

Computer Architecture Lab

Lab02

About Verilog Language

- A hardware description language that provides a means of specifying a digital system at a wide range of levels of abstraction
(Behavioral level, Register-Transfer level, Gate level)
- Supports the early conceptual stages of design with its behavioral level of abstraction and the later implementation stages with its structural level abstraction
- Provides hierarchical constructs, allowing the designer to control the complexity of description

History of Verilog

- 1983 : Developed by Gateway Design Automation based on the features of "HiLo" and "C language", which are hardware technical languages
- 1991 : Cadence Design Systems formed an organization called Open Verilog International (OVI) and released Verilog HDL
- 1993 : IEEE Working Group is configured for standardization
 - ✓ 1995 / December : IEEE Std. standardized to 1364-1995
 - ✓ 2001 : IEEE Std. revised to 1364-2001
- System Verilog, an extension of Verilog HDL, has been developed to drive IEEE standardization

Component of Verilog Module

Module Module_name(Port_list)

port declarations (if ports are present)

parameters(optional)

Data type declarations

Continuous Assignments (assign)

Procedural Blocks (initial and always)

- behavioral statements

Instantiation of lower-level modules

Task and Functions

endmodule

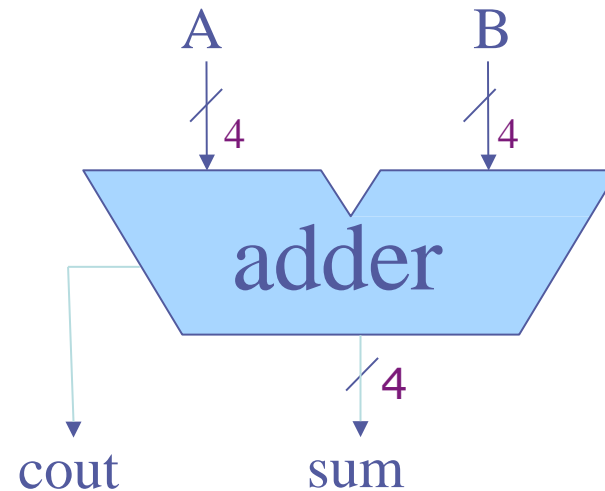
Module

```
module <module name>(<port list> )  
...  
endmodule
```

- Concept such as Function in C language
- Case sensitive (ex. Module \neq module)
- Reserved words can only be in lowercase letters
- Module name can be alphabetic, numeric, underbar(_) only
- The end of a sentence is always a semicolon(;)
- Reserved words that begin with End do not use semicolon(;) (ex. endmodule)

Ports

- Port List:
 - ✓ A listing of the port names
 - ✓ Example:
 module ANDTest(a, b, c);
- Port Types:
 - ✓ input : input port
 - ✓ output : output port
 - ✓ inout : bidirectional port
- Port Declarations:
 - ✓ <port_type> <port_name>;
 - ✓ Example :
 input [7:0] ina, inb;
 input clk, clr;
 output [15:0] out;



```
module adder(input  [3:0] A,  
             input  [3:0] B,  
             output  cout,  
             output [3:0]  
             sum );  
    // HDL modeling of  
    // adder functionality  
endmodule
```

Basic Syntax

- begin ~ end
 - Use when using initial, if, case, always etc. or when specifying block
 - Concept such as braces in C language
- Comment
 - // : Use to comment one line
 - /* ~ */ : Use to comment multiple lines
- Numerical representation
 - <Bit size>'<Radix><Value> (ex. 16'hFFFF)
 - <Radix> : b – binary, d – decimal, h – hexadecimal
 - Use 2's complement for negative numbers
 - Underbar(_) available for readability (ex. 2'b1100_0101)


Value Set

- Bit-vector is the only data type in Verilog

A bit can take on one of four values

Value	Meaning
0	Logic zero
1	Logic one
X	Unknown logic value
Z	High impedance, floating

In the simulation waveform viewer,
Unknown signals are **RED**.
There should be no red after reset



An X bit might be a 0, 1, Z, or in transition. We can set bits to be X in situations where we don't care what the value is. This can help catch bugs and improve synthesis quality.

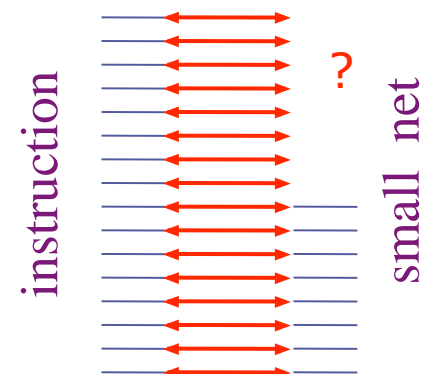
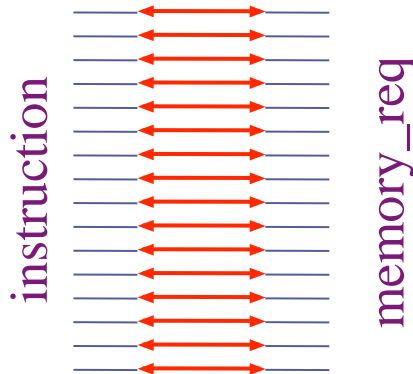
Vector

- Also known as multi-bit or bus
- Reserved word [MSB:LSB] name
- Reserved word : input, output, reg, wire
- “wire” is used to denote a hardware net
wire [MSB:LSB] <signal name>;

```
input [3:0] A, B;  
input [7:0] ABUS;  
wire [2:0] state;  
wire [2:0] H_digit;
```

```
wire [15:0] instruction;  
wire [15:0] memory_req;  
wire [7:0] small_net;
```

Absolutely no type safety when
connecting nets!



Verilog Registers “reg”

- Wires are line names – they cannot represent storage and can be assigned only once
- Regs can be assigned multiple times and holds values between assignments
- Regs are imperative variables (as in C):
 - ✓ Define storage, but not necessarily a data latch
 - ✓ Can only be changed by assigning value to them
 - ✓ reg [MSB:LSB] <signal name>;
BUT, “reg” is a variable ONLY for simulator
 - ✓ Although “reg” is not a hardware register, you must use “reg” to describe a hardware storage

Assign

- assign statement
 - ✓ Use to allocate values in combination logic
 - ✓ Signal that receives value with assign statement must be declared "wire"
 - ✓ assign statement uses the symbol "="
 - ✓ Examples:

```
wire [7:0] Y;  
assign Y = A + B; // Explicit assignment  
wire [1:0] a = x & y; // Implicit assignment  
assign o = ~((a & b) | c ^ d);  
assign w = (r == 0) ? 1'b1 : 1'b0;
```

Combinational logic's always statement

- Combinational logic's always statement
 - ✓ Use "always" statements when you want to use certain conditional statements in a combinational logic
 - ✓ Signal that receives value with "always" statement must be declared "reg"
 - ✓ 1 signal must not be changed in more than 1 "always" statement
 - ✓ Event list requires all signal values that affect the final value(or use "*")

```
always @ (event_list)
    [statement]
```

```
always @ (event_list) begin
    [multiple statements]
end
```

```
always @ (x or y or sel)
begin
    m = 0;
    if (sel == 0) begin
        m = x;
    end else begin
        m = y;
    end
end
```

if & case

➤ if statement

- ✓ Only available in "always" statement
- ✓ If the sentence is more than 2 lines, it must be bound with "begin ~ end"
- ✓ Same as "if ~ else" in C language

```
if(expression)
    statement 1;
else
    statement 2;
```

➤ case statement

- ✓ Only available in "always" statement
- ✓ If the sentence is more than 2 lines, it must be bound with "begin ~ end"
- ✓ Same as "switch ~ case" in C language

```
case(expression)
    condition 1: statement 1;
    condition 2: statement 2;
    ...
    default : statement N;
endcase
```

- ## ➤ Do not forget "else" or "default" statements when using "if" or "case" statements in a combinational logic

mux4: Using continuous assignments

```
module mux4( input a, b, c, d,  
             input [1:0] sel,  
             output out );
```

```
    wire out, t0, t1;
```

```
    assign out = ~((t0 | sel[0]) & (t1 | ~sel[0]) );  
    assign t1   = ~((sel[1] & d) | (~sel[1] & b) );  
    assign t0   = ~((sel[1] & c) | (~sel[1] & a) );  
endmodule
```

Language defined
operators



The order of these continuous assignment statements
does not matter

They essentially happen in parallel!

mux4: Behavioral style

```
// Four input multiplexer
module mux4( input    a, b, c, d
             input [1:0] sel,
             output out );

assign out = ( sel == 0 ) ? a :
             ( sel == 1 ) ? b :
             ( sel == 2 ) ? c :
             ( sel == 3 ) ? d : 1'bx;

endmodule
```

If input is undefined
we want to propagate
that information

- highest level of abstraction
- design algorithm without concern for the hardware implementation details

mux4: Using “always block”

```
module mux4( input a, b, c, d,
             input [1:0] sel, output out );

    reg out, t0, t1;

    always @( * ) begin
        t0 = ~( (sel[1] & c) | (~sel[1] & a) );
        t1 = ~( (sel[1] & d) | (~sel[1] & b) );
        out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
    end

endmodule
```

The order of these procedural assignment statements DOES matter. They essentially happen sequentially!

"Always blocks" with IF or CASE

```
module mux4( input    a,b,c,d,
             input [1:0] sel,
             output out );

    reg out;

    always @( * ) begin
        if ( sel == 2'd0 )
            out = a;
        else if ( sel == 2'd1 )
            out = b;
        else if ( sel == 2'd2 )
            out = c;
        else if ( sel == 2'd3 )
            out = d;
        else
            out = 1'bx;
    end
endmodule
```

```
module mux4( input    a,b,c,d,
             input [1:0] sel,
             output out );

    reg out;

    always @(      * ) begin
        case ( sel )
            2'd0 : out = a;
            2'd1 : out = b;
            2'd2 : out = c;
            2'd3 : out = d;
            default : out = 1'bx;
        endcase
    end
endmodule
```

Typically, we will use always blocks only to describe sequential circuits

Module's port mapping (1/2)

- Module's call
 - ✓ Concepts such as calling other functions in the main() function in C language
 - ✓ Use to include top module for testbench validation
 - ✓ A module called from a testbench or particular module is called a submodule
 - ✓ Always use instance name when calling submodule

ex)

```
module tb_test();  
    ...  
    test test_inst(<port_list>);  
endmodule
```

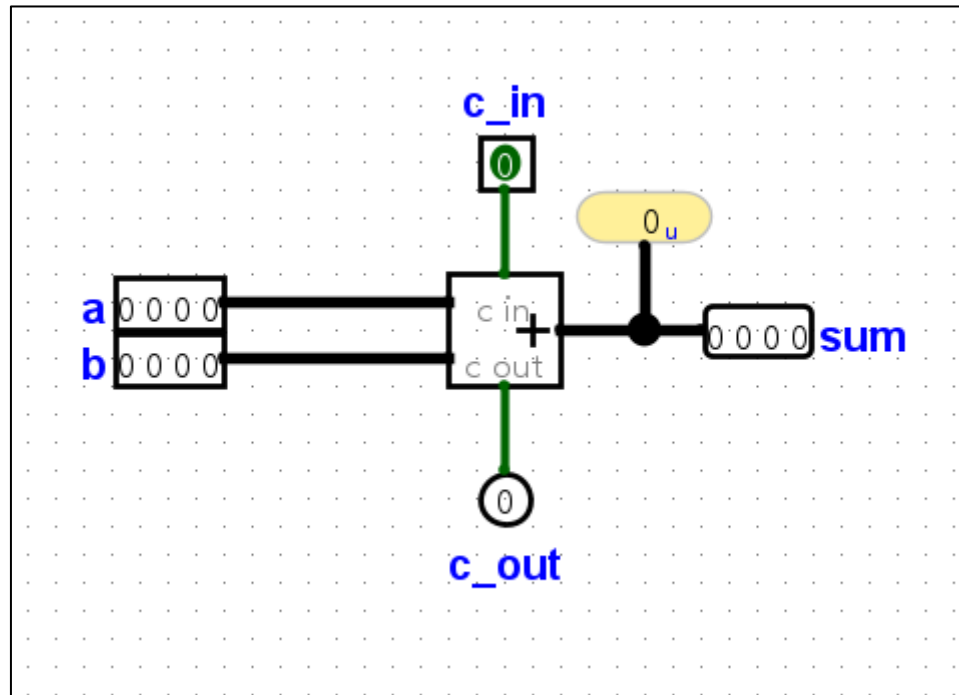
Module's port mapping (2/2)

- Module's input, output
 - ✓ Port list of the module is regardless of the order of input and output (generally listed in order of input port and output port)
 - ✓ Input signal cannot change value within module
 - ✓ All signals defined as output must be redefined with "reg" or "wire" (Output signal must generate a value from the current module)

- Module's internal signal
 - ✓ Generate when values need to be stored within module or when values need to be connected between module and module
 - ✓ Use "reg" declaration when it is necessary to generate a value using "always" statement
 - ✓ Use "wire" declaration to generate a values using "assign" or to connect values between 2 different submodules

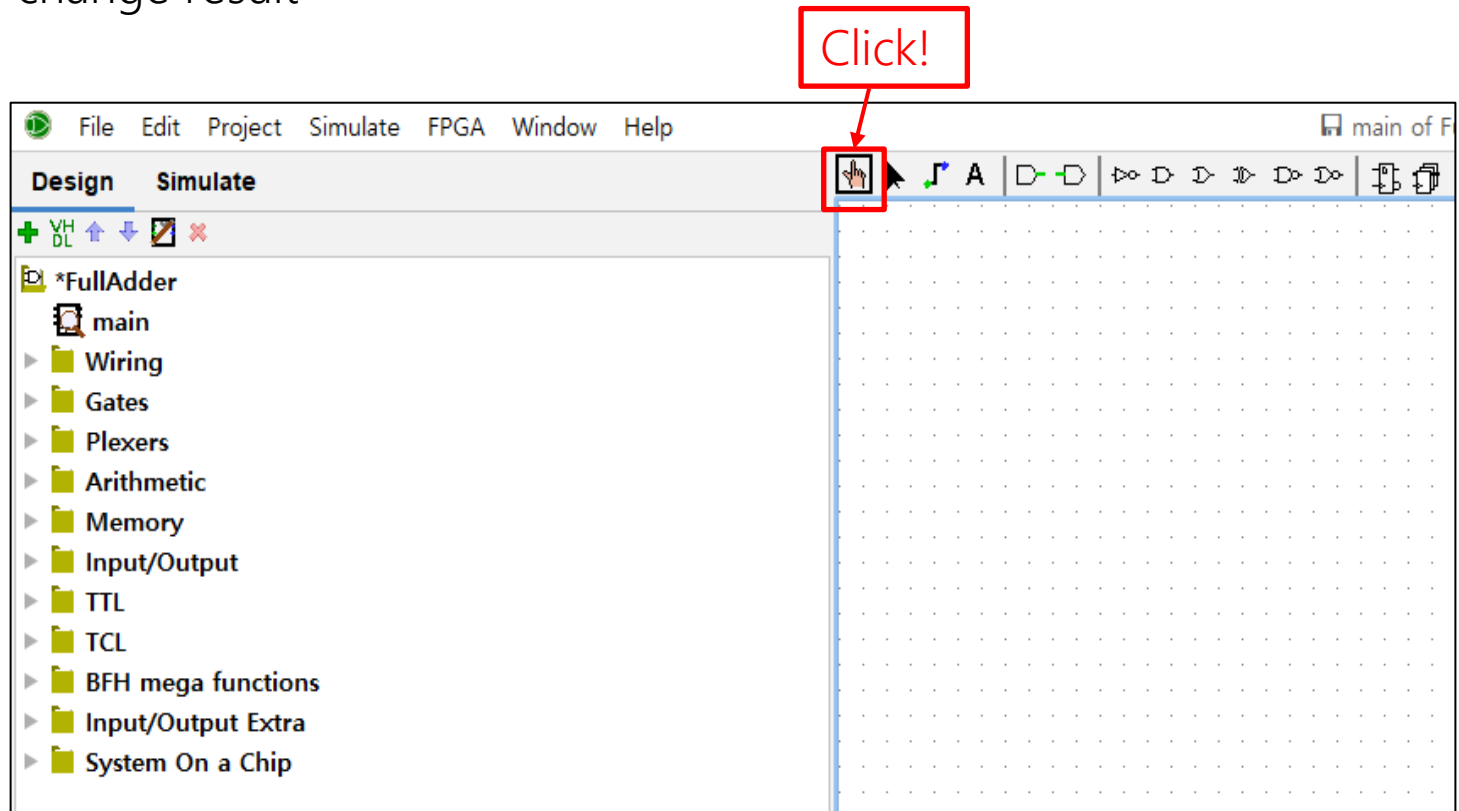
Example) Full Adder (1/11)

- Logisim – evolution



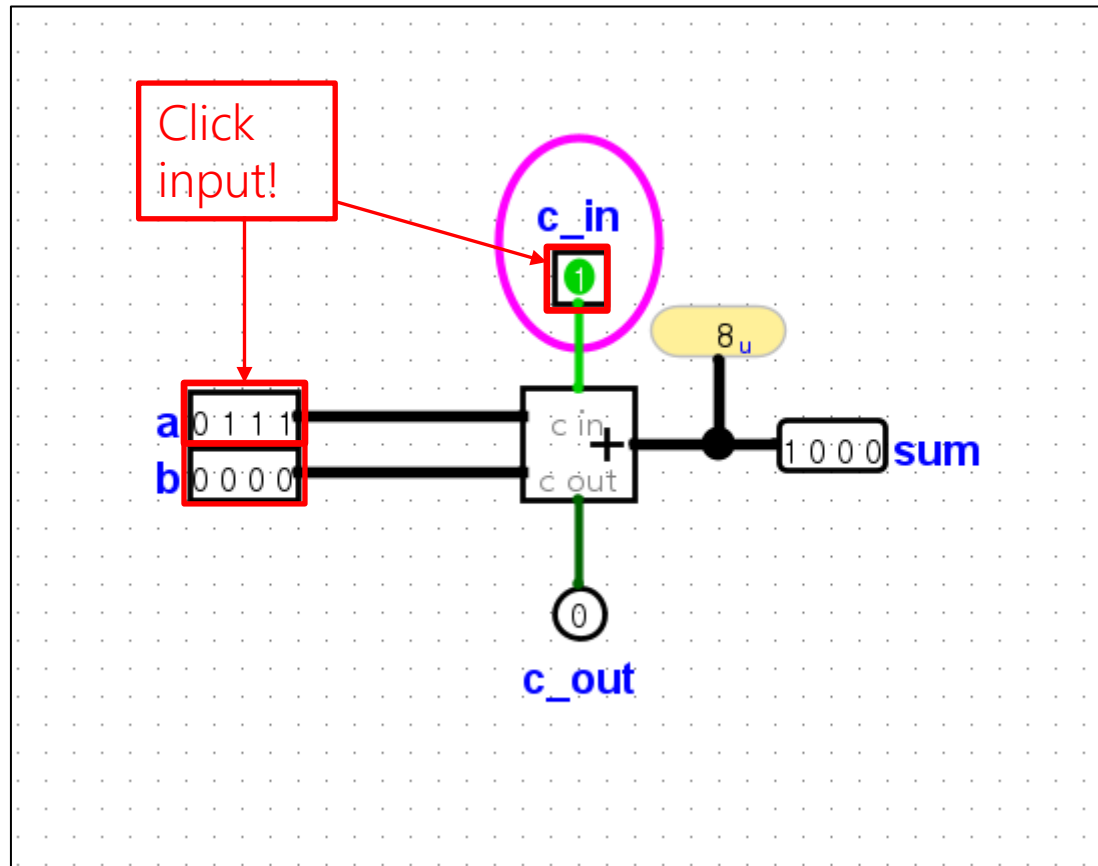
Example) Full Adder (2/11)

- Logisim – evolution
 - change result



Example) Full Adder (3/11)

- Logisim – evolution
 - change result



Example) Full Adder (4/11)

➤ Verilog code

```
FullAdder.v
1 module FullAdder(a,b,c_in,c_out,sum);
2     input a;
3     input b;
4     input c_in;
5     output c_out;
6     output sum;
7
8     assign {c_out, sum} = a + b + c_in;
9 endmodule
```

Example) Full Adder (5/11)

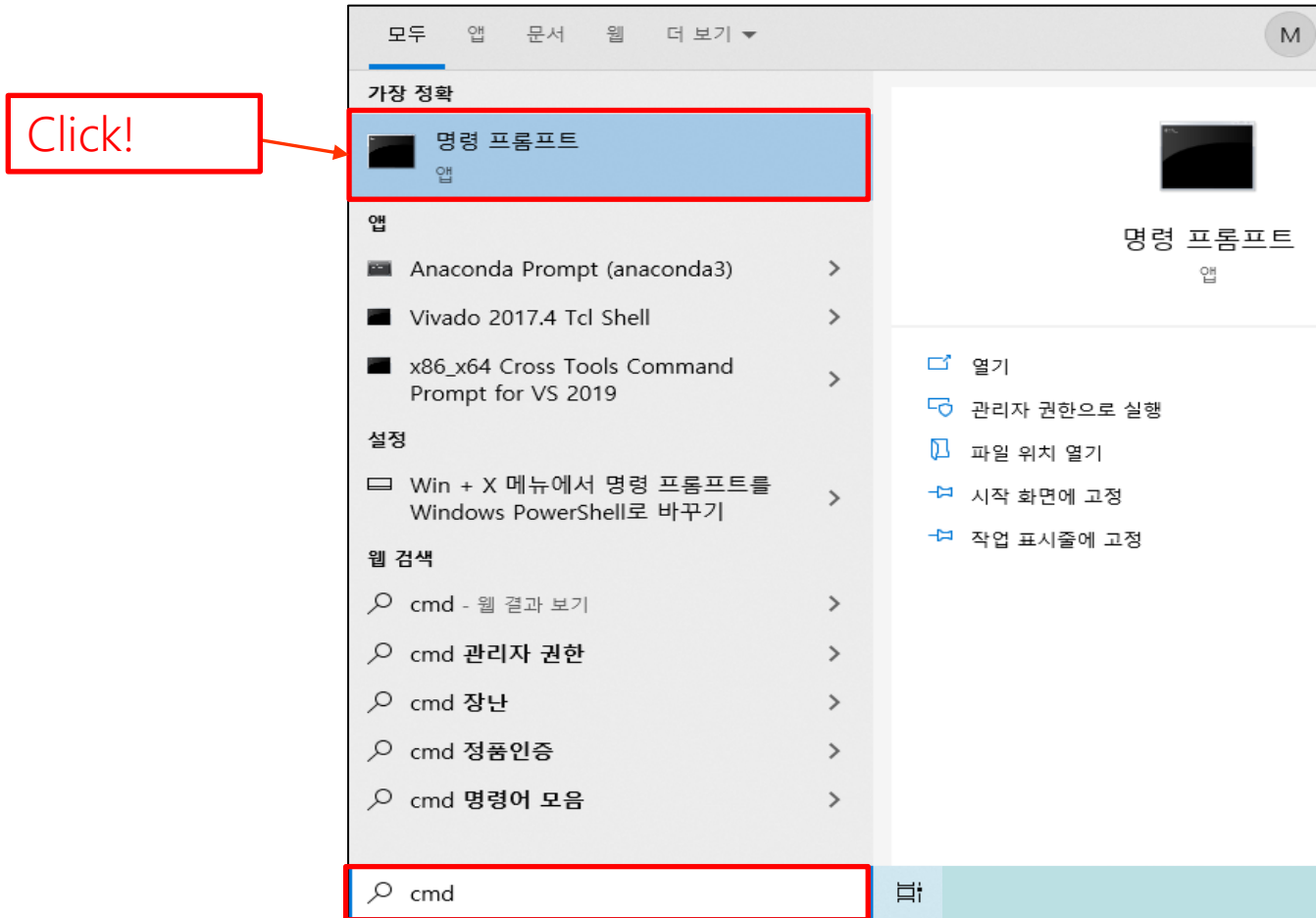
➤ Test bench

- ✓ The waveform file must have a test bench
- ✓ Testbench.v will be provided in this experiment

```
tb_FullAdder.v
1  `timescale 1ns/1ps
2  `include "FullAdder.v"
3  module tb_FullAdder;
4      reg a;
5      reg b;
6      reg c_in;
7      wire sum;
8      wire c_out;
9
10
11  FullAdder uut(
12      .a(a), .b(b), .c_in(c_in), .sum(sum), .c_out(c_out)
13  );
14
15  initial
16  begin
17      $dumpfile("tb_FullAdder.vcd");
18      $dumpvars(0,tb_FullAdder);
19
20      a = 0;
21      b = 0;
22      c_in = 1;
23      #10;
```

```
24
25      a = 1;
26      b = 0;
27      c_in = 0;
28      #10;
29
30      a = 1;
31      b = 1;
32      c_in = 0;
33      #10;
34
35      a = 1;
36      b = 0;
37      c_in = 1;
38      #10;
39
40      $display("Test complete");
41  end
42
43 endmodule
```


Example) Full Adder (6/11)



Example) Full Adder (7/11)

1. cd + file path with code

```
C:\> 명령 프롬프트
Microsoft Windows [Version 10.0.19044.1526]
(c) Microsoft Corporation. All rights reserved.
C:\Users\mplab>cd C:\Users\mplab\verilog
C:\Users\mplab\verilog>_
```

2. compile .o file

```
C:\Users\mplab\verilog>iverilog -Wimplicit -o tb_FullAdder.o tb_FullAdder.v
C:\Users\mplab\verilog>_
```

3. make .vcd file

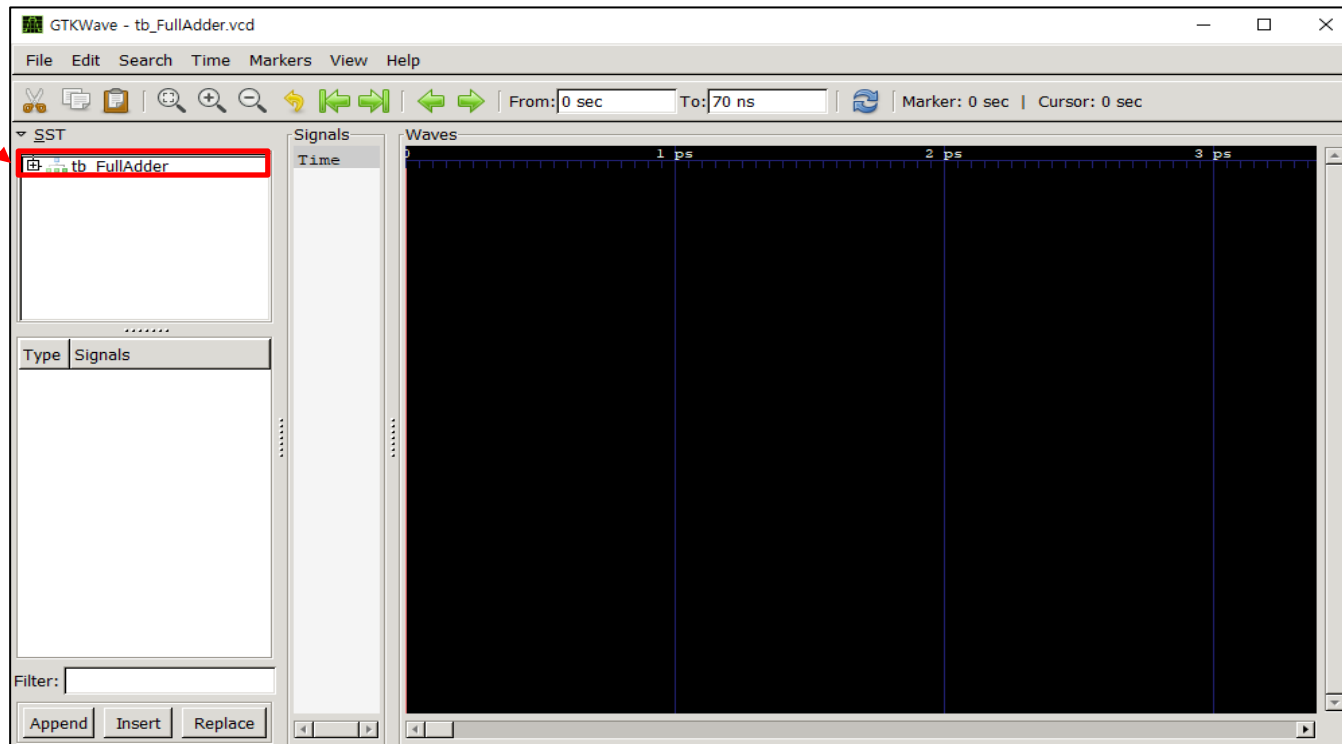
```
C:\Users\mplab\verilog>vvp tb_FullAdder.o
VCD info: dumpfile tb_FullAdder.vcd opened for output.
Test complete
C:\Users\mplab\verilog>vvp tb_FullAdder.o -fst
FST info: dumpfile tb_FullAdder.vcd opened for output.
Test complete
```

Example) Full Adder (8/11)

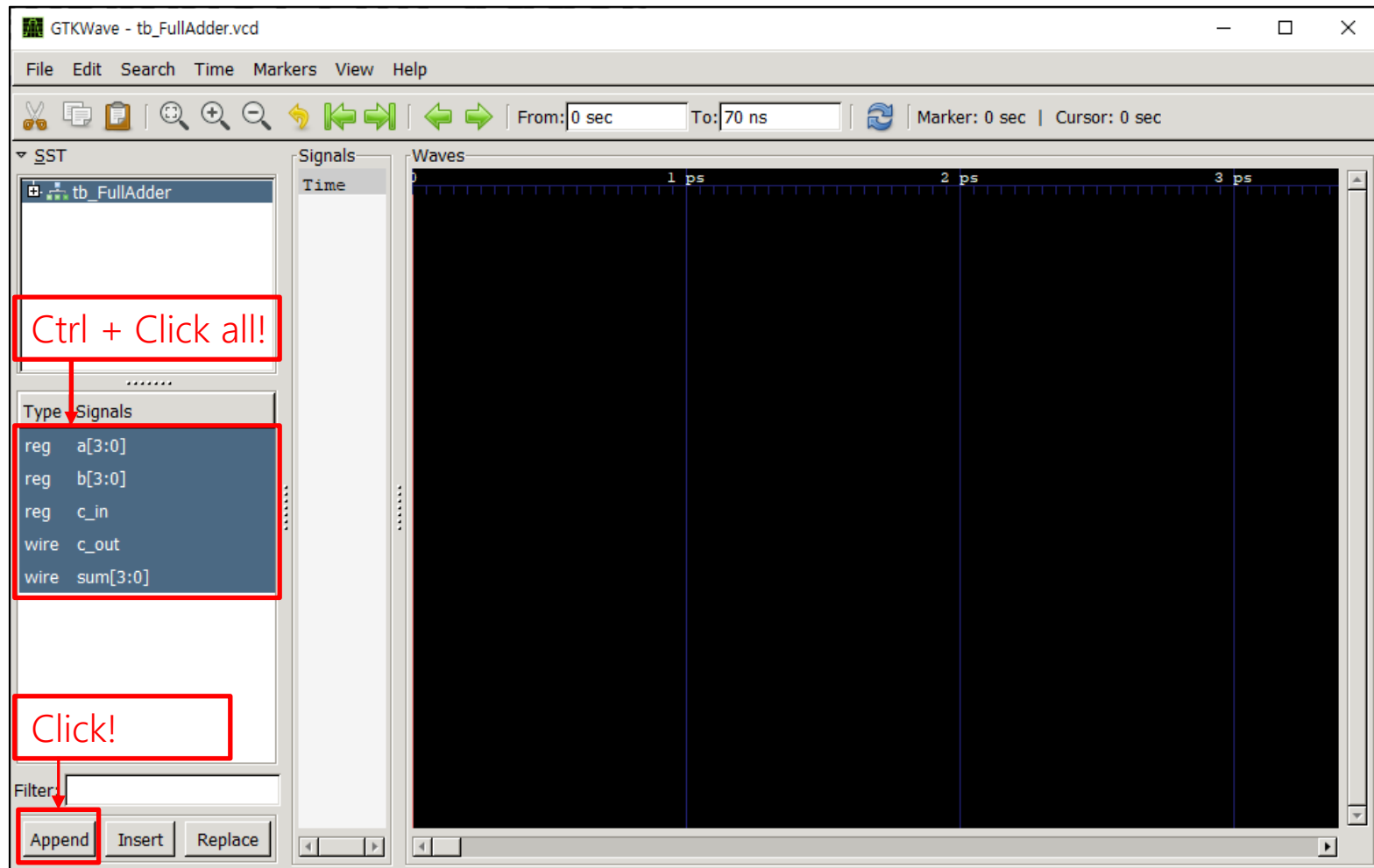
- Execute gtkwave

```
C:\Users\mplab\verilog>gtkwave tb_FullAdder.vcd  
GTKWave Analyzer v3.3.100 (w)1999-2019 BSI  
  
FSTLOAD | Processing 10 facs.  
FSTLOAD | Built 8 signals and 2 aliases.  
FSTLOAD | Building facility hierarchy tree.  
FSTLOAD | Sorting facility hierarchy tree.
```

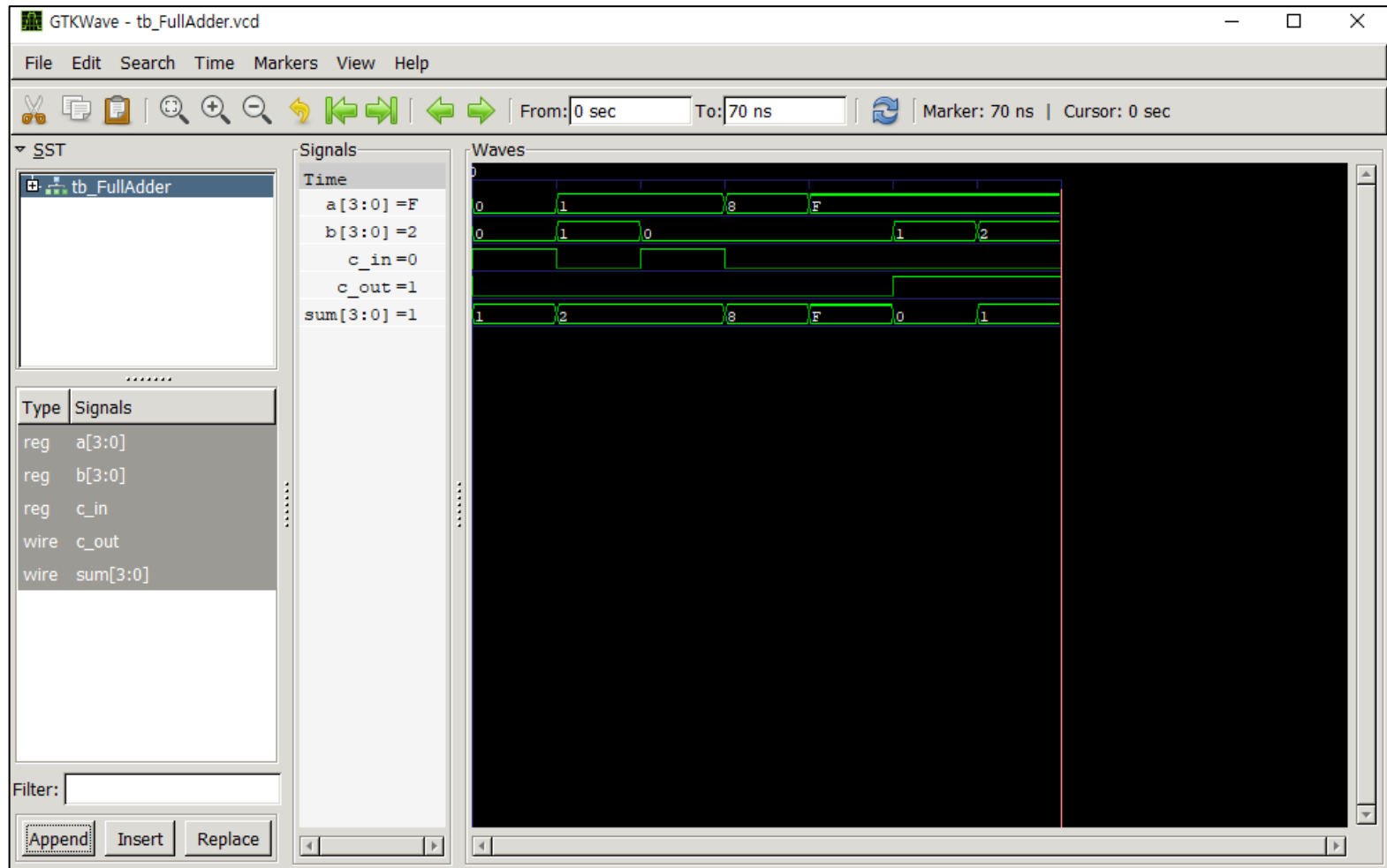
Click!



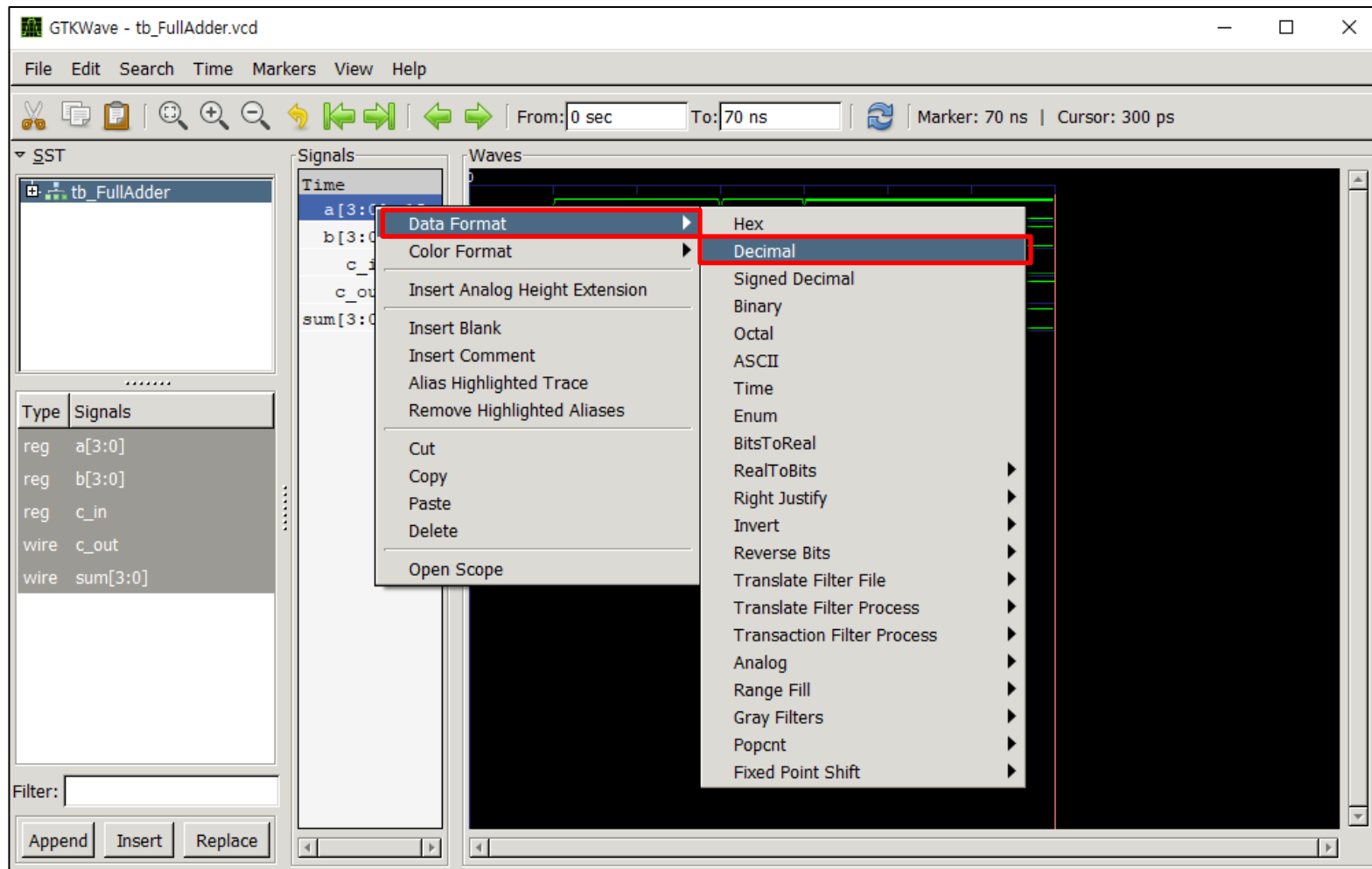
Example) Full Adder (9/11)



Example) Full Adder (10/11)

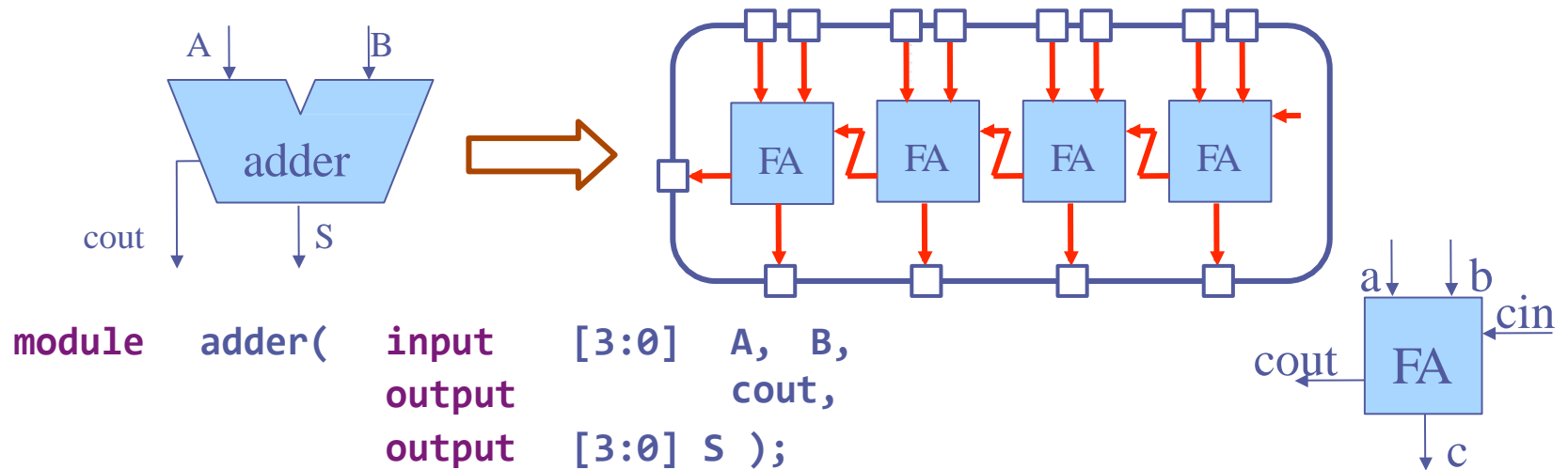


Example) Full Adder (11/11)



A module can instantiate other modules

- Higher-level designs instantiate lower-level designs
- Applies to both schematic designs and HDL designs



```
wire c0, c1, c2;  
FA fa0( ... );  
FA fa1( ... );  
FA fa2( ... );  
FA fa3( ... );  
  
endmodule
```

```
module FA( input a, b, cin  
           output cout, sum );  
    // HDL modeling of 1 bit  
    // full adder functionality  
endmodule
```

Thank You