Digital System Design

Debugging Tool and Testbench Writing

Lecturer: 王景平

Advisor: Prof. An-Yeu Wu

Date: 2024.03.20

Function and Task

Function	Task
Can call **other functions**	Can call **other functions and tasks**
No delay(combinational circuit)	**Allow delay** statement(#, @)
At least one input	**Any number** of inputs
Exact one output(function itself)	**Any number** of outputs
Called as a **RHS** value	Called as a **subroutine**
Synthesizable	**Synthesizable**(with valid content)
Allow local variables(registers/integers)	
Allow **named arguments**/ordered arguments	

Sensitivity List for Comb. Ckt

- Easy way to use sensitivity list for combinational circuit
  - Use **always @(*)**

```
always @(a or b or x) begin
    y = ~x;
    x = a | b;
end
```

```
always @(*) begin
    y = ~x;
    x = a | b;
end
```

Pitfall in Sensitivity List

```
always @(a or b or x) begin
    y = ~x;

    x = a | b;

end
```

```
// a = 0, b = 0, x = 0, y = 1
1. b changes to 1
2. y remains 1
4. y changes to 0
3. x changes to 1,
   trigger again
5. x remains 1
```

- Explicit sensitivity list **(a or b or x)**
  - **4, 5 won't happen**
  - **Order of statements** affect the simulation result
- Wildcard sensitivity list **(*)**
  - **4, 5 will happen**
  - Order of statements won't cause latch behavior
- Synthesis tool always takes (*)

Modeling of Flip-Flops

- The use of **posedge** and **negedge** makes an **always** block sequential (edge-triggered)
- Unlike combinational always block, the sensitivity list does determine the behavior of synthesis

D Flip-flop with synchronous clear

```
module dff_sync_clear(d, clearb,
clock, q);
input d, clearb, clock;
output q;
reg q;
always @ (posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```

`always` block entered only at each positive clock edge

D Flip-flop with asynchronous clear

```
module dff_async_clear(d, clearb, clock, q);
input d, clearb, clock;
output q;
reg q;

always @ (negedge clearb or posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```

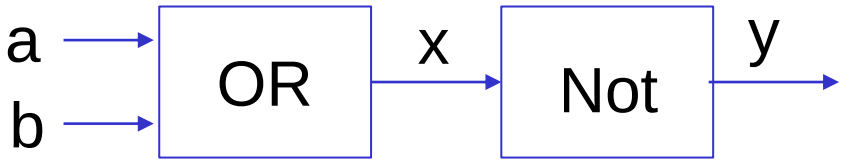
`always` block entered immediately when (active-low) clearb is asserted

Note: The following is **incorrect** syntax: `always @ (clear or negedge clock)`

If one signal in the sensitivity list uses posedge/negedge, then all signals must.

Combination & Sequential

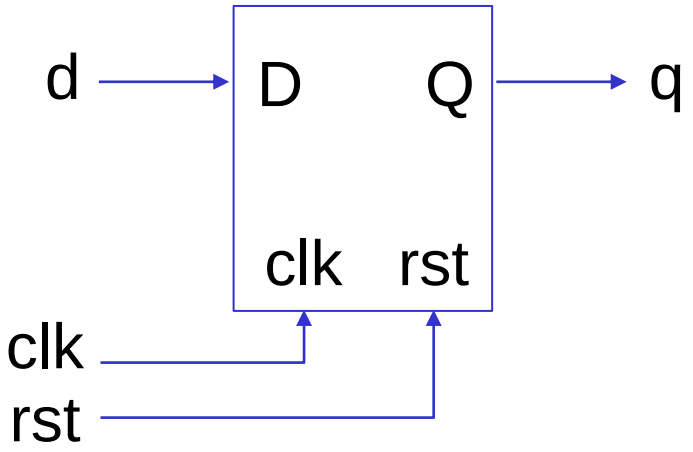
- Combinational Ckt.




```
reg x;
reg y;
always @(*) begin
    y = ~x;
    x = a | b;
end
```

Blocking assignment

- Sequential Ckt.




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

Non-Blocking assignment

Conditional Statements

- If and If-else statements

```
if (expression)
  statement
else
  statement
```

```
if (expression)
  statement
else if (expression)
  statement
else
  statement
```

```
always@(*) begin
  nxt_a = a;
  nxt_b = b;
  if (sel)
    nxt_a = data;
  else
    nxt_b = data;
end
```

- Restrictions compared with C
  - LHS in all cases should be the same!
    - **Avoid latch**
  - Conditions should be full-case, if must be followed by else!
    - **Avoid latch**
  - In short, think about MUX!

```
always@(*) begin
  if (sel)
    nxt_a = data;
  else
    nxt_b = data;
end
```

```
always@(*) begin
  if (sel)
    nxt_a = data;
end
```

Multiway Branching (Case)

- The nested ***if-else-if*** can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the ***case*** statement

```
case (expression)
  alternative1: statement1;
  alternative2: statement2;
  alternative3: statement3;
      ...
  default: default_statement;
endcase
```

if-else statement

```
if (sel==3)
  y = d;
else
  if (sel==2)
    y = c;
  else
    if (sel==1)
      y = b;
    else
      if (sel==0)
        y = a;
```

case statement

y = a;

```
case (sel)
      3: y = d;
      2: y = c;
      1: y = b;
default y = a;
endcase
```

```
always@(*)
  y= (sel==3) ? d :
     (sel==2) ? c :
     (sel==1) ? b : a;
```

Coding Overview

- Initialization
  - Input / output definition, wire / reg declaration
- Finite state machine (FSM)
  - State define (using localparameter)
  - Separating CS, NL, OL
- Combinational circuit
- Sequential circuit (only declare DFF !!!)

```
/*=======================================
Author: Yu Chuan, Chuang
Module: Counter
Description:
When getting start_i signal, counter starts
to count from 0 to 15.
=======================================*/
module counter (
    input           clk,
    input           rst,
    input           start_i,
    output  [3:0]   count_o
);

//============  Parameter ===============
    localparam  STATE_IDLE  = 1'b0;
    localparam  STATE_CNT   = 1'b1;

//=========  Reg/Wire Declaration ======
    reg             state, nxt_state;
    reg [3:0]       cnt, nxt_cnt;

//========== Finite State Machine =======
    always@(posedge clk or posedge rst) begin
        if(rst)
            state <= STATE_IDLE;
        else
            state <= nxt_state;
    end

    always@(*) begin
        case(state)
            STATE_IDLE: begin
                if(start_i)
                    nxt_state = STATE_CNT;
                else
                    nxt_state = STATE_IDLE;
            end
            STATE_CNT: begin
                if(cnt == 4'd15)
                    nxt_state = STATE_IDLE;
                else
                    nxt_state = STATE_CNT;
            end
            default: nxt_state = STATE_IDLE;
        endcase
    end

//============ Combinational ===========
    assign count_o = cnt;

    always@(*) begin
        if(state == STATE_CNT) begin
            nxt_cnt = cnt + 1;
        end
        else begin
            nxt_cnt = 0;
        end
    end

//============= Sequential ============
    always@(posedge clk or posedge rst) begin
        if(rst)
            cnt <= 0;
        else
            cnt <= nxt_cnt;
    end

endmodule
```

FSM

CS

NL

OL

```
1  /*=======================================
2  Author: Yu Chuan, Chuang
3  Module: Counter
4  Description:
5  When getting start_i signal, counter starts
6  to count from 0 to 15.
7  =======================================*/
8  module counter (
9      input            clk,
10     input            rst,
11     input            start_i,
12     output  [3:0]    count_o
13 );
14
15 //============  Parameter ===============
16     localparam  STATE_IDLE  = 1'b0;
17     localparam  STATE_CNT   = 1'b1;
18
19 //=========  Reg/Wire Declaration ======
20     reg         state, nxt_state;
21     reg [3:0]   cnt, nxt_cnt;
22
23 //========= Finite State Machine =======
24     always@(posedge clk or posedge rst) begin
25         if(rst)
26             state <= STATE_IDLE;
27         else
28             state <= nxt_state;
29     end
30
31     always@(*) begin
32         case(state)
33             STATE_IDLE: begin
34                 if(start_i)
35                     nxt_state = STATE_CNT;
36                 else
37                     nxt_state = STATE_IDLE;
38             end
39             STATE_CNT: begin
40                 if(cnt == 4'd15)
41                     nxt_state = STATE_IDLE;
42                 else
43                     nxt_state = STATE_CNT;
44             end
45             default: nxt_state = STATE_IDLE;
46         endcase
47     end
48 //============ Combinational ============
49     assign count_o = cnt;
50
51     always@(*) begin
52         if(state == STATE_CNT) begin
53             nxt_cnt = cnt + 1;
54         end
55         else begin
56             nxt_cnt = 0;
57         end
58     end
59
60 //============== Sequential =============
61     always@(posedge clk or posedge rst) begin
62         if(rst)
63             cnt <= 0;
64         else
65             cnt <= nxt_cnt;
66     end
67
68
69 endmodule
```

Header/Comment

Module instantiation

Input/output declaration

Parameter

Number Representation

Reg/WIre

If statement

FSM

Procedure Block

case statement

Combinational

Continuous Assignment

Sensitivity list

Operator

Procedure Assignment

Sequential

Outline

- Introduction to Simulation: NC-Verilog, VCS
- Introduction to Debugging Tool
  - nLint: HDL Coding Checking
  - nWave: Waveform Tracing
- Testbench Writing
  - Overview of Simulation
  - Instantiating DUT
  - Creating Clocks
  - Applying Stimulus
  - Verification
- Timing Parameter in Gate-level Simulation
  - Setup and Hold time

Introduction to NC-Verilog and VCS

- The Cadence® NC-Verilog® and Synopsys® VCS® are a Verilog digital logic simulator.
- We can use these tools to
  - Compiles the Verilog source files
  - Analyzes and Elaborates the design
  - Simulates the design

Running NC-Verilog (1/2)

- Run the Verilog simulation:

```
ncverilog testbench.v exp2.rsa.v +access+r
```

- Another choice of running Verilog simulation:

```
ncverilog -f exp2_rsa.f +access+r
```

In exp2_rsa.f

```
testbench.v
exp2_rsa.v
~
~
```

Running NC-Verilog (2/2)

- "+access+r" is added to enable waveform file dumping.

In testbench.v,

```
initial begin
    $fsdbDumpfile("exp2_rsa.fsdb");
    $fsdbDumpvars;
end
```

Or

```
initial begin
    $dumpfile("exp2_rsa.vcd");
    $dumpvars;
end
```

- *.fsdb has smaller file size than *.vcd.

Running VCS

- Run the Verilog simulation

```
vcs testbench.v design.v –full64 –R
–debug_access+all +v2k
```

  - +v2k: enable to read Verilog-2001
  - -full64: compiles and simulates on 64-bit machine

Outline

- Introduction to Simulation: NC-Verilog
- Introduction to Debugging Tool
  - nLint: HDL Coding Checking
  - nWave: Waveform Tracing
- Testbench Writing
  - Overview of Simulation
  - Instantiating DUT
  - Creating Clocks
  - Applying Stimulus
  - Verification
- Timing Parameter in Gate-level Simulation
  - Setup and Hold time

Introduction to nLint

- A design rule checker that can help hardware designers to create syntax and semantics correct HDL code (Developed by SpringSoft).

- 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
```

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
```

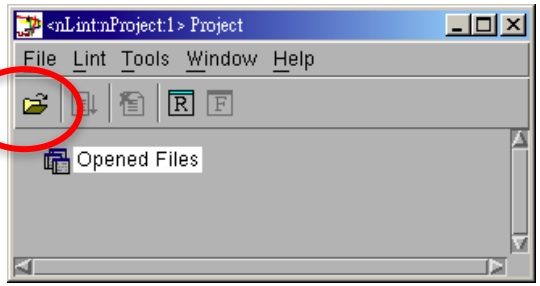
Incomplete conditional assignment ①

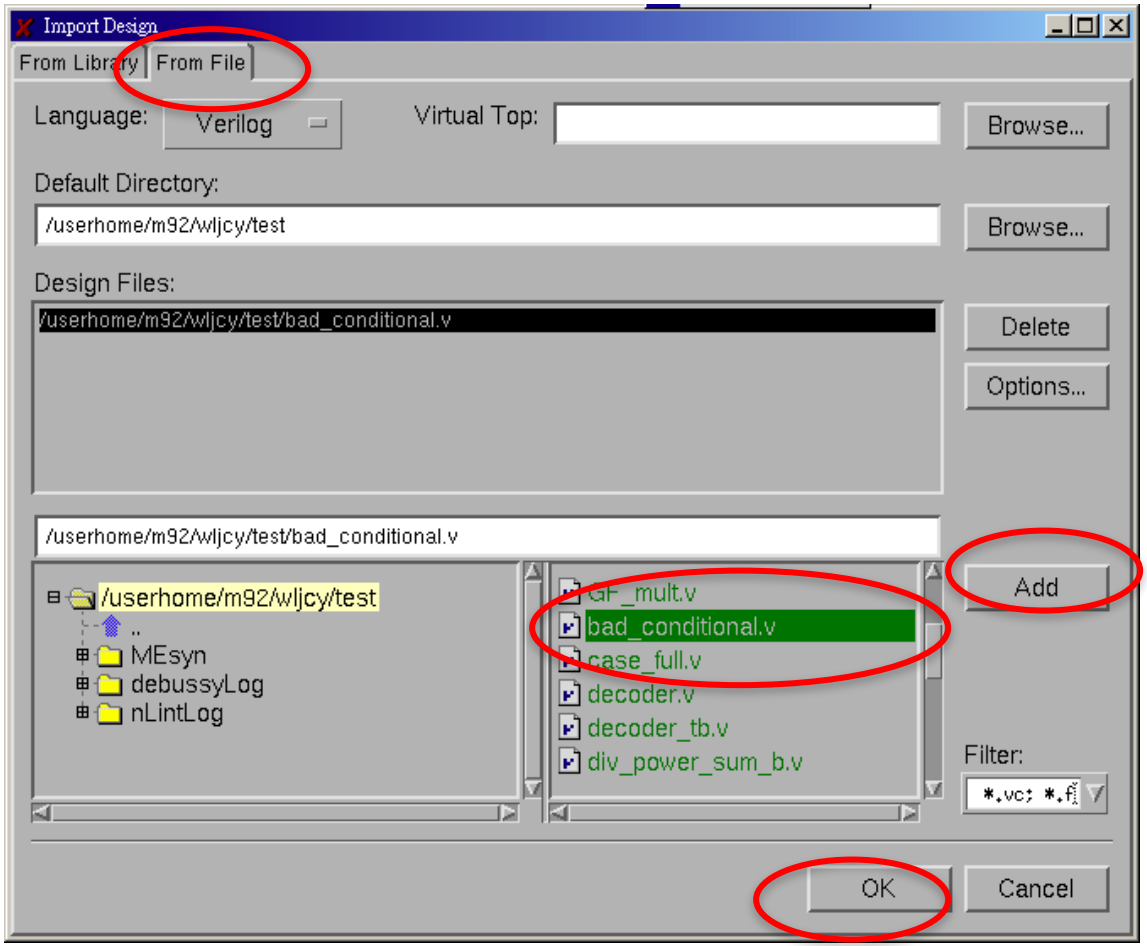
Incomplete sensitivity list ②

Error!! need " ; " ③

Invoke nLint

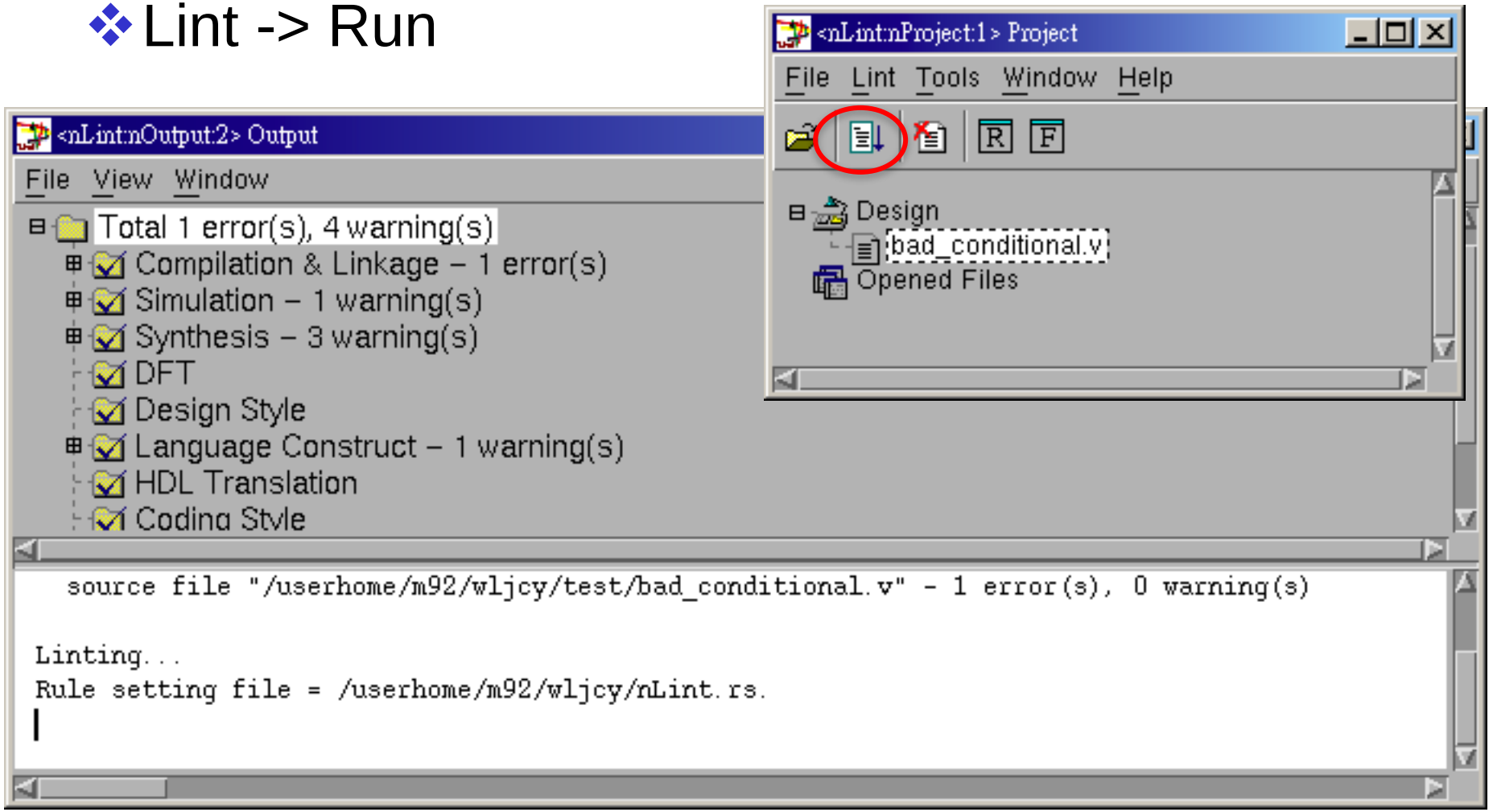
- > *nLint -gui &*
- Open verilog source file





Run nLint

- Lint -> Run



Fix Errors & Warnings

nWave

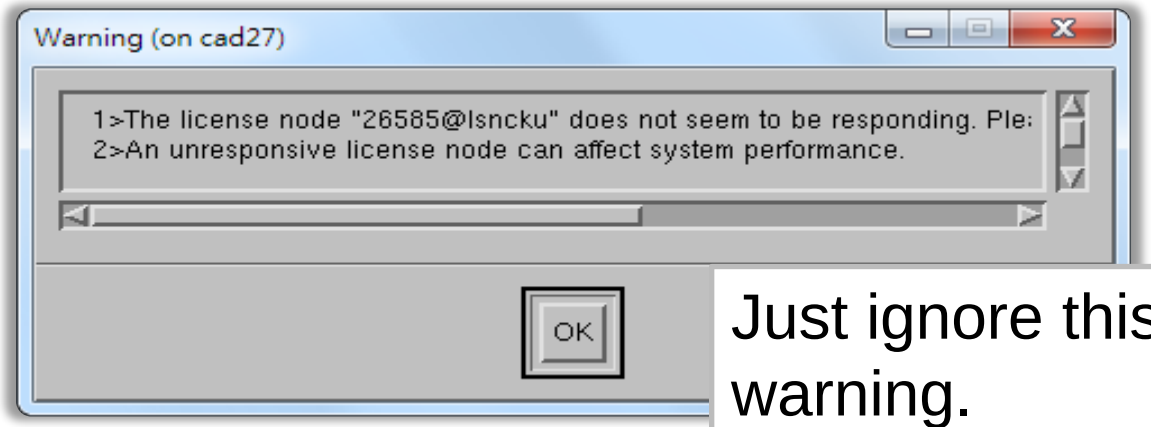
- A waveform analysis tool for viewing **.fsdb* & **.vcd* waveform files
- We can debug easily by checking the waveform file dumped during simulation.
- Invoke nWave:
  - > *nWave &*
- Open waveform file
  - Similar GUI interface with nLint

Start nWave

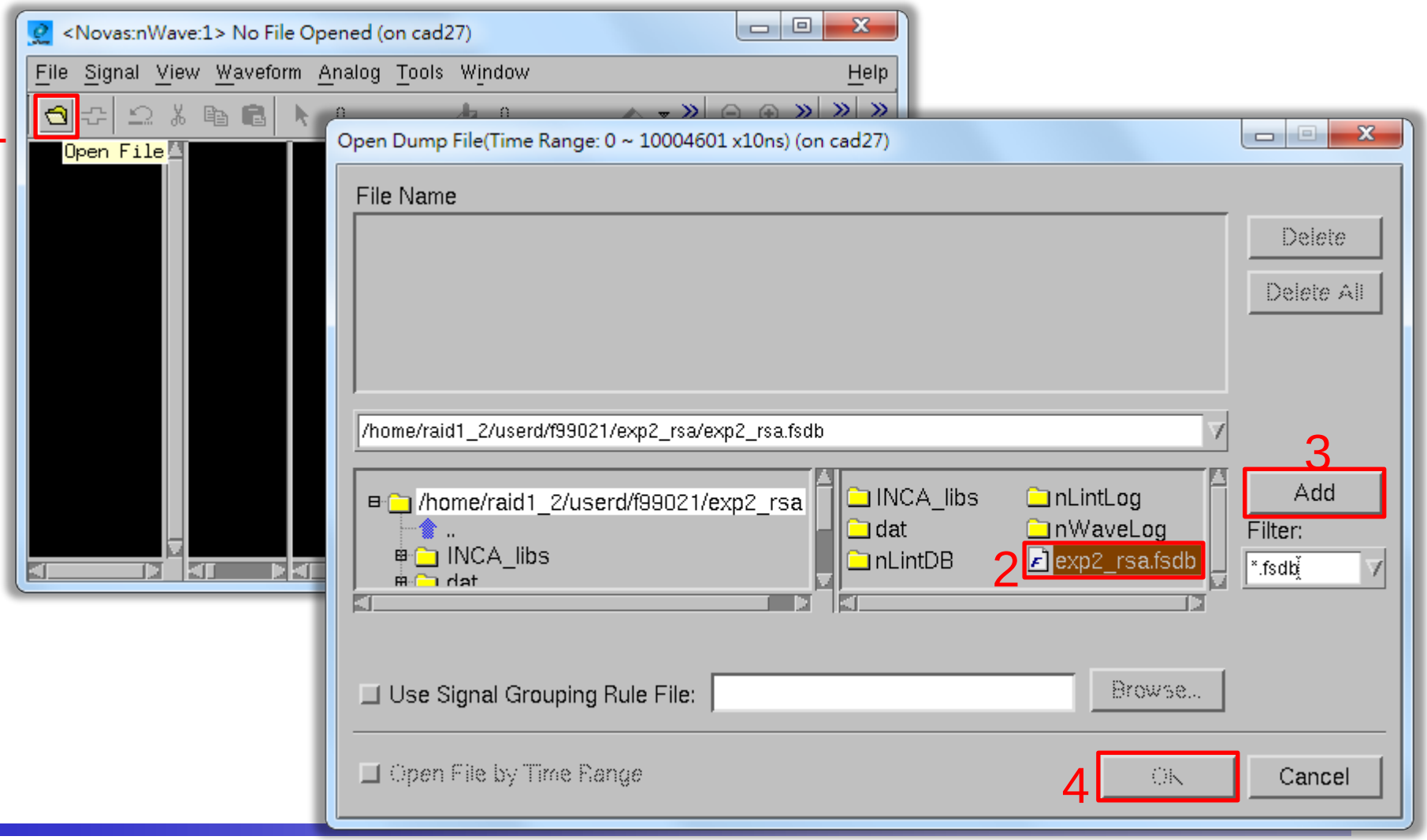
- Type the following command:

```
nWave &
```

  - Also, the token "&" enable you to use the terminal while Verdi is running in the background.

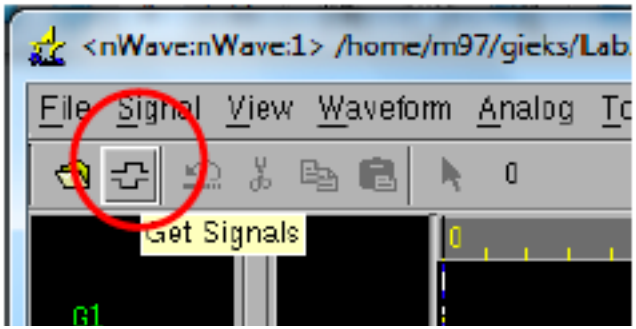


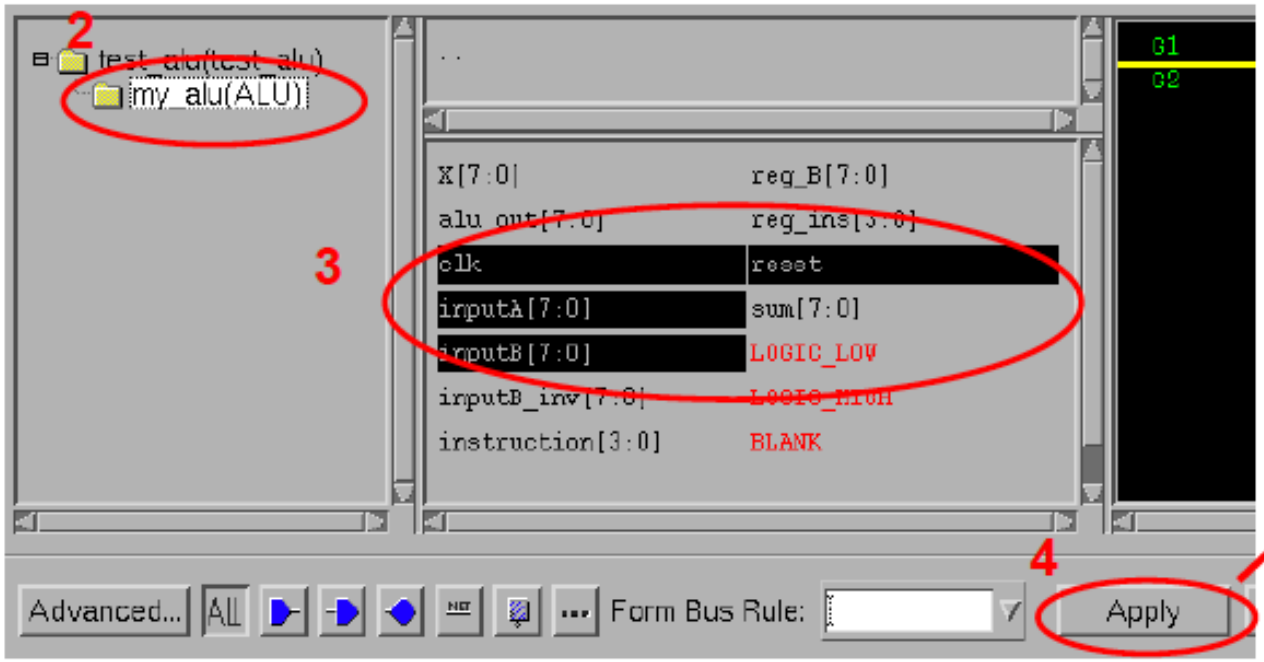
Open the FSDB File

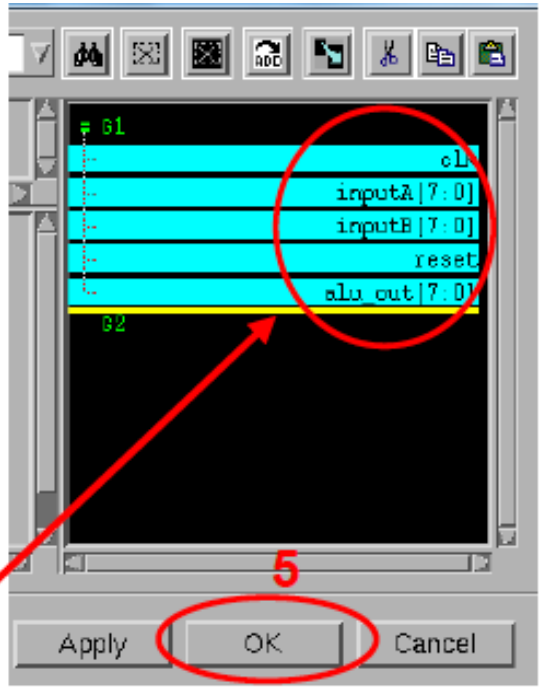


Ref: http://media.ee.ntu.edu.tw/personal/pcwu/dclab/dclab_02.pdf

Select Signals to View

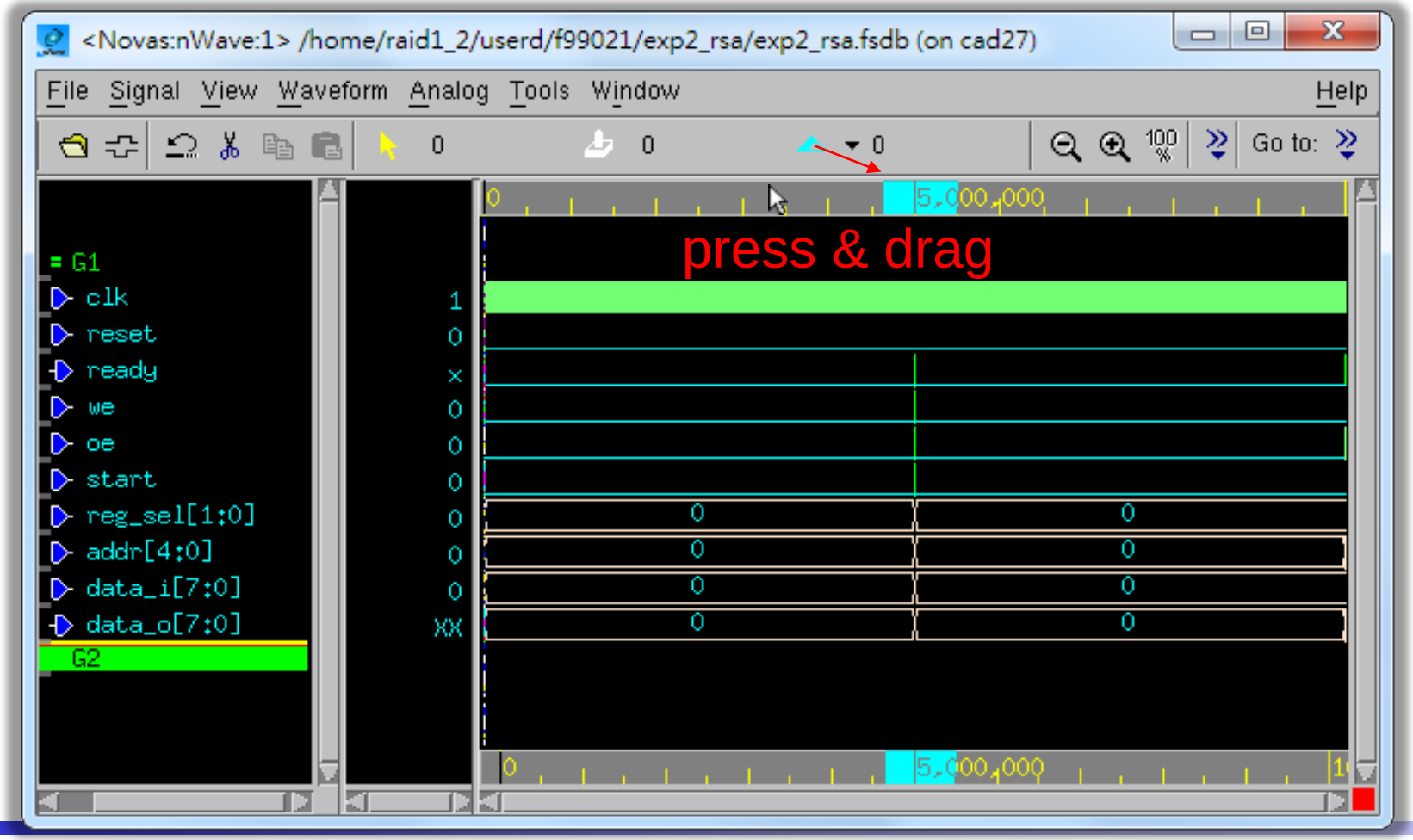


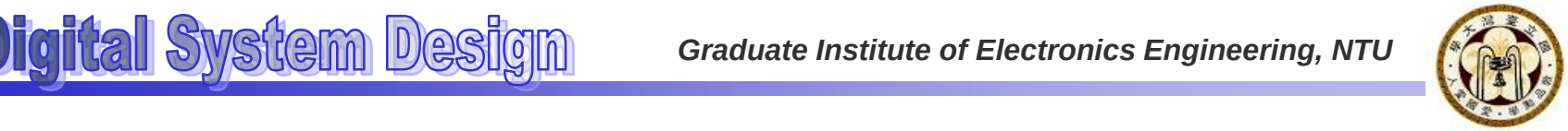




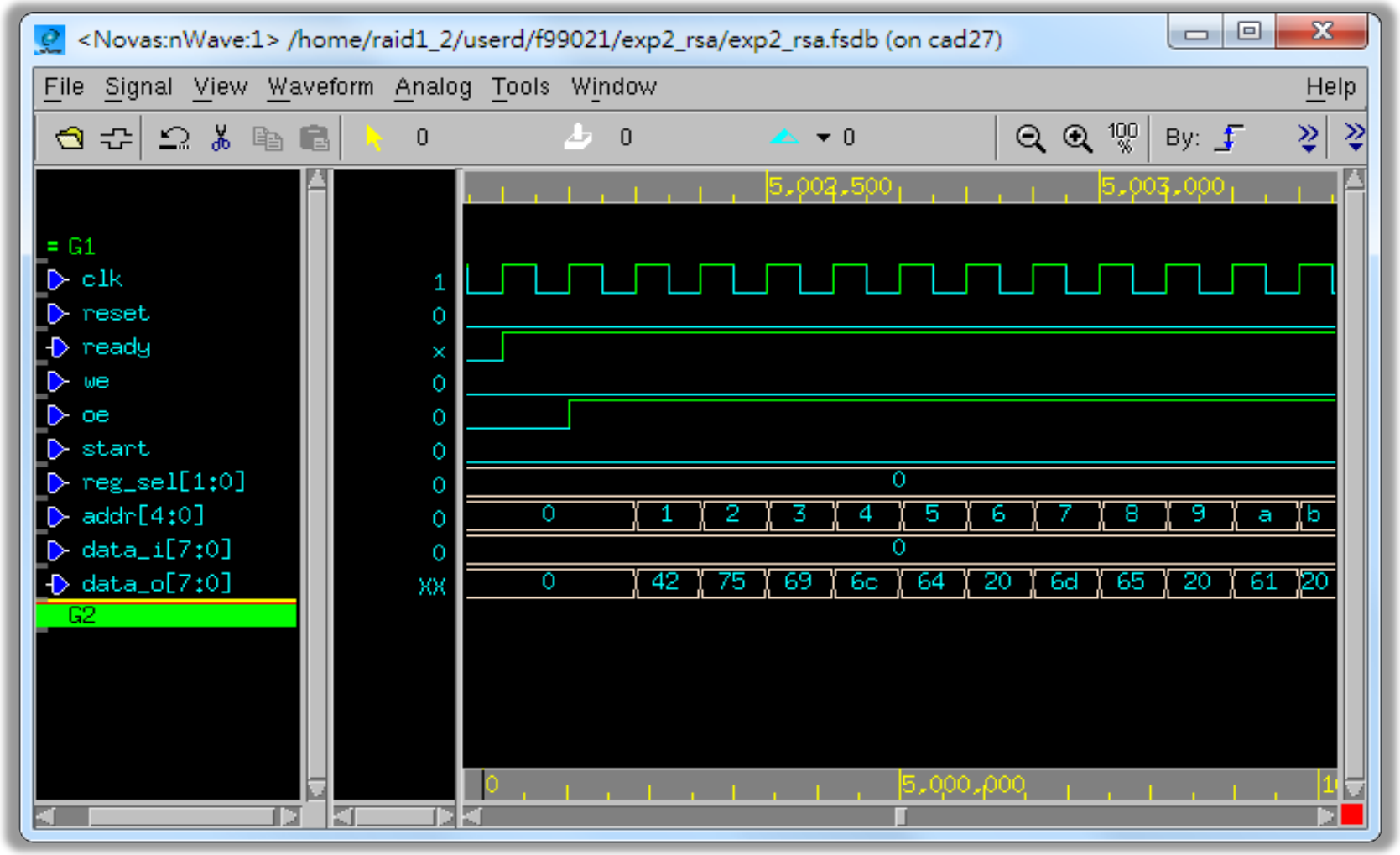
Browse the Specified Interval (1/2)

- Press “F” to see whole waveform

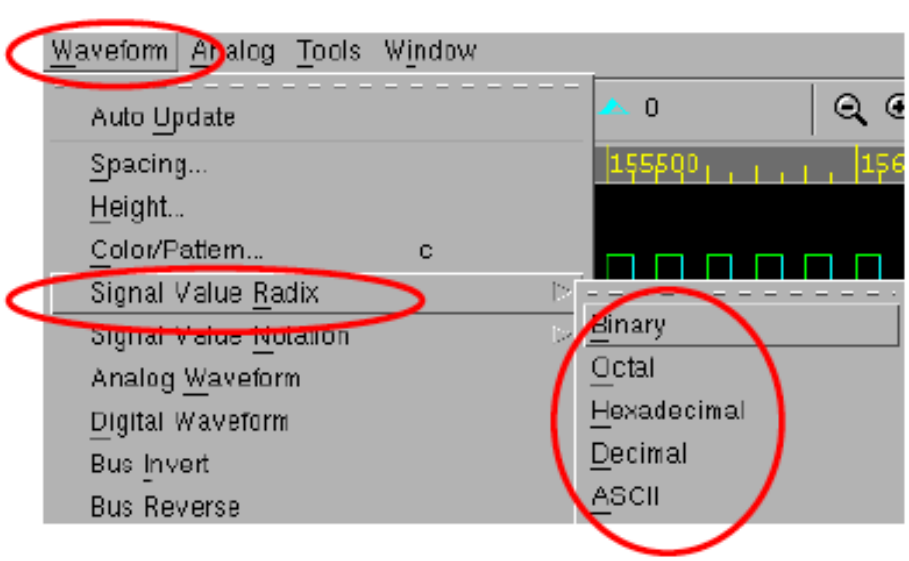


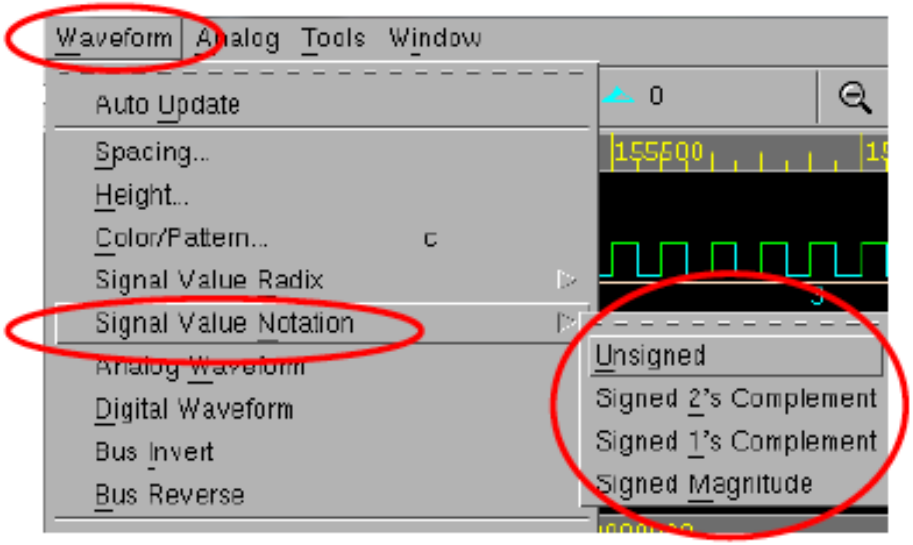


Browse the Specified Interval (2/2)

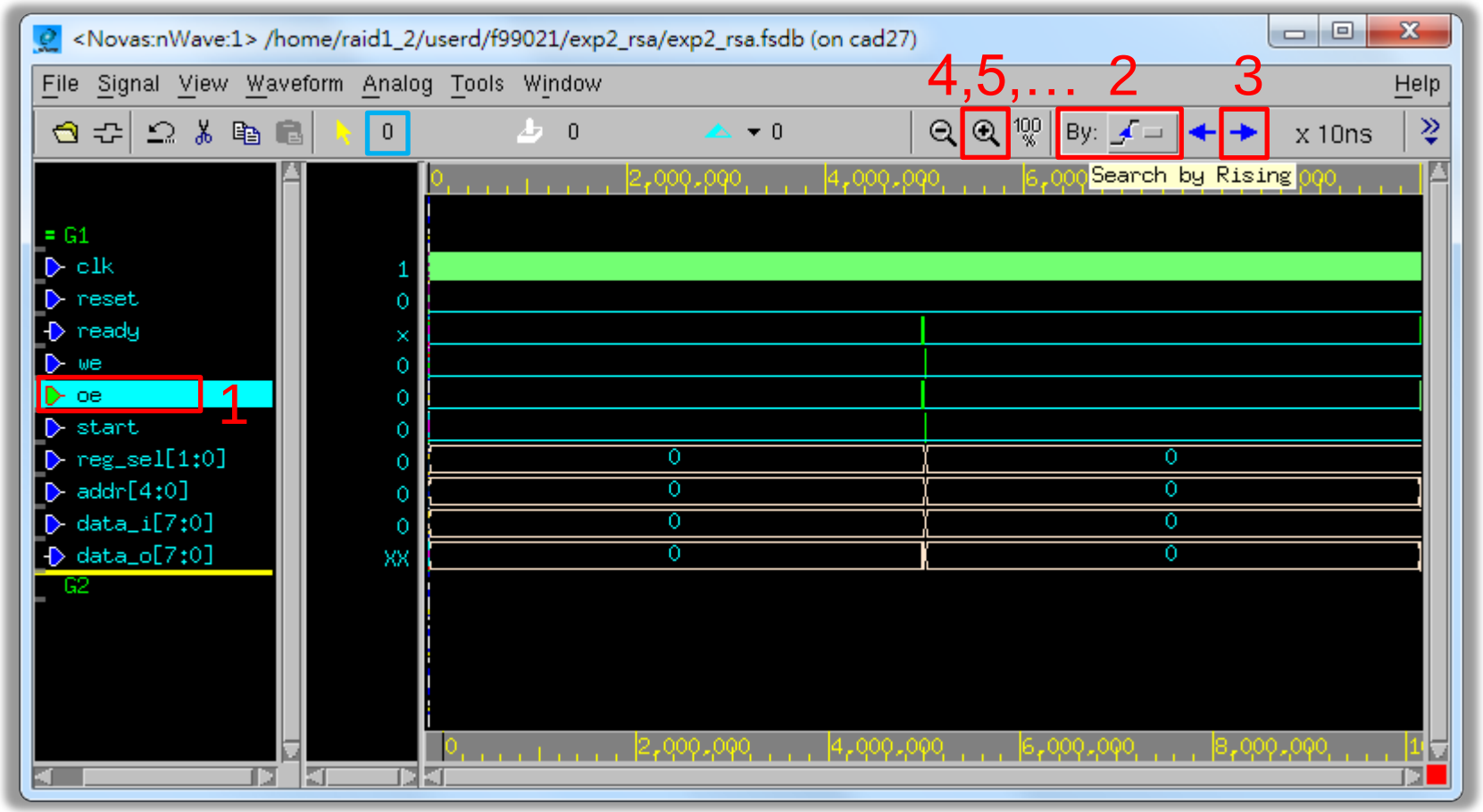


Change Number Representation

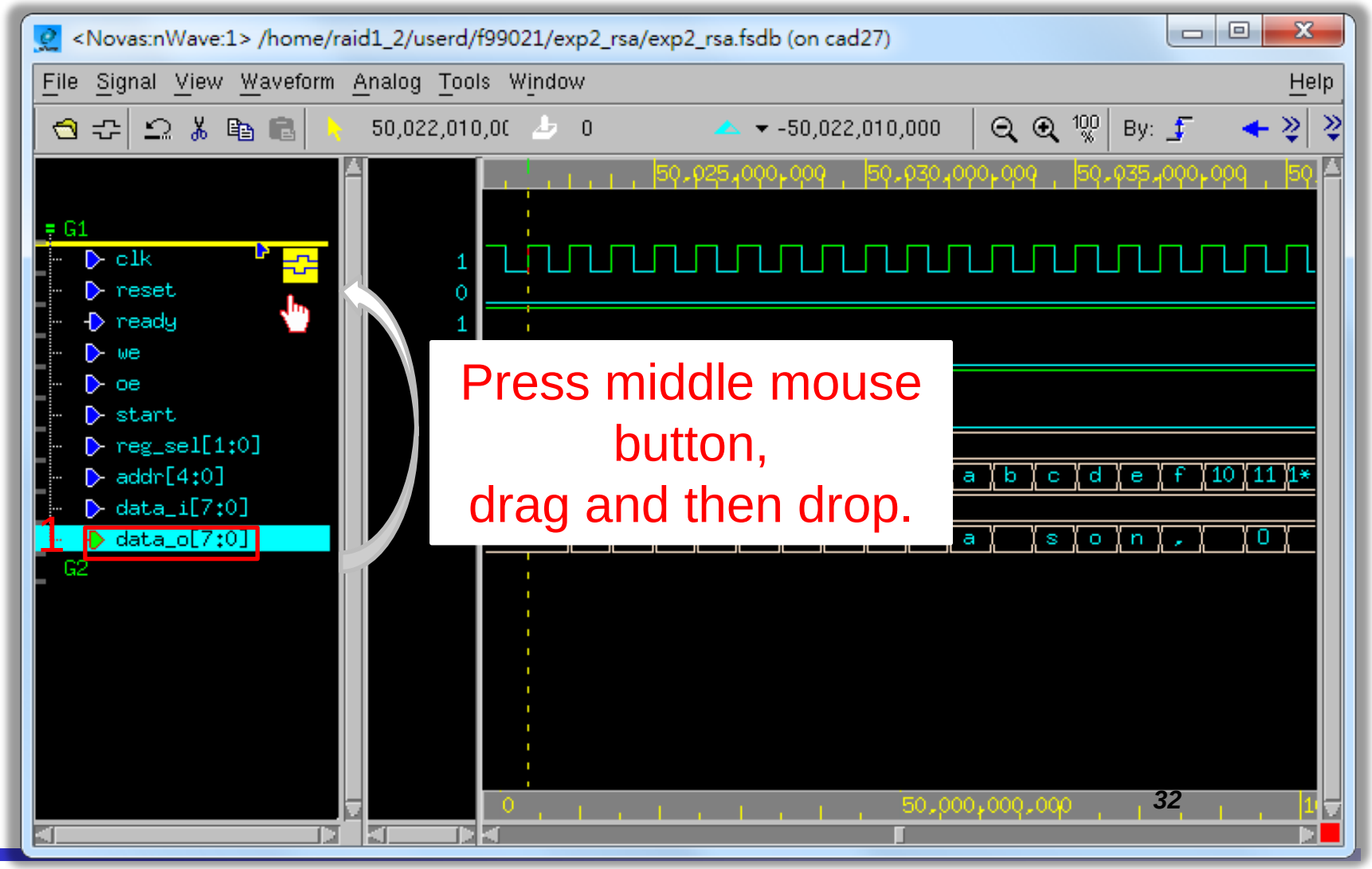




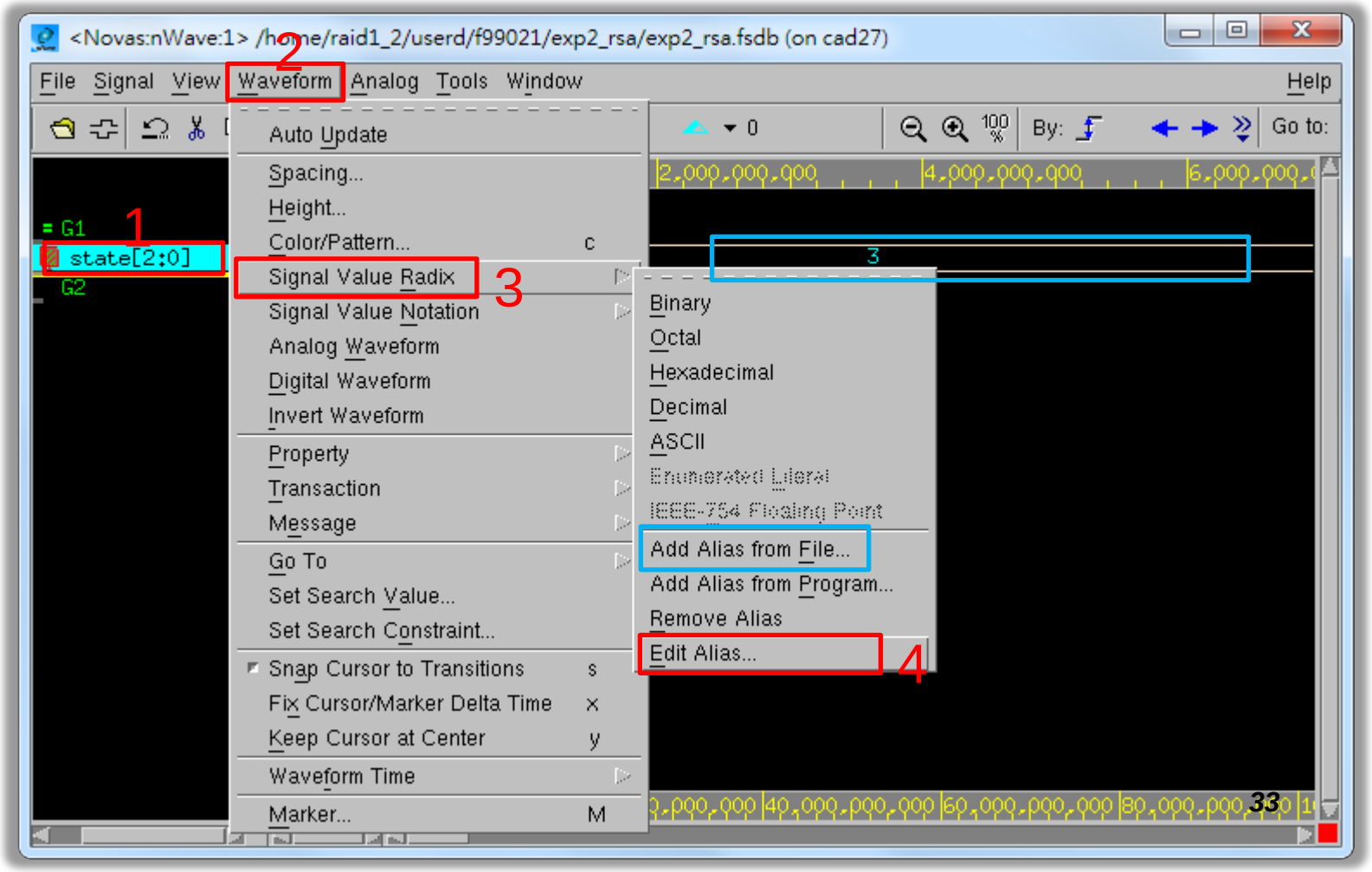
Search for Specified Signal

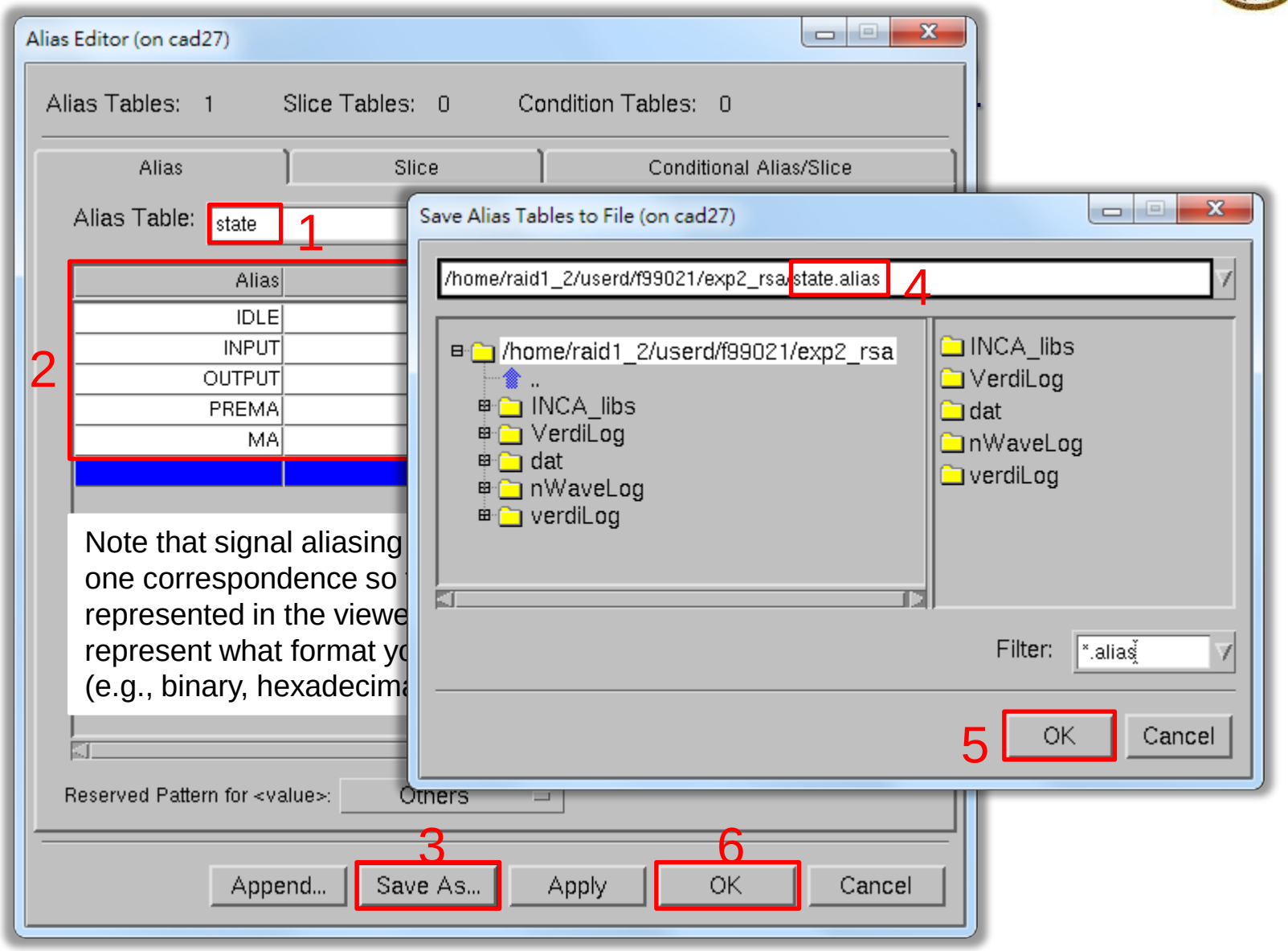


Change Signal Position

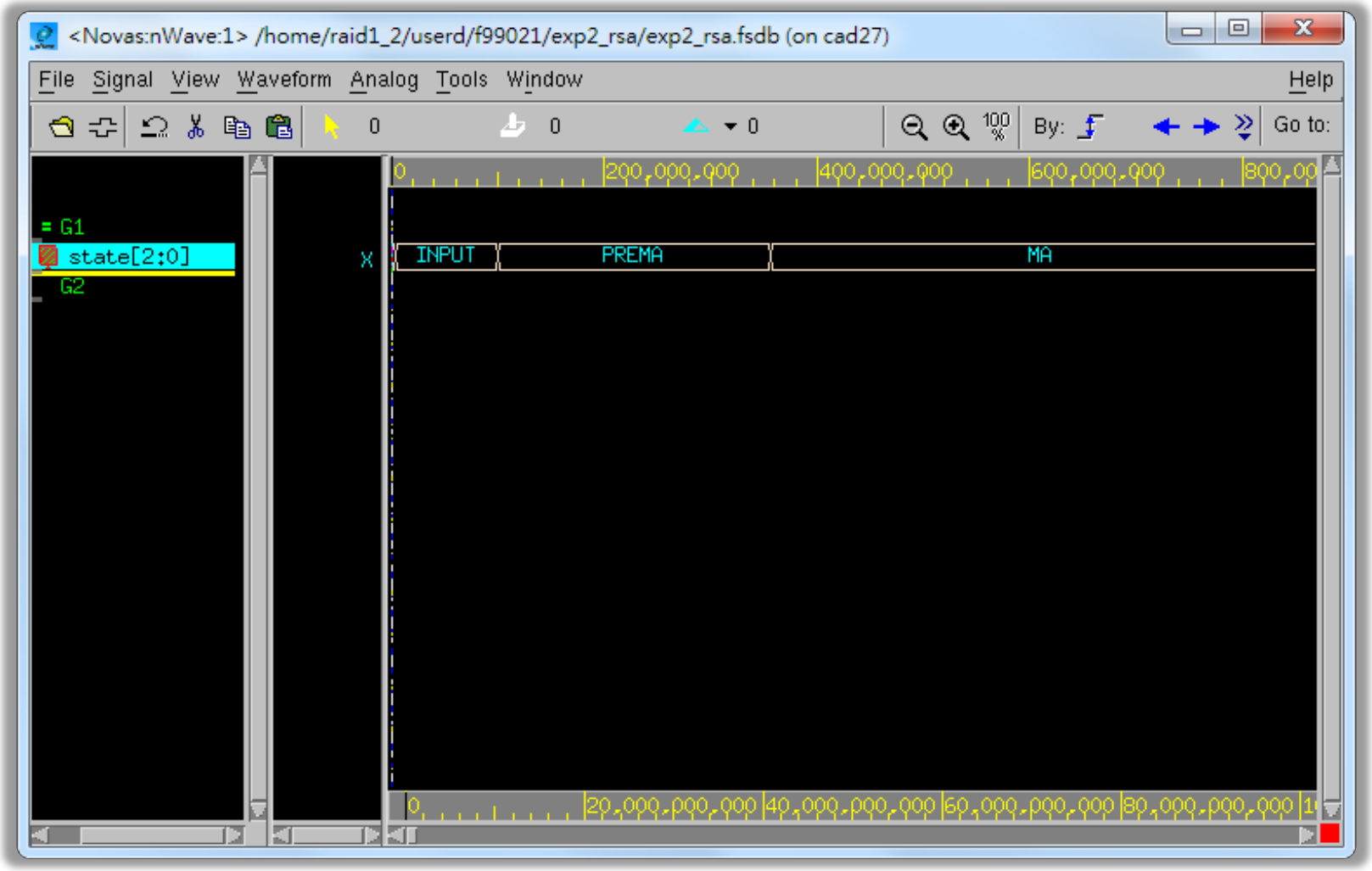


Signal Aliasing(1/3)




Alias Editor (on cad27)
Alias Tables: 1
Slice Tables: 0
Condition Tables: 0
Alias
Slice
Conditional Alias/Slice
Alias Table: state
1
Alias
IDLE
INPUT
OUTPUT
PREMA
MA
2
Note that signal aliasing
one correspondence so
represented in the viewe
represent what format yo
(e.g., binary, hexadecima
Reserved Pattern for <value>:
Others
3
Append...
Save As...
Apply
6
OK
Cancel
Save Alias Tables to File (on cad27)
/home/raid1_2/userd/f99021/exp2_rsa/state.alias
4
/home/raid1_2/userd/f99021/exp2_rsa
..
INCA_libs
VerdiLog
dat
nWaveLog
verdiLog
INCA_libs
VerdiLog
dat
nWaveLog
verdiLog
Filter:
*.alias
5
OK
Cancel

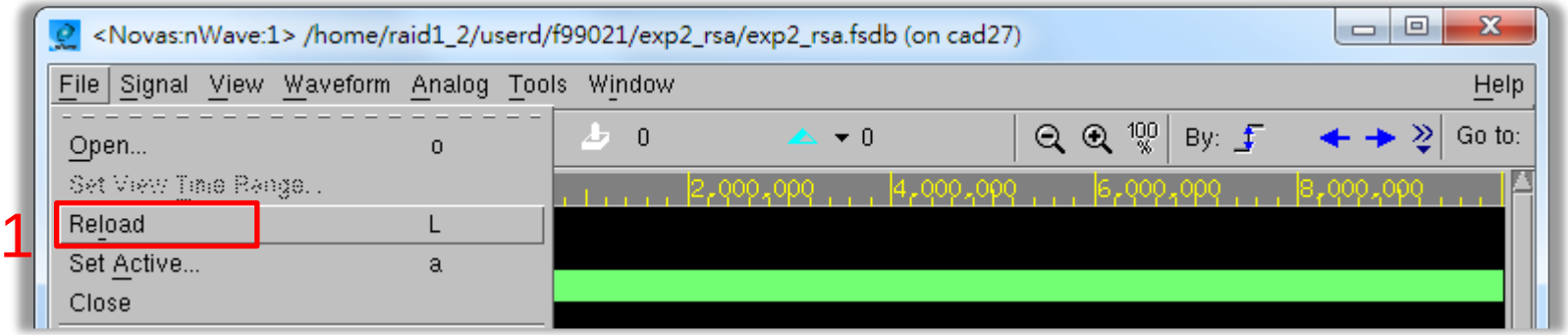
Signal Aliasing(3/3)



Reload the Waveform

- Remember to reload the waveform whenever finishing another Verilog simulation.

- File > Reload or **Shift + L**



Outline

- Introduction to Simulation: NC-Verilog
- Introduction to Debugging Tool
  - nLint: HDL Coding Checking
  - nWave: Waveform Tracing
- Testbench Writing
  - Overview of Simulation
  - Instantiating DUT
  - Creating Clocks
  - Applying Stimulus
  - Verification
- Timing Parameter in Gate-level Simulation
  - Setup and Hold time

Cell-Based IC Design Flow

Algorithm

RTL

Synthesis

Layout

Algorithm

Floating-point Analysis

Fixed-point Analysis

Spec

Parameter	Range	Descript ion
Dimension of input data (D_x)	128, 256, 512, 1024	MNIST Databas e
Number of hidden neurons (h_n)	1-256	

Architecture

Verilog

RTL Simulation

Gate-level Netlist

Gate-level Simulation

Power Optimization

Floorplan

Powerplan

Placement

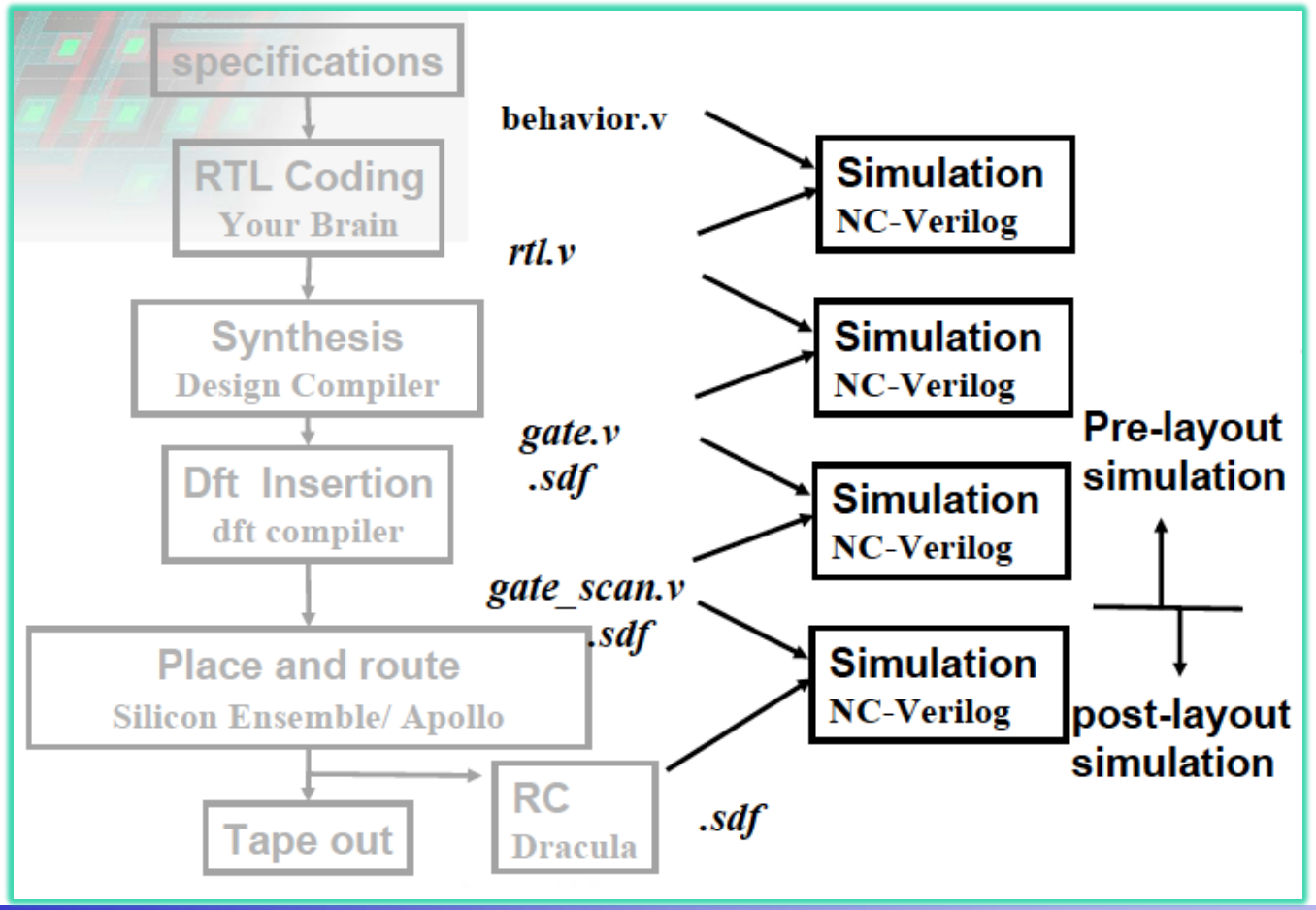
CTS

Route

Post-layout Simulation

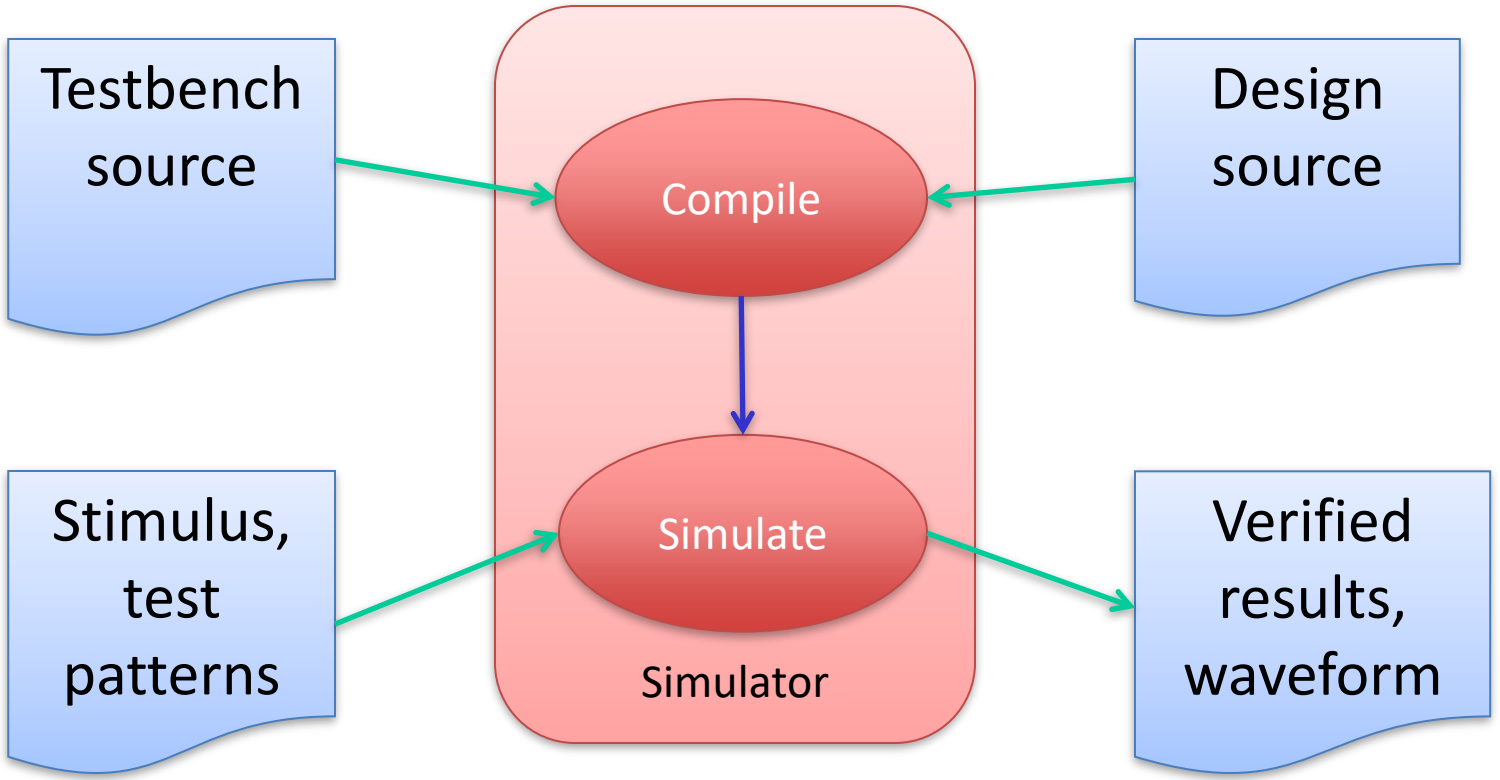
Overview of Simulation (1/2)

- Verification at every step



Overview of Simulation (2/2)

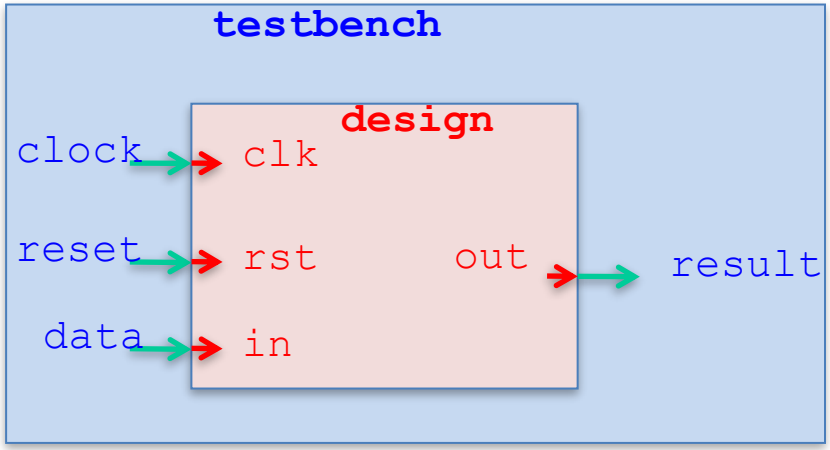
- Simulation Environment



Instantiating DUT

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

```
module testbench;

reg clock, reset, data;
wire result;

design u_design(
     .clk(clock),
     .rst(reset),
     .in(data),
     .out(result)
);
…
```



Input of DUT: use **reg** for applying stimulus
Output of DUT: use **wire** to capture signals

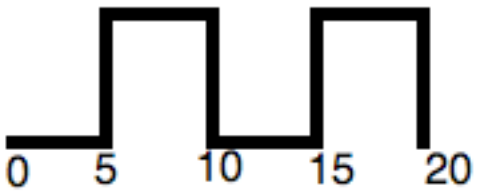
Creating Clocks (1/2)

- Initializing the clock

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

- Modeling the clock behavior

```
`timescale 1ns/10ps
`define   CYCLE 10
`define H_CYCLE  5

always #(`H_CYCLE) begin
    clock = ~clock;
end
```



Creating Clocks (2/2)

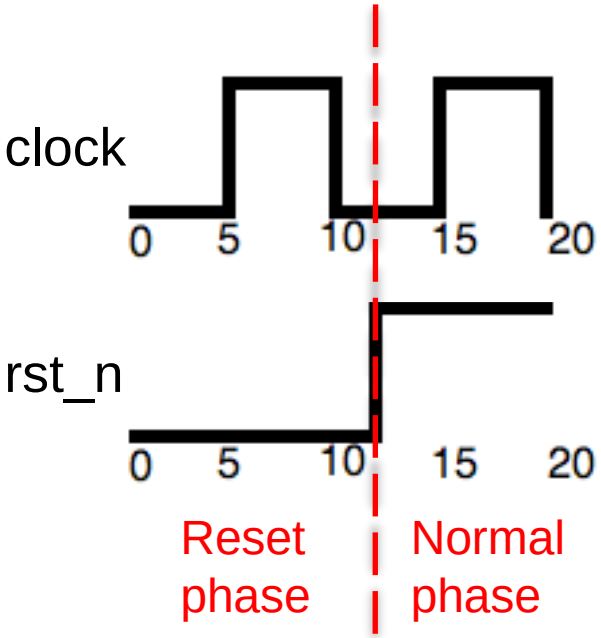
- Other syntax

```
`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 (1/5)

- 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 (2/5)

- 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 (3/5)

- 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
```

If DUT is posedge TB stimulate at negedge !

Applying Stimulus (4/5)

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

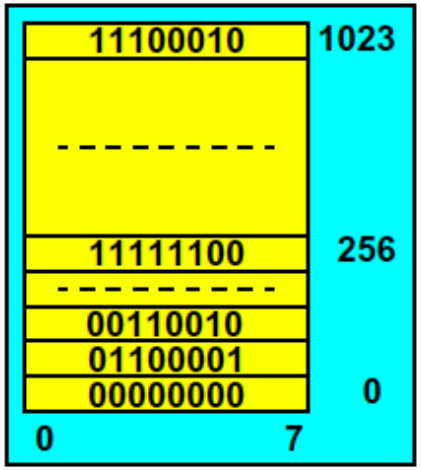
```
module stim_from_file_tb;
    wire [7:0] response;
    reg [7:0] stimulus, stim_array[0:15];
    integer i;
    DUT u1 (response, stimulus);
    initial begin
        $readmemb("datafile",stim_array);
        for (i = 0; i <= 15; i = i + 1)
            #20 stimulus = stim_array[i];
        #20 $finish;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

Applying Stimulus (5/5)

- 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 (@ indicates address)

```
/* 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
```

HEX →



Syntax for Text Monitoring

- Displaying information
  - Print at once
  - `$display([format_string], arg_list)`
  - *$display*("ID of the port is %b", port_id);
    - ID of the port is 00101
- Monitoring information
  - Print if something in `arg_list` changes
  - `$monitor([format_string], arg_list)`
  - *$monitor*("Value of signals clk = %b rst = %b", clk, rst);

```
Value of signals clk = 0 rst = 1
Value of signals clk = 1 rst = 1
Value of signals clk = 0 rst = 0
```

Timing Controls

- Procedural timing controls
  - The simple pound delay (#)
  - The event-based control (@)
  - The level-sensitive control (wait)

- Simple delay (#)

```
#2 y=1;
#4 x=0;
#(1,2,3) q=0;

#5 y=tempxz;
y = #5 tempxz;
```

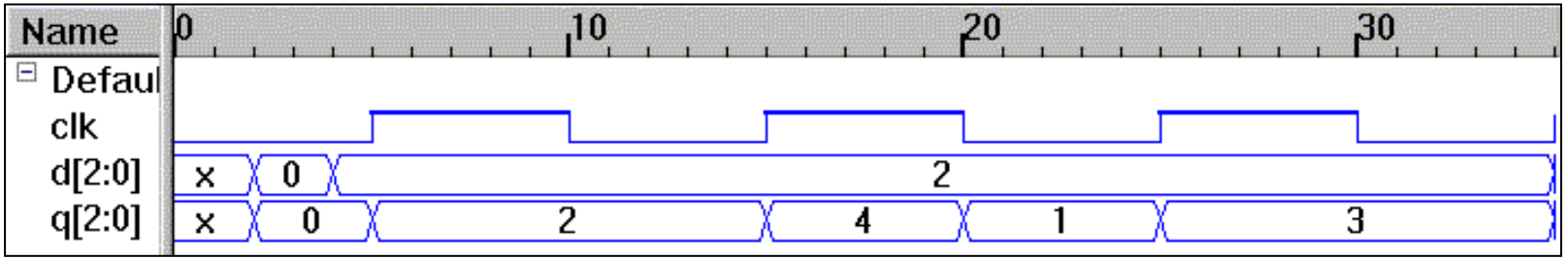
In non-intra-assignment controls (delay or event control on the left side), the right side expression is evaluated after the delay or event control

Intra-assignment controls always evaluate the right side expression immediately and assign the result after the delay or event control

Timing Controls

❖ Event-Based Timing Control (@)

```
module test;
  reg clk;
  reg [2:0] q,d;
  always #5 clk=~clk;
  initial
    begin
      clk=0;
      #2 q=0;d=0;
      #2 d=2;
      @(clk) q=d;
      @(posedge clk) q=4;
      @(negedge clk) q=1;
      q=@(posedge clk) 3;
      #10 $finish;
    end
endmodule
```



Timing Controls

- Level-Sensitive Timing Control (**wait**)
  - (wait) control suspends subsequent statement execution unless, or until, the expression is true.
  - If the expression is already true when the simulator encounters this control, it immediately proceeds to execute the next sequential statement.

```
module behavioral (req,run,ack,rst);
output req; reg req;
input run,ack,rst;
  always @(posedge run) begin
      req = 1;
      wait (ack || rst)
        req = 0;
    end
endmodule
```

Waveform Verification (2/2)

- **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(<depth>,<instance>,<option>);`

Waveform Verification (2/2)

- About `$fsdbDumpvars(<depth>,<instance>,<option>);`
  - Depth: dumping signal of n-1 level below
    - 0 (all scopes), 1 (current), 2 (current + one below)
  - Instance: targeting module
    - Often use the <test_module_name>
  - Option:

Options	Description
“+struct” (default)	Dump all structs
“+mda”	Dump all memory and MDA signals
“+all”	Dump all signals including memory, MDA, packed array, structure, union, power-related, and packed structure

  - E.g. `$fsdbDumpvars(0,test_alu,”+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 @ (negedge 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
```

Read Pattern

Test asynchronously to the triggering clock edge

Display error information

Termination

Other Tips

- Timing of Output Capturing
  - Usually capture the outputs at negative clock edge for a positive edge clock in design
- Add a Time-Out Condition
  - Because termination condition may never be reached when design is not correct.

```
...
`define TIME_OUT 10000
initial #(`TIME_OUT) $finish;
...
```

- *Learn More from Given HW Testbench!*

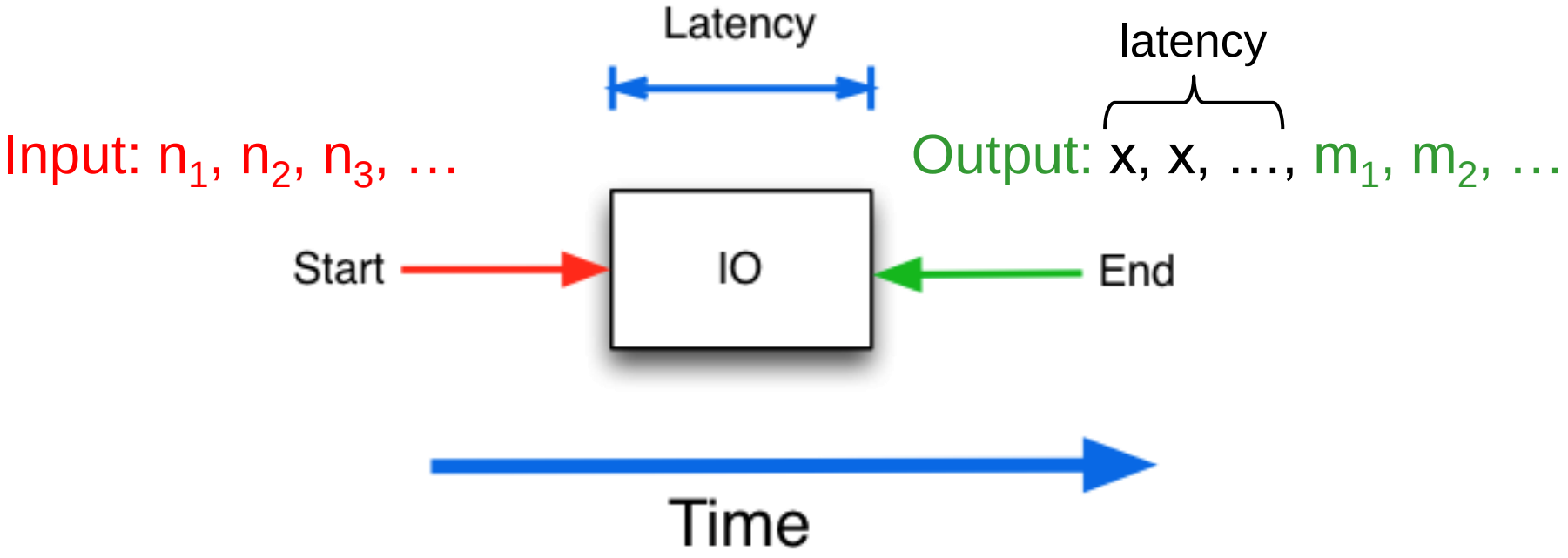
Summary (1/2)

- Step 1: Instantiate design under test (DUT)
  - Input of DUT (reg): clk, reset, input pattern
  - Output of DUT (wire): output pattern
- Step 2: Create clock
  - Use “always #(`H_CLK) clk = ~clk”
- Step 3: Read in or generate test pattern
  - Provide pattern *asynchronously to clock* (# t_1)
- Step 4: Compare test pattern
  - Dump waveform using .fsdb or .vcd
  - Check pattern *asynchronously to clock* (# t_2)

We will elaborate more at synthesis

Summary (2/2)

- Step 3/4: I/O pattern alignment
  - Notice the *latency* of your design
  - Check by *fix cycle* (simple) or *handshake* (stable)



Outline

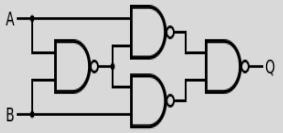
- Introduction to Simulation: NC-Verilog
- Introduction to Debugging Tool
  - nLint: HDL Coding Checking
  - nWave: Waveform Tracing
- Testbench Writing
  - Overview of Simulation
  - Instantiating DUT
  - Creating Clocks
  - Applying Stimulus
  - Verification
- Timing Parameter in Gate-level Simulation
  - Setup and Hold time

Cell-Based IC Design Flow

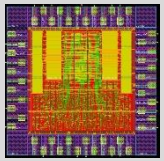
Algorithm

RTL

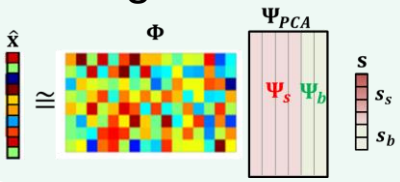
Synthesis



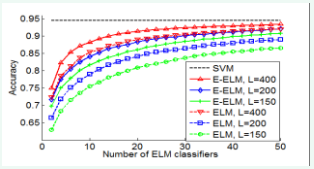
Layout



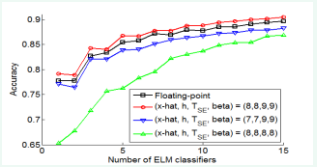
Algorithm



Floating-point Analysis



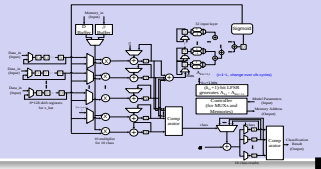
Fixed-point Analysis



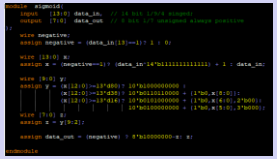
Spec

Parameter	Range	Descript ion
Dimension of input data (D_x)	128, 256, 512, 1024	MNIST Databas e
Number of hidden neurons (h_n)	1-256	

Architecture

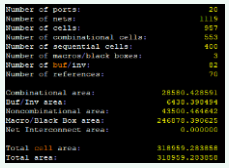


Verilog



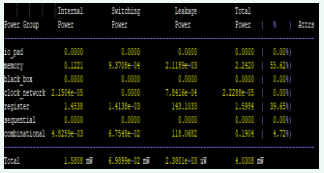
RTL Simulation

Gate-level Netlist

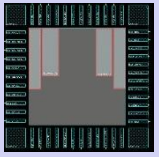


Gate-level Simulation

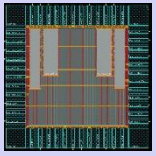
Power Optimization



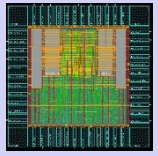
Floorplan



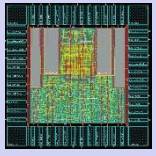
Powerplan



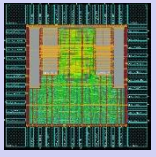
Placement



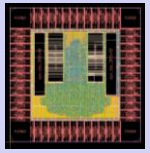
CTS

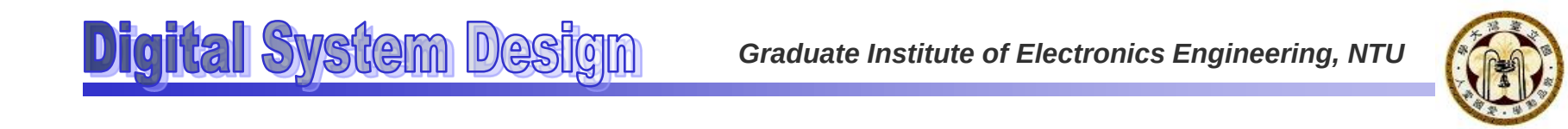


Route

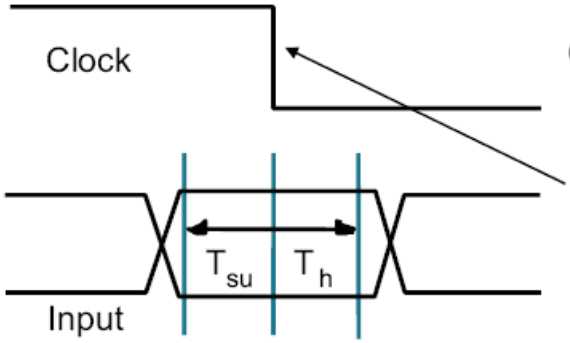


DRC/LVS





Timing Parameters (1/2)



Clock:
Periodic Event, causes state of memory element to change

memory element can be updated on the: rising *edge*, falling *edge*, high *level*, low *level*

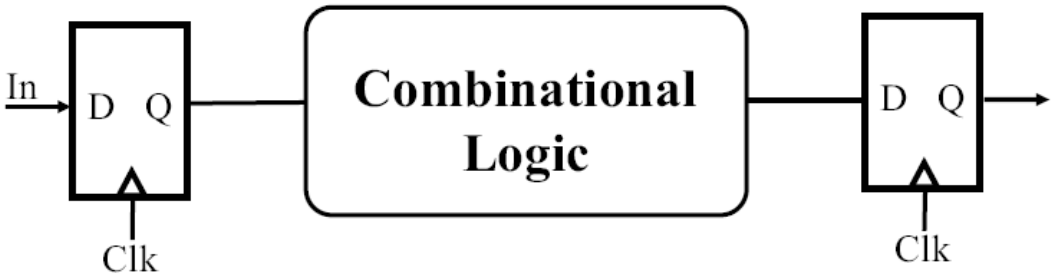
There is a timing window around the clocking event during which the input must remain stable and unchanged in order to be recognized by flip/flop or latch.

Setup Time (T_{su})
Minimum time before the clocking event by which the input must be stable

Hold Time (T_h)
Minimum time after the clocking event during which the input must remain stable

Propagation Delay (*T_{cq}* for edge-triggered flip/flop *or* *T_{dq}* *for latch*)
Delay overhead of the memory element

Timing Parameters (2/2)



Register Timing Parameters

T_{cq} : worst case rising edge clock to q delay

$T_{cq, cd}$: contamination or minimum delay from clock to q

T_{su}: setup time

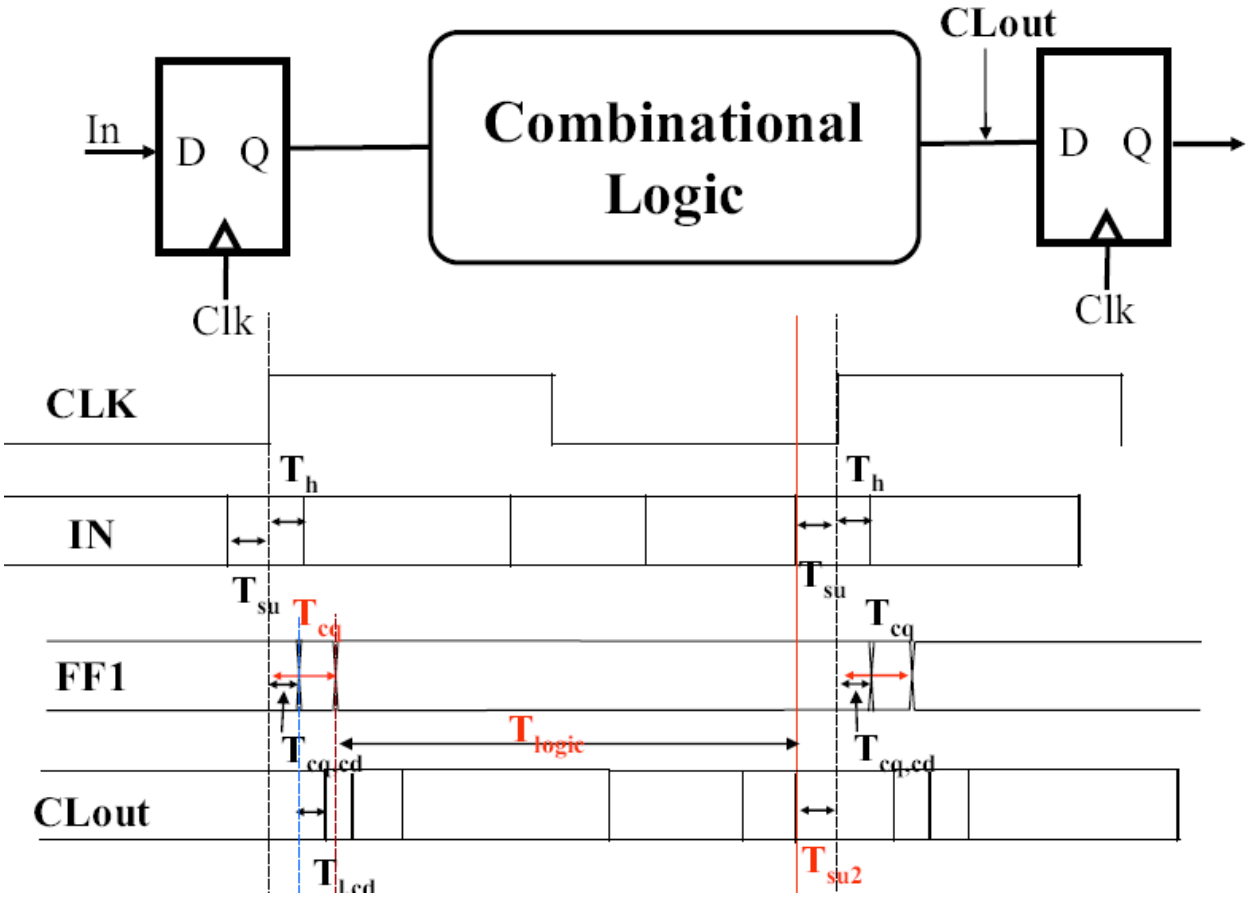
T_h: hold time

Logic Timing Parameters

T_{logic} : worst case delay through the combinational logic network

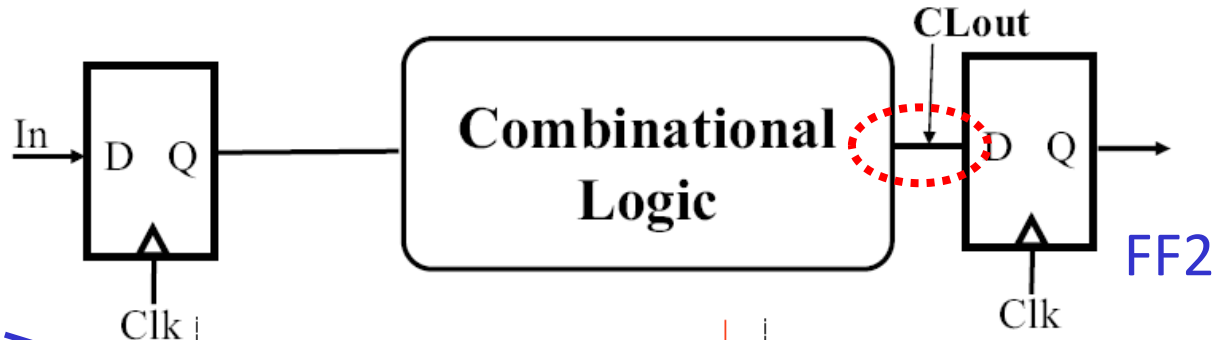
$T_{logic,cd}$: contamination or minimum delay through logic network

System Timing: Maximum Delay

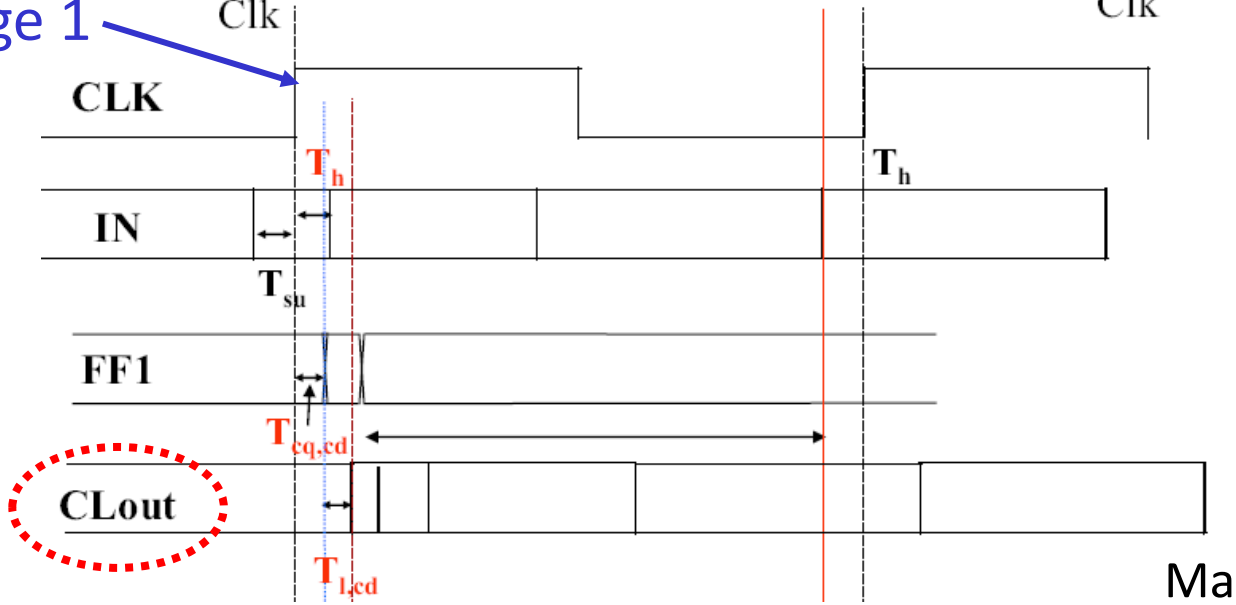


$$T_{cq} + T_{logic} + T_{su2} < T \quad \Rightarrow \quad T_{logic} < T - T_{cq} - T_{su2}$$

System Timing: Minimum Delay



Rising edge 1



$$T_{cq,cd} + T_{logic,cd} > T_{hold2}$$

Make FF2 satisfy the hold time on rising edge

Setup/Hold Timing Check (1/2)

- *Specify* Block
  - Use **specify** and **endspecify** for declaring timing checks
  - The **specify** block 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**)**
  - **$setuphold(**clock_event, FF_data, su_limit, h_limit, notifier**)**

Setup/Hold Timing Check (2/2)

- Example

```
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;
```

In Synthesis part, we will teach you how to perform a gate-level simulation to check this!