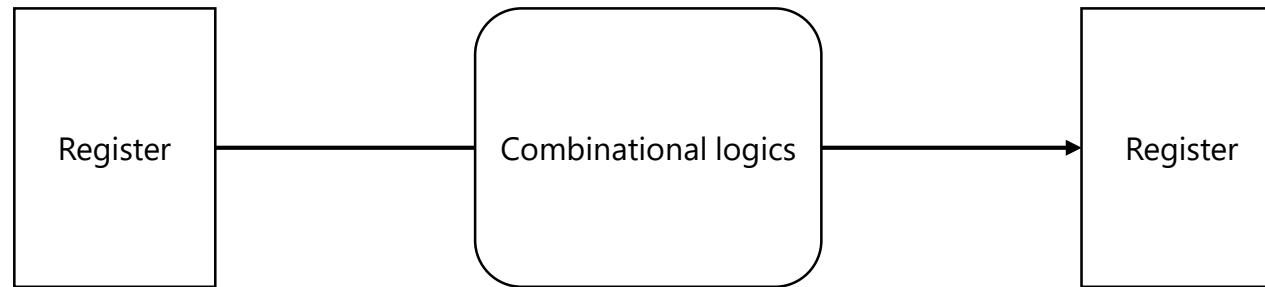


Introduction to Verilog

Sungjun Cho

What is Verilog?

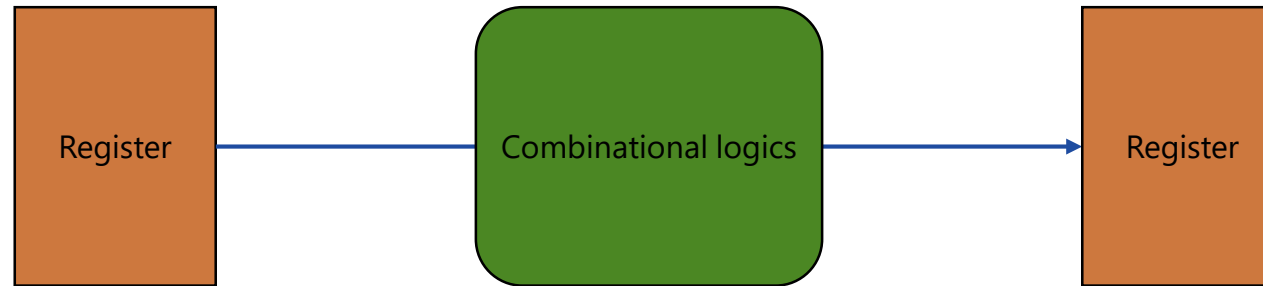
- One of **Hardware Description Languages (HDL)**
 - Describe how hardware is constructed
 - Verilog implements **register-transfer-level (RTL)** abstractions



Describe how information flows **from register to register**

What is Verilog?

- One of **Hardware Description Languages (HDL)**
 - Describe how hardware is constructed
 - Verilog implements **register-transfer-level (RTL)** abstractions

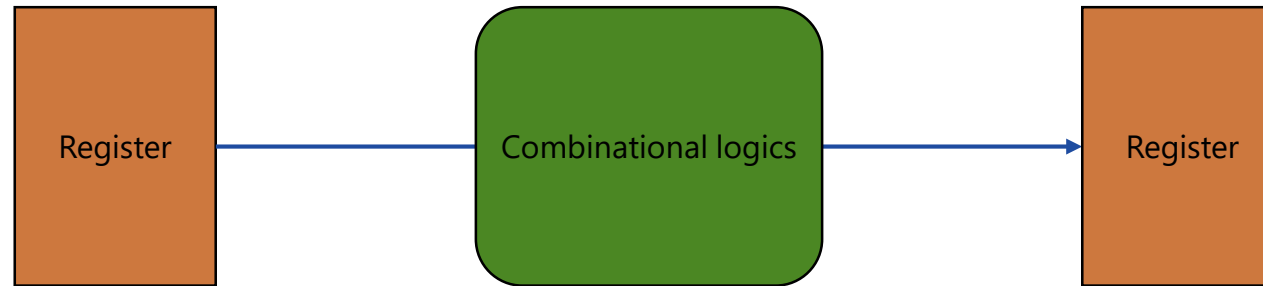


Describe how information flows **from register to register**

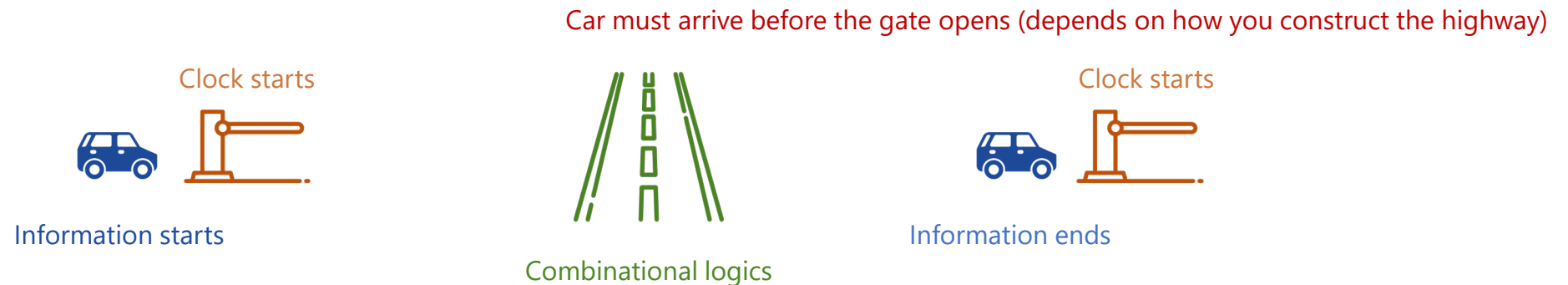


What is Verilog?

- One of **Hardware Description Languages (HDL)**
 - Describe how hardware is constructed
 - Verilog implements register-transfer-level (RTL) abstractions



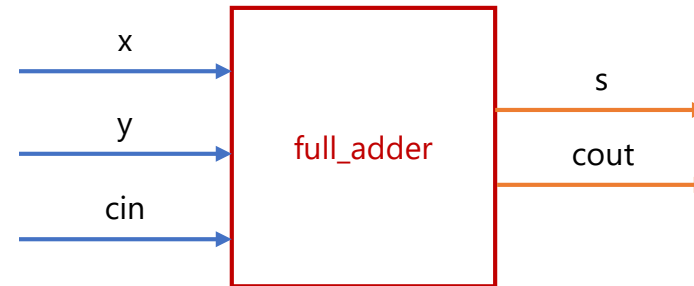
Describe how information flows from register to register



Verilog module

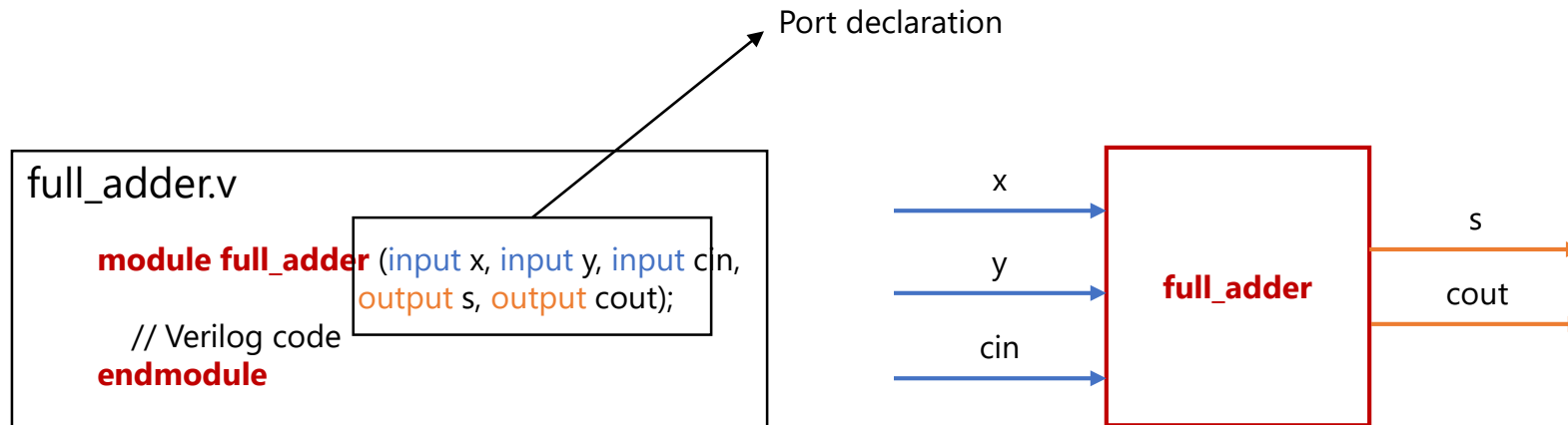
- **Modules are the building blocks of Verilog designs**
 - Similar to functions in the programming language
- **Modules are defined by port declarations (I/O) and verilog code (implementation)**
 - Port declarations => similar to function arguments in the programming languages
 - Verilog code => functionality
- **Currently, recommend to write a module in a file ended with .v**
 - Modularization

```
full_adder.v  
  
module full_adder (input x, input y, input cin,  
                  output s, output cout);  
    // Verilog code  
endmodule
```



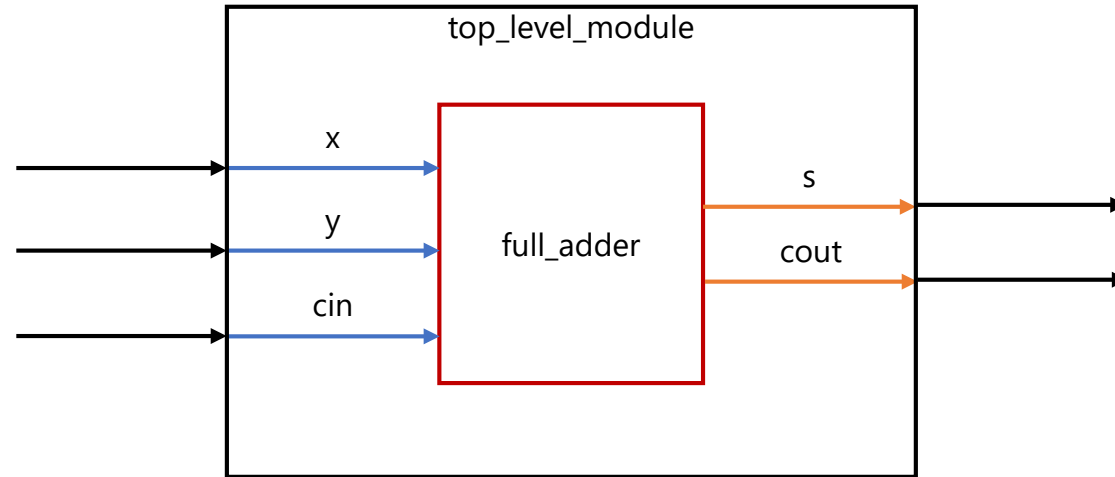
Verilog module I/O ports

- **Ports are the interface between a module and its environment**
 - Environment
 - Something that generates information flow that goes to the module
 - Something that receives the information flow generated by the module
- **Each port has a name and a type**
 - input
 - output
 - inout
 - output reg



Top-level module

- **Every Verilog design has a top-level module**
 - The highest level of the design hierarchy => Similar to the **main function** in the programming languages
- **Modules defined by the designer are instantiated within the top-level module**
 - General modules also can instantiate other modules



Module instantiation

- You have two files

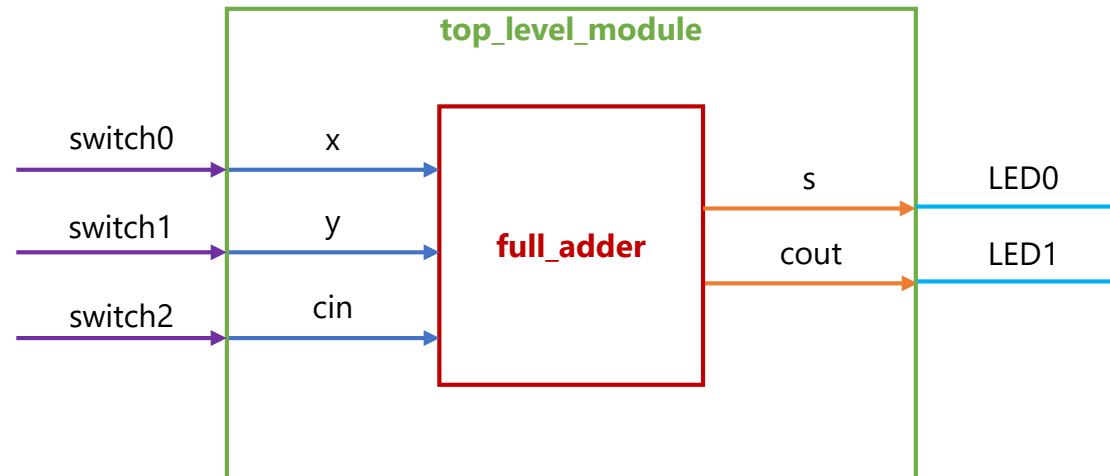
- top_level_module.v
- full_adder.v

full_adder.v

```
module full_adder (input x, input y, input cin,  
                  output s, output cout);  
    // Verilog code  
endmodule
```

top_level_module.v

```
module top_level_module (input switch0, input switch1, input switch2,  
                        output LED0, output LED1);  
  
    // Verilog code  
    // instantiate full_adder()  
    // module_name instance_name  
    full_adder my_adder(  
        .x(switch0),  
        .y(switch1),  
        .cin(switch2),  
        .s(LED0),  
        .cout(LED1));  
  
endmodule
```



Wire

- **Wires (also called nets) are analogous to wires in a circuit**
 - Wire transmits values between inputs and outputs

```
wire a; // 1 bit wire  
wire b; // 1 bit wire
```

- **Vector (multiple bit widths) wires**
 - Vectors can be declared at [high# : low#] or [low# : high#]
 - The left number is always MSB of the vector => [MSB bit index : LSB bit index]

```
wire [31 : 0] a; // 32 bit wire  
wire [63 : 0] b; // 64 bit wire (Bit at 63 is MSB)  
wire [0 : 63] c; // 64 bit wire (Bit at 0 is MSB)
```



Wire

- To use vector

module two_t

// Implement

endmodule

```
TEST                                WaitReturnTest :
Current      2800
o_return_coin 000
o_return_coin 0
o_return_coin 0
Current      1800
o_return_coin 100
o_return_coin 4
o_return_coin 4
Current      800
o_return_coin 100
o_return_coin 4
o_return_coin 4
FAILED
-200

TEST                                TriggerReturnTest :
Current      4600
o_return_coin 100
o_return_coin 4
o_return_coin 4
Current      3600
o_return_coin 100
o_return_coin 4
o_return_coin 4
Current      2600
o_return_coin 100
o_return_coin 4
o_return_coin 4
Current      1600
o_return_coin 100
o_return_coin 4
o_return_coin 4
Current      600
o_return_coin 100
o_return_coin 4
o_return_coin 4
FAILED
-400
Passed = 21, Failed = 2
1
Break in Module vending_machine_tb at C:/intelFPGA_pro/21.2/vending_machine_tb.v line 115
```

Reg

- **Reg is required whenever the state (or value) must be preserved.**
 - `reg [31 : 0] x; // 32-bit reg`
 - `reg x; // 1-bit reg`
- **Reg is used to assign the value in an `always block (procedural assignment)`.**
 - Discussed later

Array

- **Wires and regs can be declared as an array**

- An array of wire

`wire [7 : 0] wire_array [5 : 0];` // declare an array of 8-bit vector wire. The length of array is 6.

- An array of reg (Note that this usually represents the memory!)

`reg [31 : 0] memory [0 : 1023];` // declare an array of 32-bit vector reg. The length of array is 1024.

- To access an array:

`memory[15][23 : 0] = 0;` // [15] represents the index of an array. [23 : 0] accesses the bits from 23 to 0.

- **An array also can be declared as a multi-dimensional array**

- E.g., `reg [31 : 0] memory [0:1024][0:512][0:256]` // three-dimensional array

Verilog literals

- **Syntax: [bit width]'[radix][literal]**
 - Radix can be **d** (decimal), **h** (hexadecimal), **o** (octal), **b** (binary)
 - **2'd1** : 2-bit literal (decimal 1)
 - **16'hAD14** : 16-bit literal (hexadecimal 0xAD14)
 - **8'b01011010** : 8-bit literal (binary 0b01011010)
 - **8{4'b0101}** : 32-bit literal (replicating 4'b0101) (binary 0b01010101...0101)

Verilog macros

- **Macros in Verilog are similar to macros in C**
- **``include`**
 - Include a Verilog source file at specified location
- **``define <constant name> <constant value>`**
 - Declare a synthesis-time constant
 - To use defined value: ``<constant name>`

constants.v

```
`ifndef CONSTANTS_V
`define CONATANTS_V

`define ADDR_BITS 16
`define NUM_WORDS 32
`endif
```

design.v

```
`include "constants.v"

module memory(input [`ADDR_BITS - 1 : 0] addr,
               output ...);
    // implementation
endmodule
```

Verilog module parameterization

- Similar to macros, verilog provides a way to declare constant parameter **for the module**.
 - Useful to define the width of bus or others

```
module adder #(parameter data_width = 32) (  
    input [data_width - 1 : 0] a,  
    input [data_width - 1 : 0] b,  
    output [data_width : 0] c);  
    assign c = a + b;  
endmodule
```

```
module top();  
    localparam adder1width = 64; // can be used only for the module, top()  
    localparam adder2width = 32; // can be used only for the module, top()  
    reg [adder1width - 1 : 0] a, b;  
    reg [adder2width - 1 : 0] c, d;  
    wire [adder1width : 0] out1;  
    wire [adder2width : 0] out2;  
    adder #(.data_width(adder1width)) adder64 (.a(a), .b(b), .s(out1));  
    adder #(.data_width(adder2width)) adder32 (.a(c), .b(d), .s(out2));  
endmodule
```

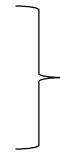
Modeling

- **Three ways to design the hardware:**

- Gate-Level modeling => low-level modeling (painful)

- Dataflow modeling

- Behavioral modeling



The term RTL design is used for a combination of these two.

- **We will learn dataflow modeling and behavioral modeling**

Dataflow modeling

- **Continuous (dataflow) assignment**

- Drive a value onto a net (wire or reg)
- A simple and natural way to **represent combinational logic**
- Declare before use

```
module example (input [1 : 0] x, input [1 : 0] y,  
                output [1 : 0] z, output [2 : 0] f)  
    // assign z = x & y;  
    // assign f = x + y; this implements two bits adder  
endmodule
```

```
module example (input [1 : 0] x, input [1 : 0] y, input a,  
                output [1 : 0] z)  
    // assign z = a ? x : y; conditional operator (multiplexer)  
endmodule
```

```
wire [7 : 0] a; // 8 bit wire  
wire [31 : 0] b; // 32 bit wire  
wire [31 : 0] c; // 32 bit wire  
wire [5 : 0] d; // 6 bit wire
```

```
// assign d = a[5 : 0]; // bit slicing  
// assign c = {a, b[25: 2]}; // concatenation + bit slicing  
// assign c = { 32{d[2]} }; // replicating d (bit at 3) 32 times => 32 bits  
// assign {a[5], b[2 : 1]} = c[23:21]; // assign the sliced value to concatenated net
```

```
module example (input [1 : 0] x, input [1 : 0] y, input a,  
                output [1 : 0] z)  
    // Assignment 1  
    // wire [1 : 0] out;  
    // assign out = x | y;  
    // assign z = out;  
  
    // Assignment 2, directly assign the input to the wire  
    // wire [1 : 0] out = x | y;  
    // assign z = out;  
  
endmodule
```

Bits are truncated if not matched or filled with zeros

- Why 'continuous' ?
 - Right-hand expression is **continuously** evaluated whenever the values are changed

Behavioral modeling

- **Two structured procedure statements in verilog**
 - always
 - Initial
- **Initial statements are executed only once at the beginning of simulation.**
 - Initial statement is not synthesizable, it only works for simulation.
 - Usually used to initialize values and used for test benches (e.g., reset signals)
- **Always statements are continuously executed.**
 - Used to model a block of activity that is repeated continuously in a digital circuit.
 - Do not think 'always' as 'while' loop in programming languages.
 - Codes in 'always' are repeated in a digital circuit until 'power off' occurs.

Behavioral modeling

- How to design clock generator by using **initial** and **always** statements

```
`timescale 1ns/10ps

module clock_generator (output reg clock);

    // initialize the value of 'clock'.
    // 'initial' statement is executed only at the beginning of the simulation (time 0).
    initial begin
        clock = 1'b0;
    end

    // toggle the value of 'clock' at every half-cycle (1 clock period = 20).
    // 'always' statement is continuously repeated.
    always begin
        #10 clock = ~clock;
        $display("current time: %d, value of the clock: %b", $time, clock)
    end

    // 'initial' statement is executed at the beginning of the simulation (time 0).
    // However, '$finish' is executed at time 1000.
    // '$finish' stops simulation.
    initial begin
        #1000 $finish;
    end

endmodule
```

Behavioral modeling

- **Timescale (`timescale)**

```
`timescale 1ns (time measurement) / 10ps (precision)
```

```
#2: 2 ns (o)
```

```
#2.2: 2.2 ns (o)
```

```
#2.22: 2.22 ns (o)
```

```
#2.222: 2.222 ns (x): because precision can be rounded off up to 10 ps
```

- **Delay (#time)**

- Intra delay assignment
- Regular delay assignment
- **Block assignment (=): executed in the ordered of program code**
- **Non-blocking assignment (<=): executed concurrently**

Behavioral modeling

- **Delay (#time)**

- Intra delay assignment
- Inter delay assignment
- Block assignment (=): executed in the ordered of program code
- Non-blocking assignment (<=): executed concurrently

```
initial begin
  x = 0; y = 1; z = 1; // time 0
  count = 0; // time 0
  reg_a = 16'b0; reg_b = reg_a; // time 0

  #15 reg_a[2] = 1'b1; // time 15
  #10 reg_b[15:13] = {x, y, z} // time 25

  count = count + 1;
end
```

```
always begin
  x = 0; y = 1; z = 1; // time 0
  count = 0; // time 0
  reg_a = 16'b0; reg_b = reg_a; // time 0

  reg_a[2] <= #15 1'b1; // time 15
  reg_b[15:13] <= #10 {x, y, z} // time 10

  count = count + 1; // time 0
end
```

Behavioral modeling

- **Procedural assignment (updating values) with `always`**

- Always statement:

```
always @(sensitivity list) begin
    ...
end
```

- Sensitivity list: The list of events or signals expressed as an OR

- Combinational logic (asynchronous)

```
module comb_logic(input x, input y, output z);
```

```
    reg reg_z;
```

```
    assign z = reg_z;
```

```
    always @(x or y) begin
        reg_z = x & y;
    end
```

```
endmodule
```

Always statement is triggered
whenever one of x and y is changed.



```
module comb_logic(input x, input y, output z);
```

```
    reg reg_z;
```

```
    assign z = reg_z;
```

```
    always @(*) begin
        reg_z = x & y;
    end
```

```
endmodule
```

'*' implies x and y

'reg_z' must be declared by 'reg'

* lvalue inside always statement must be one of reg, integer, real, and time register

Behavioral modeling

- **Procedural assignment with `always`**

- Always statement:

```
always @(sensitivity list) begin
    ...
end
```

- Sensitivity list: The list of events or signals expressed as an OR

- Combinational logic (asynchronous)

```
module comb_logic(input sel, input a, input b,
                  output z);
```

```
    reg reg_z;
```

```
    assign z = reg_z;
```

```
    always @(*) begin
        if (sel)
            reg_z = a;
        else
            reg_z = b;
    end
```

```
endmodule
```

```
module comb_logic(input sel, input a, input b,
                  output z);
```

```
    reg reg_z;
```

```
    assign z = reg_z;
```

```
    always @(*) begin
        case (sel)
            0 : reg_z = b;
            1 : reg_z = a;
            default: reg_z = 1'b1
        end
```

```
endmodule
```

Be careful! Default condition must exist (e.g., else in if-else, default in case)

Behavioral modeling

- **Procedural assignment with `always`**

- Always statement:

```
always @(sensitivity list) begin
    ...
end
```

- Sensitivity list: The list of events or signals expressed as an OR

- Synchronous logic

```
module counter(input clk);
```

```
    reg [31 : 0] counter;
```

```
    always @(posedge clk) begin
        counter <= counter + 1;
    end
```

```
endmodule
```

```
module swap_non_blocking(input clk, ...);
```

```
    always @(posedge clk) begin
        a <= b;
    end
```

```
    always @(posedge clk) begin
        b <= a;
    end
```

```
endmodule
```

1. What will be the type (wire or reg) of a and b?
2. Are always statements executed in the order?
3. What happen to a and b?

Behavioral modeling

- **Procedural assignment with `always`**

- Always statement:

```
always @(sensitivity list) begin
    ...
end
```

- Sensitivity list: The list of events or signals expressed as an OR

- Non-blocking vs blocking

```
module swap_non_blocking(input clk, ...);
```

```
    always @(posedge clk) begin
        a <= b;
    end
```

```
    always @(posedge clk) begin
        b <= a;
    end
```

```
endmodule
```

Non-blocking

```
module swap_blocking(input clk, ...);
```

```
    always @(posedge clk) begin
        a = b;
    end
```

```
    always @(posedge clk) begin
        b = a;
    end
```

```
endmodule
```

Blocking

If we want to make output b with synchronous logic,
We can declare b as output reg b
Because only reg can be lvalue

1. What happen to a and b?

Behavioral modeling

- **Procedural assignment with `always`**

- Always statement:

```
always @(sensitivity list) begin
    ...
end
```

- Sensitivity list: The list of events or signals expressed as an OR
- Synchronous write, asynchronous read memory

```
module mem(clk, wr_en, wr_addr, rd_addr, in_data, ret_data);
    localparam data_width = 512;
    localparam size = 1024;

    input clk, wr_en;
    input [size - 1] wr_addr;
    input [size - 1] rd_addr;
    input [data_width - 1] in_data;
    output [data_width - 1] ret_data;

    reg [data_width - 1 : 0] mem[size - 1 : 0];

    assign ret_data = mem[rd_addr]; // asynchronously read data
    always @(posedge clk) begin
        if (wr_en)
            mem[wr_addr] <= in_data; // synchronously write data
        end
    endmodule
```

wire vs reg

- **Rules for picking a wire or reg net type:**

- If a signal needs to be assigned inside an always block, it must be declared as a reg
- If a signal is assigned using a continuous assignment statement, it must be declared as a wire
- If any output ports in the port declaration are assigned in an always block, they must be declared as output reg
 - module a (input a, input b, output reg c); ... endmodule

- **How to know if a type of variable represents a register or a wire ?**

- A wire always represents a combinational link
- A reg represents a wire if it is assigned in an always @(*) block
- A reg represents a register if it is assigned in an always @(posedge / negedge sth) block

References

- https://inst.eecs.berkeley.edu/~eecs151/fa19/files/verilog/Verilog_Primer_Slides.pdf
- <http://courses.csail.mit.edu/6.111/f2004/handouts/L04.pdf>
- https://d1.amobbs.com/bbs_upload782111/files_33/ourdev_585395BQ8J9A.pdf