CSED311 Computer Architecture – Lab 0

Lab and Verilog Introduction

Gwangsun Kim

Department of Computer Science and Engineering
POSTECH

Lab Information

- TAs

Yunseon Shin (신윤선)
Head TA

Minsu Gong (공민수)

Jinhoon Bae (배진훈)

Okkyun Woo (우옥균)

- Use the group email alias (csed311-ta@postech.ac.kr) to contact TAs
- Use PLMS to submit the assignments and reports
- Use the Q&A board on PLMS to ask questions
  - General questions sent to the TA email will not be answered
  - Student-specific questions can be asked through the email

Lab Coverage

- Verilog HDL (Lab 0 ~ Lab 1) and tools

- CPU Design in Verilog (Lab 2 ~ Lab 5)
  - Register
  - Datapath
  - Control Unit
  - Pipelined CPU
  - Cache

POSTECH

Lab Schedule

- There will be a lab session in week 3 (although lectures will be rescheduled)

Week	Date	Topic
1	2/20	Lab 0: Lab intro and Verilog HDL
2	2/27	Lab 1a: ALU, Vending machine
3	3/5	Lab 1b:Vending machine
4	3/12	Lab 2a: Single-Cycle CPU (L1 demo)
5	3/19	Lab 2b: Single-Cycle CPU
6	3/26	Lab 3a: Multi-Cycle CPU (L2 demo)
7	4/2	Lab 3b: Multi-Cycle CPU
8	4/9	MID-TERM EXAM

Week	Date	Topic
9	4/16	Lab 4-1a: Pipelined CPU (L3 demo)
10	4/23	Lab 4-1b: Pipelined CPU
11	4/30	Lab 4-2a: Pipelined CPU with control flow (L4-1 demo)
12	5/7	Lab 4-2b: Pipelined CPU with control flow
13	5/14	Lab 5a: Cache (L4-2 demo)
14	5/21	Lab 5b: Cache
15	5/28	(L5 demo)
16	6/4	FINAL EXAM

Grading

- Labs account for 35% of grading in this course
  - Report: 20%
  - Demonstration: 80%
- Testbenches will be provided for each lab

Lab	Topic	Weight
1	ALU, Vending Machine	5
2	Single-cycle CPU	10
3	Multi-cycle CPU	25
4	Pipelined CPU	35
5	Cache	25
Total		**100**

Rule

- Each team has a **1-day late submission token**
  - You can use the token to avoid the late submission penalty for one day.
  - If you want to use it, let TAs know at the time of submission.

- For late submission (when the token is not used)
  - 1 day late: -10%
  - 2 days late: -20%
  - 3 days late: -50%
  - **4 or more days late: -100%**
    (you would still need to finish it to work on the remaining labs.)

- Cheating will be taken very seriously. Refer to the first lecture note.

Team

- Form a team of two students for the labs in this course
  - If you cannot, the TAs can assign your teammate

- Submit the following google form by 2/26, 23:59 (Monday)
  - https://forms.gle/LrhZrtwp3HDGHMGL8

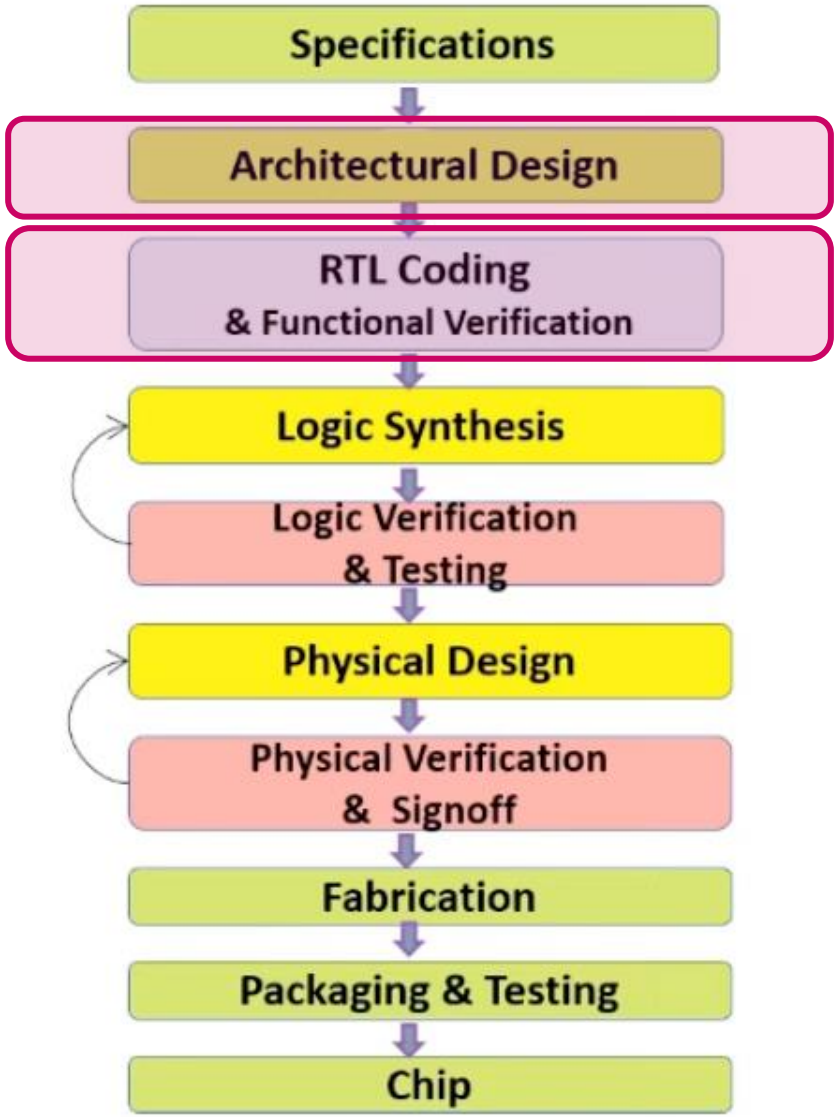
POSTECH

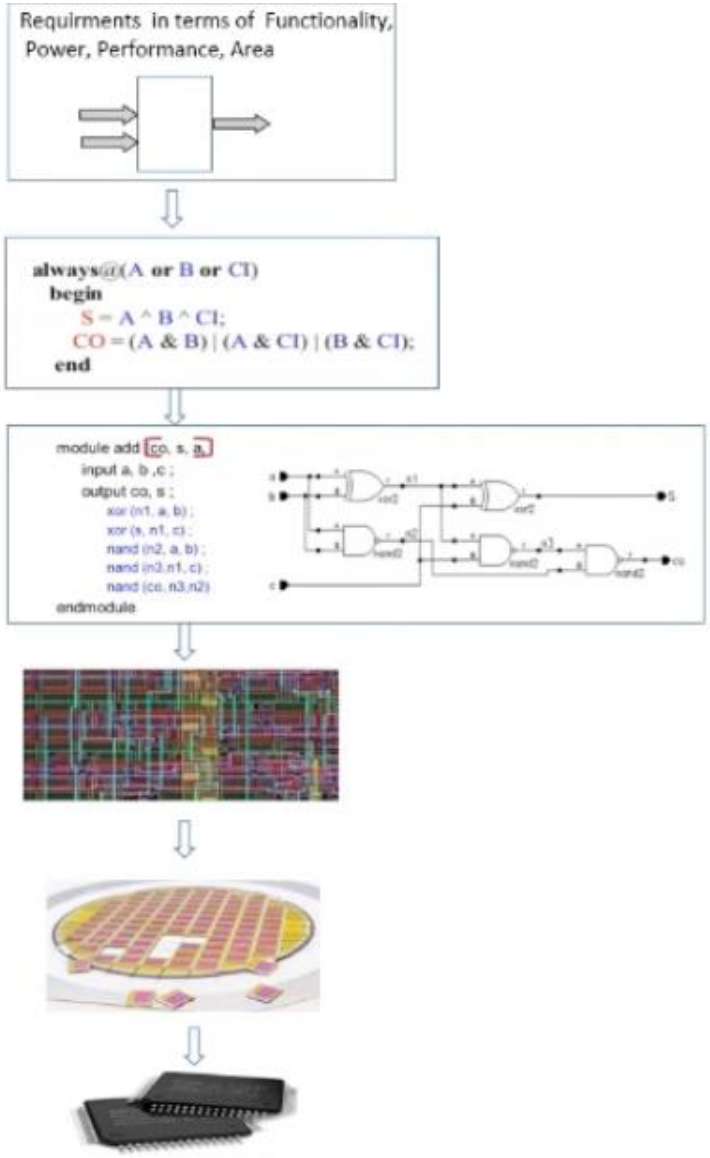
Verilog Tutorial

VLSI Design Flow

Focus of our lectures!

Focus of our labs!



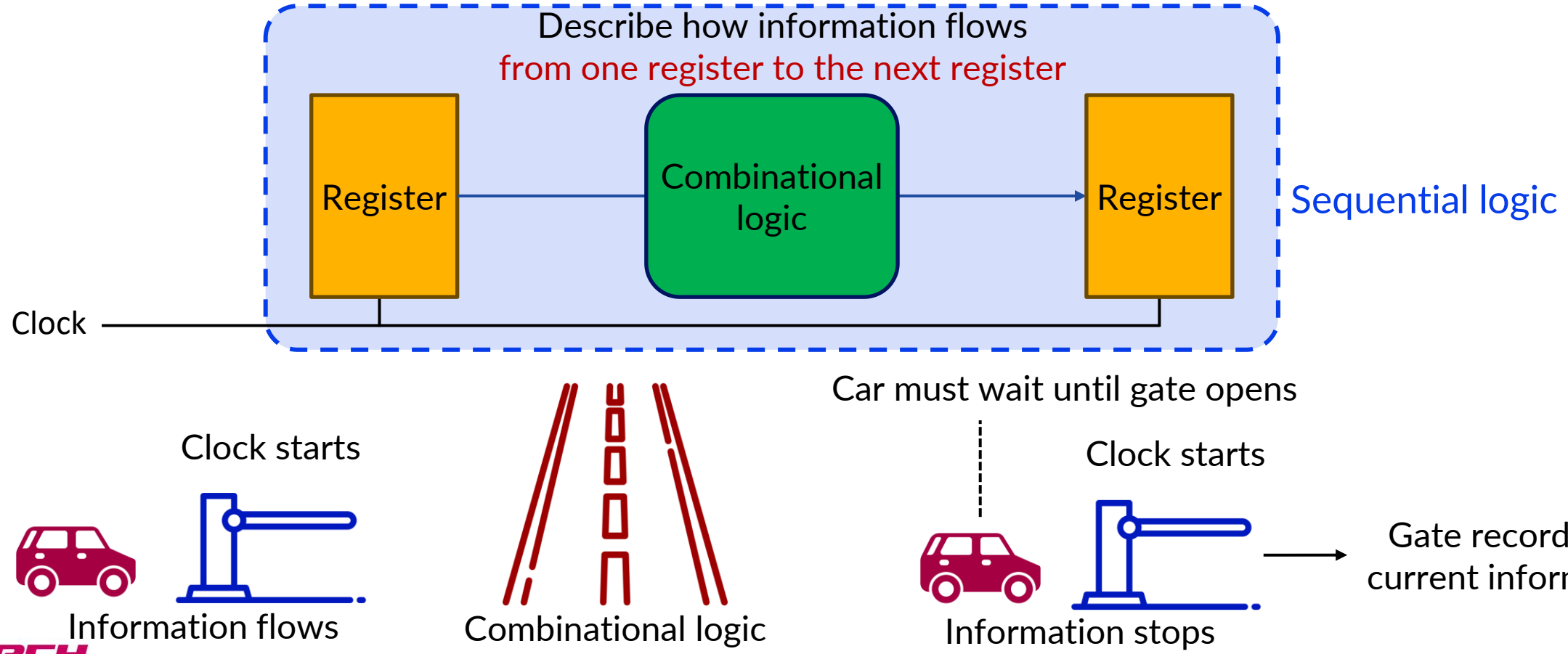


Description with images:
https://www.asic-world.com/verilog/design_flow1.html

Image source: https://www.youtube.com/watch?v=zpOioOiKYp4

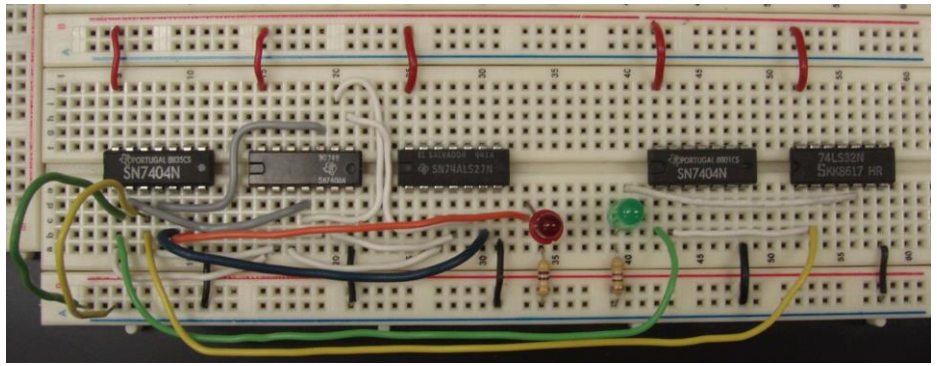
What is Verilog HDL?

- One of Hardware Description Languages (HDLs)
  - Describe how hardware is constructed (말로 풀어서 설명하는 그림 또는 동작방식)
  - Verilog implements register-transfer-level (RTL) abstractions

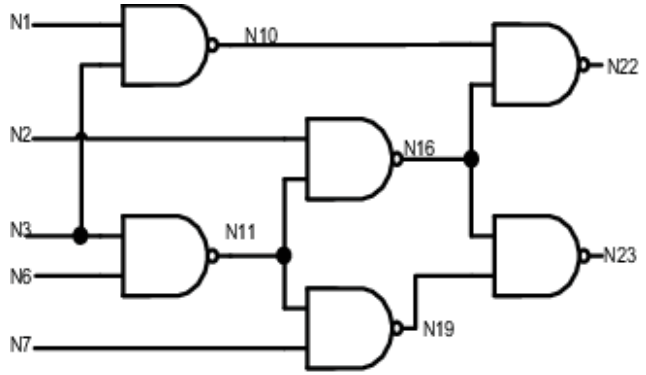


What is Verilog HDL?

- Hardware Description Language
  - Not a programming language



Breadboard



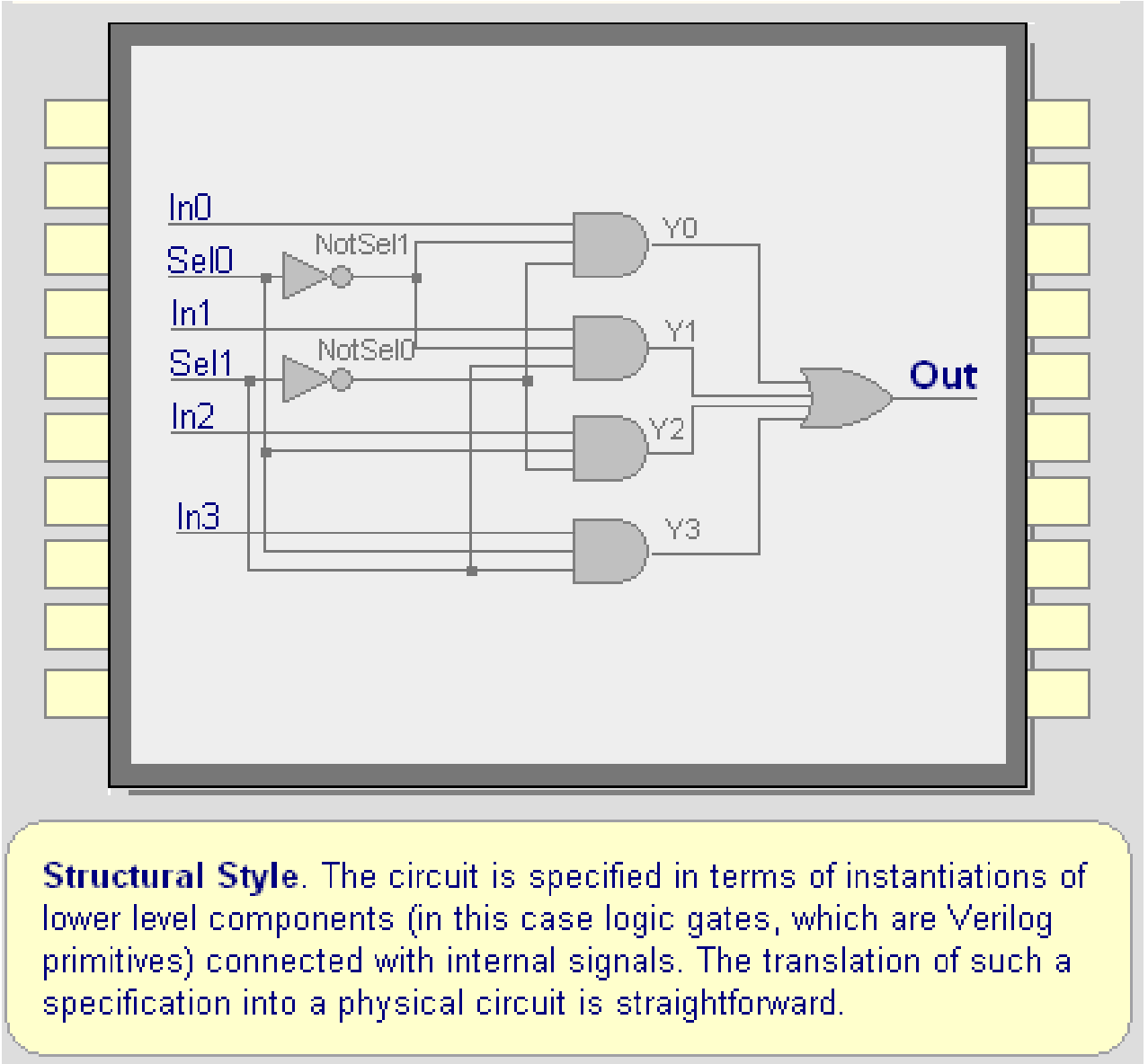
Circuit diagram

```
module c17 (N1,N2,N3,N6,N7,N22,N23);
input N1,N2,N3,N6,N7;
output N22,N23;
wire N10,N11,N16,N19;
nand NAND2_1 (N10, N1, N3);
nand NAND2_2 (N11, N3, N6);
nand NAND2_3 (N16, N2, N11);
nand NAND2_4 (N19, N11, N7);
nand NAND2_5 (N22, N10, N16);
nand NAND2_6 (N23, N16, N19);
endmodule
```

Verilog

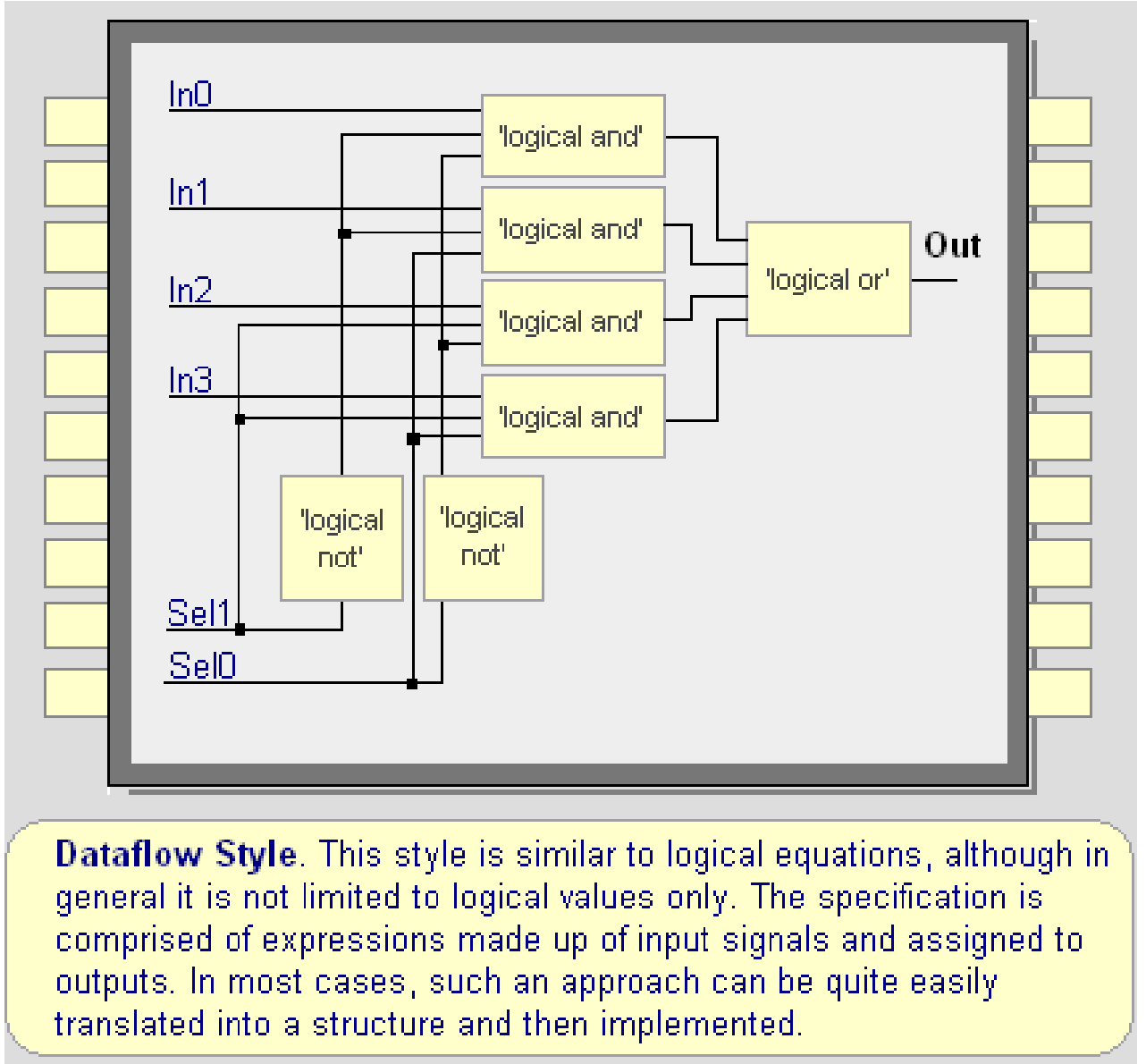
- Keep in mind
  - Hardware components operate in parallel
  - Your code should be synthesizable

One Language, Many Coding Styles



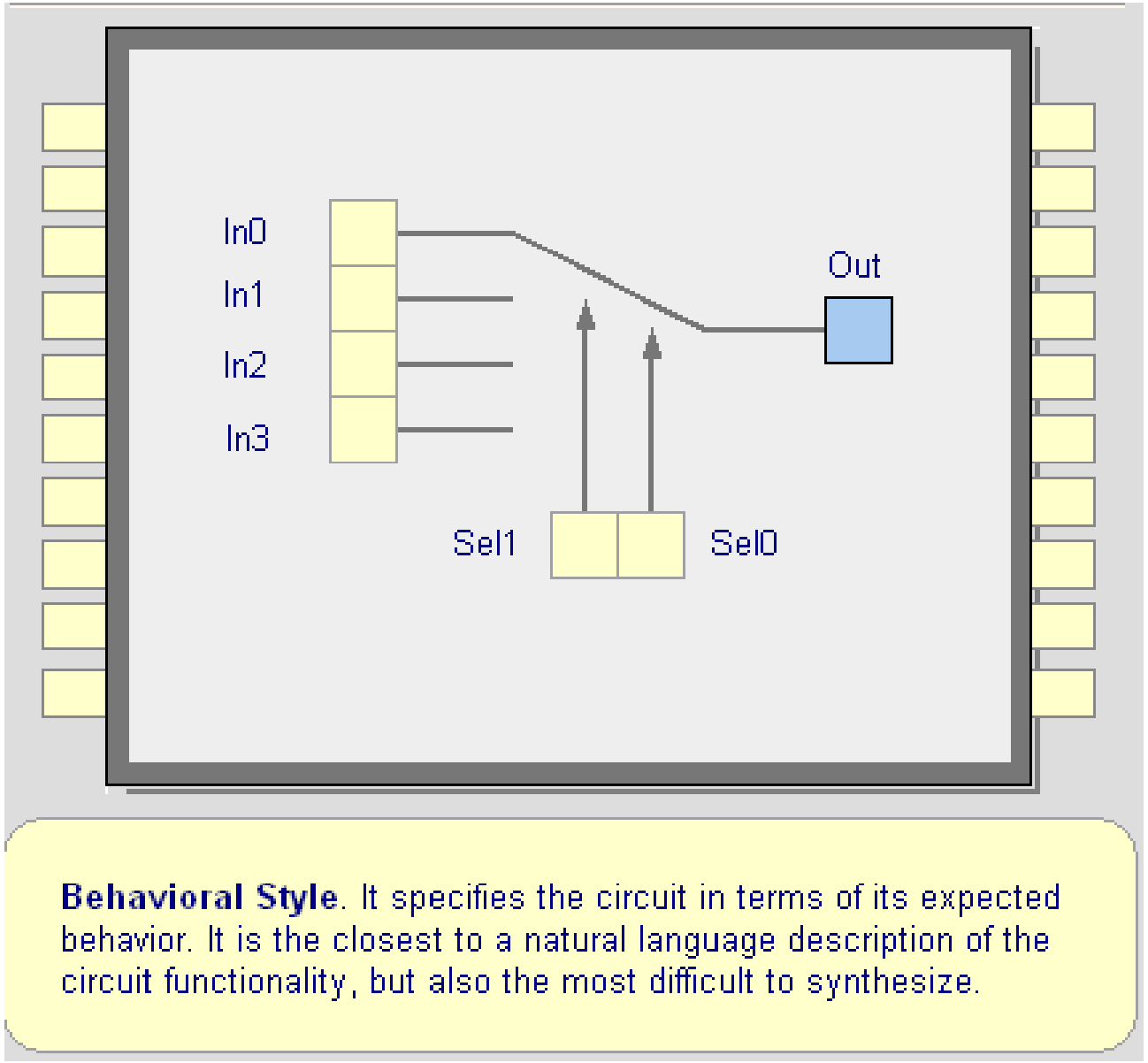
Structural Style. The circuit is specified in terms of instantiations of lower level components (in this case logic gates, which are Verilog primitives) connected with internal signals. The translation of such a specification into a physical circuit is straightforward.

One Language, Many Coding Styles (contd.)



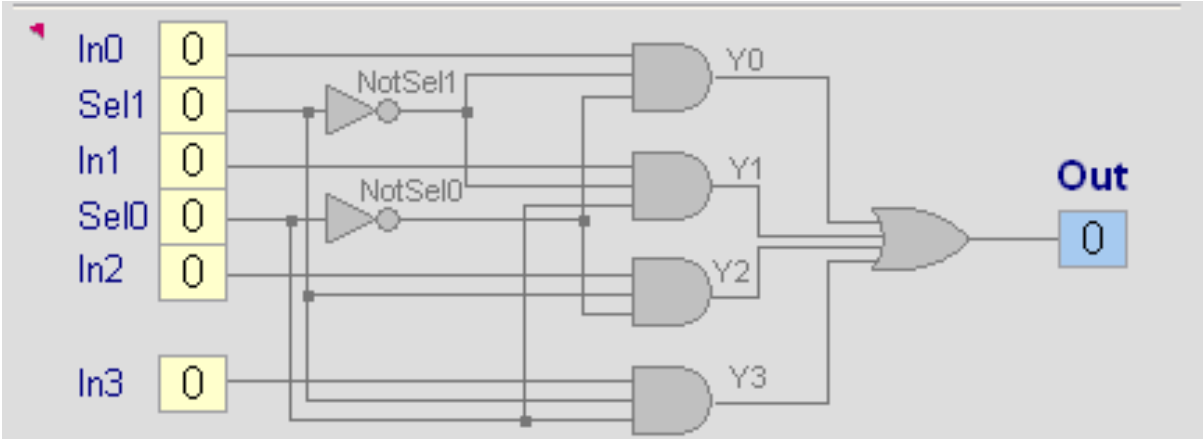
Dataflow Style. This style is similar to logical equations, although in general it is not limited to logical values only. The specification is comprised of expressions made up of input signals and assigned to outputs. In most cases, such an approach can be quite easily translated into a structure and then implemented.

One Language, Many Coding Styles (contd.)



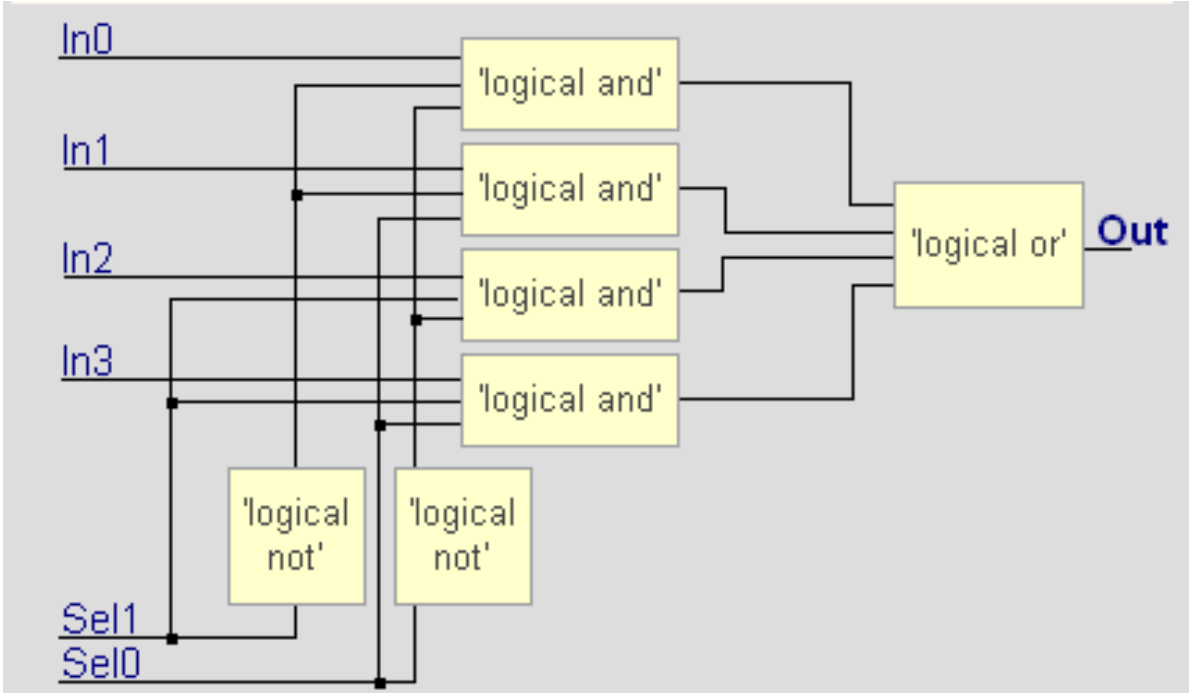
Behavioral Style. It specifies the circuit in terms of its expected behavior. It is the closest to a natural language description of the circuit functionality, but also the most difficult to synthesize.

Structural Style: Verilog Code



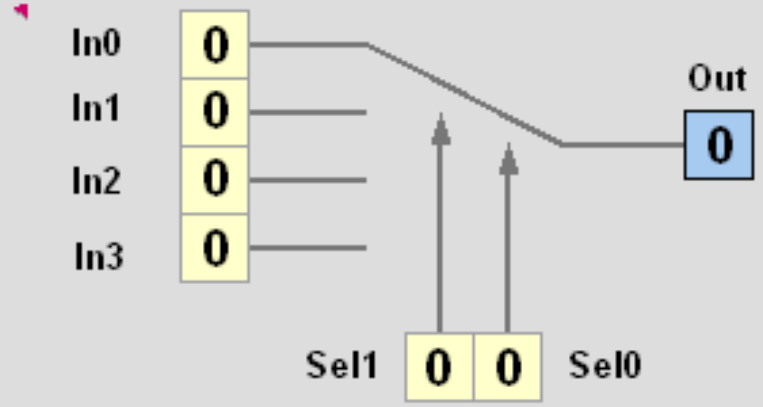

```
module mux_4_to_1 (Out,In0,In1,In2,In3,Sel1,Sel0);
output Out;
input In0, In1, In2, In3, Sel0, Sel1;

wire NotSel0, NotSel1;
wire Y0, Y1, Y2, Y3;

not (NotSel0, Sel0);
not (NotSel1, Sel1);
and (Y0, In0, NotSel1, NotSel0);
and (Y1, In1, NotSel1, Sel0);
and (Y2, In2, Sel1, NotSel0);
and (Y3, In3, Sel1, Sel0);
or (Out, Y0, Y1, Y2, Y3);

endmodule
```

Dataflow Style: Verilog Code




```
module mux_4_to_1 (Out, In0, In1, In2, In3, Sel1, Sel0);

output Out;
input In0, In1, In2, In3, Sel0, Sel1;

assign Out = (~Sel1 & ~Sel0 & In0) | (~Sel1 & Sel0 & In1)
             | (Sel1 & ~Sel0 & In2)| (Sel1 & Sel0 & In3);

endmodule
```

Behavioral Style: Verilog Code




```
module mux_4_to_1 (Out,In0,In1,In2,In3,Sel1,Sel0);
output Out;
input In0, In1, In2, In3, Sel0, Sel1;
reg Out;

always @(Sel1 or Sel0 or In0 or In1 or In2 or In3)
  begin
    case ({Sel1, Sel0})
    2'b00 : Out = In0;
    2'b01 : Out = In1;
    2'b10 : Out = In2;
    2'b11 : Out = In3;
    default : Out = 1'bx;
    endcase
  end

endmodule
```

Verilog Module

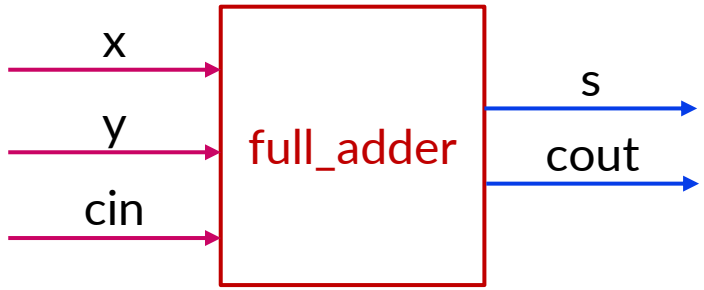
- Modules are the building blocks of Verilog designs
  - Similar to functions in the programming language

- 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

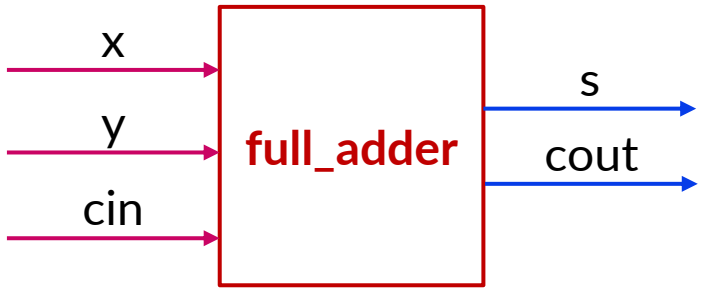
```
module full_adder (input x, input y, input cin,
                   output s, output cout);
   // Verilog code
endmodule
```



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



Port declaration

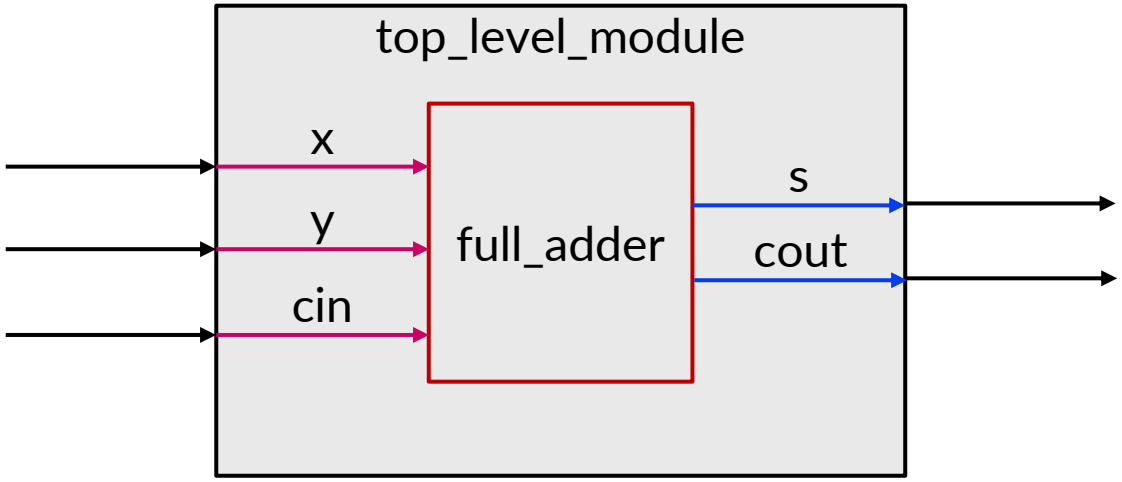
full_adder.v

```
module full_adder (input x, input y, input cin,
                   output s, output cout);

   // Verilog code
endmodule
```

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

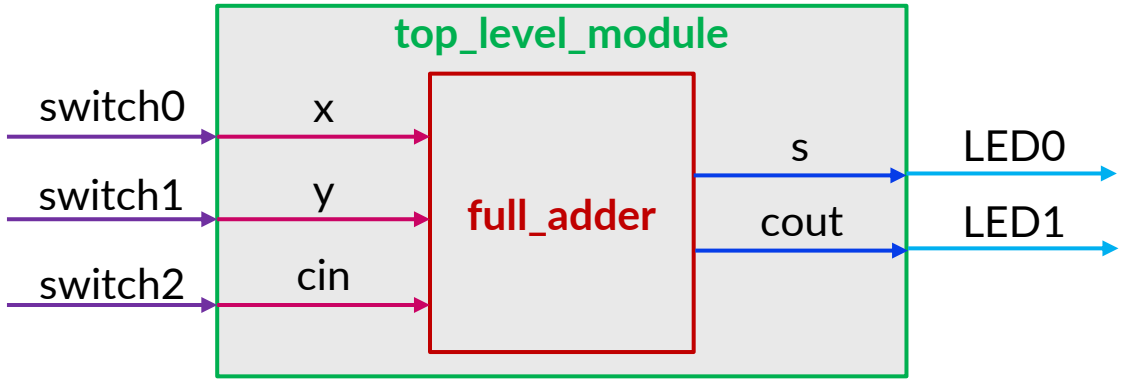
- You have two files
  - top_level_module.v
  - full_adder.v

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 switch,
                         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
```



Module Instantiation

Port connection by name

```
1  module design_top;
2      wire [1:0]  a;
3      wire        b, c;
4
5      mydesign d0  ( .x (a[0]),
6                     .y (b),
7                     .z (a[1]),
8                     .o (c));
9  endmodule
```

Always use this style!

Port connection by ordered list

```
1  module mydesign ( input  x, y, z,
2                    output o);
3
4  endmodule
5
6  module tb_top;
7      wire [1:0]  a;
8      wire        b, c;
9
10     mydesign d0  (a[0], b, a[1], c);
11
12
13
14 endmodule
```

Do not use this style!

Reference: https://www.chipverify.com/verilog/verilog-module-instantiations

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]

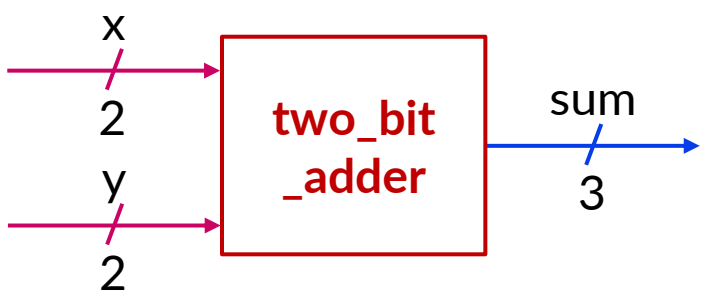
```
wire [31 : 0] a;     // 32 bit wire
wire [63 : 0] b;     // 64 bit wire (Bit at 63 is MSB)
wire [0 : 63] c;     // 64 bit wire (Bit at 0 is MSB)
```

Recommended

Wire

- To use vector wires (wider bitwidth) for declaring ports in a module:

```
module two_bit_adder (input [1 : 0] x, input [1 : 0] y,
                      output [2 : 0] z);
   // Implement two-bit adder here
endmodule
```



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 may or may not create a hardware register (i.e., flip-flop)
  - Combinational logic → hardware wire
  - Sequential logic → hardware register

- 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 size of the array is 6.

  - An array of reg (can be used to represent a memory!)

    reg [31 : 0] memory [0 : 1023]; // declare an array of 32-bit vector reg. The size of the 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) (   // 32 is the default value
                    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:
  - Structural (gate-level) modeling => low-level, painful
  - Dataflow modeling
    - Using continuous assignments
  - Behavioral modeling
    - Using procedural assignments

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

- We will learn about 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

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

```
module example1 (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-bit adder
endmodule
```

```
module example2 (input [1 : 0] x, input [1 : 0] y, input a,
                 output [1 : 0] z)
   assign z = a ? x : y;  // conditional operator (multiplexer)
endmodule
```

Dataflow Modeling, cont'd

```
module example3 (input [1 : 0] x, input [1 : 0] y, input a,
                 output [1 : 0] z)
   wire [1 : 0] out;
   assign out = x | y;
   assign z = out;

   // Alternatively, directly assign the input to the wire
   // wire [1 : 0] out = x | y;
   // assign z = out;

endmodule
```


* Refer to Chapter 6 at https://d1.amobbs.com/bbs_upload782111/files_33/ourdev_585395BQ8J9A.pdf

Dataflow Modeling, cont'd

```
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 the value at the bit position 3 of d 32 times => 32 bits
// assign {a[5], b[2 : 1]} = c[23:21]; // assign the sliced value to concatenated net
```

Caution: always match the bit width of both sides!

* Refer to Chapter 6 at https://d1.amobbs.com/bbs_upload782111/files_33/ourdev_585395BQ8J9A.pdf

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 implement a clock generator 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

Behavioral Modeling

- Delay (#time)
  - Intra-assignment delay
  - Inter-assignment delay

Do not use delays in your designs! They are not synthesizable
These examples are not synthesizable.
The purpose of showing them here is to illustrate the difference between blocking and non-blocking assignments.

- Blocking assignment (=): executed in the order of the 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 blocks loop to execute over and over again; in other words, as the name suggests, it executes always
    - Can be used for both sequential and combinational logic

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

    - Sensitivity list: The list of events or signals that triggers execution of the always block

Reference: https://www.asic-world.com/verilog/vbehave1.html#Example_-_always

Behavioral Modeling – Combinational Logic

- Combinational (asynchronous) logic example
  - lvalue inside always statement must be one of reg, ~~integer, real, and time register~~
    - 'reg_z' must be declared by 'reg'
  - Use blocking assignments when modeling combinational logic in an always block

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

reg reg_z;
assign z = reg_z;

// always @(x or y) begin  -- error-prone, not recommended for combinational logic
always @(*) begin          //'*' implies x or y
    reg_z = x & y;
end

endmodule
```

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

Reference: https://www.chipverify.com/verilog/verilog-combinational-logic-always

Behavioral Modeling – Combinational Logic

- Combinational (asynchronous) logic example, cont'd

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

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

endmodule
```

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

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

endmodule
```

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

Behavioral Modeling – Sequential Logic

- Sequential (synchronous) logic example

```
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 should be the type (i.e., wire or reg) of a and b?

2. Are always statements executed in order?

3. What happen to a and b?

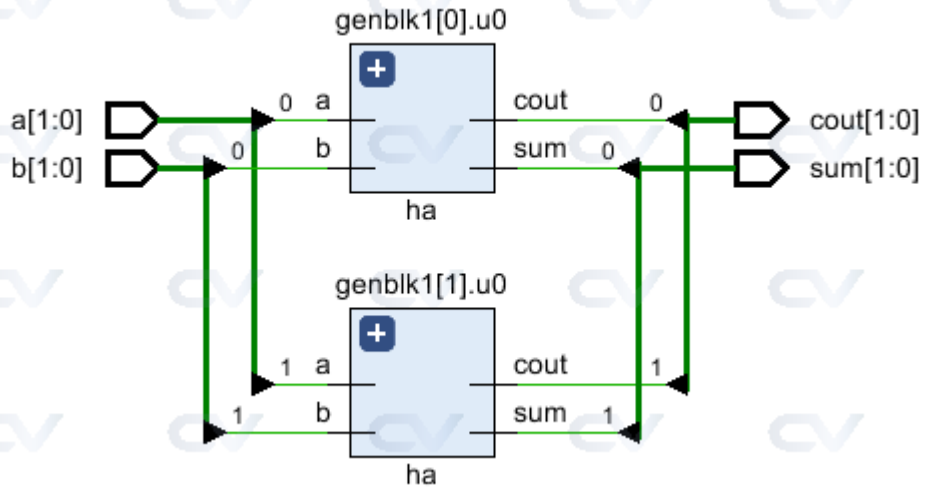
Behavioral Modeling – Sequential Logic

- Sequential (synchronous) logic example, cont'd
  - 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
```

Generate Block

- Used for creating multiple instances (for loop) or conditional instantiation (if or case)
- Used within module block
- Example 1: generating multiple (N) half-adders



Reference: https://www.chipverify.com/verilog/verilog-generate-block

```
1  // Design for a half-adder
2  module ha ( input   a, b,
3              output  sum, cout);
4
5    assign sum  = a ^ b;
6    assign cout = a & b;
7  endmodule
8
9  // A top level design that contains N instances of half adder
10 module my_design
11     #(parameter N=4)
12         (   input [N-1:0] a, b,
13             output [N-1:0] sum, cout);
14
15     // Declare a temporary loop variable to be used during
16     // generation and won't be available during simulation
17     genvar i;
18
19     // Generate for loop to instantiate N times
20     generate
21         for (i = 0; i < N; i = i + 1) begin
22           ha u0 (a[i], b[i], sum[i], cout[i]);
23         end
24     endgenerate
25 endmodule
```

Generate Block, cont'd

- Example 2: conditional instantiation (if)

```
32 // Top Level Design: Use a parameter to choose either one
33 module my_design (  input a, b, sel,
34                     output out);
35   parameter USE_CASE = 0;
36
37   // Use a "generate" block to instantiate either mux_case
38   // or mux_assign using an if else construct with generate
39   generate
40     if (USE_CASE)
41       mux_case mc (.a(a), .b(b), .sel(sel), .out(out));
42     else
43       mux_assign ma (.a(a), .b(b), .sel(sel), .out(out));
44   endgenerate
45
46 endmodule
```

```
1  // Design #1: Multiplexer design uses an "assign" statement to assign
2  // out signal
3  module mux_assign ( input a, b, sel,
4                      output out);
5    assign out = sel ? a : b;
6
7    // The initial display statement is used so that
8    // we know which design got instantiated from simulation
9    // logs
10   initial
11     $display ("mux_assign is instantiated");
12 endmodule
13
14 // Design #2: Multiplexer design uses a "case" statement to drive
15 // out signal
16 module mux_case (input a, b, sel,
17                  output reg out);
18   always @ (a or b or sel) begin
19     case (sel)
20         0 : out = a;
21         1 : out = b;
22     endcase
23   end
24
25   // The initial display statement is used so that
26   // we know which design got instantiated from simulation
27   // logs
28   initial
29     $display ("mux_case is instantiated");
30 endmodule
```

Reference: https://www.chipverify.com/verilog/verilog-generate-block

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

Guideline

- Combinational logic
  - “always @(*)” block (LHS: reg, blocking assignment)
  - “assign” statement (LHS: wire)

- Sequential logic
  - “always @(posedge clk)” block (LHS: reg, non-blocking assignment)

- Use the highlighted styles always if confused

- Carefully read this article on “Synthesizable Verilog Coding Style”
  - https://hongcezh.people.ust.hk/post/verilog/

References

- https://www.chipverify.com/tutorials/verilog
- 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
- https://www.asic-world.com/verilog/veritut.html