



CT Verilog Series

My Very First Verilog Coding

黃稚存

Chih-Tsun Huang

cthuang@cs.nthu.edu.tw



國立清華大學
資訊工程學系

聲明

- ◎ 本課程之內容 (包括但不限於教材、影片、圖片、檔案資料等)，僅供修課學生個人合理使用，非經授課教師同意，不得以任何形式轉載、重製、散布、公開播送、出版或發行本影片內容 (例如將課程內容放置公開平台上，如 Facebook, Instagram, YouTube, Twitter, Google Drive, Dropbox 等等)。如有侵權行為，需自負法律責任。

Outline

- ◉ Tool Environment
- ◉ A Simple Example to Start with
- ◉ Simulation
- ◉ Waveform Viewer
- ◉ Summary

Tool Environment

Tool Environment for IC Design

- ◉ We use multi-million-dollar commercial tools in class
 - ◆ Verilog simulator (from Cadence):
NCVerilog (Native Compiled-Code Verilog)
Incisive irun / Xcelium xrun
 - ◆ Waveform viewer (from Synopsys):
nWave (a component in Verdi)
- ◉ Please also refer to the lecture notes:
 - ◆ A Quick Tour to Access NTHU CAD Lab
 - ◆ Getting Started with Verilog Simulation
- ◉ They run on Linux-based workstations
 - ◆ Be familiar with Linux
 - ◆ Recommended text editor: Vim
 - ◆ Reference: 鳥哥的 Linux 私房菜
<http://linux.vbird.org/>

Tool Environment for FPGA Design



- ◉ We use Xilinx FPGA (Filed Programmable Gate Array) for the Logic Design Laboratory
- ◉ Xilinx **Vivado** from simulation, all the way down to synthesis and implementation
- ◉ Please also refer to the lecture notes:
 - ◆ Vivado Installation and Setting Up
 - ◆ Vivado Tutorial for FPGA Implementation
- ◉ You may use built-in specific Verilog editor or
 - ◆ Vim
 - ◆ Visual Studio Code

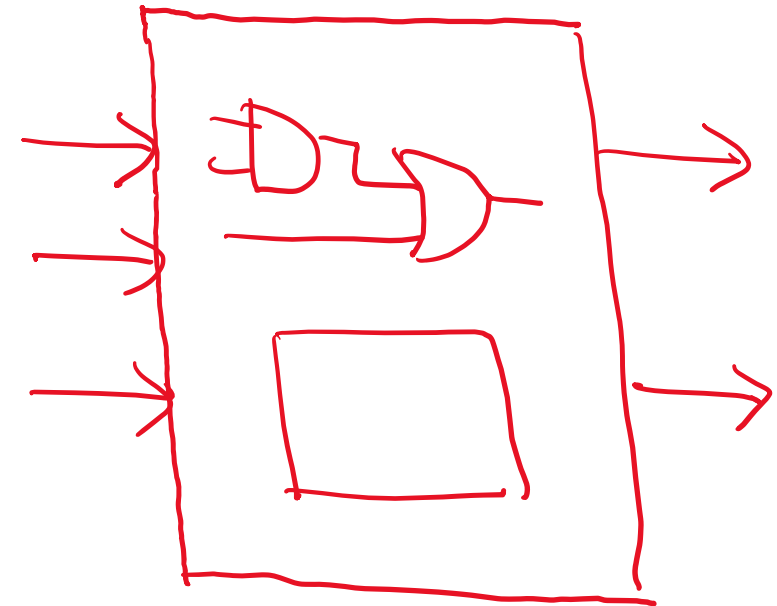
A Simple Example to Start with

Basic Concepts using Verilog HDL

- ◉ Verilog HDL (**Hardware Description Language**) can
 - ◆ Describe the operations of a circuit at various level of abstraction
 - ✓ □ **Structure**
 - ✓ □ **Dataflow** (or function)
 - **Behavior**
 - ◆ Describe the **timing** of a circuit
 - ◆ Express the **concurrency** of circuit operation
- ◉ Verilog simulator
 - ◆ **Event-based** simulation for the efficiency
 - ◆ **Simulated parallel execution** of hardware instances and `always/initial` blocks

Basic Construct

- Basic unit of Verilog HDL is **module**
- Modules have
 - ◆ Module declaration: name + ports
 - ◆ Input and output declarations
 - ◆ Internal signal declarations
 - ◆ Logic definition
 - Submodule instantiations
 - **assign** statements
 - **always** and **initial** blocks



Majority Function

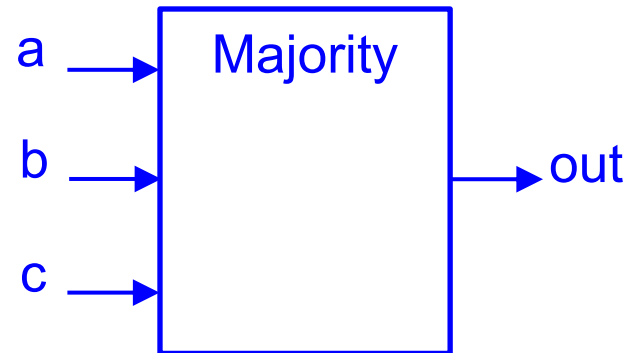
Truth table

<i>a</i>	<i>b</i>	<i>c</i>	<i>out</i>
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

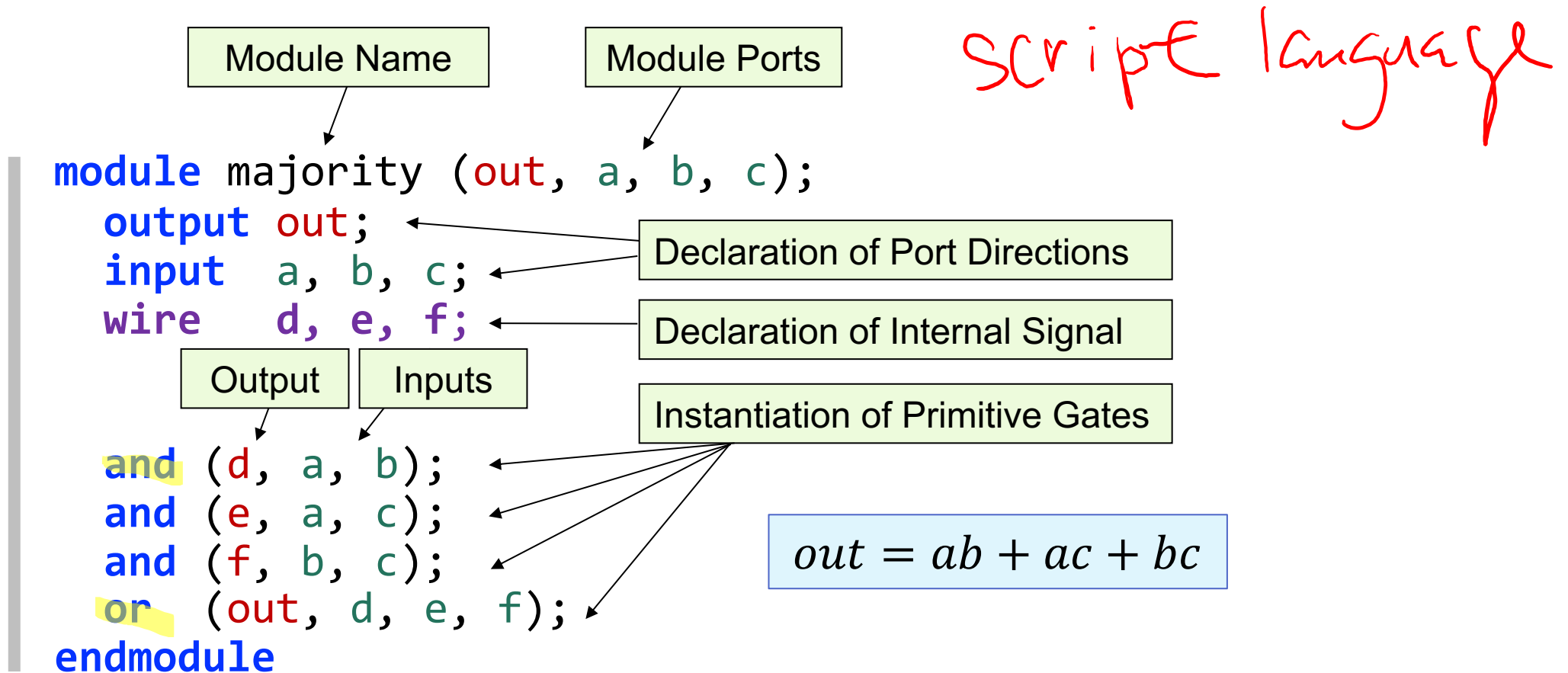
Boolean function

$$\text{out} = ab + ac + bc$$

Block Diagram



Verilog **Structural** (or **Gate-Level**) Description (majority_gate.v) *netlist*



Note: All bold-faced items are Verilog keywords.

Primitive Gates in Verilog

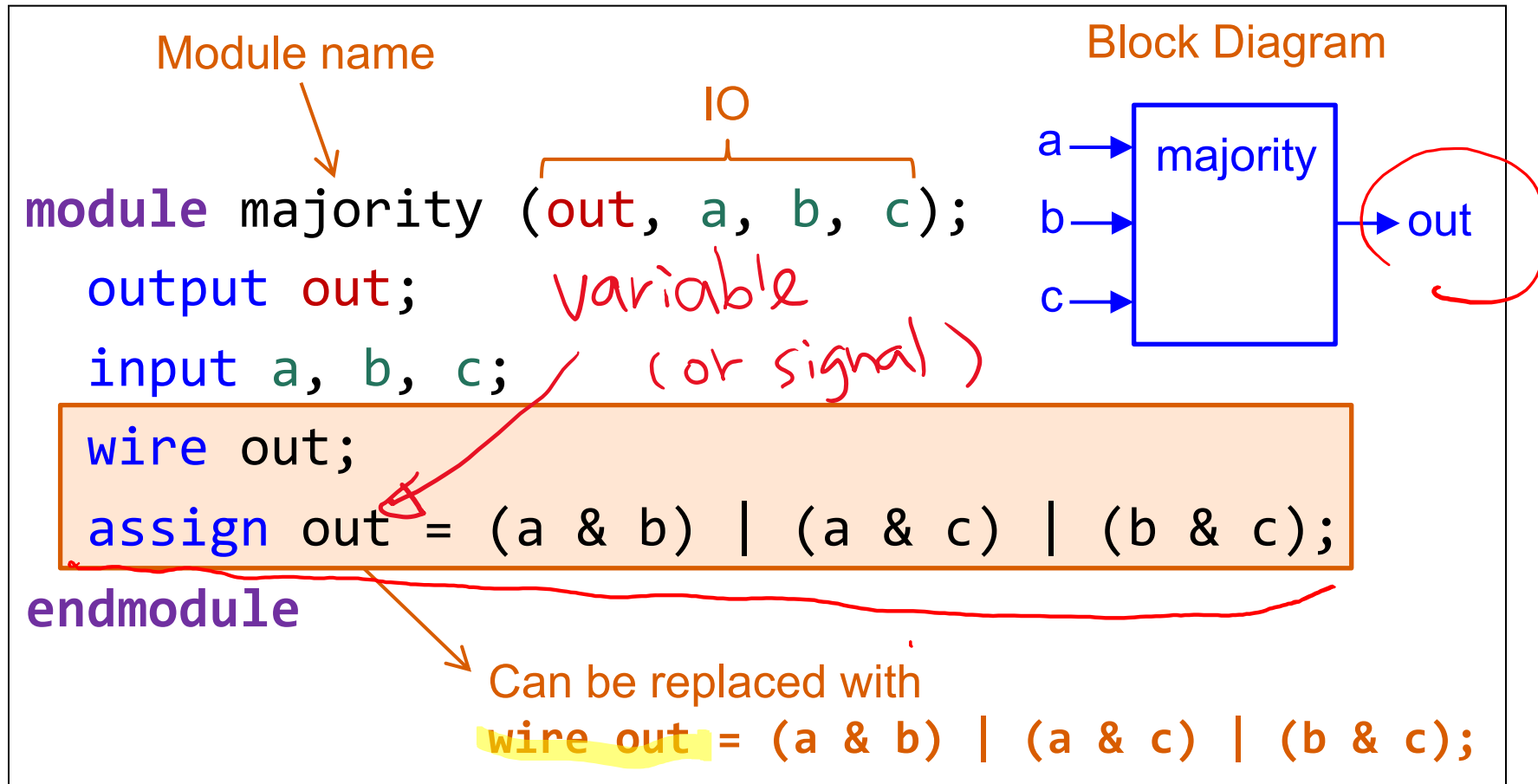
Common built-in logic gates

- ◆ and, nand, or, nor, xor, xnor
- ◆ buf, not

Examples

- ◆ Two-input named NAND gate
`nand gate1 (f, i1, i2);`
- ◆ Three-input unnamed AND gate
`and (f, x, y, z);`
- ◆ Two instances of four-input OR gates
`or b[1:0] (f, i1, i2, i3, i4);`
- ◆ The instance of buffer with four outputs and one input
`buf (o1, o2, o3, o4, i);`

Verilog Dataflow (or Functional) Description (majority_func.v)



*DO NOT copy-paste Verilog code from the lecture notes. There may have hidden special characters that cause errors!!!

Data Types for Signals or Variables

- ◉ Syntax
 - ◆ `<data_type> [<MSB>:<LSB>] <list_of_identifier>`
- ◉ **Wire type**: physical connections between ports
(most popular type of input/internal signals)
 - ◆ **wire**
`wire reset, clock;`
`wire [7:0] address;`
- ◉ **Register types**: abstract data storage elements
(only these types of signals can be on the left-hand side of assignments in procedural blocks)
 - ◆ **reg**: unsigned, varying width
(most popular type of output/internal signals)
`reg carry_out;`
`reg [31:0] data_a, data_b;`
 - ◆ **integer**: two's complement, 32-bit
`integer i, j, k;`
 - ◆ (Other data types are not discussed here)

Boolean Function in Verilog

◉ Combinational logic in Verilog

```
wire out;  
assign out = (a & b) | (a & c) | (b & c);
```

◆ Boolean function $out = ab + ac + bc$

◉ Alternatively, $out = ((ab)'(ac)'(bc)')'$

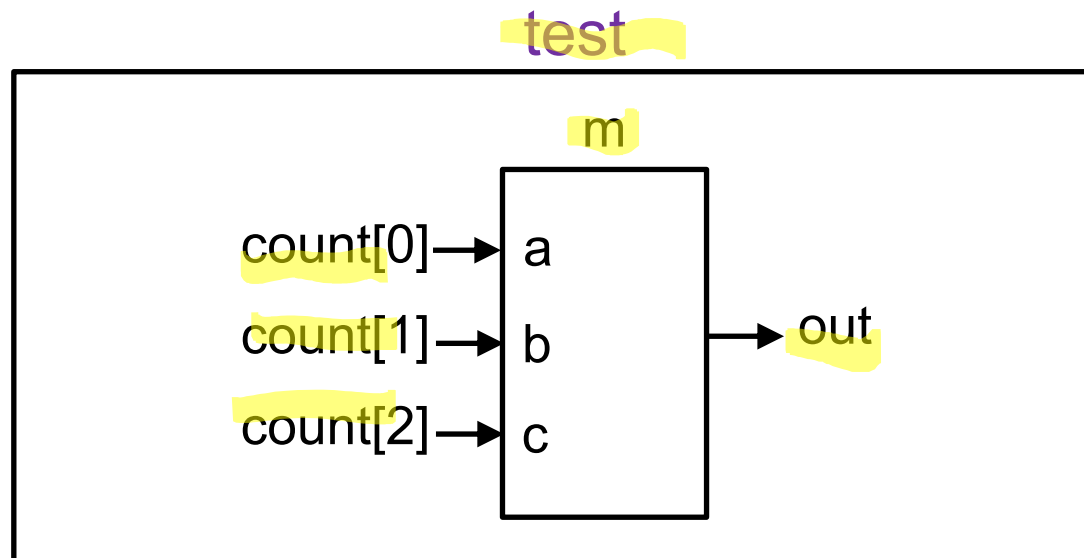
```
assign out = ~(~(a & b) & ~(a & c) & ~(b & c));
```

◆ Less readable

◆ Not necessary since synthesis tool performs optimization for you

Simulation

- ⦿ Design is only complete with verification
 - ◆ Verification by simulation
- ⦿ Design + test stimulus
 - ◆ Stimulus (test patterns) and control
 - ◆ Response verification



Test Stimulus (Testbench) (majority_t1.v)

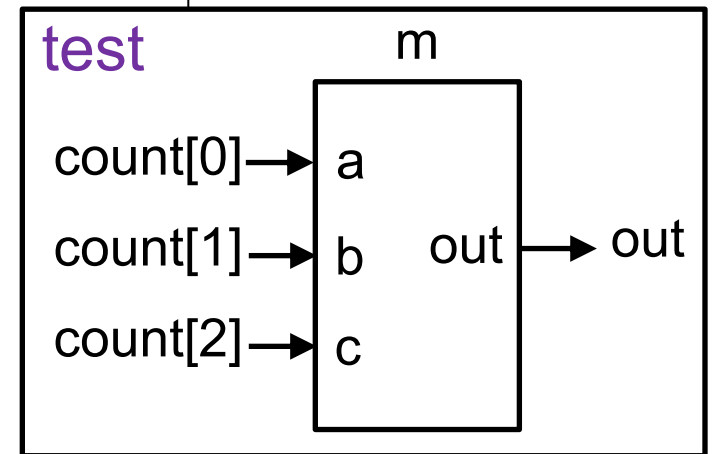
```
timescale 1ns/100ps
module test;
  reg [2:0] count; // 3-bit input
  wire out;        // output
  // instantiate the majority block
  majority m(out, count[0], count[1], count[2]);
  // generate the input patterns
  initial begin
    count = 3'b000;
    #10
    $display("in = %b, out = %b", count, out);
    count = 3'b001;
    #10
    $display("in = %b, out = %b", count, out);
  end
endmodule
```

reference (pointing to `majority`)

instance name (pointing to `m`)

Delay (pointing to `#10`)

Can you generate all the eight input patterns?



Time Scale for Simulation

- ⦿ Add the time scale setting in the beginning of the testbench

``timescale 1ns/100ps`

 ↑ ↑

Reference Time Unit Time Precision

- ⦿ Basic time unit is usually 1ns for modern technology
- ⦿ Precision will affect
 - ◆ The simulation speed
 - ◆ Also, the time scale in the waveform viewer
- ⦿ In Vivado, you will get a warning if not every Verilog file has the timescale setting
 - ◆ In NCVerilog, you only need to put timescale in the beginning of the very first Verilog file



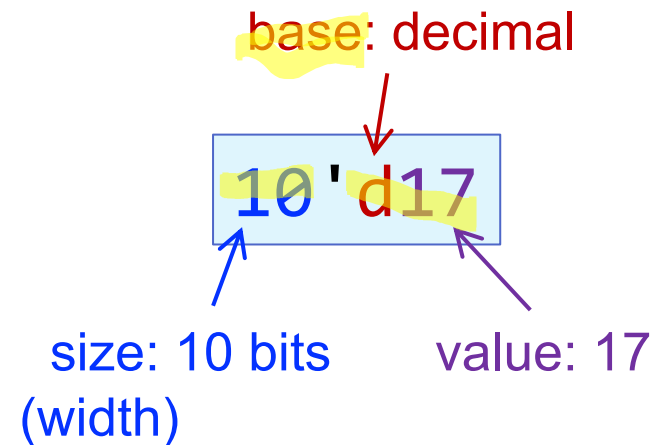
initial Blocks

- ⦿ Cannot be nested
- ⦿ Starts at time 0, executes exactly once
- ⦿ Usually for generating testbench
- ⦿ initial is a procedural block
 - ◆ Procedural statements such as for-loop and repeat-loop can only be placed within initial blocks (and always blocks which are procedural too)
- ⦿ Multi-statement block must be enclosed by begin/end
 - ◆ Similar to { } in C/C++
 - ◆ One-line block may or may not have begin/end enclosure

Number Representation

⦿ Number can be binary (**b**), octal (**o**), decimal (**d**) or hexadecimal (**h**)

- ◆ Value: 0123456789abcdef_
 - Unsized number: 32 bits
- ◆ Sized or unsigned
 - Default numbers are unsigned
- ◆ Examples
 - 10'd17
 - 8'b0010_0111
 - 'h6f
 - 1'b0
 - 7'b1
 - 'b1
 - 1



Simulation

Invoke The Verilog Simulator

- ◉ With the **shell prompt**, type

```
$ ncverilog majority_t1.v majority_func.v
```

or

```
$ irun majority_t1.v majority_func.v
```

to execute Verilog simulation.

- ◆ '\$' is the shell prompt, you don't need to type it
- ◆ To enable waveform dumping, you must add the options +access+r

```
$ irun majority_t1.v majority_func.v +access+r
```

Verilog Simulation Output

- ⦿ There is 10-unit delay time between two outputs
- ⦿ Simulation result (if you manage to apply all 8 input patterns)

```
in = 000, out = 0
in = 001, out = 0
in = 010, out = 0
in = 011, out = 1
in = 100, out = 0
in = 101, out = 1
in = 110, out = 1
in = 111, out = 1
```

Test Stimulus (Testbench) (majority_t2.v)

```
`timescale 1ns/100ps
module test;
    reg [2:0] count;    // three-bit input
    wire out;          // output of majority

    majority m(out, count[0], count[1], count[2]);

    // generate all eight input patterns
    initial begin
        count = 3'b000;
        repeat (8) begin
            #10
            $display("in = %b, out = %b", count, out);
            count = count + 3'b001;
        end
    end
endmodule
```

long long
→ t
Simulated time

↑ ↑ ↑
Simulation time

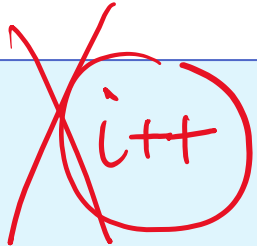
Loops in Verilog

Repeat loop

```
count = 3'b000;  
repeat (8) begin  
    #10  
    $display("in = %b, out = %b", count, out);  
    count = count + 3'b001;  
end
```

For loop

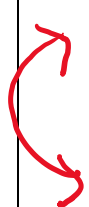
```
integer i;  
...  
count = 3'b000;  
for (i=0; i<8; i=i+1) begin  
    #10  
    $display("in = %b, out = %b", count, out);  
    count = count + 3'b001;  
end
```



Test Stimulus (Testbench) (majority_t3.v)

```
`timescale 1ns/100ps
module test;
    reg [2:0] count; // three-bit input
    wire out;        // output of majority
    // instantiate the block
    majority m(out, count[0], count[1], count[2]);
    initial #1000 $finish;
    // generate all eight input patterns
    initial begin
        for (count=0; count<8; count=count+1) begin
            #10
            $display("in = %b, out = %b", count, out);
        end
    end
endmodule
```

parallel



Infinite Loop?

- Press **CTRL-C** to stop the simulation, and **exit** to quit

```
in = 101, out = 1
in = 110, out = 1
in = 111, out = 1
in = 000, out = 0
in = 001, out = 0
in = 100, out = 0
Simulation interrupted at 30138100 NS + 0
[ncsim> exit
ic22 [cthuang 11:04am] ~/workbench/eecs1010/majority$
```

- You can add a failsafe termination

```
initial
    #1000 $finish;
```

- ◆ Note: for one-line block, begin/end can be omitted
- ◆ Two or more initial blocks are simulated in parallel

Test Stimulus (Testbench) (majority_t4.v)

```
`timescale 1ns/100ps
module test;
    reg [3:0] count; // why using four-bit counter?
    wire out;
    // instantiate the block
    majority m(out, count[0], count[1], count[2]);
    // generate all eight input patterns
    initial begin
        for (count=0; count<8; count=count+1) begin
            #10
            $display("in = %b, out = %b", count, out);
        end
    end
endmodule
```

Simulation with Waveform Dumping

- For Verdi **FSDB** format, add the following code segment

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

or \$fsdbDumpvars(0, test);

task

Mandatory option! Do not forget it!!

```
$ irun majority_t5.v majority_func.v +access+r
```

Simulation with Waveform Dumping

- For standard VCD (Value Change Dump) format:

```
initial begin
```

```
    $dumpfile("counter.vcd");
```

```
    $dumpvars(0, stimulus);
```

```
end
```

- For compressed Debussy/Verdi FSDB format:

```
initial begin
```

```
    $fsdbDumpfile("counter.fsdb");
```

```
    $fsdbDumpvars;
```

```
end
```

Test Stimulus (Testbench) (majority_t5.v)

```
`timescale 1ns/100ps
module test;
    reg [2:0] count;    // three-bit input
    wire out;          // output of majority
    integer i;
    majority m(out, count[0], count[1], count[2]);
    initial begin
        $fsdbDumpfile("majority.fsdb");
        $fsdbDumpvars;
    end
    initial begin
        for (i=0; i<8; i=i+1) begin
            count = i[2:0];
            #10
            $display("in = %b, out = %b", count, out);
        end
    end
endmodule
```

Separate Design and Test Stimulus

regression test

```
$ ncverilog majority_t1.v majority_func.v +access+r  
majority_t2.v majority_gate.v  
majority_t3.v  
majority_t4.v  
majority_t5.v
```

- ⦿ You can replace with different implementations as long as the interface (of IOs) is the same between design and test stimulus

Verilog Simulation

- Method 1

- \$ ncverilog counter_test.v counter1.v +access+r

- Method 2

- Using additional file to define the project: counter.f:

counter_test.v
counter1.v

Project

- ncverilog -f counter.f +access+r

- Method 3

- Using shell script

- Method 4

- Using Makefile

- Syntax checking (no simulation)

- ncverilog -c counter1.v

- ncverilog -c -f counter.f

Simulation with Shell Script

- Create a script file: cnt.sh

~~#!/bin/sh~~

```
ncverilog counter_test.v counter1.v +access+r
```

- Execute the script file

\$ **sh** ./cnt.sh

(Using the up/down keys can access the previous commands in the shell.)

Simulation with Makefile

- **Makefile** is a good friend to a programmer
 - make**
 - make sim**
 - make check**
 - make clean**
- The most important thing is that Makefile uses **tab** strictly
 - ◆ ~~make1:7: *** missing separator. Stop.~~
- You can also avoid the use of many ***.f** files, integrating them into one Makefile

```
VLOG          = ncverilog
SRC           = -f counter.f
VLOGARG       = +access+r
TMPFILE       = *.log \
               verilog.key \
               nWaveLog

DBFILE        = *.fsdb *.vcd *.bak
RM            = -rm -rf

all :: sim

sim :
    $(VLOG) $(SRC) $(VLOGARG)
check :
    $(VLOG) -c $(SRC)
clean :
    $(RM) $(TMPFILE)

veryclean :
    $(RM) $(TMPFILE) $(DBFILE)
```

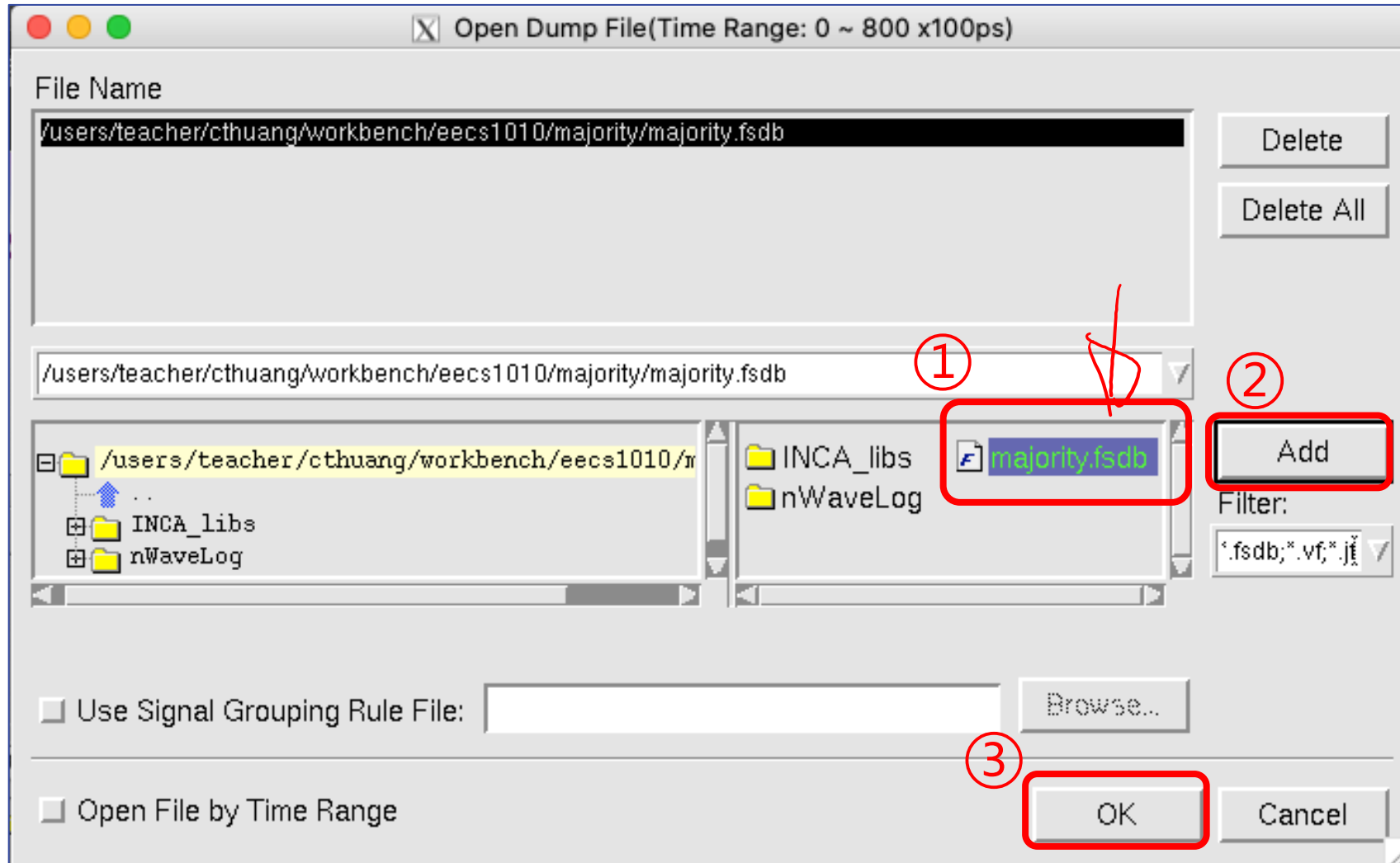
Waveform Viewer

Invoke the Waveform Viewer

- ⦿ Using nWave on Linux workstations
 - ◆ One of the components in Verdi
 - \$ nWave
- ⦿ You should learn the basic concepts of Linux UI

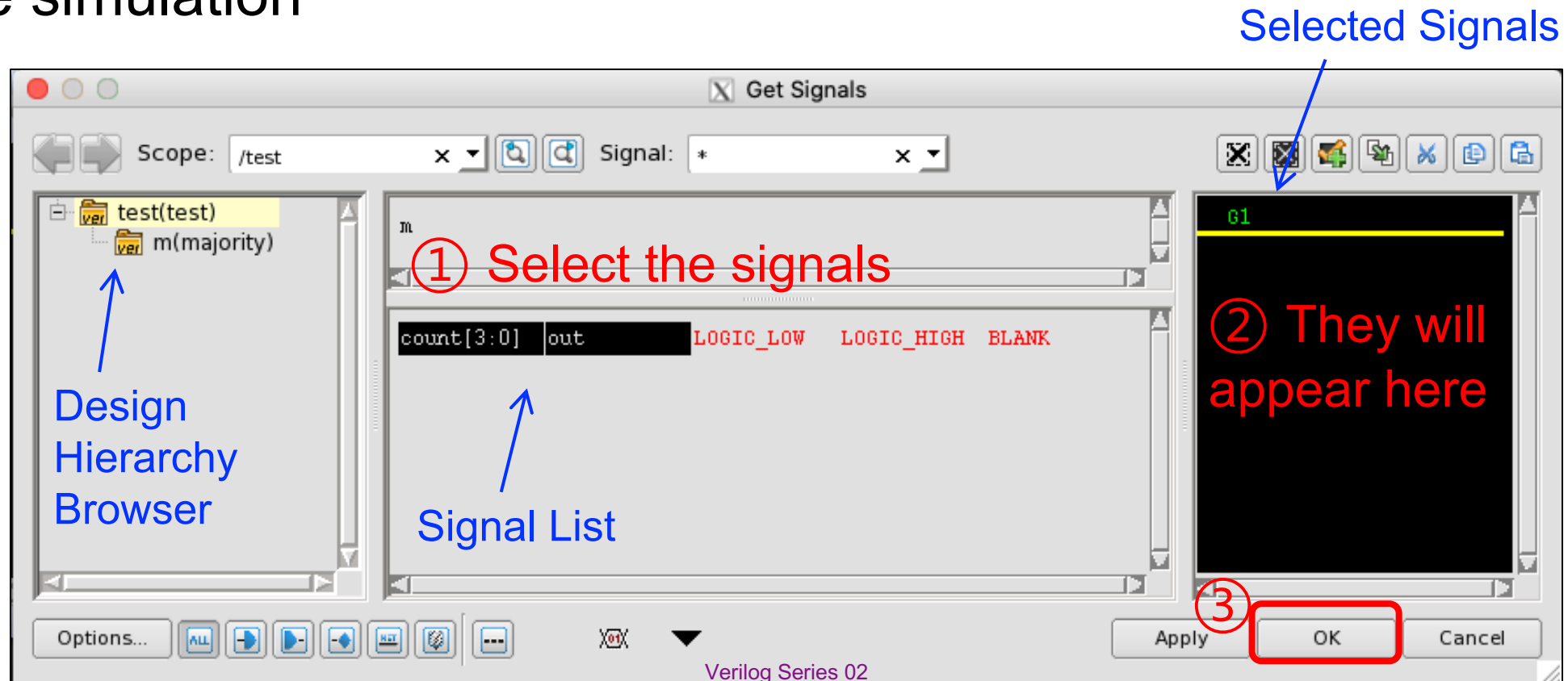
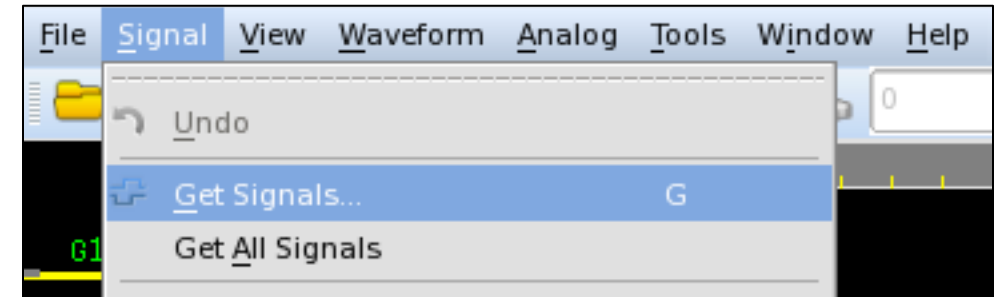
Open FSDB-format Waveform

- Open FSDB file: majority.fsdb (File → Open)



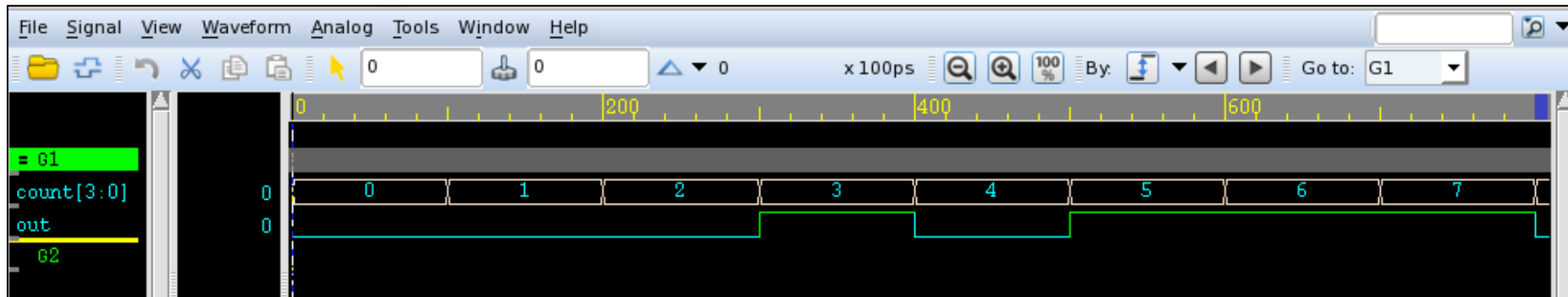
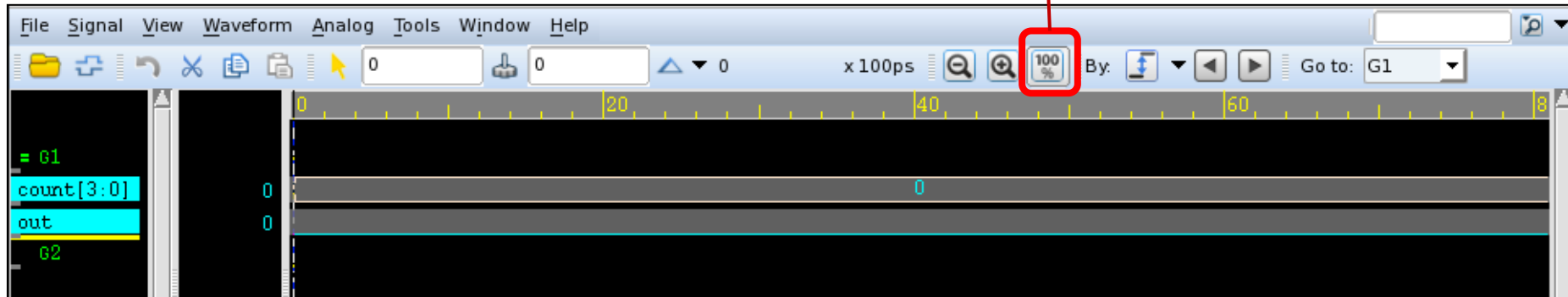
Select Signals to Observe

- Signal → Get Signals
(or Get All Signals)
- Press “L” to reload after relauching the simulation

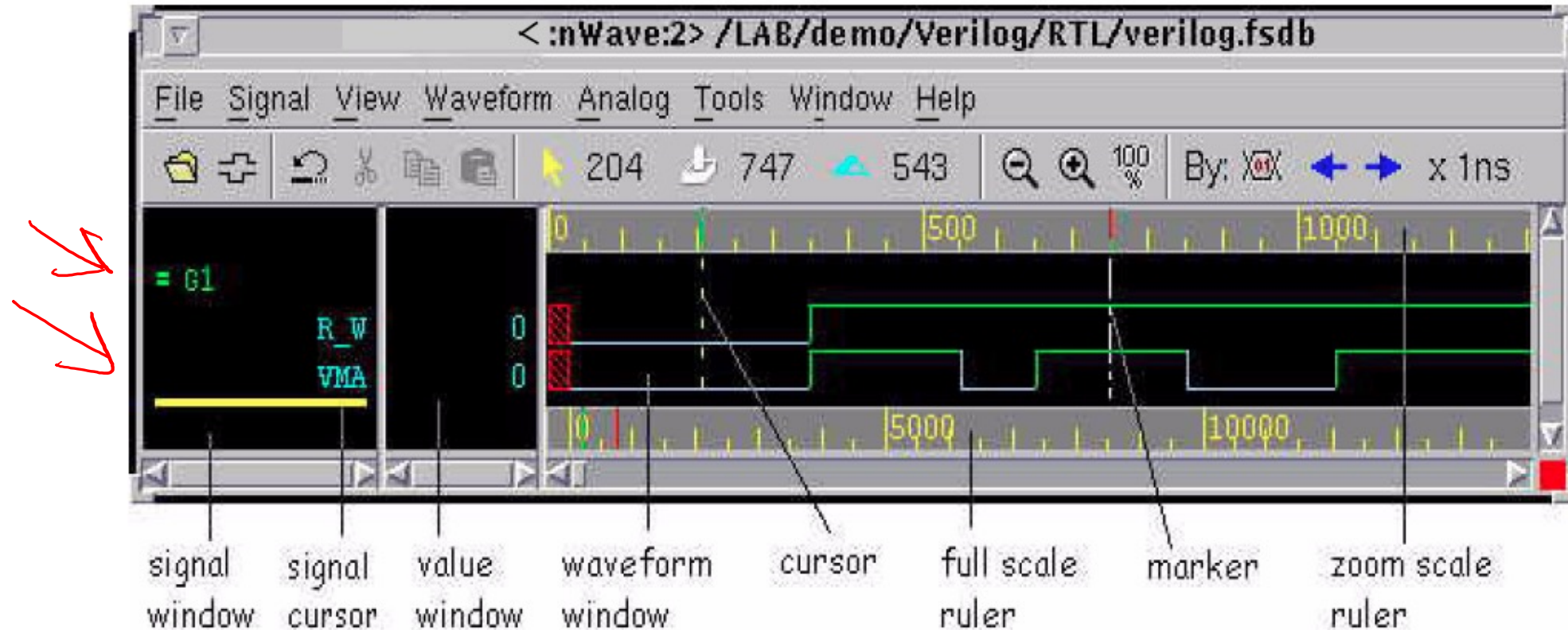


Signal Display

Zoom to fit

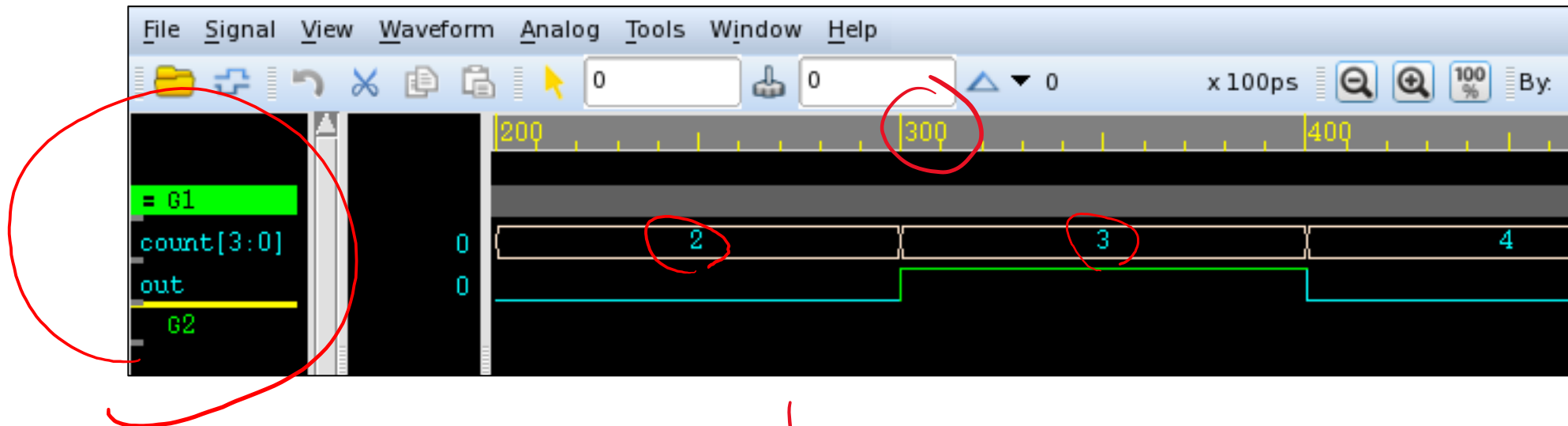


nWave's Waveform Window




Time Unit in nWave

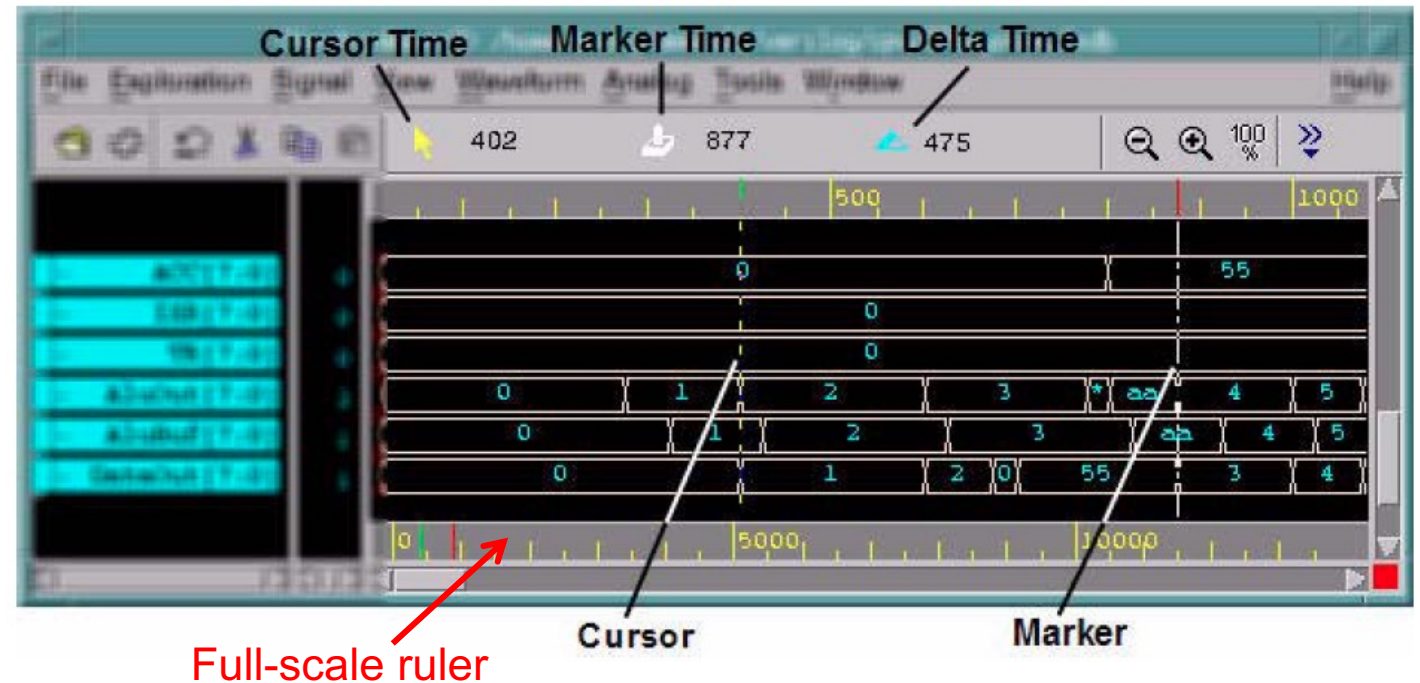
300x100ps



- What does “300” mean along the time axis?
 - If you use ``timescale 1ns/100ps`
 - Then 300 units indicate 30ns ($100\text{ps} \times 300 = 30\text{ns}$)

Cursor/Marker Position

- ⦿ Left click: cursor
- ⦿ Middle click: marker
- ⦿ 3-click zoom
 - ◆ Set cursor (left click) and marker (middle click)
 - ◆ Click Delta Time icon to zoom 
- ⦿ Fast zoom on full-scale ruler: drag-left



Bus Value Settings

- ◉ Waveform → Signal Value Radix →
 - ◆ Binary
 - ◆ Octal
 - ◆ Hexadecimal
 - ◆ Decimal
 - ◆ ASCII
 - ◆ Alias
- ◉ Waveform → Signal Value Notation →
 - ◆ Unsigned
 - ◆ Signed 2's Complement
 - ◆ Signed 1's Complement
 - ◆ Signed Magnitude

1
10
100

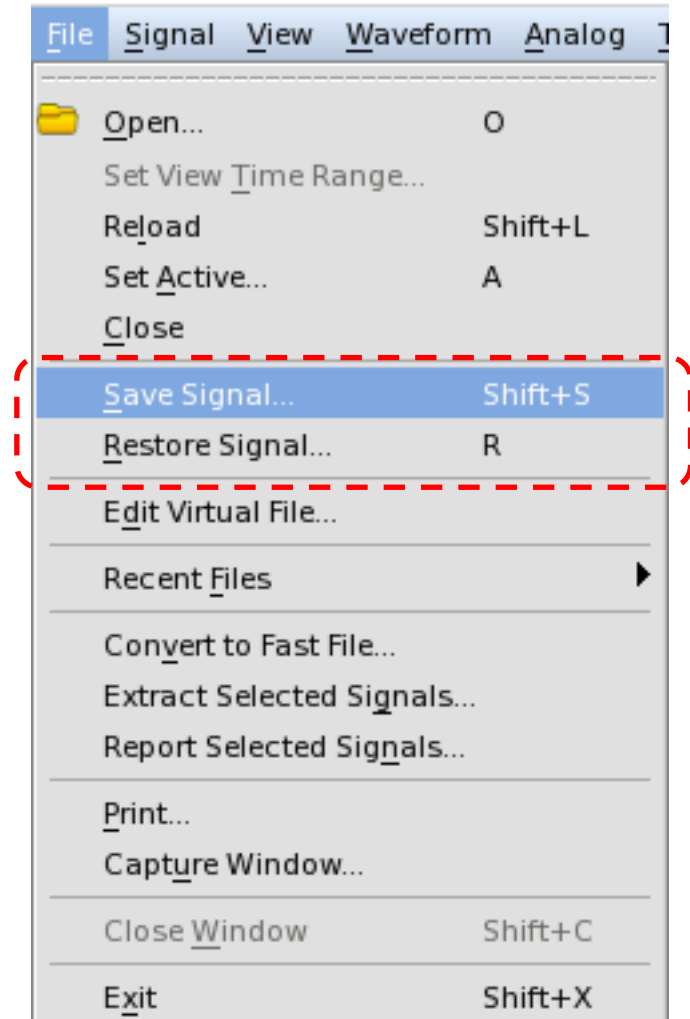
"leading zero"



001
010
100

Save/Restore the Signals

- You can save the signal view into a file (*.rc)
- So, you can restore it next time
- Command line argument
`nWave -ssr wave.rc`



Summary

⊙ Tool Environment

- ◆ NCVerilog for IC design
- ◆ Vivado for FPGA

structural
dataflow

⊙ Verilog design flow

- ◆ Design
- ◆ Coding
- ◆ Simulation
 - ▣ Testbench
 - ▣ Debugging from waveform and signal values
- ◆ Synthesis