

# Computer-Aided VLSI System Design

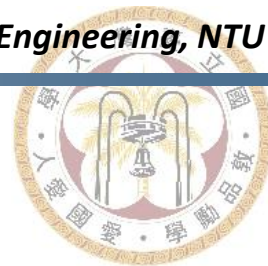
## Chapter 3-1. Synthesizable Verilog Coding

Lecturer: Chun-Hao Chang

*Graduate Institute of Electronics Engineering,  
National Taiwan University*



**NTUGIEE**



# Outline

- **Introduction to Logic Synthesis**
- Synthesizable RTL Coding
  - Syntax
  - Structure
- Circuit-Level Optimization
  - Translating RTL to Circuits
  - Circuit Refining Tips
- Checking Synthesizability

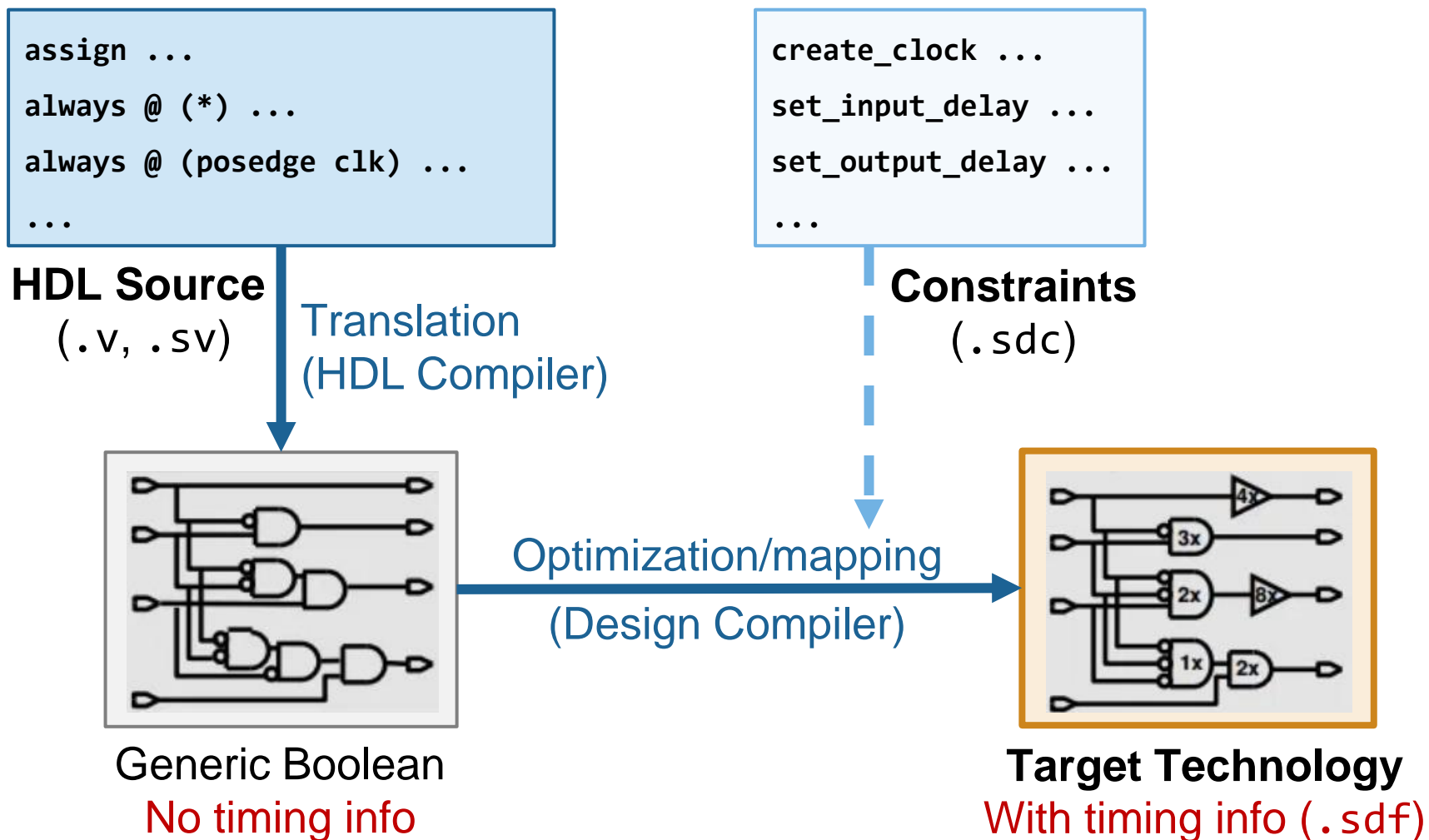


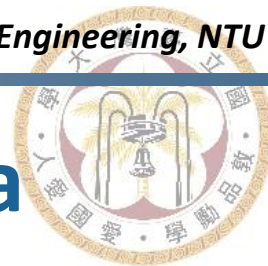
# Introduction to Logic Synthesis

- **Definition**
  - Converting a **high-level description** of hardware (HDL) to an optimized **gate-level representation** (netlist)
- Logic synthesis uses **standard cell library**
  - Basic logic gates (and, or, xor, ...)
  - Macro cells (flip-flop, adder, mux, memory, ...)
- Logic synthesis is **constraint-driven**
  - Timing, area, and power



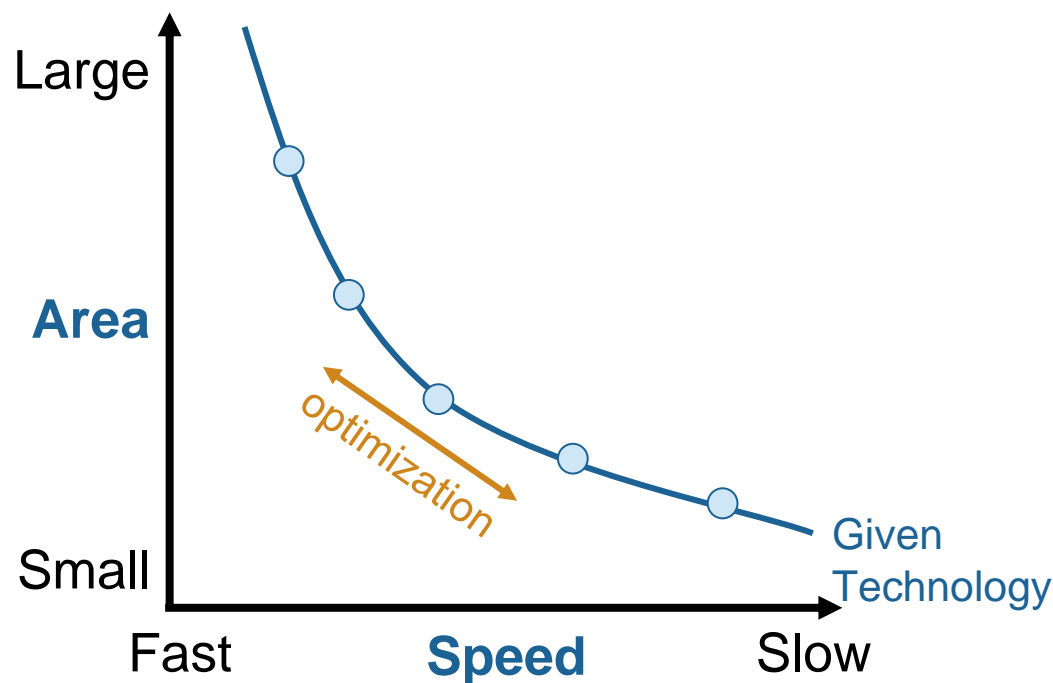
# Logic Synthesis Flow

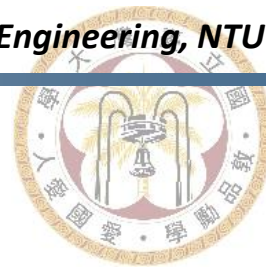




# Tradeoff between Timing and Area

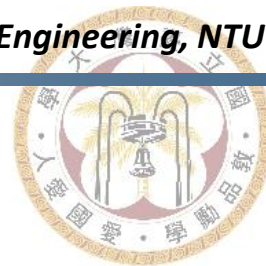
- Given the technology and the same RTL code, we can:
  - Sacrifice area for higher speed (frequency)
  - Sacrifice speed for lower area
- Set constraints properly to obtain preferable results





# Translating Verilog to Logic Gates

- Some Verilog syntax are easy to translate:
  - Primitive gates (and, or, xor, ...)
  - Continuous assignments (assign)
- **Behavioral statements may not be synthesizable**
- To design digital circuit with RTL coding, we should:
  1. Know the **synthesizable syntax** of Verilog
  2. Follow the **RTL structural conventions**
  3. Know **how** RTL is mapped to circuits



# Outline

- Introduction to Logic Synthesis
- **Synthesizable RTL Coding**
  - Syntax
  - Structure
- Circuit-Level Optimization
  - Translating RTL to Circuits
  - Circuit Refining Tips
- Checking Synthesizability



# Synthesizable Verilog

- Verilog is not only designed for synthesizable designs
  - Testbench
  - Behavioral modeling
- Not all Verilog syntax are synthesizable
- **Only a subset of Verilog syntax can be synthesized**
- **RTL codes containing only these syntax are synthesizable**





# Synthesizable Verilog Syntax

- Basics

- reg, wire
- logic (SystemVerilog)
- input, output
- always blocks
- module and instantiation
- Operators (arithmetic, logical, relational...)
- Continuous assignments (assign)
- Blocking assignments (=)
- Non-blocking assignments (<=)

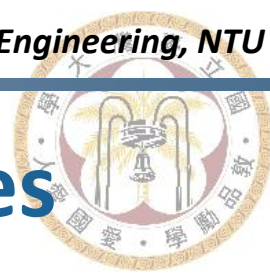
- Conditional

- if ... else
- case



# Synthesizable Verilog Syntax

- Constants
  - parameter
  - localparam
  - enum (SystemVerilog)
- Compiler directives (``define`, ``ifdef`, ...)
- for loop (as long as the iteration is constant)
- task (as long as there is no timing/delay constructs)
- function (as long as there is no timing/delay constructs)

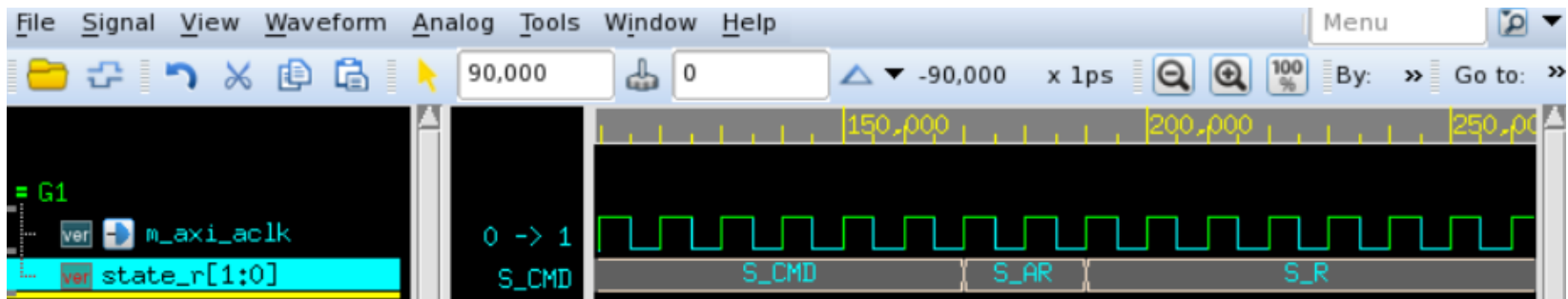


# Using SystemVerilog enum for States

- In SystemVerilog, we can use `typedef enum` to define states:

```
typedef enum logic [1:0] {  
    S_CMD,  
    S_AR,  
    S_R,  
    S_DONE  
} State;  
State state_r, state_w;
```

- The state names will be automatically shown in nWave:





# Synthesizable Verilog Operators

- Unary `!, &, |, ^, ~&, ~|`
- Binary bit-wise `&, |, ^, ~^`
- Binary logical `&&, ||`
- Arithmetic `+, -, *`
- Relational `>, <, >=, <=`
- Equality `==, !=`
- Logical shift `>>, <<`
- Conditional `?:`
- Division `/` (with DesignWare, not recommended)



## Example

$a = 1011$   
 $b = 0010$

Bit-wise  
 $a|b = 1011$   
 $a\&b = 0010$

Unary reduction  
 $|a = 1$   
 $\&a = 0$

Logical  
 $a||b = 1$   
 $a\&\&b = 1$

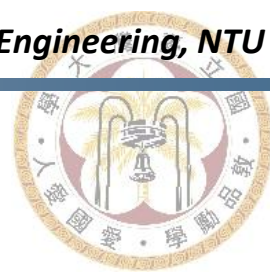


# Not Synthesizable Verilog Syntax

- **Delay (#)**
- **Identity (==, !=)**
- **initial**
- **repeat**
- **forever**
- **wait**
- **fork ... join**

Common syntax in testbench  
but not synthesizable

- event
- deassign
- force
- release
- UDP (user defined primitive)
- time



## Pitfall: Comparing to X or Z

- **A comparison to an X or Z is always False**
  - May cause simulation to disagree with synthesis
- If you need to compare to X or Z in testbench, **use === and !==**

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

Simulation warning message



# Equality Operators

- `==` is the **logical** equality operator
  - Which the logical equality operator, an X in either of the operand is logicality unknown.
- `===` is the **case** equality operator
  - Which can still evaluate to true(1) or false(0) when X or Z values are present in the operands.

<code>==</code>	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

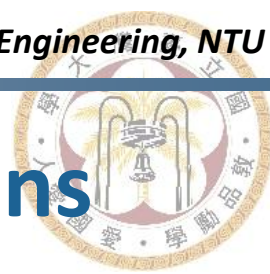
<code>===</code>	0	1	X	Z
0	1	0	0	0
1	0	1	0	0
X	0	0	1	0
Z	0	0	0	1





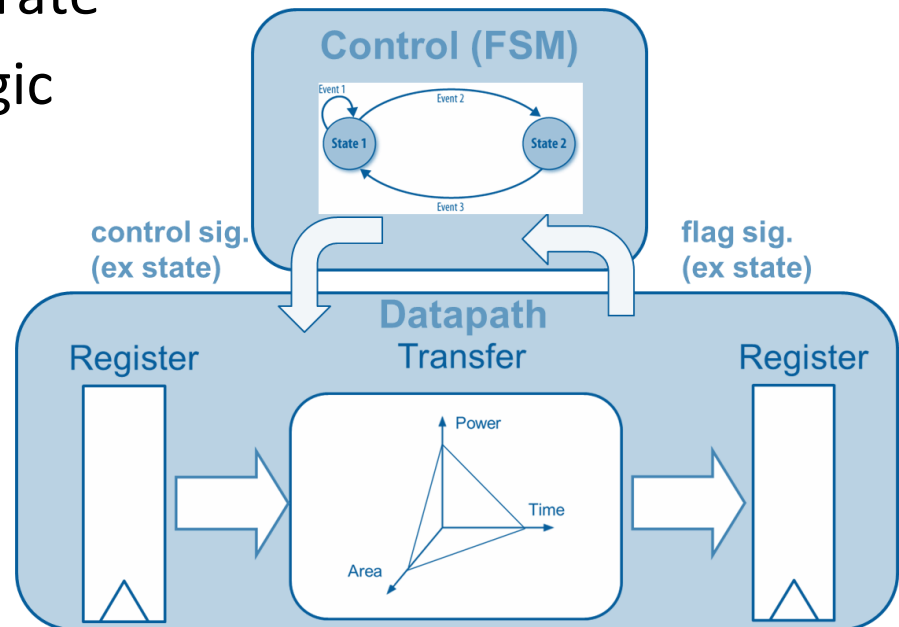
# Outline

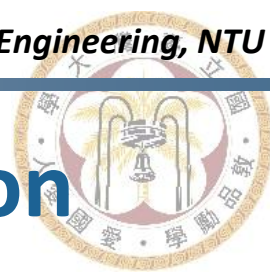
- Introduction to Logic Synthesis
- **Synthesizable RTL Coding**
  - Syntax
  - **Structure**
- Circuit-Level Optimization
  - Translating RTL to Circuits
  - Circuit Refining Tips
- Checking Synthesizability



# Synthesizable RTL Coding Conventions

- Separate **combinational** and **sequential** parts
  - Use `always @ (*)` to model combinational logic
  - Use `always @ (posedge clk)` for sequential logic
- Separate **control** and **datapath** modules
- Keep major design blocks separate
- Keep related combinational logic in the same module
- Register at hierarchical output





# Combinational-sequential Separation

## ▪ Purely combinational blocks

```
always @ (*) begin
    if (valid)
        data_w = data_in;
    else
        data_w = data_r;
end
```

## ▪ Purely sequential blocks

```
always @ (posedge clk) begin
    if (rst)
        data_r <= 0;
    else
        data_r <= data_w;
end
```

## ▪ Avoid logic in sequential blocks

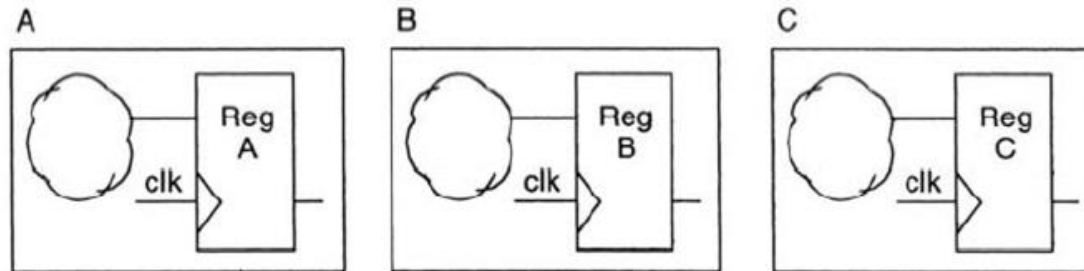
```
always @ (posedge clk) begin
    if (rst) begin
        data_r <= 0;
    end
    else begin
        if (valid)
            data_r <= data_in;
        end
    end
end
```

**Note:** writing logic in sequential blocks is still synthesizable, but does not model hardware architecture as clear, so use it carefully.



# Register All Output Ports

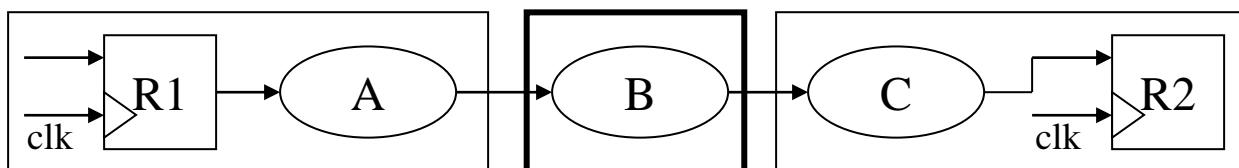
- **It is recommended to register all output signals in a module**
  - All the inputs of each block arrive with the same delay
  - Output drive strength is equal to the drive strength of a DFF



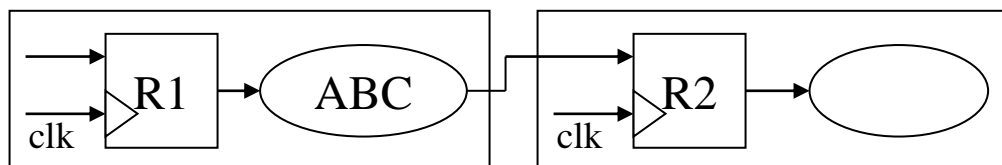
- However, modern synthesis tools can perform optimization across hierarchical boundaries:
  - `compile -boundary_optimization`
  - `compile -auto_ungroup`
  - `compile_ultra`



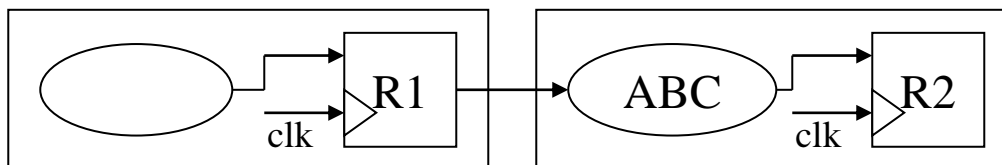
# Register All Output Ports



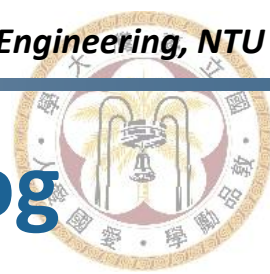
No output  
register



**Input register**



**Output register**



# Example: Register Outputs in Verilog

```
module mac (a, b, c);  
    input  [ 7:0] a, b;  
    output [15:0] c;  
  
    reg [15:0] c_r, c_w;  
    always @ (*) begin  
        c_w = c_r + a*b;  
    end  
    always @ (posedge clk) begin  
        if (rst) c_r <= 0;  
        else    c_r <= c_w;  
    end  
  
    assign c = c_r;  
endmodule
```

## Naming conventions

- $X\_r, X\_w$  (reg, wire)
- $X, X\_next$  (next value)
- $X\_reg, X\_next$
- $X, X\_n$
- ...

$c\_r$  is a register that accumulates  $axb$

Assign  $c\_r$  to output port  $c$ ,  
now  $c$  is an output register



# Outline

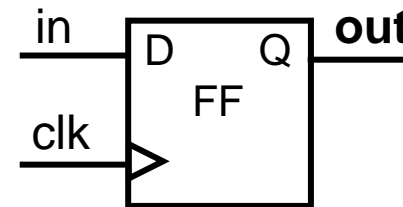
- Introduction to Logic Synthesis
- Synthesizable RTL Coding
  - Syntax
  - Structure
- **Circuit-Level Optimization**
  - Translating RTL to Circuits
  - Circuit Refining Tips
- Checking Synthesizability



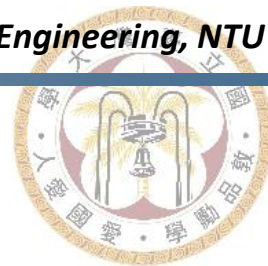
# Mapping of Sequential Circuits

- Purely sequential circuits can be mapped as **flip-flops**
- **Example:** a sequential `always` block -> D flip-flop

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





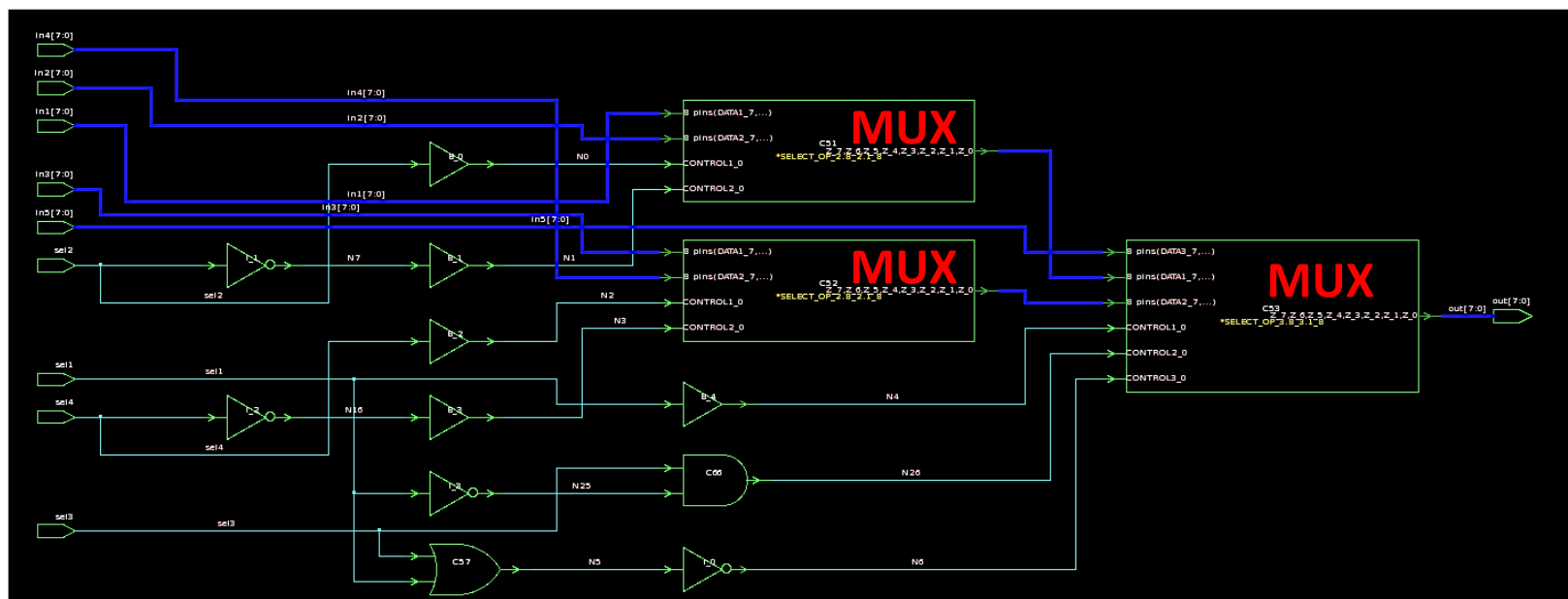


# Mapping of if Statements

- An **if statement** is mapped to a **multiplexer (MUX)**
- if statements can be nested
  - Multiple levels of MUXs

```

always @ (*) 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 Statements

- Different coding styles of if statements:

- **Multiple if**

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

- **Single if...else**

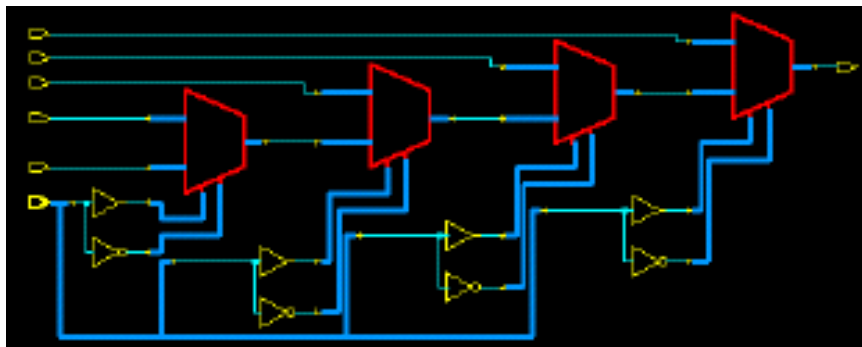
```
module single_if (  
    a, b, c, d, e, sel, f  
);  
    input a, b, c, d, e;  
    input [3:0] sel;  
    output reg f;  
  
    always @ (*) begin  
        f = e;  
        if (sel[3]) f = d;  
        else if (sel[2]) f = c;  
        else if (sel[1]) f = b;  
        else if (sel[0]) f = a;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

- The **single if...else** style infer a **priority encoder** circuit



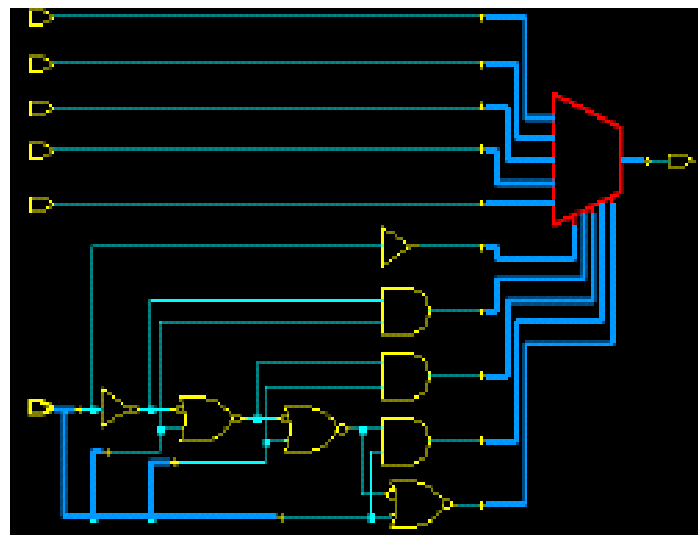
# Mapping of if Statements

- Multiple if



longer delay,  
smaller area

- Single if...else



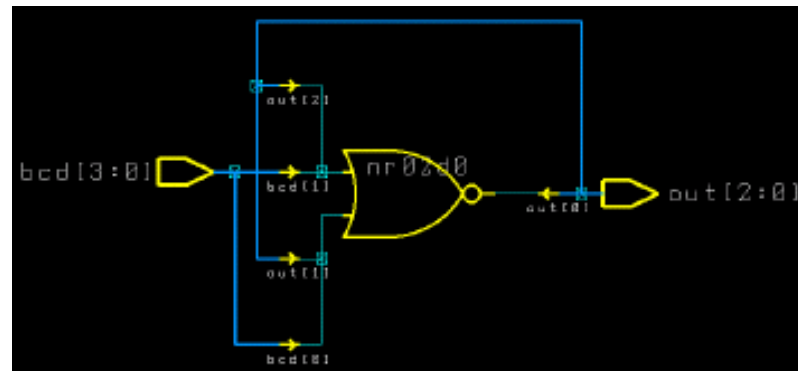
(priority encoder)  
shorter delay,  
larger area

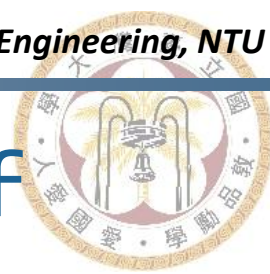


# Mapping of case Statements

- A case statement is **full** if all possible branches are specified
- Similar to the `if...else` statement, a **full case** statement is mapped to a **multiplexer (MUX)**

```
case (sel)
    2'd0:    out = 3'b001;
    2'd1:    out = 3'b010;
    2'd2:    out = 3'b100;
    default: out = 3'b000;
endcase
```

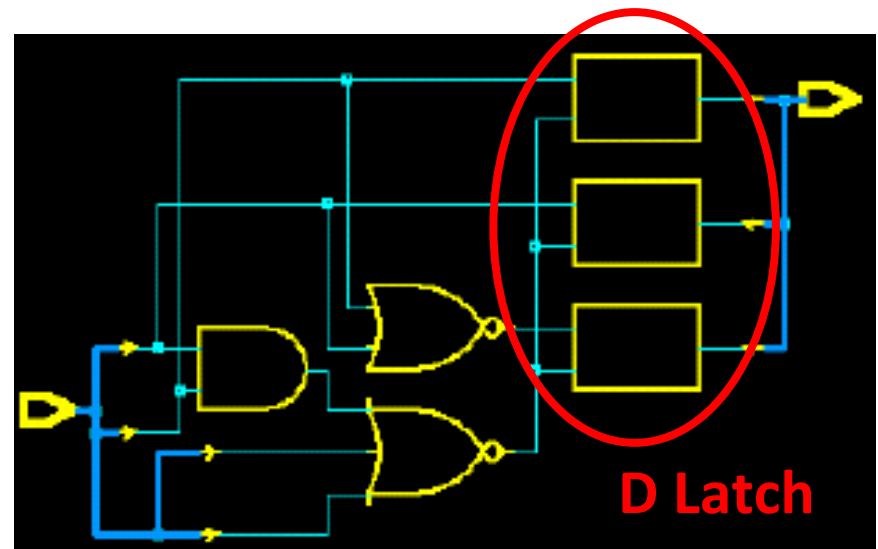
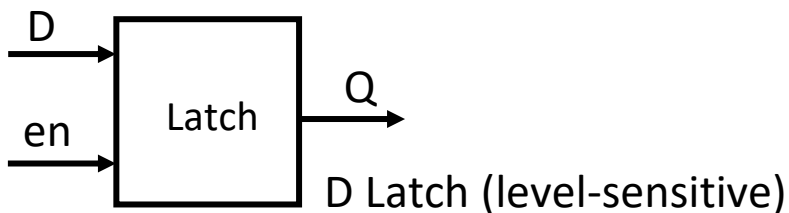




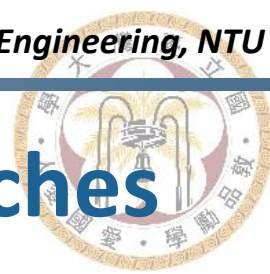
# Inferred Latches from case and if

- If a case statement is not full, it will infer latches
- If an if statement does not specify all possible branches, it will also infer latches

```
case (sel)
  2'd0: out = 3'b001;
  2'd1: out = 3'b010;
  2'd2: out = 3'b100;
endcase
```



- Latches are not fully supported by synthesis tools for static timing analysis (STA), so we should avoid latches in our design



# Assigning Default Values to Avoid Latches

## ▪ case

```
case (sel)
    2'd0:    out = 3'b001;
    2'd1:    out = 3'b010;
    2'd2:    out = 3'b100;
    default: out = 3'b000;
endcase
```

or

```
out = 3'b000;
case (sel)
    2'd0: out = 3'b001;
    2'd1: out = 3'b010;
    2'd2: out = 3'b100;
endcase
```

## ▪ if...else

```
if (sel == 2'd0)
    out = 3'b001;
else if (sel == 2'd1)
    out = 3'b010;
else if (sel == 2'd2)
    out = 3'b100;
else
    out = 3'b000;
```

or

```
out = 3'b000;
if (sel == 2'd0)
    out = 3'b001;
else if (sel == 2'd1)
    out = 3'b010;
else if (sel == 2'd2)
    out = 3'b100;
```

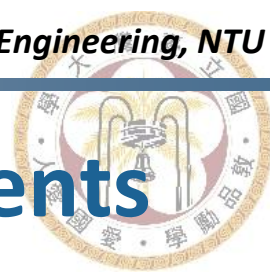


## Note: Logic in Sequential Blocks

- Though we do not recommend it, you can implement logic in sequential blocks, which is synthesizable
- **In a sequential block, if a case is not full, or if an if is not full, NO latches will be inferred** (since there are flip-flops already):

```
always @ (posedge clk) begin
    if (rst) begin
        out <= 3'b000;
    end
    else begin
        if (sel == 2'd0)      out <= 3'b001;
        else if (sel == 2'd1) out <= 3'b010;
        else if (sel == 2'd2) out <= 3'b100;
    end
end
```

```
always @ (posedge clk) begin
    if (rst) begin
        out <= 3'b000;
    end
    else begin
        case (sel)
            2'd0: out <= 3'b001;
            2'd1: out <= 3'b010;
            2'd2: out <= 3'b100;
        endcase
    end
end
```

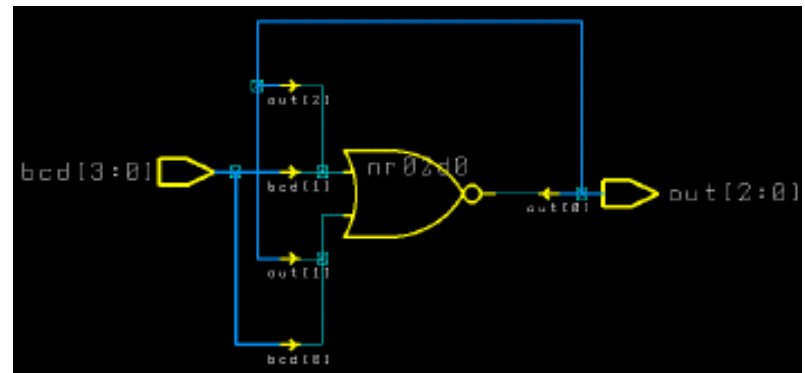


# Synthesis Directives for case Statements

- If you do not specify all possible branches, but other branches will never occur, you can use this directive to **remove latches**:

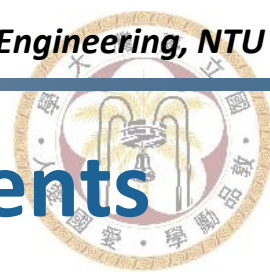
`//synopsys full_case`

```
case (sel) //synopsys full_case
  2'd0: out = 3'b001;
  2'd1: out = 3'b010;
  2'd2: out = 3'b100;
endcase
```



- Note that if you miss a variable assignment in a specified branch, latches will still be inferred for that variable



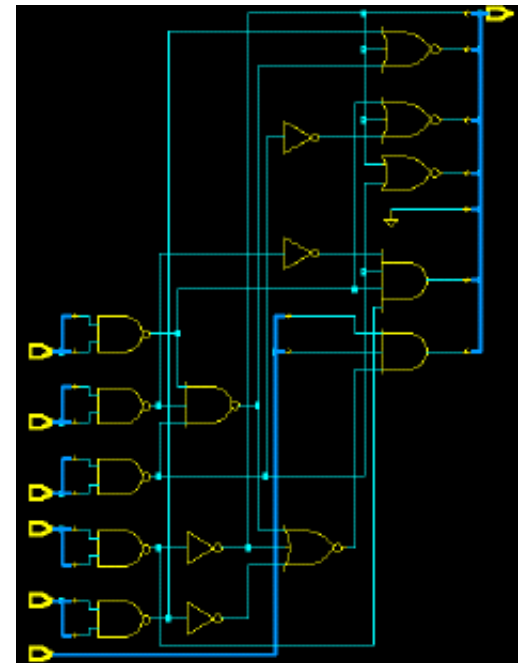


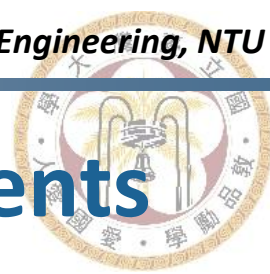
# Synthesis Directives for case Statements

- If only one branch occurs at the same time in a case statement, you can use this directive to **remove priority encoder**:

```
//synopsys parallel_case
```

```
always @ (*) begin
    case (2'b11)
        u: out = 6'b000001;
        v: out = 6'b000010;
        w: out = 6'b000100;
        x: out = 6'b001000;
        y: out = 6'b010000;
        z: out = 6'b100000;
        default: out = 6'b000000;
    endcase
end
```



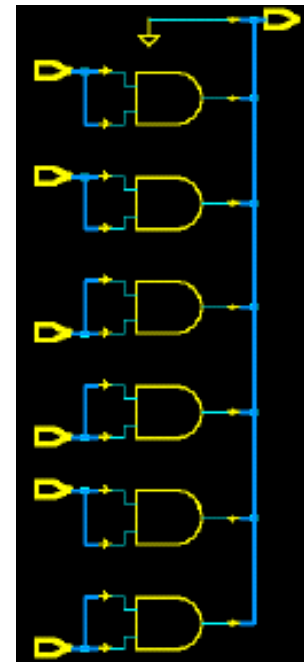


# Synthesis Directives for case Statements

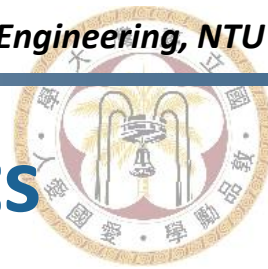
- If only one branch occurs at the same time in a case statement, you can use this directive to **remove priority encoder**:

```
//synopsys parallel_case
```

```
always @ (*) begin
    case (2'b11) //synopsys parallel_case
        u: out = 6'b000001;
        v: out = 6'b000010;
        w: out = 6'b000100;
        x: out = 6'b001000;
        y: out = 6'b010000;
        z: out = 6'b100000;
        default: out = 6'b000000;
    endcase
end
```



Smaller area



# Synthesis Directives: Final Thoughts

- **Try to avoid using these directive**
  - Less predictable behavior than assigning default values
  - Possible pre-synthesis and post-synthesis mismatch



# Mapping of for Statements

- **for loops are synthesizable only when the number of iterations is a compile-time constant integer value:**
  - Integer literal
  - Constants macro (``define`)
  - Parameters (`parameter`, `localparam`)
- Start, end, and step should all be constant integer
- For synthesis tools, **for loops are unrolled**, then synthesized

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

unroll →

```
always @ (*) 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** ( $\&$ ,  $|$ ,  $\wedge$ ,  $\sim\wedge$ )
  - Mapped to **logic gates** directly
- **Unary Logical Operators** ( $\&$ ,  $|$ ,  $\wedge$ ,  $\sim\wedge$ ,  $\sim$ ,  $!$ )
  - 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

- **Addition**
  - Full adder
- **Subtraction**
  - Full adder with 2's complement inverter
- **Multiplication**
  - Full adder array
- **Division & Modulo**
  - May need to instantiate DesignWare modules
  - No direct mapping to simple elements



# Mapping of Arithmetic Operators

- **Shift operations (<<, >>)**

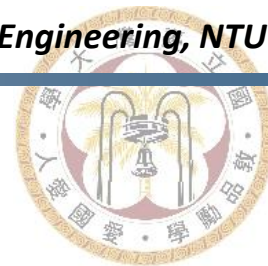
- Shift by constant: wire assignment and bit select

```
// equivalent  
assign c = a[7:0] >> 2;  
assign c = {2'b0, a[7:2]};
```

- Shift by variable: barrel shifter

- **Multiplication & Division of  $2^N$**

- Simplified as shift
- Left shift by N bit:      Multiply by  $2^N$
- Right shift by N bit:      Divide by  $2^N$

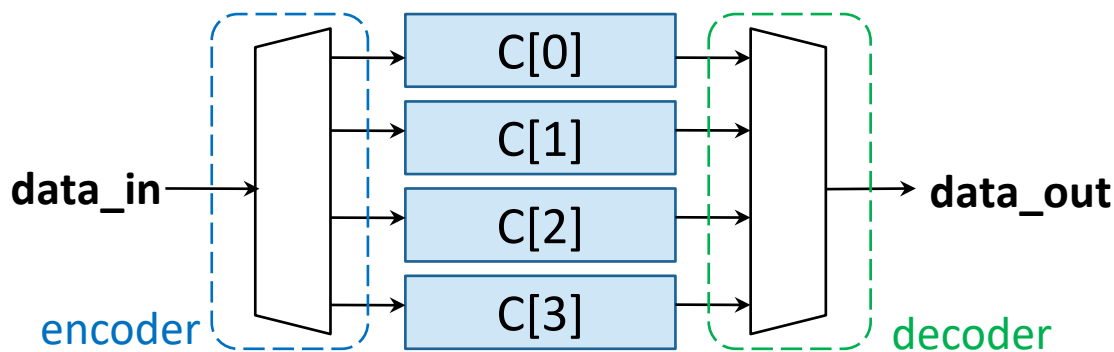


# Vector Arrays

- **Example:** A 4x8b vector array

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

- Hardware translation



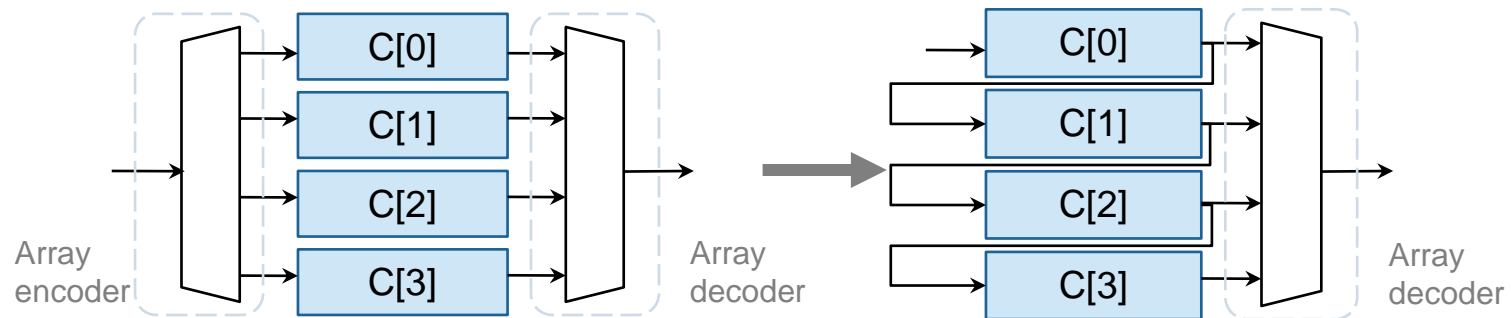




# Vector Arrays

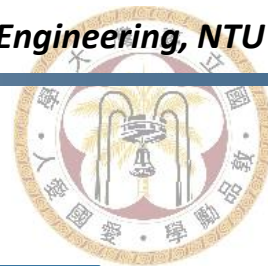
## ▪ Encoder/decoder issue

- Encoder and decoder may be too large for larger arrays
- Sometimes we can reduce the vector array to shift registers, so encoder or decoder will not be generated



## ▪ Dumping vector arrays in waveform

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



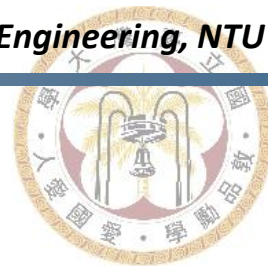
## Example: Shift Registers

```
integer i;
reg [7:0] data_r [0:127];
reg [7:0] data_w [0:127];

always @ (*) begin
    for (i = 0; i < 128; i = i + 1) data_w[i] = data_r[i];

    if (valid) begin
        for (i = 1; i < 128; i = i + 1) data_w[i] = data_r[i-1];
        data_w[0] = data_in;
    end
end

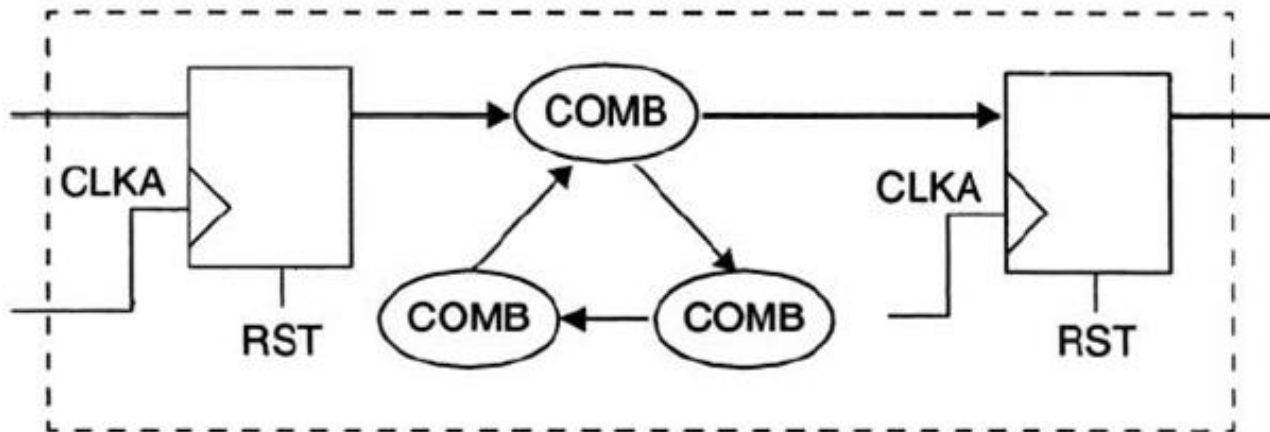
always @ (posedge clk or negedge rst_n) begin
    if (~rst_n) begin
        for (i = 0; i < 128; i = i + 1) data_r[i] <= 0;
    end
    else begin
        for (i = 0; i < 128; i = i + 1) data_r[i] <= data_w[i];
    end
end
```



# Combinational Loop

- An output of a combinational block feeds back to an input of the same block
- **Combinational loops should be avoided**

**Bad: Combinational processes are looped**





# Outline

- Introduction to Logic Synthesis
- Synthesizable RTL Coding
  - Syntax
  - Structure
- **Circuit-Level Optimization**
  - Translating RTL to Circuits
  - **Circuit Refining Tips**
- Checking Synthesizability



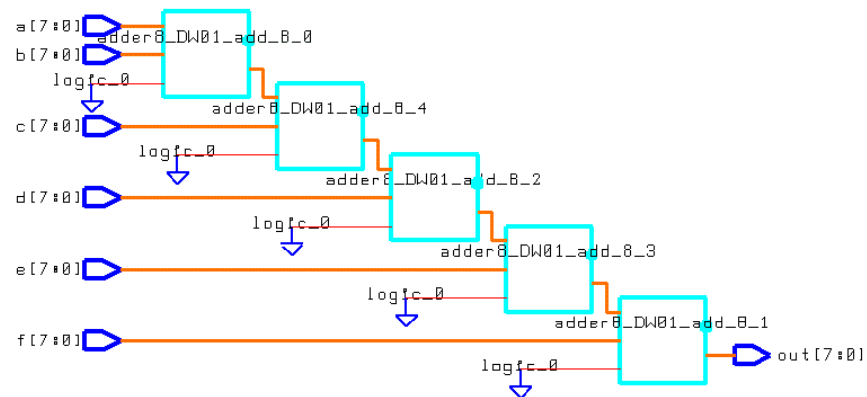
# Circuit-level Refinement

- **Be aware of the translation between circuits and codes**
  - Operators -> computation units
  - Conditional statements -> multiplexers
  - Sequential blocks -> registers
  
- Planning your design using a **block diagram**
  - Understand your design architecture
  - Easier analysis of design cost (area/timing/critical path)

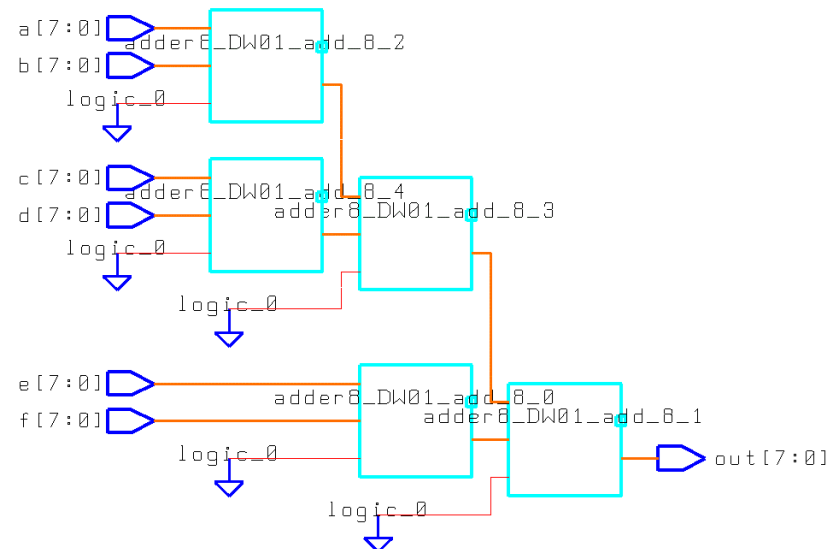


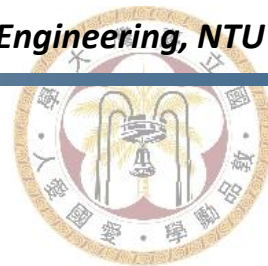
# Parentheses

•  $out = a+b+c+d+e+f;$



•  $out = (a+b)+(c+d)+(e+f);$





# Operator Bit-width

```
module test(a,b,out);  
  input [7:0] a,b;  
  output [8:0] out;  
  assign out=add_lt_10(a,b);
```

```
  function [8:0] add_lt_10;  
    input [7:0] a,b;  
    reg [7:0] temp;  
    begin  
      if (b<10) temp=b;  
      else temp=10;  
      add_lt_10=a+temp[3:0]; //use [3:0] for temp  
    end  
  endfunction  
  
endmodule
```

**Not involve redundant bits**



# Different Types of Addition

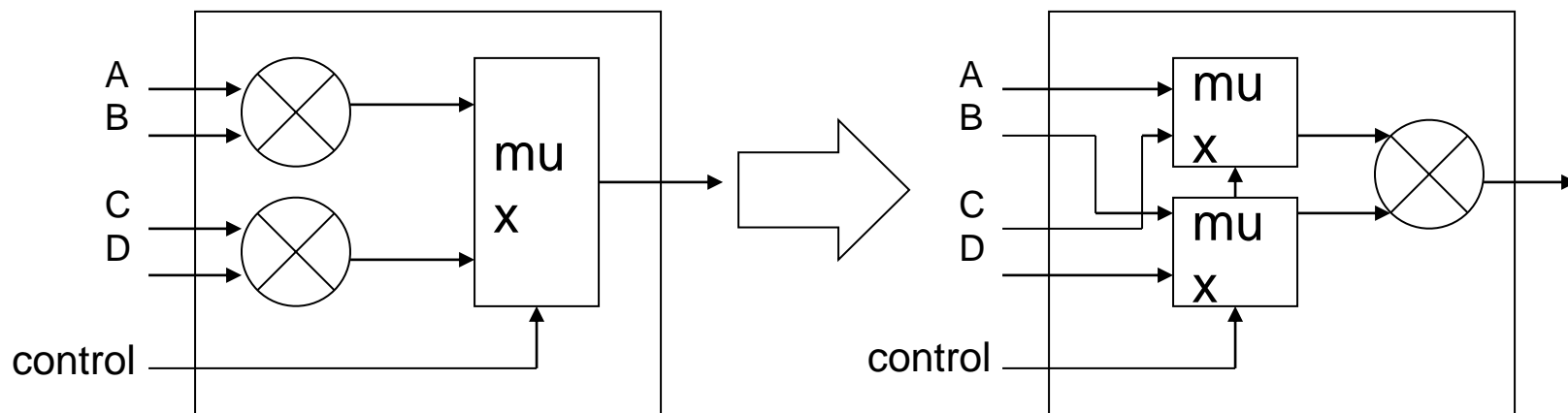
```
parameter size = 8;  
wire  [3:0] a,b,c,d,e;  
assign c = size + 2;    // constant  
assign d = a + 1;       // incrementer  
assign e = a + b;       // adder
```





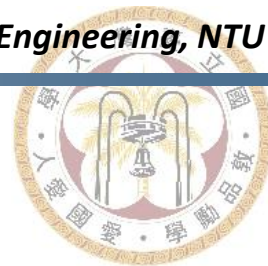
# Resource Reusing

- 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
```



# Data-Path Duplication

**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'b100000000;
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**

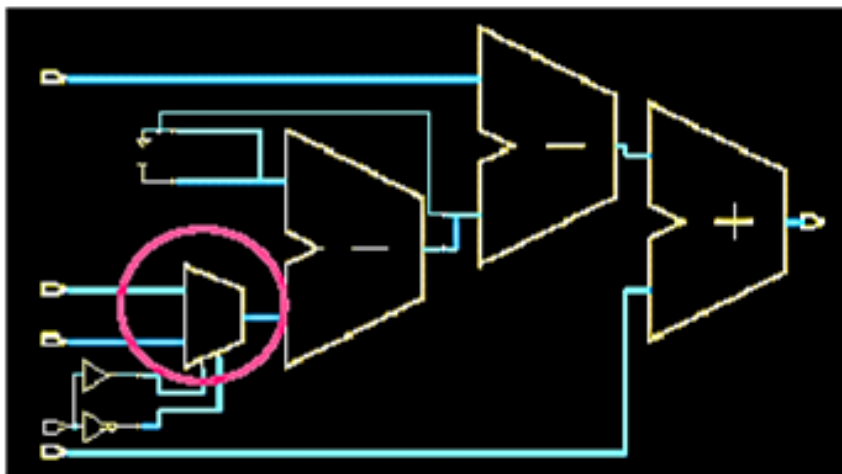
```
module PRECOMPUTED (ADDRESS, PTR1, PTR2, B, CONTROL, COUNT);
input [7:0] PTR1, PTR2;
input [15:0] ADDRESS, B;
input CONTROL;
output [15:0] COUNT;
parameter [7:0] BASE = 8'b100000000;
wire [7:0] OFFSET1, OFFSET2;
wire [15:0] ADDR1, ADDR2, COUNT1, COUNT2;
assign OFFSET1 = BASE - PTR1; // Could be f(BASE, 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
```



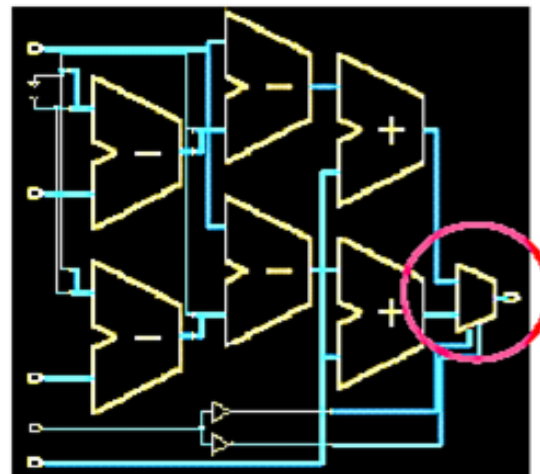
# Data-Path Duplication

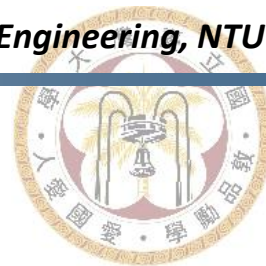
- Assume that signal "CONTROL" is the latest arrival pin
- Sacrifice area to gain latency reduction

No Duplicated



Duplicated





# Comparison Refinement

- 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
```

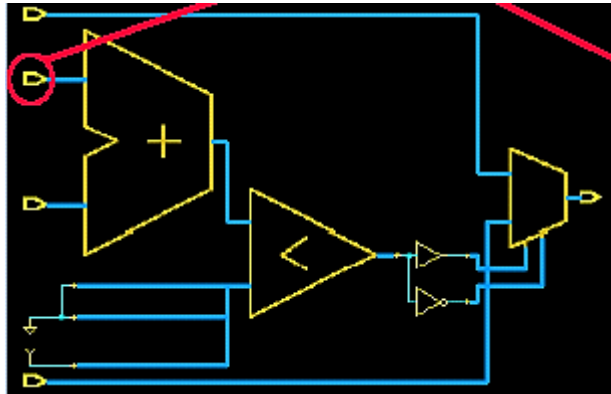
## Improved

```
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
```

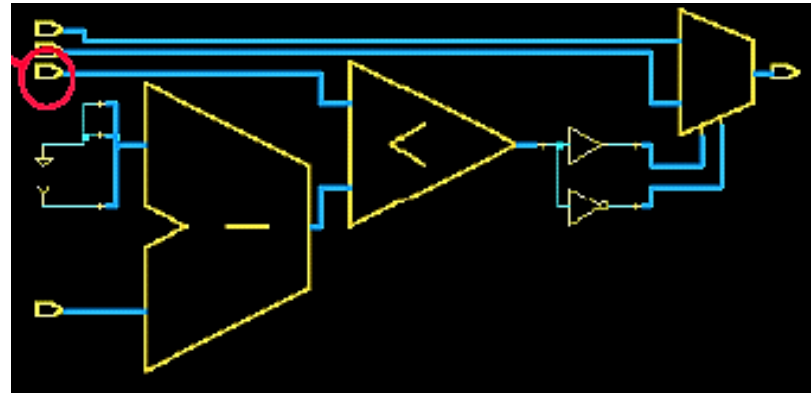


# Comparison Refinement

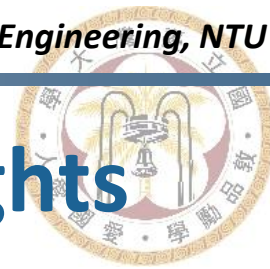
- In this example, latency reduced



Before\_improved

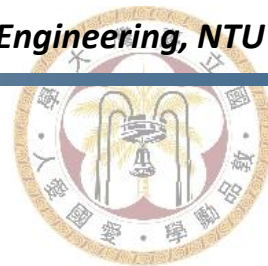


Improved



# Circuit-level Refinement: Final Thoughts

- Modern synthesizers *may* optimize these kinds of statements
- Optimize the **critical** part of your code first:
  - Critical paths
  - A module that is instantiated multiple times



# Outline

- Introduction to Logic Synthesis
- Synthesizable RTL Coding
  - Syntax
  - Structure
- Circuit-Level Optimization
  - Translating RTL to Circuits
  - Circuit Refining Tips
- **Checking Synthesizability**



# Checking Synthesizability

- ***Synopsys VC SpyGlass***
  - Verilog/SystemVerilog linting tool
  - Early structural and functional analysis for logic designs
  
- ***Design Compiler (DC)***
  - Synthesis tool
  - DC can list flip-flops and latches in your design
    - > `dc_shell`
    - > `read_verilog yourdesign.v`





# Checking Latches with dc

```
Inferred memory devices in process
in routine test line 11 in file
'/home/raid7_2/user08/r08011/synth_test/test.v'.
```

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
h_w_reg	Latch	16	Y	N	N	N	-	-	-

```
Inferred memory devices in process
in routine test line 16 in file
'/home/raid7_2/user08/r08011/synth_test/test.v'.
```

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
g_r_reg	Flip-flop	16	Y	N	N	N	N	N	N
h_r_reg	Flip-flop	16	Y	N	N	N	N	N	N

```
Presto compilation completed successfully.
Current design is now '/home/raid7_2/user08/r08011/synth_test/test.db:test'
Loaded 1 design.
Current design is 'test'.
test
```

# Computer-Aided VLSI System Design

## Chapter 3-2. Debugging and Testbench

Lecturer: Chun-Hao Chang

*Graduate Institute of Electronics Engineering,  
National Taiwan University*



**NTUGIEE**



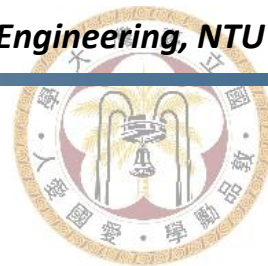
# Outline

- **Debugging Tools**
- Testbench
  - Simulation Overview
  - Instantiating DUT
  - Creating Clocks
  - Applying Stimulus
  - Verification
- Other Tips

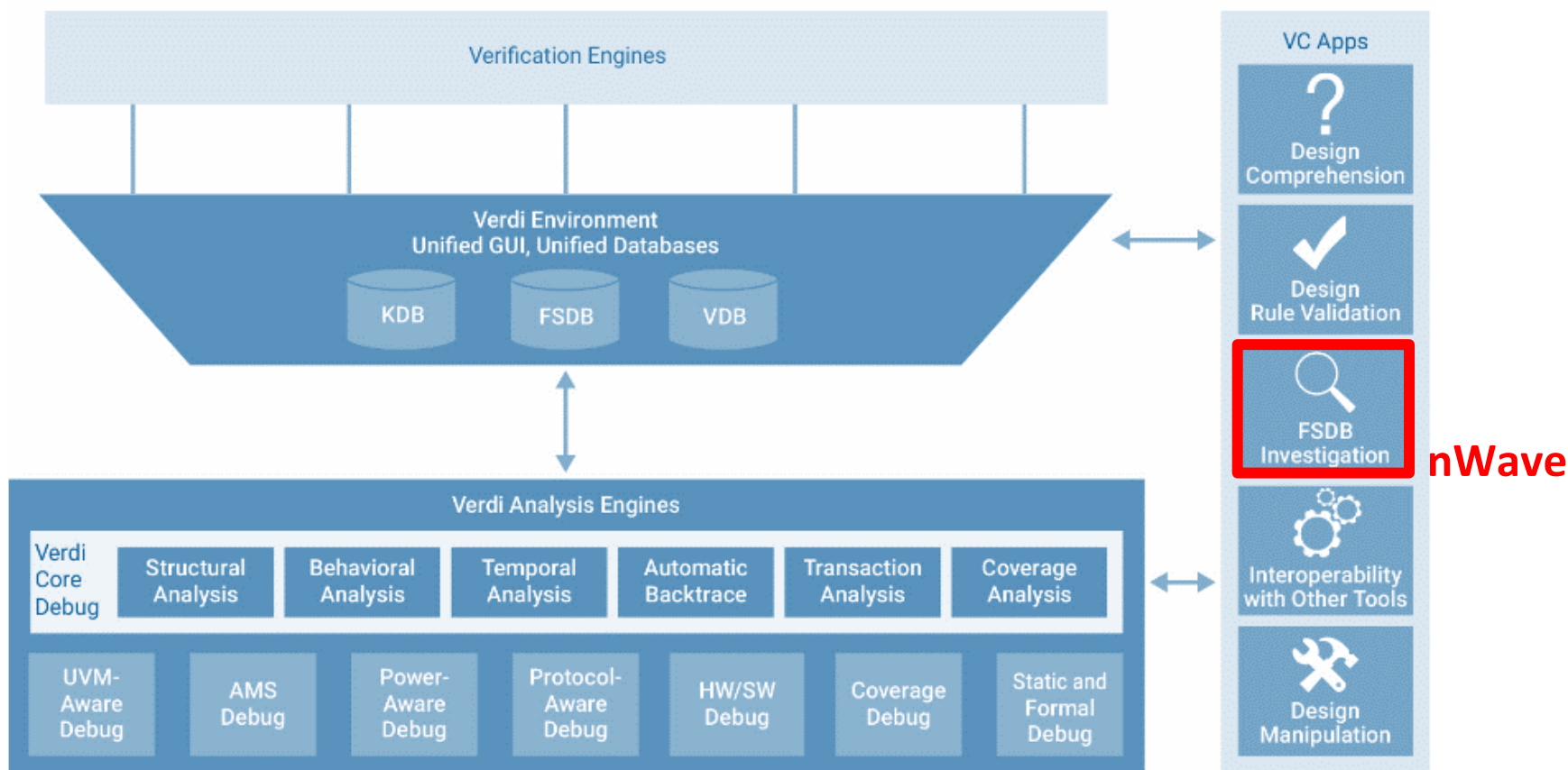


# Synopsys Verdi Debug System

- **Verdi is a debug system by Synopsys, featuring:**
  - **Waveform viewer (nWave)**
  - **Source code tracing (nTrace)**
  - Schematics and block diagrams
  - Visualizing state machines
  - Tracing of signal activity across clock cycles
  - ... and more



# Synopsys Verdi Debug System





# Waveform Formats

- **Value Change Dump (.vcd) format**
  - Indigenously supported by most simulators
  - Record all values at all timestamps, large file size
  - `$dumpfile("filename");`  
`$dumpvars();`
  
- **Fast Signal Database (.fsdb) format**
  - Defined by *Synopsys Verdi debug system*
  - Record value change events, smaller file size
  - `$fsdbDumpfile("filename");`  
`$fsdbDumpvars(0, test_module_name, "+mda");`



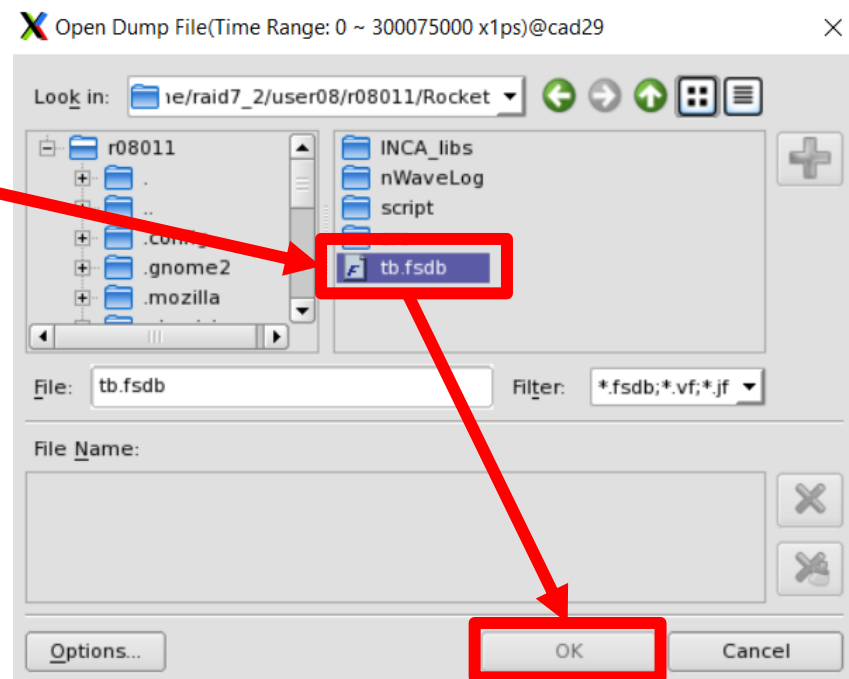
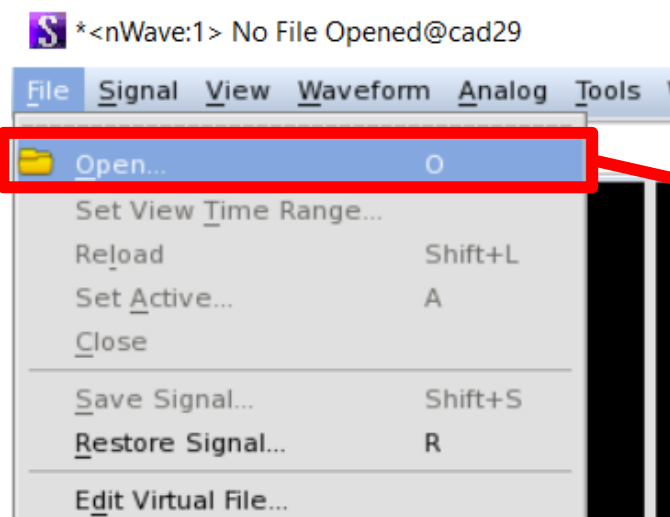
# nWave: Waveform Viewer

- A waveform analysis tool for .fsdb and .vcd waveform files
- Invoking nWave:
  - > `nWave &`
  - or > `nWave your_waveform.fsdb &`
- You can also run nWave along with Verdi GUI:
  - > `verdi your_waveform.fsdb &`



# nWave: Loading Waveform

- Loading waveform
  - Terminal: nWave waveform.fsdb &
  - GUI: "File" – "open"

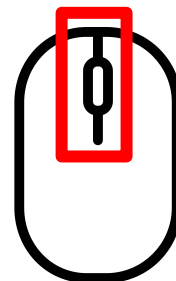




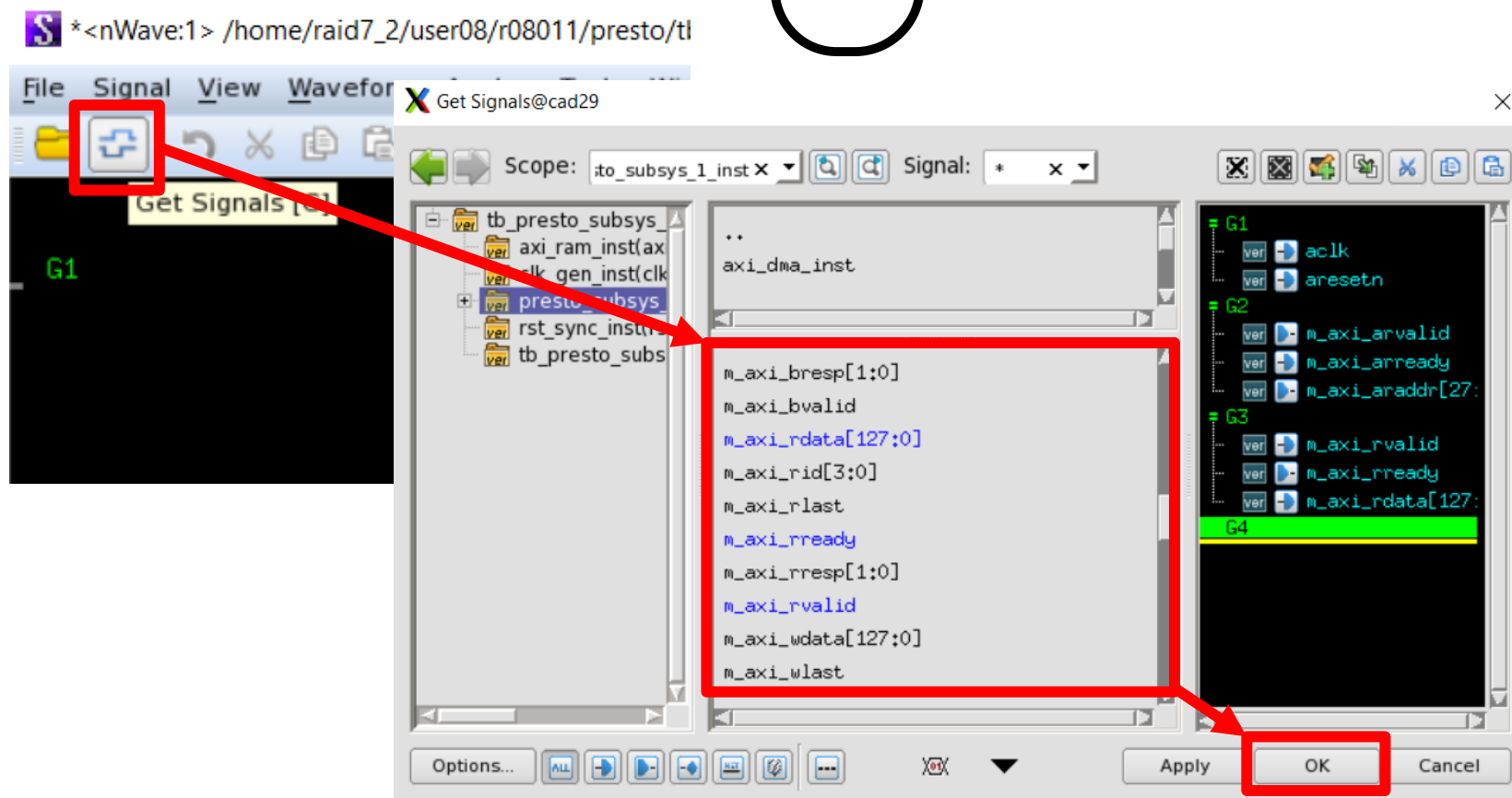


# nWave: Selecting Signals

- Press "Get Signals"
  - Shortcut 'G'



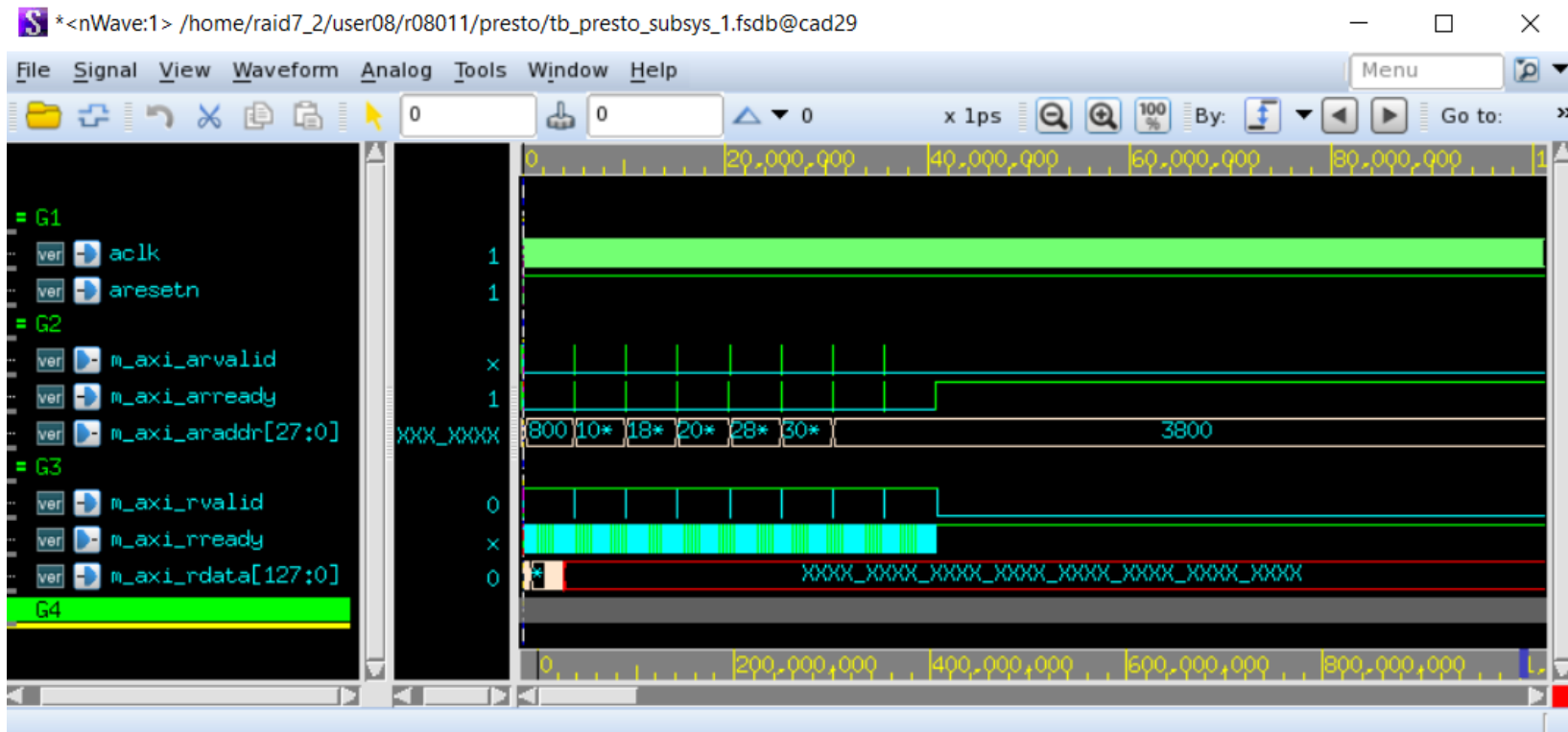
Use mouse middle button to reorder and group signals





# nWave: Viewing Waveforms

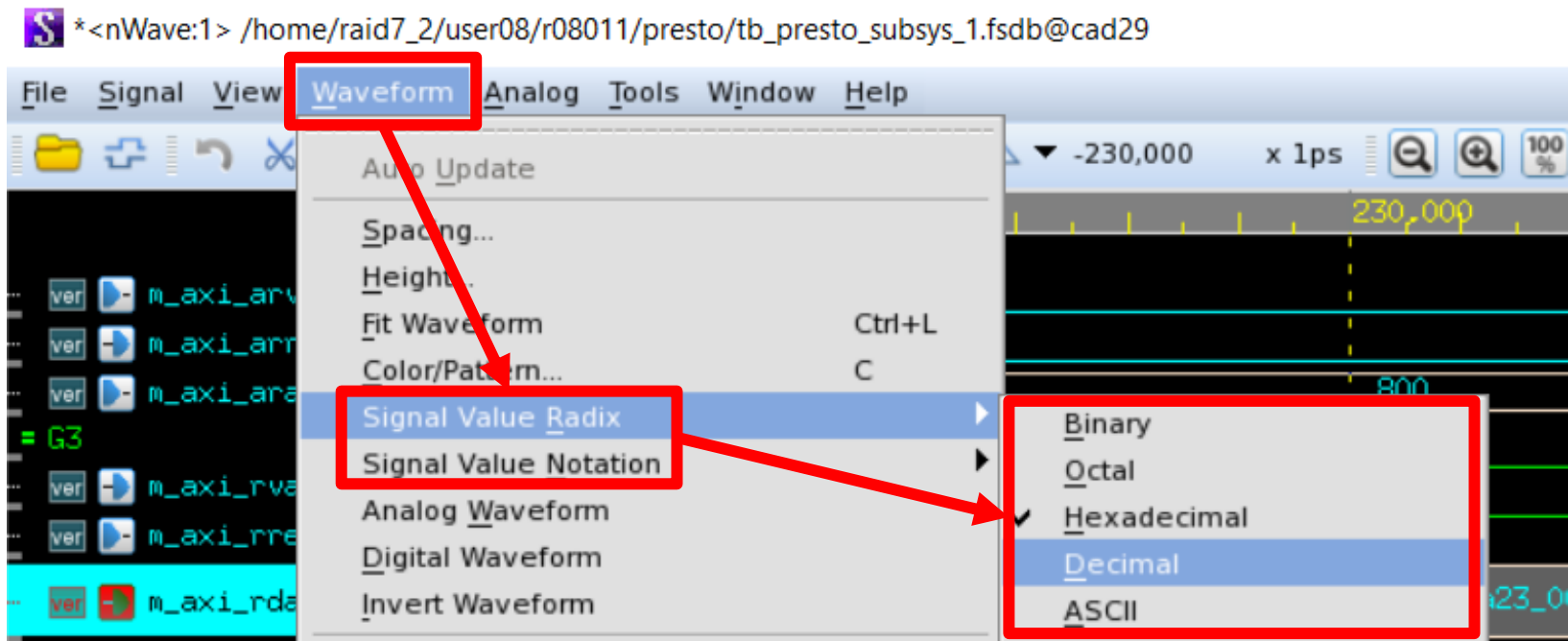
- **Shortcut 'h'**: toggle hierarchical naming
- **Shortcut 'y'**: center cursor
- You can reorder signals and rename groups in this view

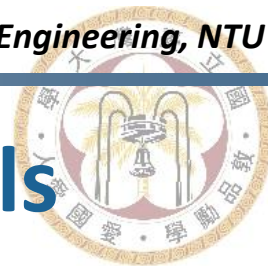




# nWave: Radix and Notation

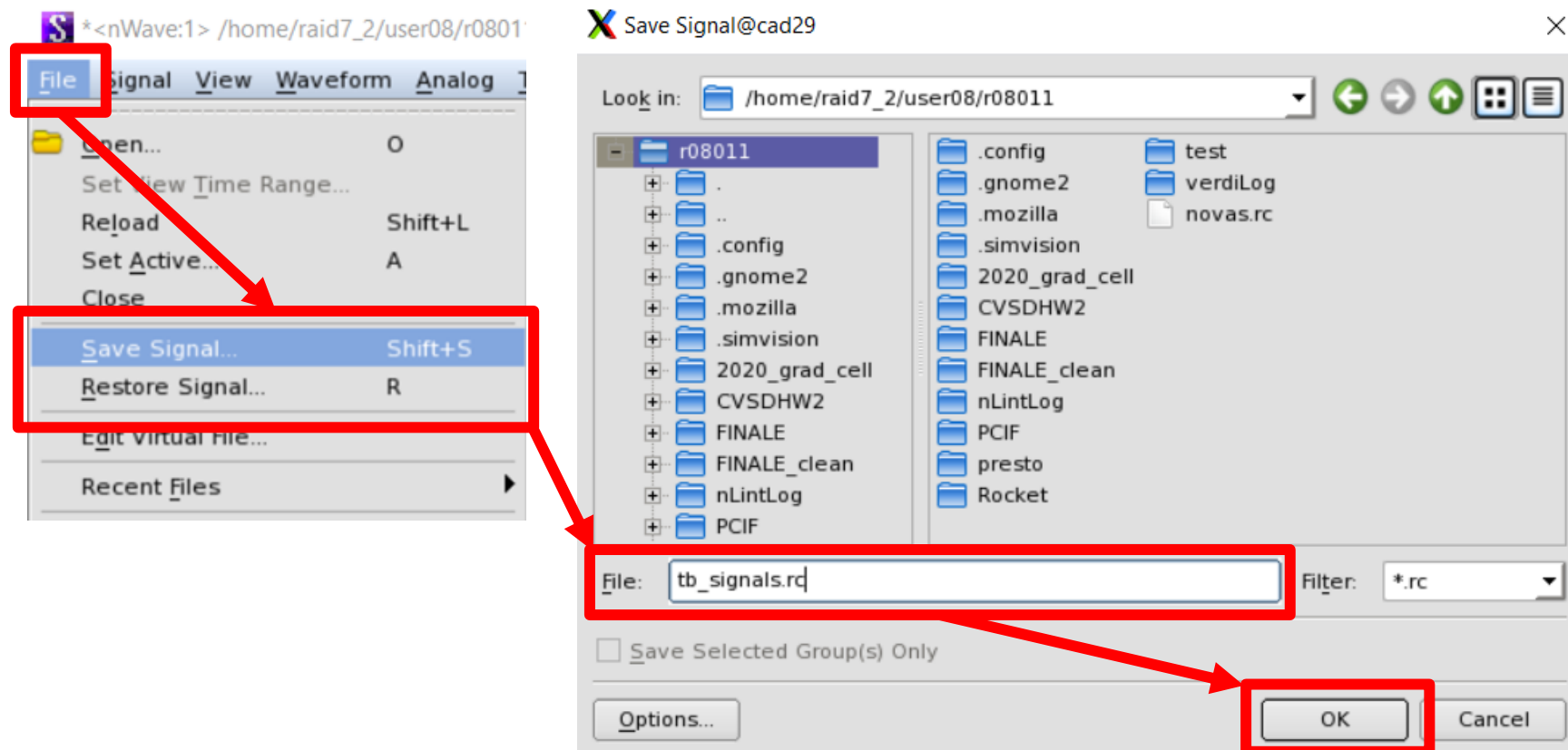
- **Setting Radix** (bin, oct, hex, dec)  
"Waveform" – "Signal Value Radix"
- **Setting Notation** (unsigned, signed 2's complement)  
"Waveform" – "Signal Value Notation"





# nWave: Saving and Restoring Signals

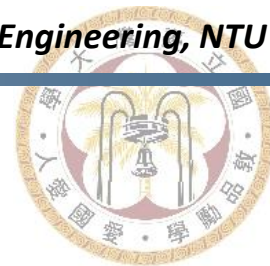
- After properly grouping and renaming signals, you can save the current view as a .rc file, which can be restored later





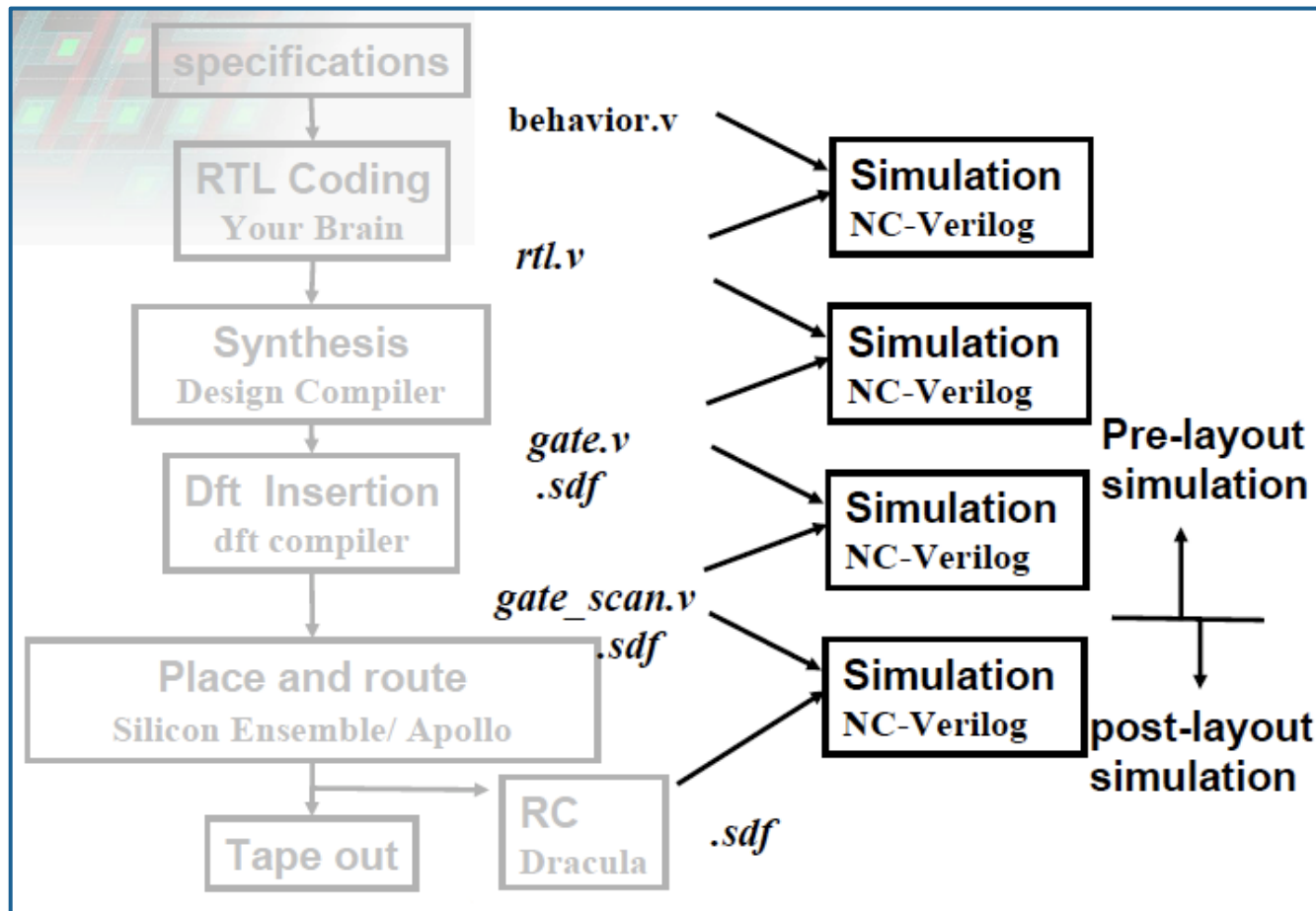
# Outline

- Debugging Tools
- **Testbench**
  - Simulation Overview
  - Instantiating DUT
  - Creating Clocks
  - Applying Stimulus
  - Verification
- Other Tips



# Simulation Overview

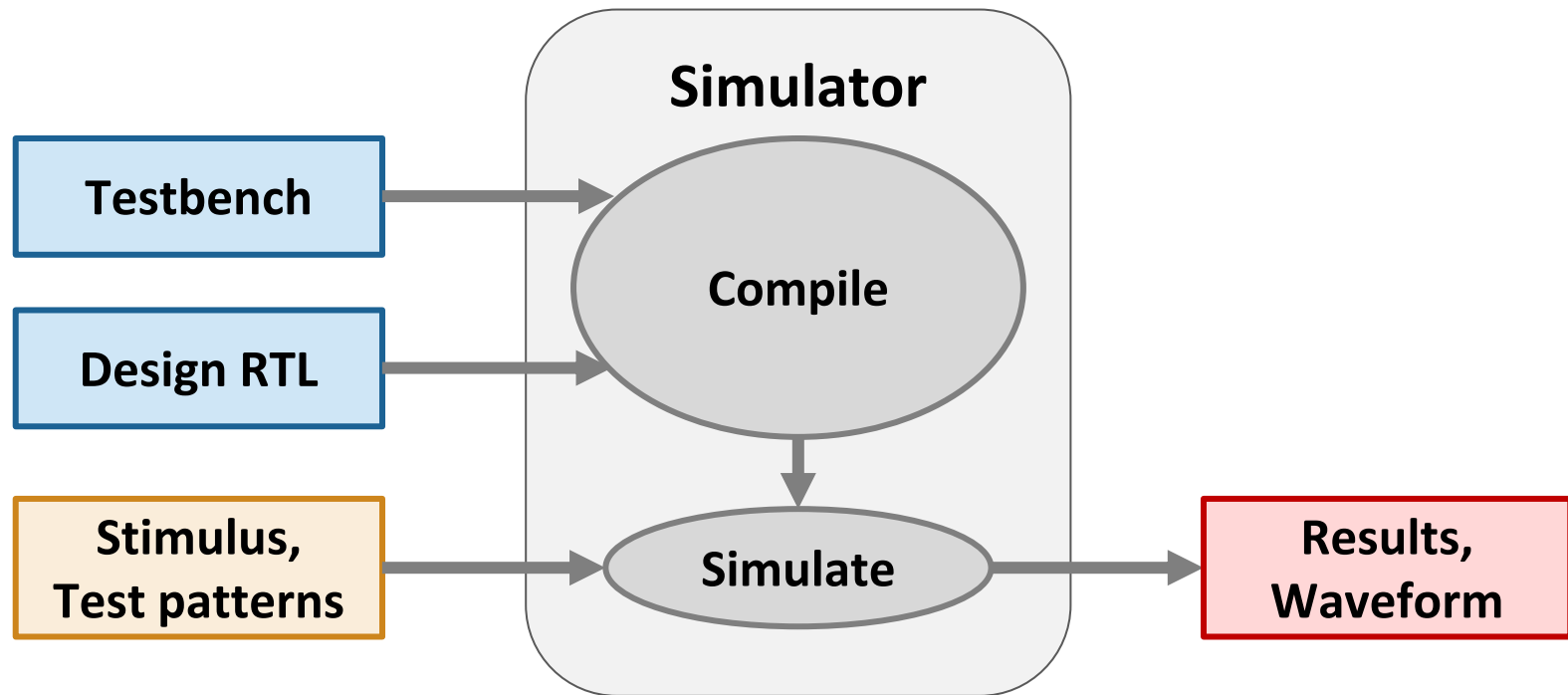
- Verification at each step





# Simulation Overview

- Simulation environment



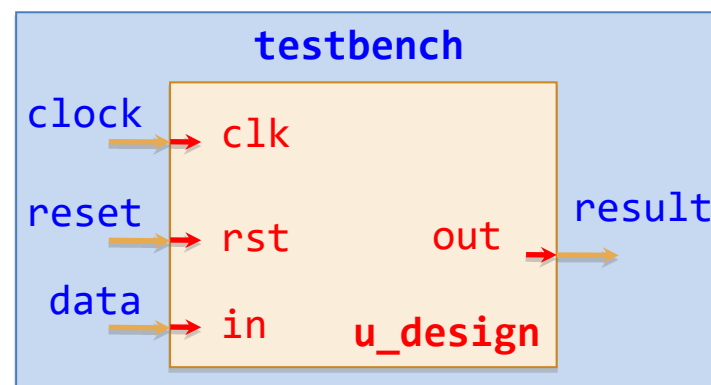


# Instantiating DUT

## ▪ Design Under Test (DUT)

- Instantiate the top module of the design in testbench

```
module testbench;  
  reg clock, reset, data;  
  wire result;  
  
  design u_design  
    .clk (clock ),  
    .rst (reset  ),  
    .in  (data   ),  
    .out (result )  
  );  
  
  ...  
endmodule
```



DUT Inputs: use **reg** to apply stimulus

DUT Outputs: use **wire** to capture signals

(In SystemVerilog, use **logic** for both)



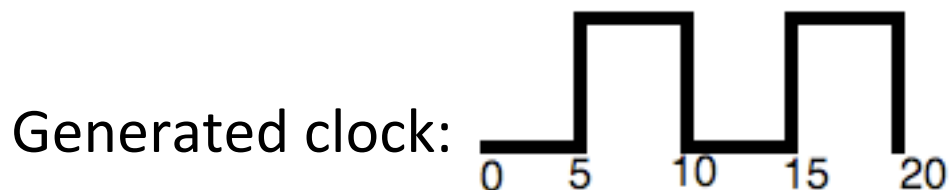


# Creating Clocks

- Creating clocks using always

```
`timescale 1ns/10ps
`define CYCLE 10.0
`define H_CYCLE 5.0

module tb;
    reg clock;
    initial clock = 0;
    always #(`H_CYCLE) clock = ~clock;
endmodule
```





# Creating Clocks

- Creating clocks using forever

```
`timescale 1ns/10ps
`define CYCLE 10.0
`define H_CYCLE 5.0

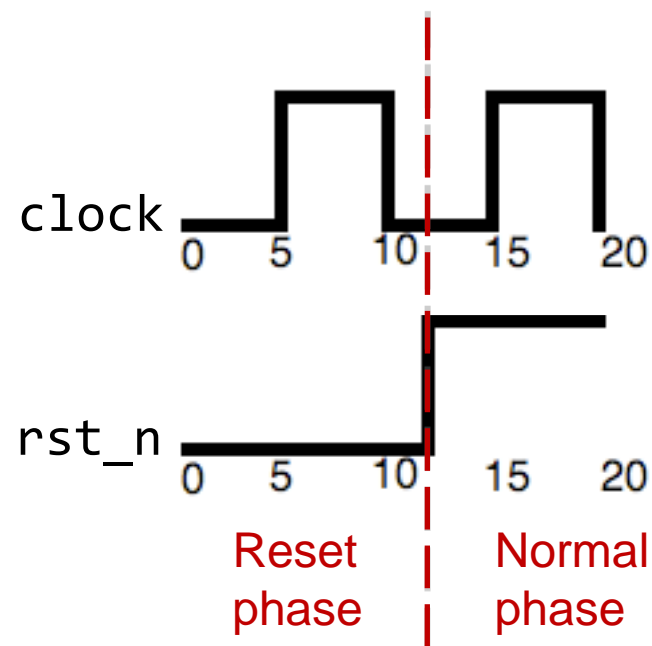
module tb;
    reg clock;
    initial begin
        clock = 0;
        forever #(`H_CYCLE) clock = ~clock;
    end
endmodule
```



# Applying Stimulus: Reset

- Generating the initialization reset signal

```
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: Timestamp

- **Manually assign values at each timestamp**
  - Can result in very long code, not scalable to larger patterns

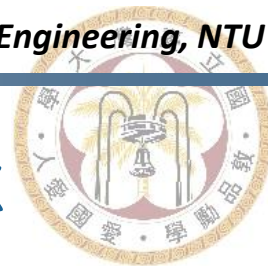
```
module inline_tb;
  wire [7:0] results;
  reg  [7:0] data_bus, addr;
  DUT u1 (results, data_bus, addr);
  initial fork
    #10 addr = 8'h01;
    #10 data_bus = 8'h23;
    #20 data_bus = 8'h45;
    #30 addr = 8'h67;
    #30 data_bus = 8'h89;
    #40 data_bus = 8'hAB;
    #45 $finish;
  join
endmodule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



# Applying Stimulus: Looping

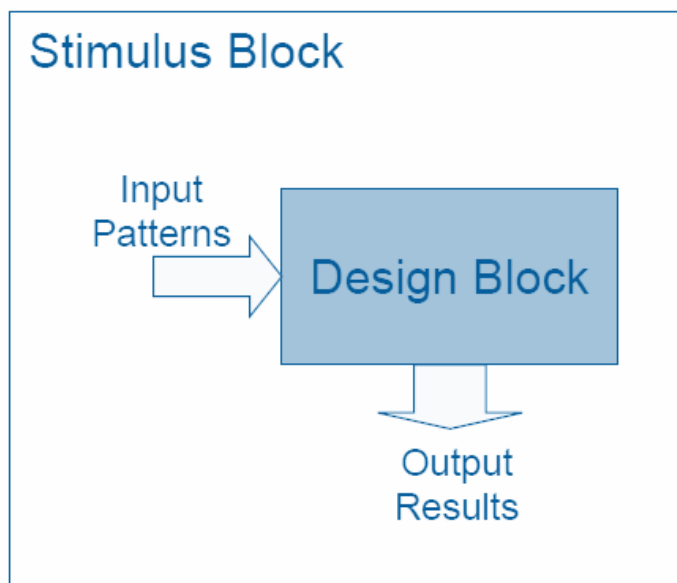
- Using for loops to apply data at given clock edge or condition
  - Compact testbench code
  - Combining other syntax for more flexible control, such as:
    - @(posedge clk), @(negedge clk)
    - #delay
    - if...else
    - wait

```
DUT i_DUT (.clk(clk), .rst(rst),  
           .rdy(rdy), data(data));  
initial begin  
    for (i = 0; i <= 255; i = i + 1) begin  
        wait (rdy == 1);  
        @(negedge clk)  
        data = i;  
    end  
end
```

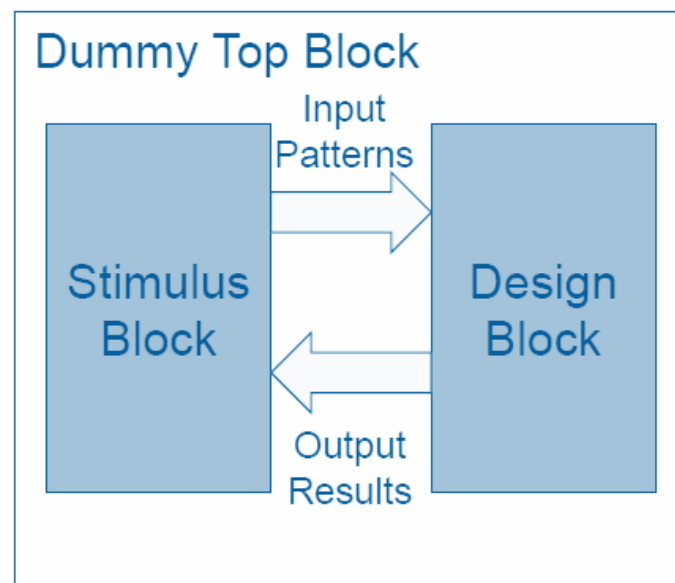


# Applying Stimulus: Stimulus Block

- We can implement a behavioral model as a stimulus block



The output results are verified by console/waveform viewer



The output results are verified by testbench or stimulus block



# Reading Test Pattern from Files

- Use Verilog built-in functions to load data into vector arrays

```
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
```



# Reading Test Pattern from Files

## 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

- Use **@address** to put data to different place
- **Address are always in hexadecimal format**

```
/* Data File */

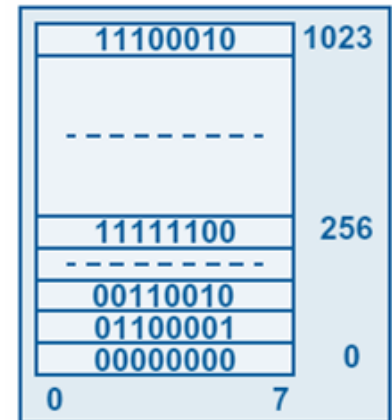
@0 // address always hex
0000_0000
0110_0001 0011_0010

// addresses 3-255 undefined

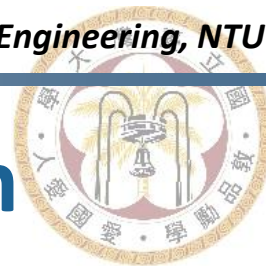
@100
1111_1100

// addresses 257-1022 undefined

@3FF
1110_0010
```







# Verify Output with Golden Pattern

- Using pre-calculated golden pattern to check output correctness

```
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
```



## Pitfall: Implicitly Comparing to X

- **Always use `===` and `!==` in a testbench for equivalence check**
  - In this example, if `output_data` is always x, error count `err_cnt` will still be 0:

```
err_cnt = 0;
for (i = 0; i < 256; i = i + 1) begin
    if (output_data[i] != output_golden[i]) begin
        err_cnt = err_cnt + 1;
    end
end

if (err_cnt == 0) begin
    $display("PASS!!! All output pattern correct.");
end
```



# 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



# Syntax for Text Monitoring

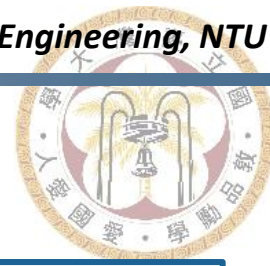
- Verilog format string syntax

Format Specifiers:

<code>%b</code>	<code>%c</code>	<code>%d</code>	<code>%h</code>	<code>%m</code>	<code>%o</code>	<code>%s</code>	<code>%t</code>	<code>%v</code>
binary	ASCII	decimal	hex	module	octal	string	time	strength

Escaped Literals:

<code>\"</code>	<code>\&lt;1-3 digit octal number&gt;</code>	<code>\\</code>	<code>\n</code>	<code>\t</code>
double quote	ASCII representation of number	backslash	newline	tab



## Example: Monitoring

```

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

```

### Results

time	realtime	stime	in1	o1
0.00ns	0.00ns	0.00ns	0	x
10.00ns	9.53ns	10.00ns	0	1
10.00ns	10.00ns	10.00ns	1	1
20.00ns	19.53ns	20.00ns	1	0



# Timing Checks

- **specify Block**
  - Use **specify** and **endspecify** for declaring timing checks
  - Separates module timing from its functionality
- **Checking Setup/Hold-Time Violation**
  - `$setup(FF_data, clock_event, su_limit, notifier)`
  - `$hold(clock_event, FF_data, h_limit, notifier)`

**Note:** Gate-level simulation(Post-sim):

1. Correct timing information from **sdf files** (from dc static timing analysis)
2. If there is no available sdf file, read default timing from **tsmc13.v**
  - May cause hold time violation
3. **Timing checks are written in tsmc13.v flip-flop modules**



## Example: Timing Checks

```
reg flag1, flag2; // notifier should be one-bit reg

specify
    $setup(data, posedge CLK &&& RESET, (`SETUPTIME), flag1);
    $hold(posedge CLK &&& RESET, data, (`HOLDTIME), flag2);
endspecify

always @(flag1)
    // avoid unknown (X) toggling of notifier
    if(flag1 == 1'b1 || flag1 == 1'b0)
        s_violation = s_violation +1; // +1 when flag1 is toggled

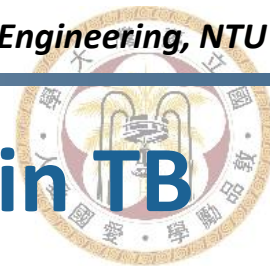
always @(flag2)
    if(flag2 == 1'b1 || flag2 == 1'b0)
        h_violation = h_violation +1;
```



# Outline

- Debugging Tools
- Testbench
  - Simulation Overview
  - Instantiating DUT
  - Creating Clocks
  - Applying Stimulus
  - Verification
- **Other Tips**



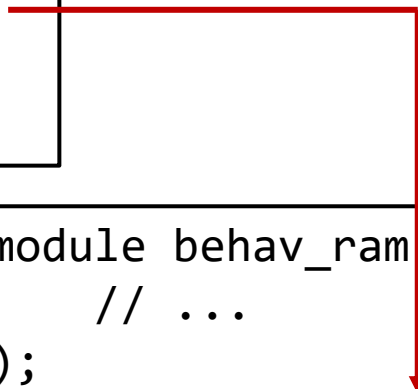


# Accessing Modules from Lower Levels in TB

- Use . to access members of lower level

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

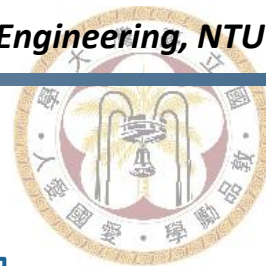
```
module behav_ram(  
    // ...  
);  
    reg [31:0] mem [0:32767];  
endmodule
```





# Accelerating Simulation

- **Parameters of \$fsdbDumpvars** could slow down simulation and increase file size
  - \$fsdbDumpvars(**depth**, instance, options);
    - **depth=0**: all signals in all scopes
    - **depth=n**: all signals in current scope and n-1 levels below
- **To speed up simulation:**
  - Set **depth** to positive numbers
  - Set **instance** to the module you want to observe
  - **Do not use "+mda"** when there are large vector arrays



# Modularized Clock Generator

```
module clk_gen # (  
    parameter CYCLE      = 10.0,  
    parameter MAX_CYCLE  = 10000,  
    parameter RST_DELAY  = (5 * CYCLE_TIME)  
)  
{  
    output logic clk,  
    output logic rst_n  
};  
    localparam H_CYCLE = (CYCLE / 2.0);  
    initial clk = 0;  
Clock → always # (H_CYCLE) clk = ~clk;  
  
    initial begin  
        rst_n = 1; # (0.25 * CYCLE);  
Reset → rst_n = 0; # (RST_DELAY - 0.25 * CYCLE);  
        rst_n = 1;  
    end  
  
    initial begin  
        # (MAX_CYCLE * CYCLE);  
Finish → $finish;  
    end  
endmodule
```



## Indexed Part Select Syntax

- 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 sixteen 8-bit numbers

```
for (i = 0; i < 16; i = i + 1)
    data_byte[i] <= data_chunk[(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



# Indexed Part Select Syntax

- **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]
```