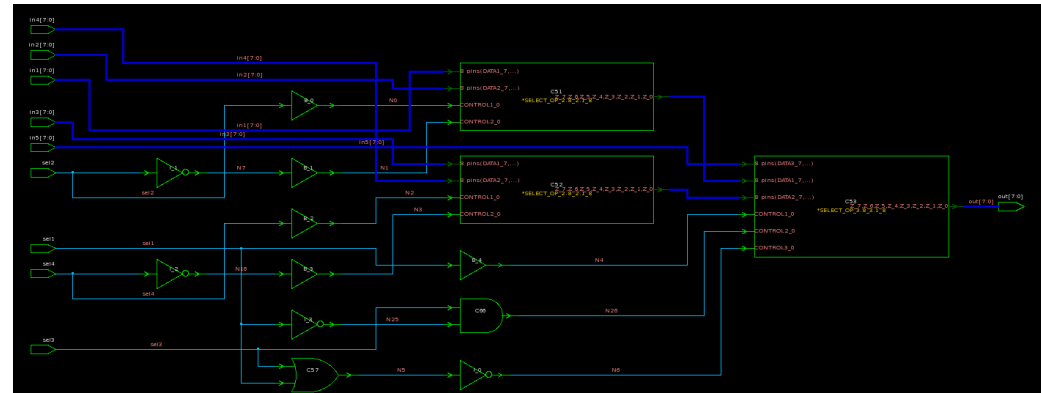
```
reg out;  
wire in, clk;  
always @(posedge clk)  
    out <= in;  
end
```



# Mapping of *if* Statement (1/2)

- Mapped to a Multiplexer
- *if* statement can be nested

```
always @(sel1 or sel2 or sel3 or  
sel4 or in1 or in2 or in3 or in4  
or in5)  
begin  
    if (sel1) begin  
        if (sel2) out=in1;  
        else out=in2;  
    end  
    else if (sel3) begin  
        if (sel4) out=in3;  
        else out=in4;  
    end  
    else out=in5;  
end
```



# Mapping of *if* Statement (2/2)

- What's the difference between these two coding styles?

```
module mult_if(a, b, c, d, e, sel, z);
input a, b, c, d, e;
input [3:0] sel;
output z;
reg z;
always @(a or b or c or d or e or sel)
begin
z = e;
if (sel[0]) z = a;
if (sel[1]) z = b;
if (sel[2]) z = c;
if (sel[3]) z = d;
end
endmodule
```

if sel==4'b1001  
z = d;

後寫的 if 優先

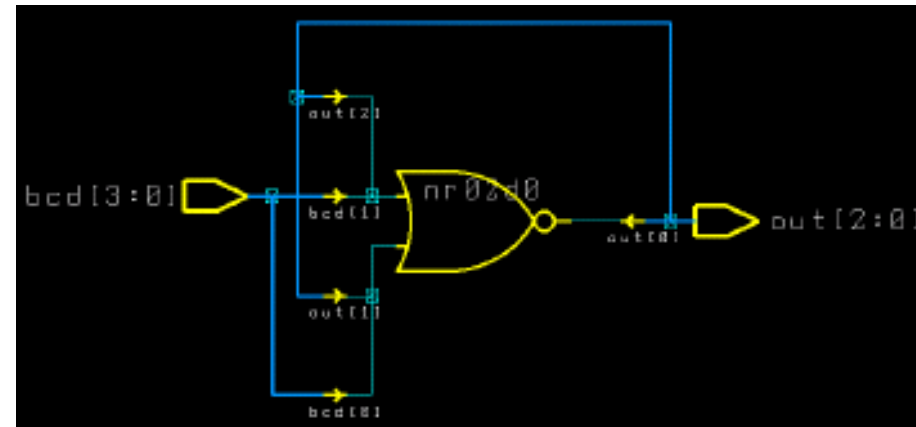
```
module single_if(a, b, c, d, e, sel, z);
input a, b, c, d, e;
input [3:0] sel;
output z;
reg z;
always @(a or b or c or d or e or sel)
begin
z = e;
if (sel[3])
z = d;
else if (sel[2])
z = c;
else if (sel[1])
z = b;
else if (sel[0])
z = a;
end
endmodule
```

if sel==4'b1001  
z = d;

# Mapping of *case* Statement (1/7)

- A case statement is called a **full case** if all possible branches are specified
- Also mapped to a **Multiplexer**

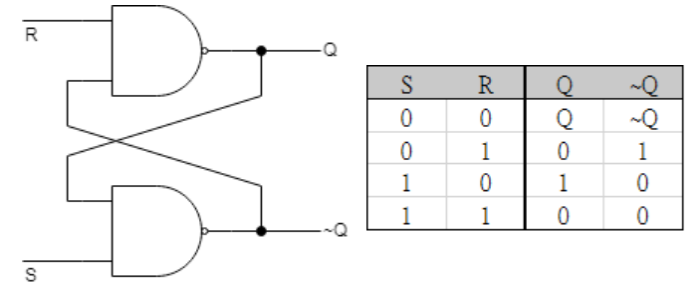
```
always @(bcd) begin
  case (bcd)
    4'd0:out=3'b001;
    4'd1:out=3'b010;
    4'd2:out=3'b100;
    default:out=3'bxxx;
  endcase
end
```





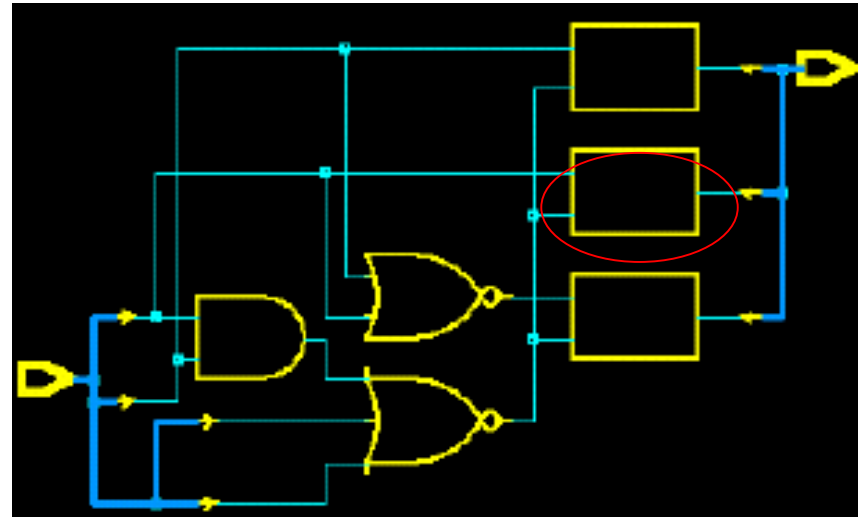
# Mapping of *case* Statement (2/7)

- If a case statement is not a full case, it will infer a **latch**
- **Latch may arise timing violation easily!**
  - So try to avoid this situation!



Latches

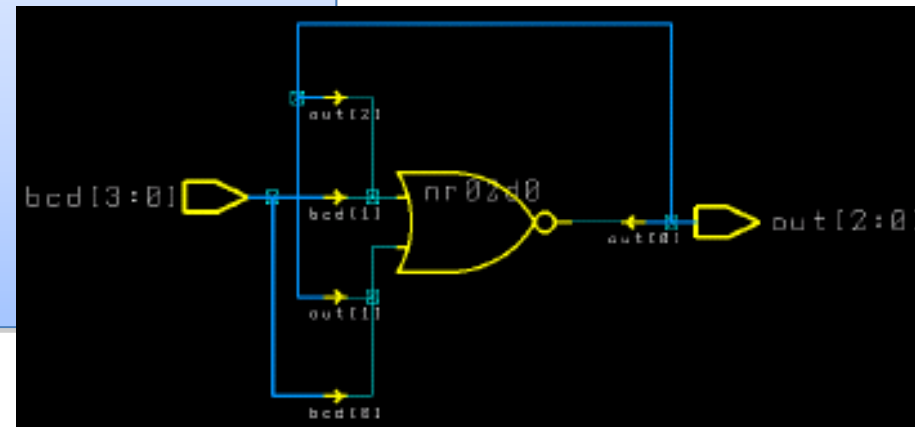
```
always @(bcd) begin
  case (bcd)
    4'd0: out = 3'b001;
    4'd1: out = 3'b010;
    4'd2: out = 3'b100;
  endcase
end
```



# Mapping of *case* Statement (3/7)

- If you do not specify all possible branches, but you know the other branches will never occur, you can use “`//synopsys full_case`” directive to specify full case when synthesizing

```
always @(bcd) begin
  case (bcd) //synopsys full_case
    4'd0:out=3'b001;
    4'd1:out=3'b010;
    4'd2:out=3'b100;
  endcase
end
```



# Mapping of *case* Statement (4/7)

---

- **Note:** the second case item does not modify reg2, causing it to be inferred as a latch (to retain last value).

```
case (cntr_sig) // synopsys full_case
2'b00 : begin
        reg1 = 0 ;
        reg2 = v_field ;
    end
2'b01 : reg1 = v_field ; /* latch will be inferred for reg2*/

2'b10 : begin
        reg1 = v_field ;
        reg2 = 0 ;
    end
endcase
```

# Mapping of *case* Statement (5/7)

---

- Two possible ways we can assign a default value to a variable to avoid latch
  - Second way is more common!

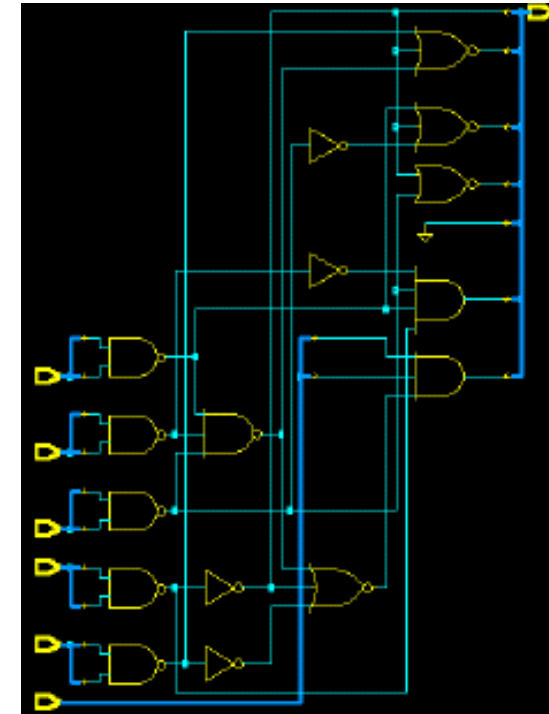
```
(1)  out = 3'b000 ; // this is called unconditional assignment
      case (condition)
      ...
      endcase
```

```
(2)  case (condition)
      ...
      default : out = 3'b000 ; // out=0 for all other cases
      endcase
```

# Mapping of *case* Statement (6/7)

- If HDL Compiler can't determine that case branches are parallel, its synthesized hardware will include a **priority decoder**.

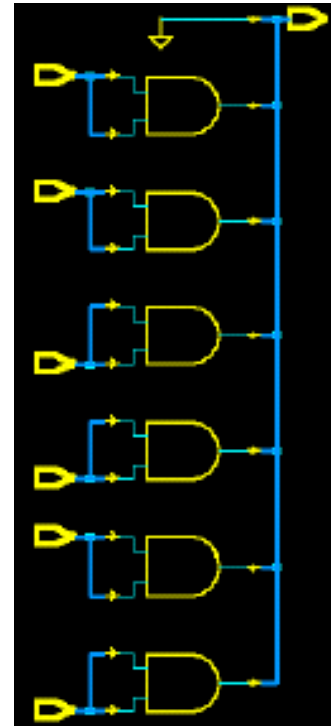
```
always @(u or v or w or x or y or z)
begin
  case (2'b11)
    u:out=10'b0000000001;
    v:out=10'b0000000010;
    w:out=10'b0000000100;
    x:out=10'b0000001000;
    y:out=10'b0000010000;
    z:out=10'b0000100000;
    default:out=10'b0000000000;
  endcase
end
```



# Mapping of *case* Statement (7/7)

- You can declare a case statement as parallel case with the “`//synopsys parallel_case`” directive

```
always @(u or v or w or x or y or z)
begin
  case (2'b11) //synopsys parallel_case
    u:out=10'b0000000001;
    v:out=10'b0000000010;
    w:out=10'b0000000100;
    x:out=10'b0000001000;
    y:out=10'b0000010000;
    z:out=10'b0000100000;
    default:out=10'b0000000000;
  endcase
end
```



# Mapping of *for* Loop

- Provide a shorter way to express a series of statements.
- Loop index variables must be **integer** type
- Step, start & end value must be **constant**
- For synthesis tools, for loops are “**unrolled**”, and then synthesized.

```
always@( a or b )  
begin  
    for( i=0; i<4; i=i+1 )  
        c[i] = a[i] & b[i];  
end
```



```
always@( a or b )  
begin  
    c[0] = a[0] & b[0];  
    c[1] = a[1] & b[1];  
    c[2] = a[2] & b[2];  
    c[3] = a[3] & b[3];  
end
```

# Mapping of Logical Operators

---

- Binary Logical Operators (&, |, ^, ~^)
  - Mapped to **logic gates** directly
- Unary Logical Operators (&, |, ^, ~^, ~, !)
  - Each bit mapped to a **logic gate**
- Comparison Operators (>, <, >=, <=)
  - Mapped to **full adders for subtraction**
  - Comparison result = MSB of subtraction output
- Equality Operators (==, !=)
  - Mapped to **full adders for subtraction**
  - Or/And each bit of subtraction output for result



# Mapping of Arithmetic Operators (1/2)

---

- Addition
  - Full adder
- Subtraction
  - Full adder with 2's complement inverter
- Multiplication
  - Full adder array
- Division & Modulo
  - May need to instantiate [DesignWare's modules](#)
  - No direct mapping to any simple elements

# Mapping of Arithmetic Operators (2/2)

---

- Multiplication & Division of Radix-2
  - Simplified as shift operations
  - Left shift by 1 bit: Multiply by 2
  - Right shift by 1 bit: Divide by 2
- Shift operations (<<, >>)
  - Shift by constant: Simply wire assignment

```
// c is the same as b  
assign b = a[7:0] >> 2;  
assign c = {2'b0, a[7:2]};
```

- Shift by variable: **Shifter** (can be derived from truth table)

# Signed Signal (1/4)

- “Regard as” basis of signed signal
  - MSB is regarded as sign bit and nothing changes, that’s all!
  - NO automatic sign extension will be done by compiler

- Usage of signed signal

- Addition/Subtraction

- equivalent to unsigned signals

- Multiplication/Division

- all inputs should be defined as signed signals

- Comparison

```
wire [7:0] a;  
wire [15:0] ax; // a sign-extended to 16-bit  
assign ax = {{8{a[7]}}, a};
```

```
wire [7:0] a;  
wire [7:0] b;  
wire less;  
assign less = ($signed(a) < $signed(b));
```

```
wire signed [7:0] a;  
wire signed [7:0] b;  
wire less;  
assign less = (a < b);
```

# Signed Signal (2/4)

- Signed addition bit length
  - $A(8 \text{ bits}) + B(8 \text{ bits}) \rightarrow C(8+1 \text{ bits})$

```
// Verilog 1995
wire [7:0] A, B;
wire [8:0] C;
assign C = {A[7], A} + {B[7], B};
```

```
// Verilog 2001
wire signed [7:0] A, B;
wire signed [8:0] C;
assign C = A + B;
```

- Signed multiplication bit length
  - $A(3 \text{ bits}) \times B(5 \text{ bits}) \rightarrow C(((3-1)+(5-1)+1)+1 \text{ bits})$
  - $A(-4 \sim 3) \times B(-16 \sim 15) \rightarrow C(-60 \sim \underline{64})$

```
wire signed [2:0] A;
wire signed [4:0] B;
wire signed [7:0] C;
assign C = A * B;
```

# Signed Signal (3/4)

- Signed comparison

- If  $-3 < A(4 \text{ bits}) < 4$ , raise flag, otherwise not

- Signed representation from 4'b0000 to 4'b1111

|   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |
|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|

- Unsigned representation from 4'b0000 to 4'b1111

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|

- Correct Verilog behavior modeling

```
wire signed [3:0] A;  
wire          flag;  
assign flag = (A<4'd4 || A>4'd13)? 1'b1: 1'b0;
```

# Signed Signal (4/4)

---

If any section of a comparison is unsigned then the comparison is unsigned. **Selecting bit widths, even if the whole range, is unsigned**

```
reg signed [8:0] sin_hall2;

initial begin
    sin_hall2 = -9'd169 ;
    $display( "Comparison unsigned : %b ", sin_hall2 > 9'd1 );
    $display( "Comparison cast      : %b ", sin_hall2 > $signed(9'd1) );
    $display( "Comparison signed   : %b ", sin_hall2 > 9'sd1 );
    $display( "Comparison signed [8:0]: %b ", sin_hall2[8:0] > 9'sd1 );
end
```

Returns:

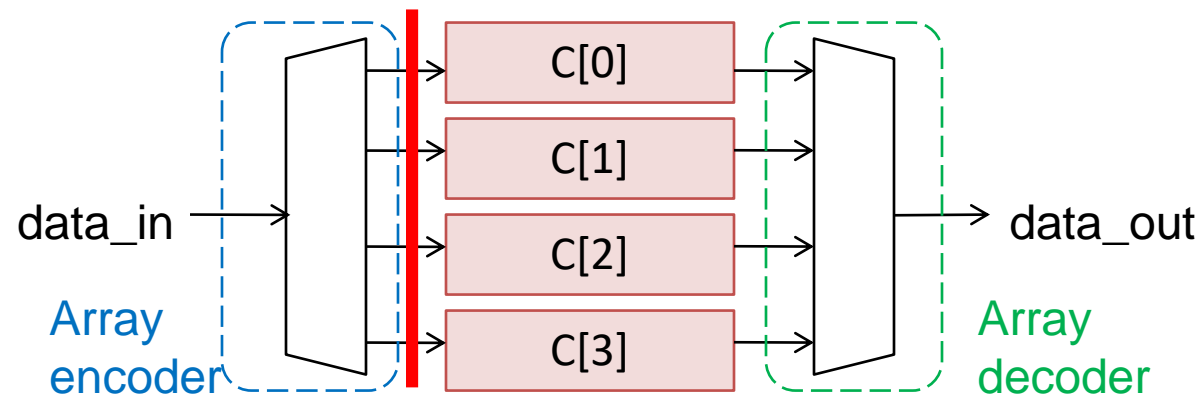
```
# Comparison unsigned : 1
# Comparison cast      : 0
# Comparison signed   : 0
# Comparison signed [8:0]: 1
```

# Vector Array (1/2)

- Vector array
  - Declaration and usage of vector array (4 vectors of 8 bits)

```
reg[7:0] C[0:3];  
assign data_out = C[index_o];  
always@(posedge clock) begin  
    C[index_i] <= data_in;  
end
```

- Hardware translation



# Vector Array (2/2)

- Encoder/decoder issue
  - If the array size is large or the array is accessed by multiple signals, encoder & decoder may be very large!
  - When input is coming **in order**, the **encoder can be reduced by using shift registers**



- Decoder MUX can be customized if some are not used
- Debugging issue

```
reg[7:0] C[0:3];  
wire [7:0] dbg_C0 = C[0]; // for vcd waveform debugging
```

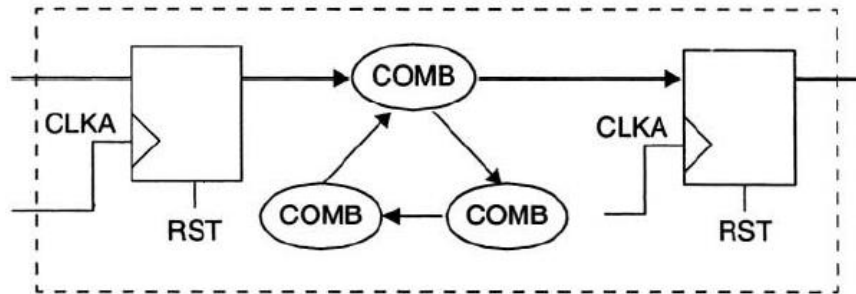
```
$fsdbDumpfile("filename");  
$fsdbDumpvars(0, test_module_name, "+mda");
```



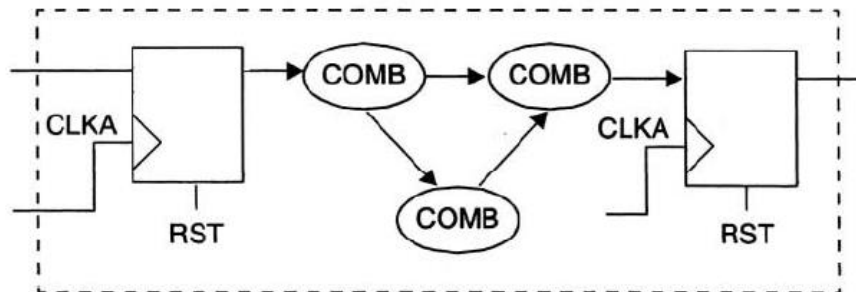
# Combinational Loop

- An output of a combinational block feeds back to an input of the same block
- Should be avoided!

**Bad: Combinational processes are looped**



**Good: Combinational processes are not looped**



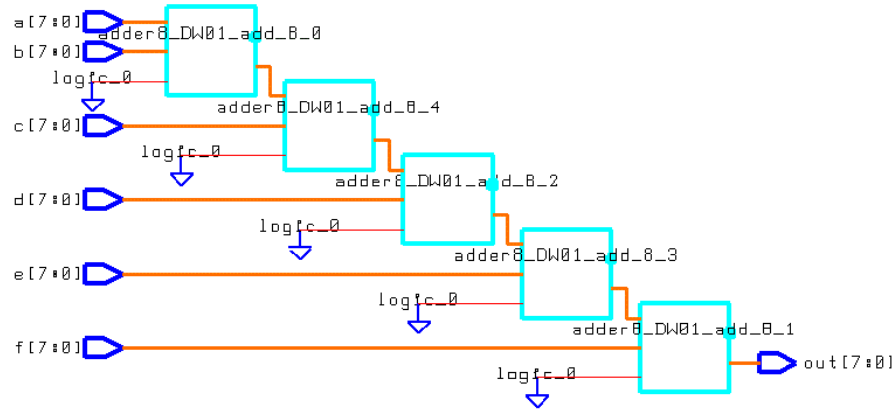
# Circuit-Level Refinement

---

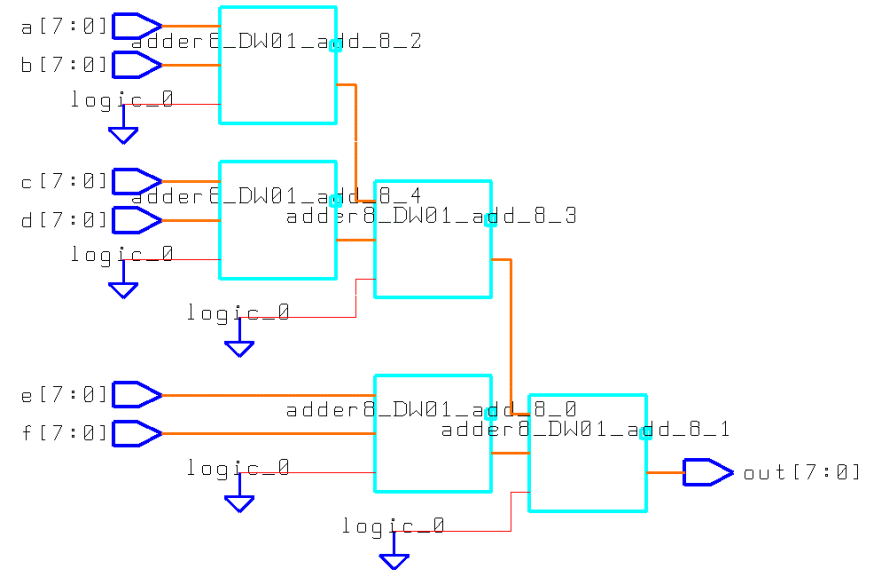
- Be aware of the translation between circuits and codes
  - Operators means computation units
  - Datapath controllers mean FSM or multiplexers
- Plan a design using the block diagram instead of a pseudo code of data flow
  - Easy to understand your design cost (area/timing/critical path)

# Use Parentheses Properly

- Out=a+b+c+d+e+f;



- Out=(a+b)+(c+d)+(e+f);



# Propagate Constant Value

---

```
parameter size = 8;  
wire  [3:0] a,b,c,d,e;  
assign c = size + 2;    // constant  
assign d = a + 1;       // incrementer  
assign e = a + b;       // adder
```

# Data-Path Duplication (1/2)

**No\_duplicated**

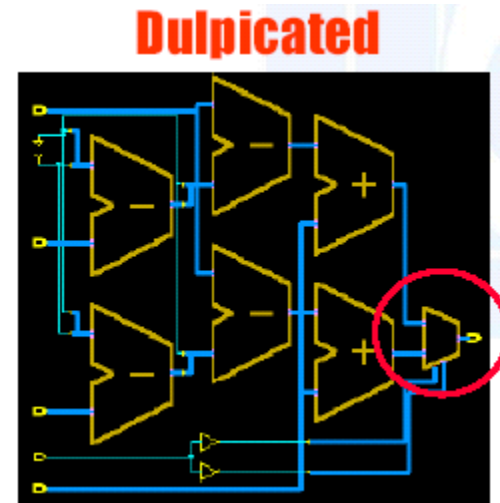
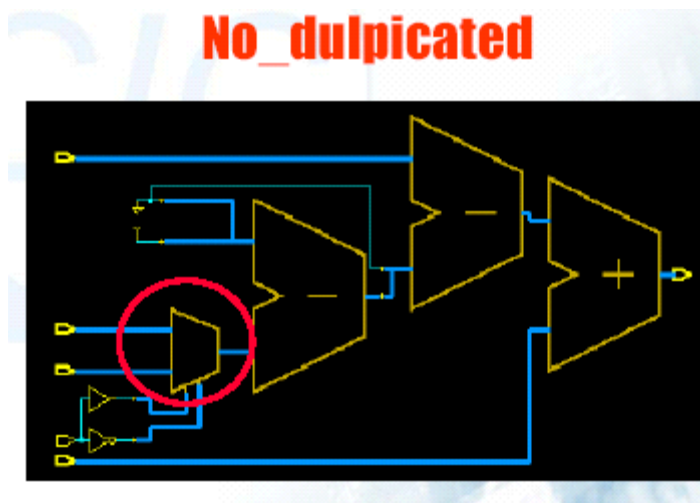
```
module BEFORE (ADDRESS, PTR1, PTR2, B, CONTROL, COUNT);
input [7:0] PTR1, PTR2;
input [15:0] ADDRESS, B;
input CONTROL;          // CONTROL is late arriving
output [15:0] COUNT;
parameter [7:0] BASE = 8'b10000000;
wire [7:0] PTR, OFFSET;
wire [15:0] ADDR;
assign PTR = (CONTROL == 1'b1) ? PTR1 : PTR2;
assign OFFSET = BASE - PTR; //Could be any function f(BASE, PTR)
assign ADDR = ADDRESS - {8'h00, OFFSET};
assign COUNT = ADDR + B;
endmodule
```

**Duplicated**

```
module PRECOMPUTED (ADDRESS, PTR1, PTR2, B, CONTROL, COUNT);
input [7:0] PTR1, PTR2;
input [15:0] ADDRESS, B;
input CONTROL;
output [15:0] COUNT;
parameter [7:0] BASE = 8'b10000000;
wire [7:0] OFFSET1, OFFSET2;
wire [15:0] ADDR1, ADDR2, COUNT1, COUNT2;
assign OFFSET1 = BASE - PTR1; // Could be f(BASE, PTR)
assign OFFSET2 = BASE - PTR2; // Could be f(BASE, PTR)
assign ADDR1 = ADDRESS - {8'h00, OFFSET1};
assign ADDR2 = ADDRESS - {8'h00, OFFSET2};
assign COUNT1 = ADDR1 + B;
assign COUNT2 = ADDR2 + B;
assign COUNT = (CONTROL == 1'b1) ? COUNT1 : COUNT2;
endmodule
```

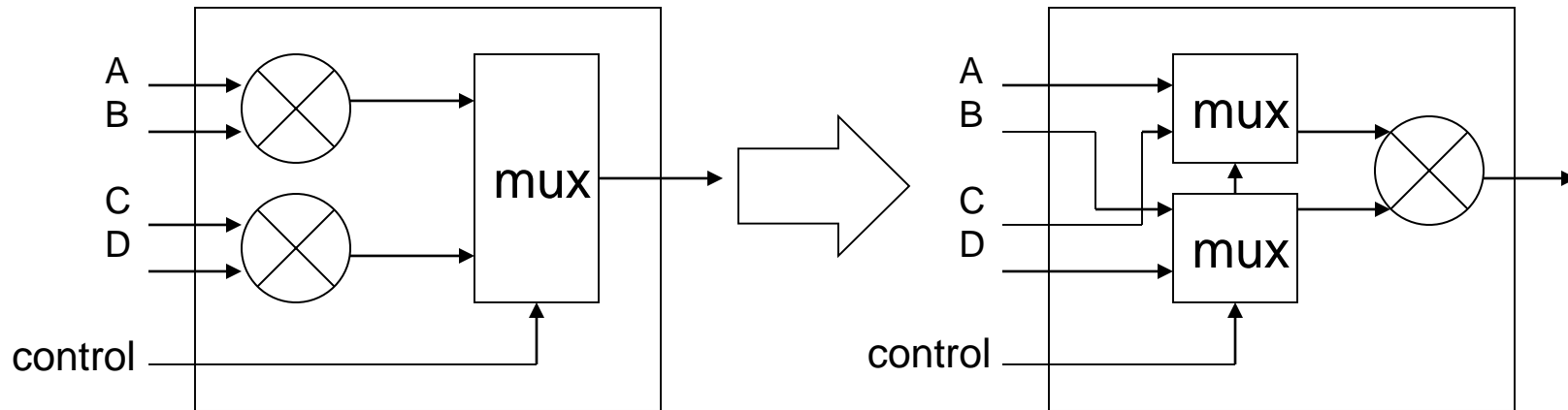
# Data-Path Duplication (2/2)

- We assume that signal “CONTROL” is the latest arrival pin.
- Sacrifice area to gain latency reduction



# Resource Reusing

- Keep sharable resources in the same block



```
always@(*) begin
    if(control) z = a*b;
    else      z = c*d;
end
```

```
always@(*) begin
    z = ((control)? a:c)
        * ((control)? b:d);
end
```

# Comparison Refinement (1/2)

- We assume that signal “A” is latest arrival signal

## Before\_improved

```
module cond_oper(A, B, C, D, Z);
parameter N = 8;
input [N-1:0] A, B, C, D;
//A is late arriving
output [N-1:0] Z;
reg [N-1:0] Z;

always @(A or B or C or D) begin
if (A + B < 24)
    Z <= C;
else
    Z <= D;
end
endmodule
```

## Improved

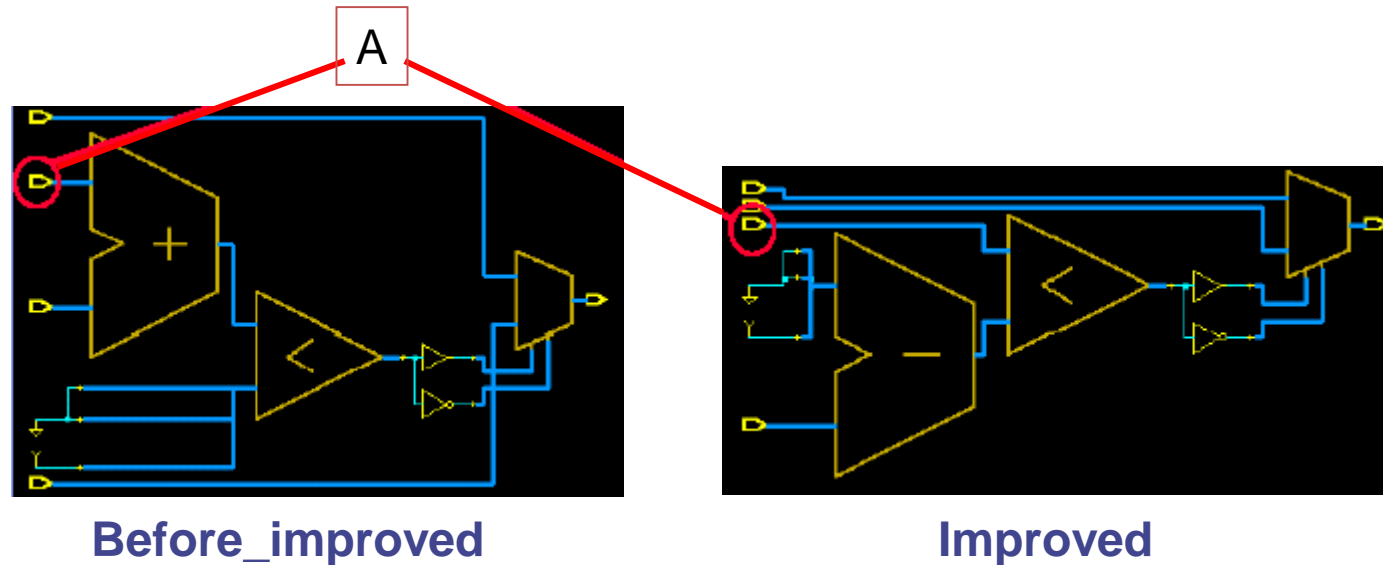
```
module cond_oper_improved (A, B, C, D, Z);
parameter N = 8;
input [N-1:0] A, B, C, D;
// A is late arriving
output [N-1:0] Z;
reg [N-1:0] Z;

always @(A or B or C or D) begin
if (A < 24 - B)
    Z <= C;
else
    Z <= D;
end
endmodule
```



## Comparison Refinement (2/2)

- In this example, not only latency reduced, but also area reduced.



# Outline

---

- Brief Introduction to Logic Synthesis
- Syntax for Synthesis
- Partition for Synthesis
- Circuit-Level Coding Skills
  - Translation between circuits and codes
  - Circuit Refining
- Check for Synthesizability

# Check for Synthesizability (1/2)

---

- *SpringSoft nLint*
  - Check for correct mapping of your design
  - Not so powerful in detecting latches
- *Synopsys Design Compiler*
  - Synthesis Tool
  - The embedded *Presto Compiler* can list your flip-flops and latches in details
    - `> dv -no_gui`
    - `> read_verilog yourdesign.v`

## Check for Synthesizability (2/2)

```
Inferred memory devices in process
in routine cache line 281 in file
'/home/m97/gieks/cache/cache.v'.
```

| Register Name    | Type      | Width | Bus | MB | AR | AS | SR | SS | ST |
|------------------|-----------|-------|-----|----|----|----|----|----|----|
| block6_reg       | Flip-flop | 155   | Y   | N  | Y  | N  | N  | N  | N  |
| block7_reg       | Flip-flop | 155   | Y   | N  | Y  | N  | N  | N  | N  |
| block0_reg       | Flip-flop | 155   | Y   | N  | Y  | N  | N  | N  | N  |
| state_reg        | Flip-flop | 2     | Y   | N  | Y  | N  | N  | N  | N  |
| block1_reg       | Flip-flop | 155   | Y   | N  | Y  | N  | N  | N  | N  |
| mem_fetching_reg | Flip-flop | 1     | N   | N  | Y  | N  | N  | N  | N  |
| block3_reg       | Flip-flop | 155   | Y   | N  | Y  | N  | N  | N  | N  |
| block5_reg       | Flip-flop | 155   | Y   | N  | Y  | N  | N  | N  | N  |
| block2_reg       | Flip-flop | 155   | Y   | N  | Y  | N  | N  | N  | N  |
| block4_reg       | Flip-flop | 155   | Y   | N  | Y  | N  | N  | N  | N  |

```
Presto compilation completed successfully.
```

```
Current design is now '/home/m97/gieks/cache/cache.db:cache'
```

```
Loaded 1 design.
```

```
Current design is 'cache'.
```

```
cache
```

```
design_vision> █
```

**Checking latches using  
Design Compiler**

# Debugging and Testbench Writing

# Outline

---

- Introduction to Debugging Tool: Verdi
  - nLint: HDL Coding Checking
  - nWave: Waveform Tracing
- Testbench Writing
  - Overview of Simulation
  - Instantiating DUT
  - Creating Clocks
  - Applying Stimulus
  - Verification

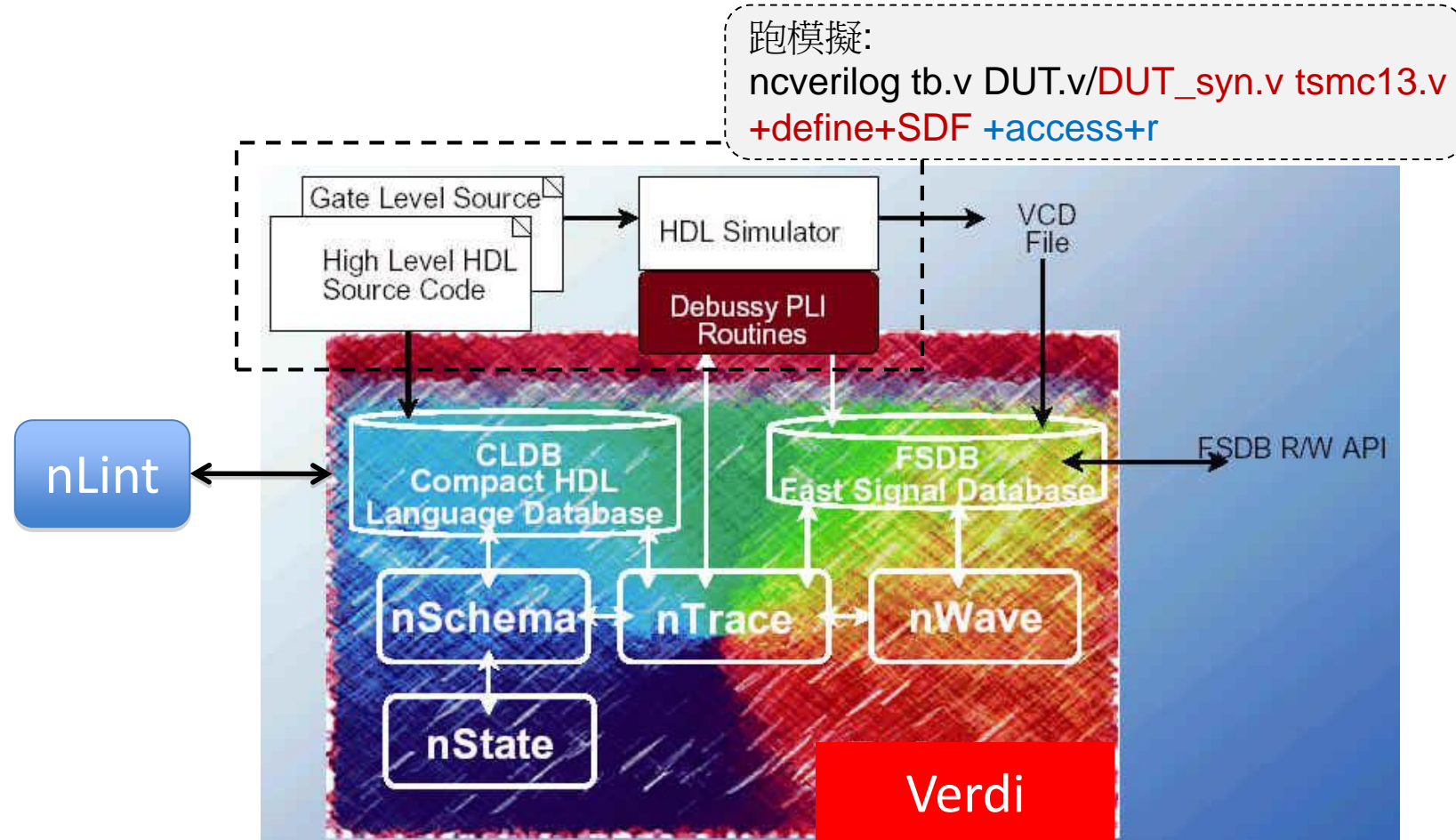
# Introduction to Verdi (1/2)

---

- Automated Debugging Solution by SpringSoft
- Originally developed by Novas
  - SpringSoft's company in the U.S., now bought back by SpringSoft
- Three key features
  - nTrace (main Verdi debugging environment)
  - nLint (RTL design rule checking tool)
  - nWave (waveform analysis tool)

# Introduction to Verdi (2/2)

- Verdi debugging system





# nLint

---

- A **design rule checker** that can help hardware designers to create syntax and semantics correct HDL code.
- nLint reads in HDL source code, analyzes it, and outputs warnings and errors.
  - Including position and message.

# Example: Bad\_conditional.v

```
always@(in1 or select1)begin
    case(select1)
        2'b00: out1 = 1'b0;
        2'b01: out1 = in1;
        2'b10: out1 = ~in1;
    endcase
end

always@(in2)begin
    if(select2)begin
        out2 = in2;
    end else begin
        out2 = ~in2
    end
end
```

Incomplete  
conditional  
assignment ①

Incomplete sensitivity list ②

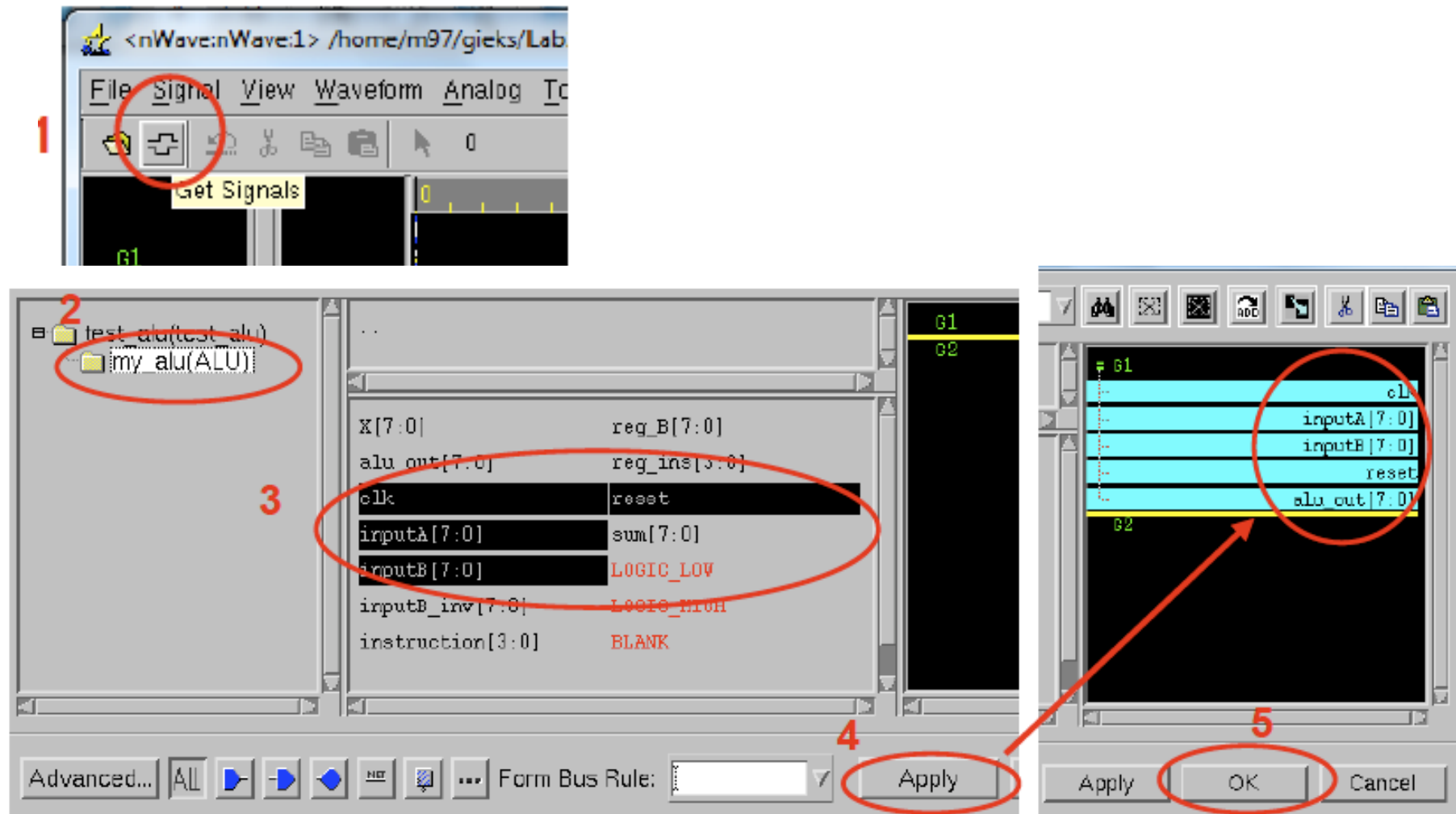
Error!!  
need “;” ③

# nWave

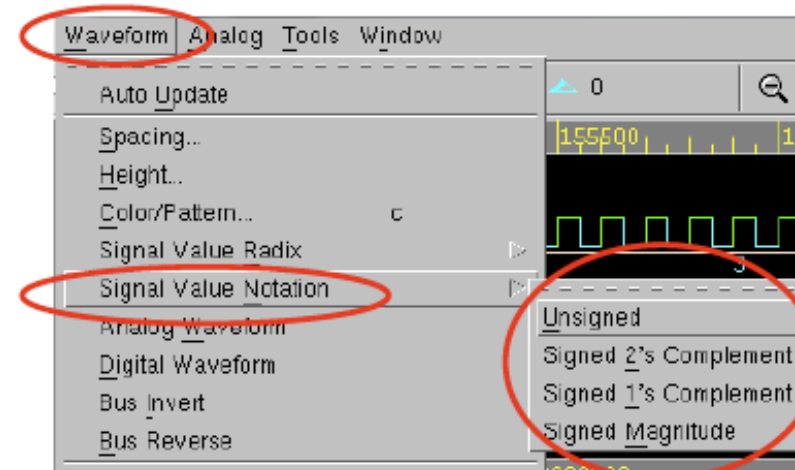
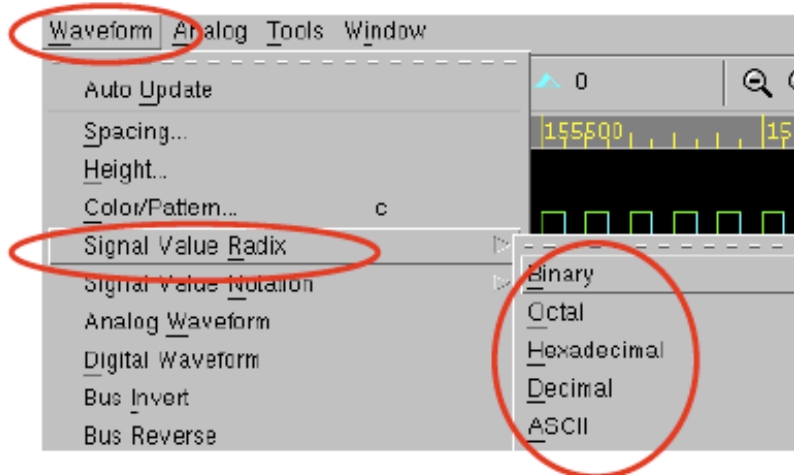
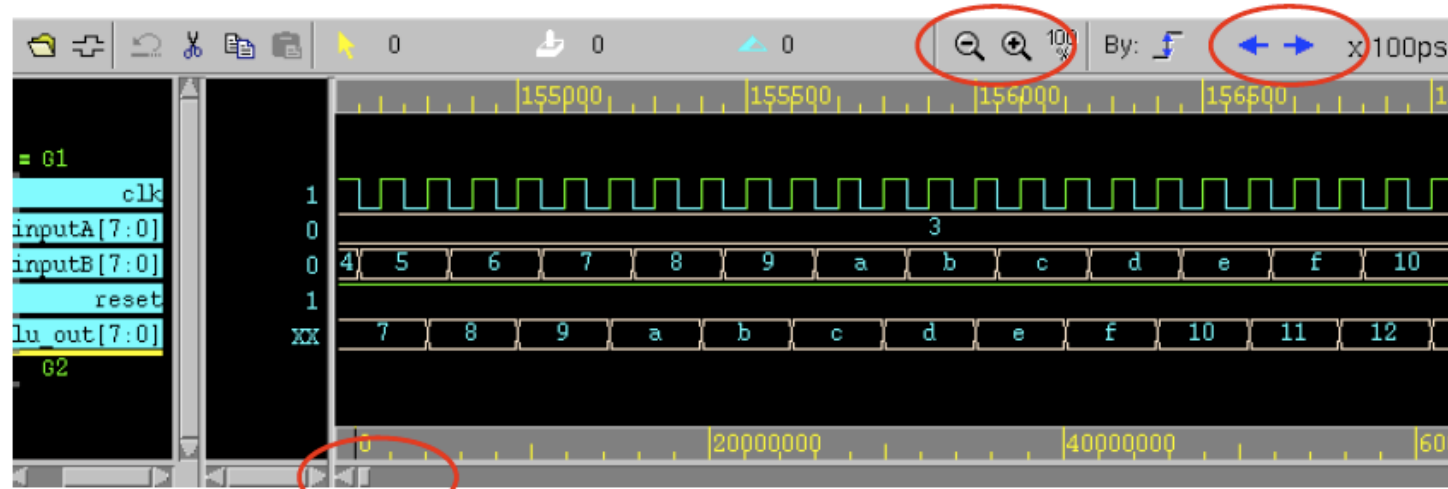
---

- A waveform analysis tool for viewing *\*.fsdb* & *\*.vcd* waveform files
- Invoke nWave:
  - *> nWave &*
- Open waveform file

# Select Signals to View



# Viewing Tools



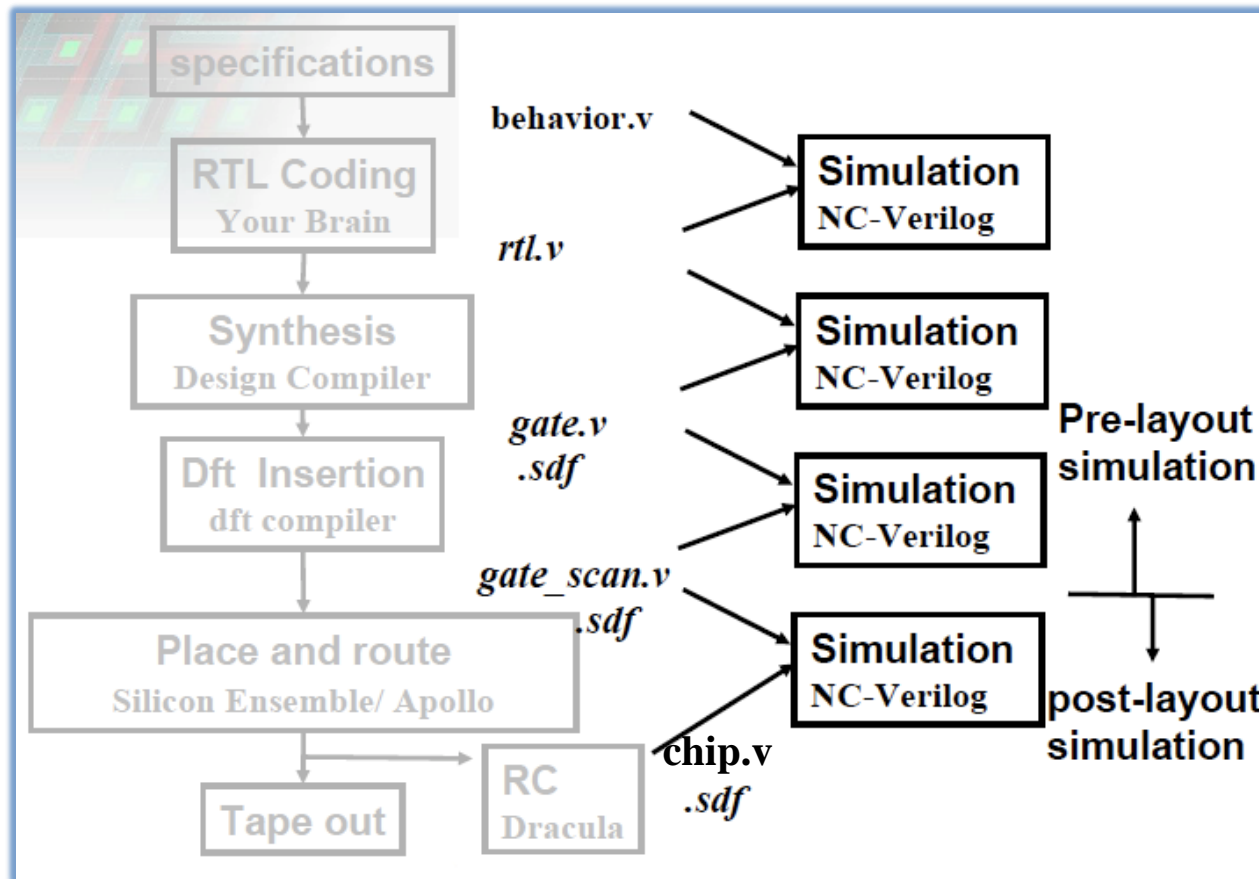
# Outline

---

- Introduction to Debugging Tool: Verdi
  - nLint: HDL Coding Checking
  - nWave: Waveform Tracing
- Testbench Writing
  - Overview of Simulation
  - Instantiating DUT
  - Creating Clocks
  - Applying Stimulus
  - Verification

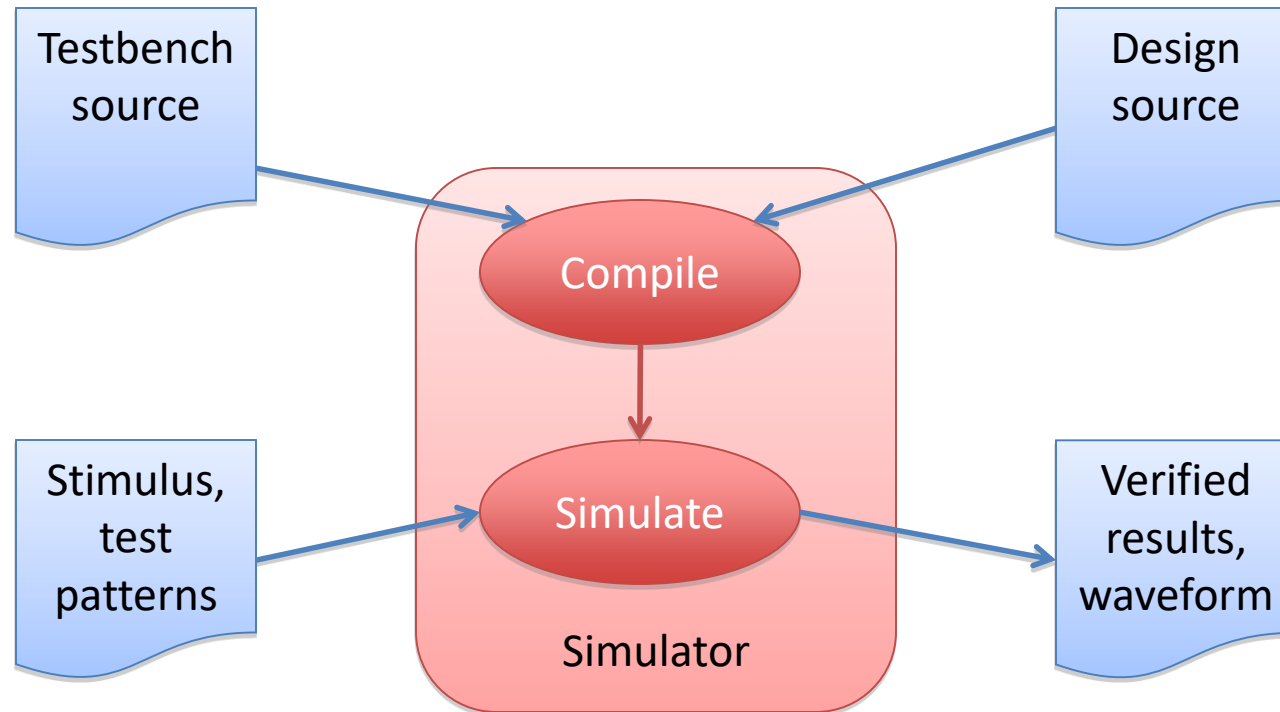
# Overview of Simulation (1/2)

- Verification at every step



# Overview of Simulation (2/2)

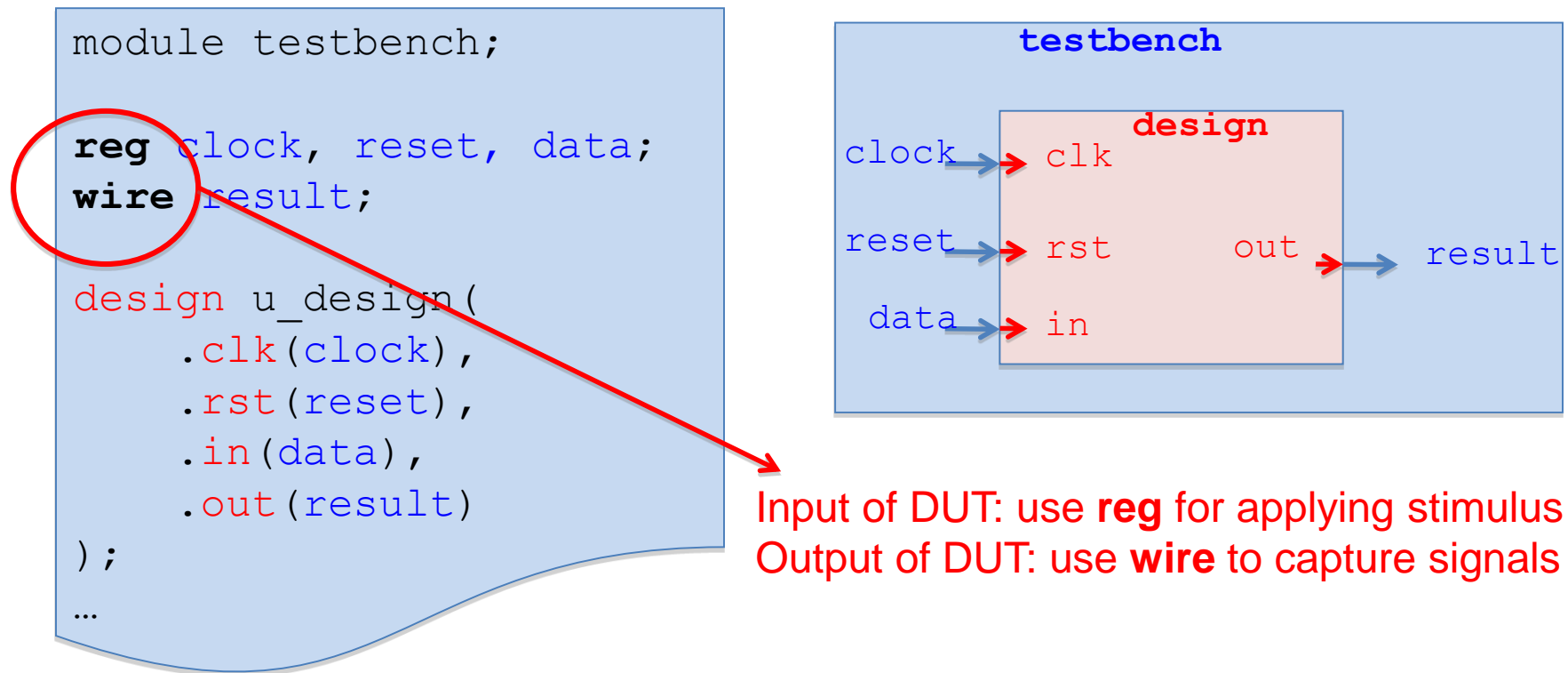
- Simulation Environment





# Instantiating DUT

- Device Under Test (DUT)
  - Top module of the design should be instantiated inside the testbench



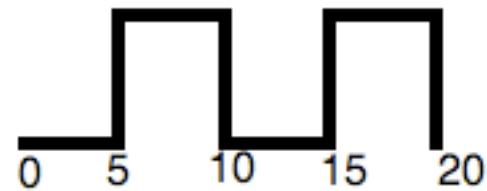
# Creating Clocks (1/2)

- Initializing the clock

```
reg clock;  
initial begin  
    clock = 0;  
end
```

- Modeling the clock behavior

```
(unit/precision)  
`timescale 1ns/10ps  
`define CYCLE 10  
`define H_CYCLE 5  
  
always #(`H_CYCLE) begin  
    clock = ~clock;  
end
```



# Creating Clocks (2/2)

---

- Other syntax

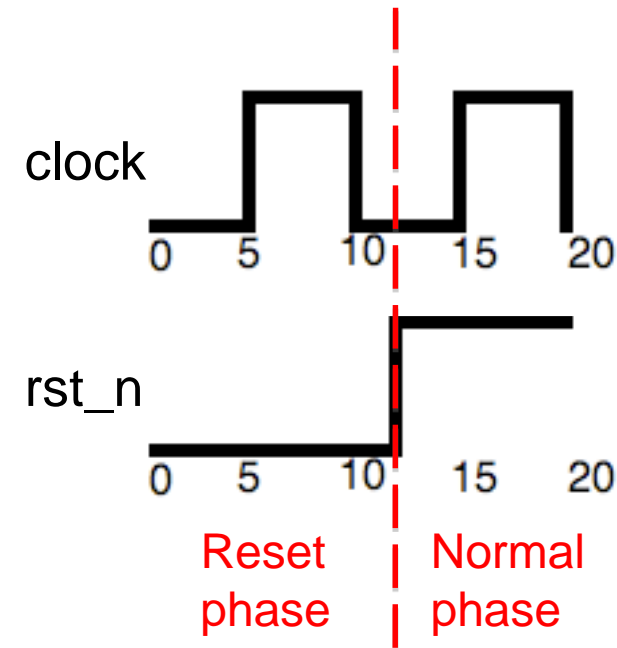
```
`timescale 1ns/10ps
`define CYCLE 10
`define H_CYCLE 5
reg clock;

initial begin
    clock = 0;
    forever begin
        #(`H_CYCLE) clock = 1;
        #(`H_CYCLE) clock = 0;
    end
end
```

# Applying Stimulus (1/6)

- Initialization using *reset* signal

```
`timescale 1ns/10ps
`define CYCLE 10
`define H_CYCLE 5
reg clock, rst_n;
always #(`H_CYCLE) begin
    clock = ~clock;
end
initial begin
    clock = 0;
    rst_n = 0;
    #(`CYCLE*1.2) rst_n = 1;
end
```



# Applying Stimulus (2/6)

- In-Line Style
  - Pros: easily define complex timing relationship between signals
  - Cons: the testbench can be very long for massive test patterns

```
module inline_tb;
  wire [7:0] results;
  reg [7:0] data_bus, addr;
  DUT u1 (results, data_bus, addr);
  initial fork
    #10 addr = 8'h01;
    #10 data_bus = 8'h23;
    #20 data_bus = 8'h45;
    #30 addr = 8'h67;
    #30 data_bus = 8'h89;
    #40 data_bus = 8'hAB;
    #45 $finish;
  join
endmodule
```

# Applying Stimulus (3/6)

- Looping Style
  - Pros: testbench may be compact
  - Cons: only adequate for test patterns with **regular** timing and values

```
module loop_tb;
    wire [7:0] response;
    reg [7:0] stimulus;
    reg clk;
    integer i;
    DUT u1 (response, stimulus);
    initial clk = 0;
    always #10 clk = ~clk;
    initial begin
        for (i = 0; i <= 255; i = i + 1)
            @(negedge clk) stimulus = i;
        #20 $finish;
    end
endmodule
```

# Applying Stimulus (4/6)

---

- Stimulus From File
  - Most popular way with well-considered test patterns

```
module stim_from_file_tb;
    wire [7:0] response;
    reg [7:0] stimulus, stim_array[0:15];
    integer i;
    DUT u1 (response, stimulus);
    initial begin
        $readmemb("datafile", stim_array);
        for (i = 0; i <= 15; i = i + 1)
            #20 stimulus = stim_array[i];
        #20 $finish;
    end
endmodule
```

# Applying Stimulus (5/6)

- File Input
  - Verilog support two methods to load data into a *reg* array
  - Read binary data:
    - `$readmemb("filename", reg_array_name);`
  - Read hexadecimal data
    - `$readmemh("filename", reg_array_name);`
- Data file format

```
/* Data File */

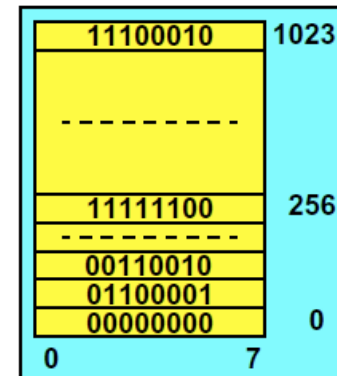
@0 // address always hex
0000_0000
0110_0001 0011_0010

// addresses 3-255 undefined

@100
1111_1100

// addresses 257-1022 undefined

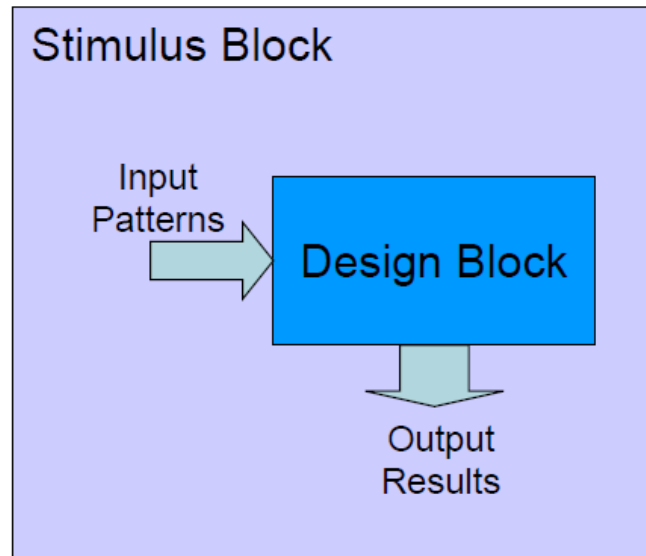
@3FF
1110_0010
```



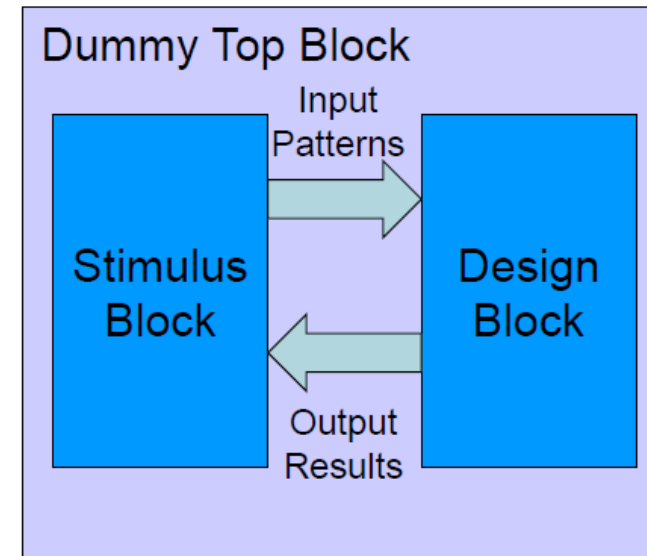


# Applying Stimulus (6/6)

---



The output results are verified by console/waveform viewer



The output results are verified by testbench or stimulus block

# Syntax for Text Monitoring

---

- Define Time Format (%t)
  - `$timeformat(unit, precision, suffix, min_width)`
  - `$timeformat(-9, 2, " ns", 10)` stands for:
    - 10E-9 second as time unit
    - 2 decimals as floating-point precision
    - Print " ns" after time information
    - Preserve 10 characters for displaying
- Display & Monitor
  - Display: print at once
    - `$display([format_string], arg_list)`
  - Monitor: print if something in `arg_list` changes
    - `$monitor([format_string], arg_list)`
  - Similar syntax as `printf()` in C language

# Display Format

---

- Similar to C Language

## Format Specifiers:

|        |       |         |     |        |       |        |      |          |
|--------|-------|---------|-----|--------|-------|--------|------|----------|
| %b     | %c    | %d      | %h  | %m     | %o    | %s     | %t   | %v       |
| binary | ASCII | decimal | hex | module | octal | string | time | strength |

## Escaped Literals:

|              |                                |           |         |     |
|--------------|--------------------------------|-----------|---------|-----|
| \"           | \<1-3 digit octal number>      | \\        | \n      | \t  |
| double quote | ASCII representation of number | backslash | newline | tab |

# Terminal Monitoring

- Example

```
...
initial begin
    $display(" time stime realtime in1 o1");
    $display("-----");
    $timeformat(-9, 2, " ns", 10);
    $monitor("%d %d %t %b %b", $time, $stime, $realtime, in1, o1);
    in1 = 0;
    #10 in1 = 1;
    #10 $finish;
end
...
```

| time | stime | realtime | in1 | o1 |
|------|-------|----------|-----|----|
| 0    | 0     | 0.00 ns  | 0   | x  |
| 10   | 10    | 9.53 ns  | 0   | 1  |
| 10   | 10    | 10.00 ns | 1   | 1  |
| 20   | 20    | 19.53 ns | 1   | 0  |

# Waveform Verification

---

- Value Change Dump (VCD) format
  - Indigenously supported by most simulators
  - Using ASCII text for waveform recording, extremely huge file size
  - `$dumpfile("filename");`  
`$dumpvars();`
- Fast Signal Database (FSDB) format
  - Defined by *SpringSoft Verdi debugging system*
  - More compact format, small file size
  - `$fsdbDumpfile("filename");`  
`$fsdbDumpvars(0, test_module_name, "+mda");`

# Waveform Verification

---

- Some notes on waveform output
  - **Setting "+mda" could slow down simulation and increase waveform file size**
    - E.g., when there is a large behavioral RAM
  - **Setting the first argument to 0 could also slow down simulation and increase file size**
    - The first argument (`level`)
      - 0: all signals in all scopes
      - n: all signals in current scope and all scopes n-1 levels below.
    - `$fsdbDumpvars(2, test_module_name, "+mda");`

# Verification with Golden Patterns

- Very popular way for verification with massive test patterns

```
initial begin
    $readmemh( "GoldenPattern.txt",  golden_pattern);
    pattern_num = 0; err = 0;
end
always @ (posedge CLK) begin
    if (OUTPUT_READY) begin
        current_golden = golden_pattern[pattern_num];
        if ( data_out !== current_golden ) begin
            $display("ERROR at %d:output (%h) !=expect (%h)",
                    pattern_num, data_out, current_golden);
            err = err + 1 ;
        end
        pattern_num = pattern_num + 1 ;
    end
    if( pattern_num == N_PAT ) begin
        if (err == 0) $display("All correct, congratulations!");
        else        $display("There are %d errors!", err);
        $finish;
    end
end
end
```

# Other Tips

---

- Verilog *indexed part select*

- In Verilog, we may want to select a fixed number of bits using variables (instead of compile-time constants)
- E.g., separate an 128-bit input into 16 8-bit numbers

```
for (i = 0; i < 16; i = i + 1)
    data_mem[i] <= data_input[(i+1)*8 : i*8];
```

- However, **the syntax is illegal**:
  - ncvlog: \*E,NOTPAR: Illegal operand for constant expression [4(IEEE)].
- Why?
  - The variable `i` is not a compile-time constant



# Other Tips

- Verilog *indexed part select*
  - Solution: indexed part select

```
for (i = 0; i < 16; i = i + 1)
    data_mem[i] <= data_input[i*8 +: 8];
```

- Syntax:

```
reg [31:0] A;
reg [0:31] B;

A[ 0 +: 8] // == A[ 7 : 0]
A[15 -: 8] // == A[15 : 8]
B[ 0 +: 8] // == B[ 0 : 7]
B[15 -: 8] // == B[ 8 : 15]
```

# Other Tips

---

- Verilog constants
  - Verilog provides 3 ways for constant definition
  - ``define`
    - Affects all files
  - `parameter`
    - Affects only the current module
    - Can be parameterized when instantiated
  - `localparam`
    - Affects only the current module
    - Cannot be parameterized when instantiated

# Other Tips


---

- Verilog constant conventions
  - Use **`define** for system-level constants
    - E.g., clock frequency, maximum simulation cycles
  - Use **parameter** for configurable module constants
    - E.g., data width, address width
  - Use **localparam** for unconfigurable module constants
    - E.g., FSM state indices

# Other Tips

```
`timescale 1ns/10ps
`define CYCLE      10.0
`define MAX_CYCLE 10000
```

```
module tb;
    // ...
    my_module # (
        .DATA_WIDTH(32),
        .ADDR_WIDTH(16)
    ) my_module_inst (
        .clk(clk),
        .rst_n(rst_n),
        .data(data),
        .addr(addr)
    );
    // ...
endmodule
```



```
module my_module # (
    parameter DATA_WIDTH = 32,
    parameter ADDR_WIDTH = 16
) (
    input          clk,
    input          rst_n,
    input [DATA_WIDTH-1:0] data,
    input [ADDR_WIDTH-1:0] addr
);
    reg [1:0] state;
    localparam S_IDLE   = 0;
    localparam S_READ   = 1;
    localparam S_CALC   = 2;
    localparam S_WRITE  = 3;
    // ...
endmodule
```

# Other Tips

- Use === and !== in testbench for equivalence check
  - Comparisons with x/z are always false
  - In this case, if output\_data is always x, errors will still be 0:

```
if (output_data !== output_golden) begin
    err = err + 1;
end
```

- Use . to access members of lower level

```
module tb;
    behav_ram i_mem(
        // ...
    );
    initial begin
        $readmemh("data.mem",
            i_mem.mem);
    end
endmodule
```

```
module behav_ram(
    // ...
);
    reg [31:0] mem [0:32767];
endmodule
```

