Imperial College
London

Lecture 8

SystemVerilog HDL

Peter Cheung
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/EE2_CAS/
E-mail: p.cheung@imperial.ac.uk

Lecture Objectives

- By the end of this lecture, you should understand:
  - The basic structure of a module specified in SystemVerilog HDL
  - Commonly used syntax of SystemVerilog HDL
  - Continuous vs Procedural Assignments
  - **always** block in SystemVerilog and sensitivity list
  - The use of arithmetic and logic operations in SystemVerilog
  - The danger of **incomplete specification**
  - How to specify **clocked circuits**
  - Differences between **blocking** and **nonblocking** assignments

Schematic vs HDL

Schematic

- ✓ Good for multiple data flow
- ✓ Give overview picture
- ✓ Relate directly to hardware
- ✓ Don't need good programming skills
- ✓ High information density
- ✓ Easy back annotations
- ✓ Useful for mixed analogue/digital

- × Not good for algorithms
- × Not good for datapaths
- × Poor interface to optimiser
- × Poor interface to synthesis software
- × Difficult to reuse
- × Difficult to parameterise

HDL

- ✓ Flexible & parameterisable
- ✓ Excellent input to optimisation & synthesis
- ✓ Direct mapping to algorithms
- ✓ Excellent for datapaths
- ✓ Easy to handle electronically (only needing a text editor)

- × Serial representation
- × May not show overall picture
- × Need good programming skills
- × Divorce from physical hardware

SystemVerilog HDL

- Similar to C language to describe/specify hardware
- Description can be at different levels:
  - **Behavioural level**
  - **Register-Transfer Level (RTL)**
  - **Gate Level**
- Not only a specification language, also with associated **simulation environment**
- Easier to learn and “lighter weight” than its competition: VHDL
- Very popular with chip designers

- For this lecture, we will:
  - Learn through examples and practical exercises
  - Use examples: e.g. 2-to-1 multiplexer and 7 segment decoder

HDL to Gates

Simulation

- Inputs applied to circuit
- Outputs checked for correctness
- Millions of dollars saved by debugging in simulation instead of hardware

Synthesis

- Transforms HDL code into a netlist describing the hardware (i.e., a list of gates and the wires connecting them)

Physical design

- Placement, routing, chip layout, …… – not considered in this module

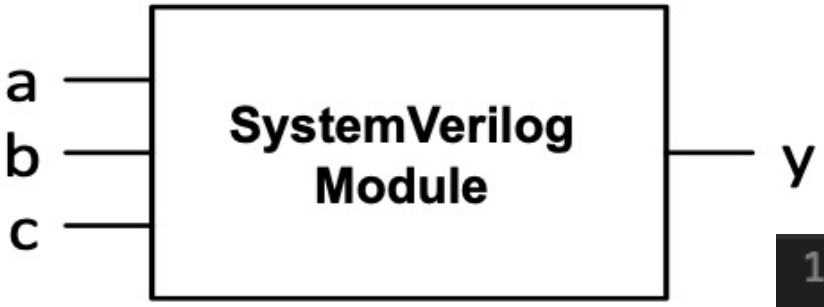
IMPORTANT:

When using an HDL, think of the **hardware** the HDL should produce, then write the appropriate idiom that implies that hardware.

Beware of treating HDL like software and coding without thinking of the hardware.

SystemVerilog: Module Declaration

- **Two types of Modules:**
  - **Behavioral**: describe what a module does
  - **Structural**: describe how it is built from simpler modules



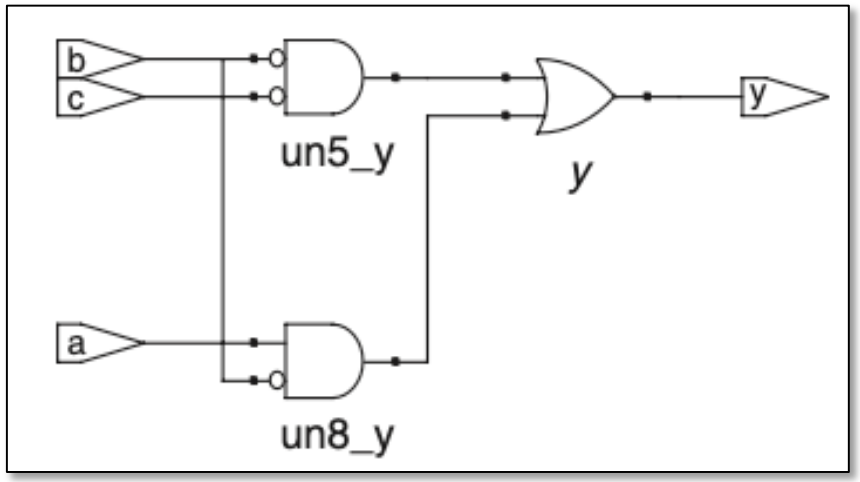

```
1   module example(input  logic a, b, c,
2                  output logic y);
3     // module body goes here
4   endmodule
```

- **module/endmodule: required to begin/end module**
- **example: name of the module**

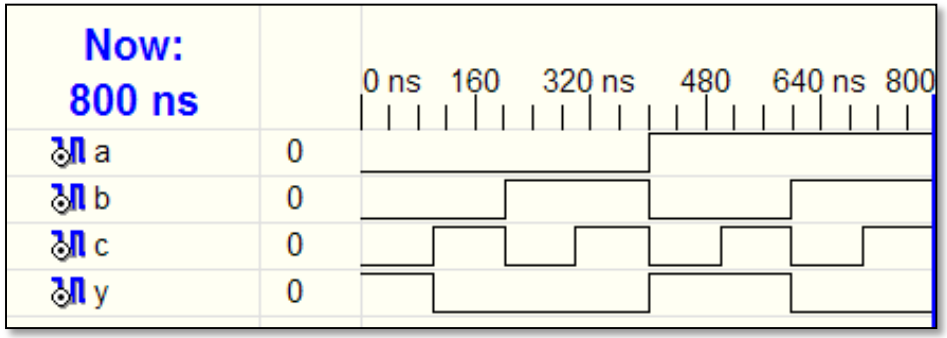
System Verilog: Behavioural Description

```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

synthesis



simulation



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

System Verilog: Syntax

- **Case sensitive**
  - e.g.: reset and Reset are not the same signal.
- **No names that start with numbers**
  - e.g.: 2mux is an invalid name
- **Whitespace ignored**
- **Comments:**
  - // single line comment
  - /* multiline
  - comment */

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*"
by Sarah Harris and David Harris (H&H),

System Verilog: Structural Description

Behavioural

```
module and3(input  logic a, b, c,
            output logic y);
  assign y = a & b & c;
endmodule
```

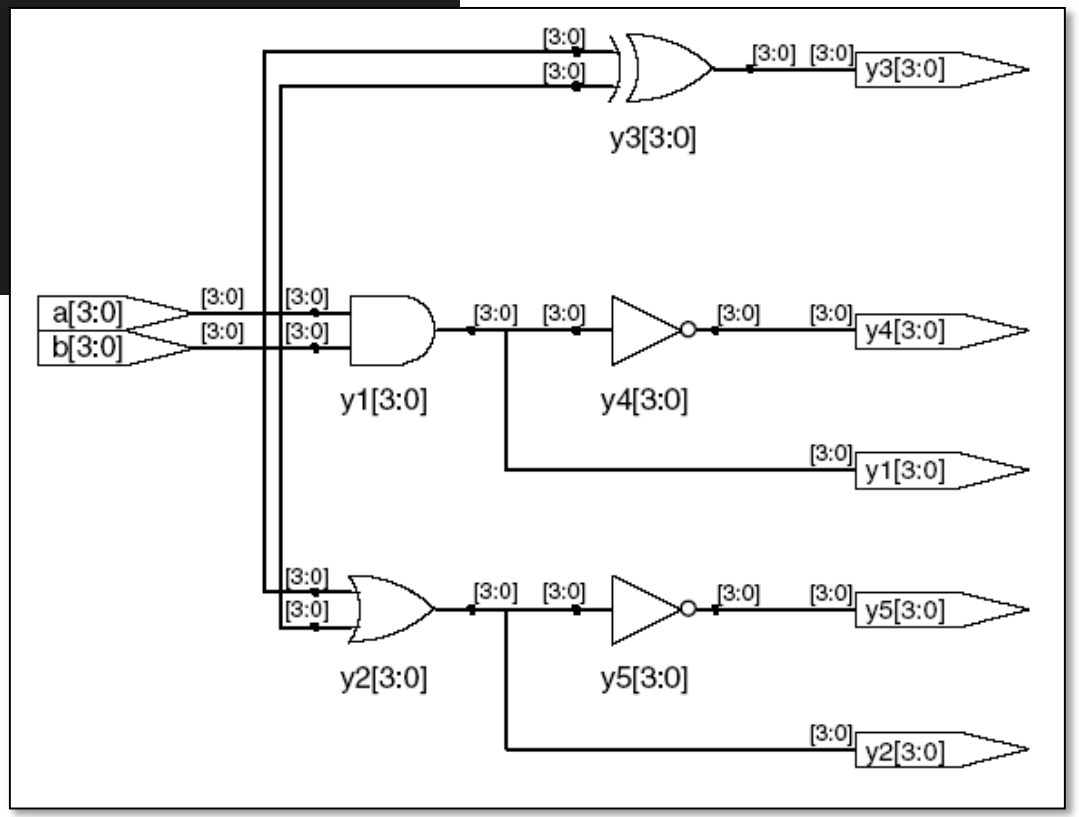
```
module inv(input  logic a,
           output logic y);
  assign y = ~a;
endmodule
```

Structural

```
module nand3(input  logic a, b, c
             output logic y);
  logic n1;                    // internal signal

  and3 andgate(a, b, c, n1);  // instance of and3
  inv  inverter(n1, y);       // instance of inv
endmodule
```

System Verilog: Bitwise Operators

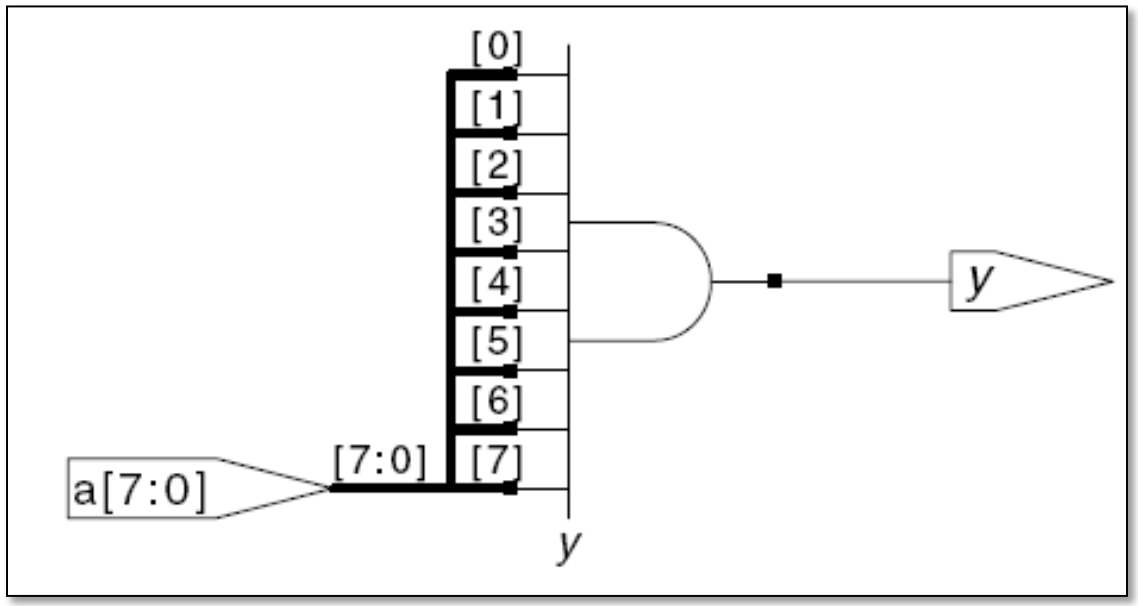
```
module gates(input  logic [3:0]  a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
   /* Five different two-input logic
      gates acting on 4 bit busses */
   assign y1 = a & b;    // AND
   assign y2 = a | b;    // OR
   assign y3 = a ^ b;    // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR
endmodule
```



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

SysytemVerilog: Reduction Operators

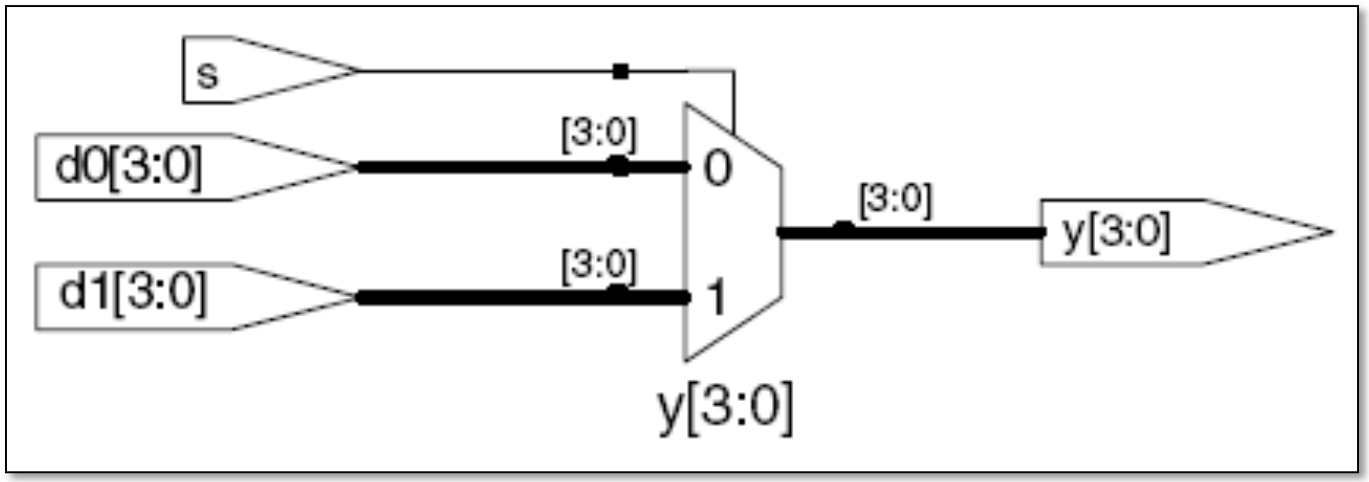
```
module and8(input  logic [7:0] a,
            output logic       y);
  assign y = &a;
  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*"
by Sarah Harris and David Harris (H&H),

System Verilog: Conditional Assignment

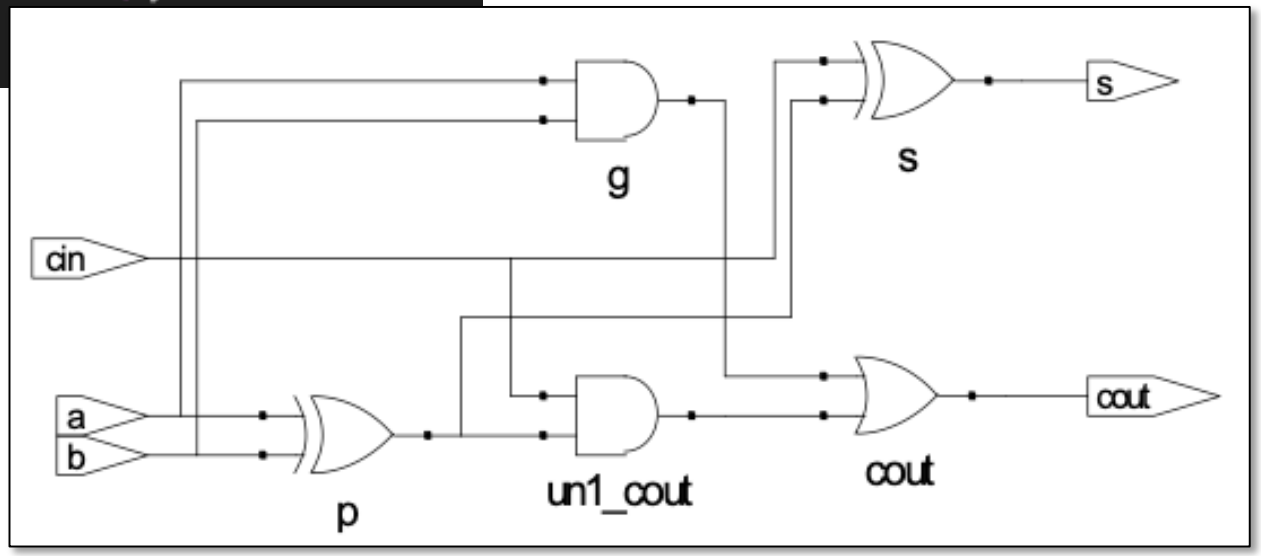
```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);
   assign y = s ? d1 : d0;
endmodule
```



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*"
by Sarah Harris and David Harris (H&H),

System Verilog: Internal Signals

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;   // internal nodes

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

System Verilog: Precedence of operators

Highest

~	**NOT**
*, /, %	**mult, div, mod**
+, -	**add, sub**
<<, >>	**shift**
<<<, >>>	**arithmetic shift**
<, <=, >, >=	**comparison**
==, !=	**equal, not equal**
&, ~&	**AND, NAND**
^, ~^	**XOR, XNOR**
|, ~|	**OR, NOR**
?:	**ternary operator**

Lowest

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*"
by Sarah Harris and David Harris (H&H),

System Verilog: Number Format

Format: N'Bvalue

N = number of bits, **B** = base

N'B is optional but recommended (default is decimal)

Number	# Bits	Base	Decimal Equivalent	Stored
3'b101	3	binary	5	101
'b11	unsized	binary	3	00…0011
8'b11	8	binary	3	00000011
8'b1010_1011	8	binary	171	10101011
3'd6	3	decimal	6	110
6'o42	6	octal	34	100010
8'hAB	8	hexadecimal	171	10101011
42	Unsized	decimal	42	00…0101010

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

System Verilog: Bit Manipulations (1)

```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};
```

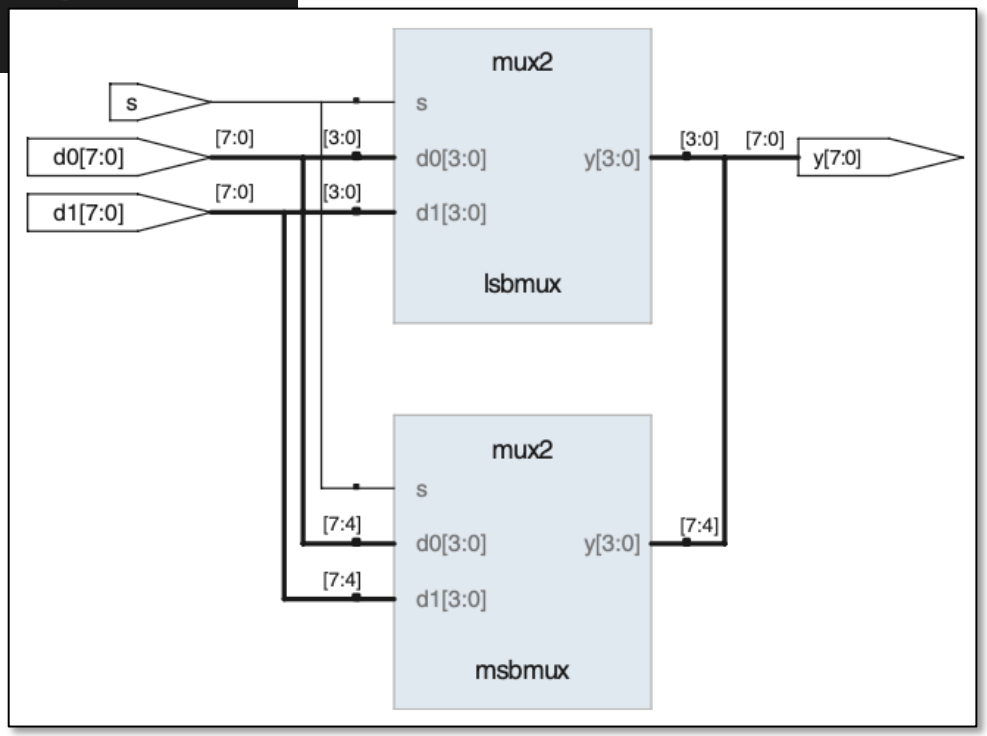
- If y is a 12-bit signal, the above statement produces:

```
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0
```

- Underscores (_) are used for formatting only to make it easier to read. **System Verilog ignores them**.

System Verilog: Bit Manipulations (2)

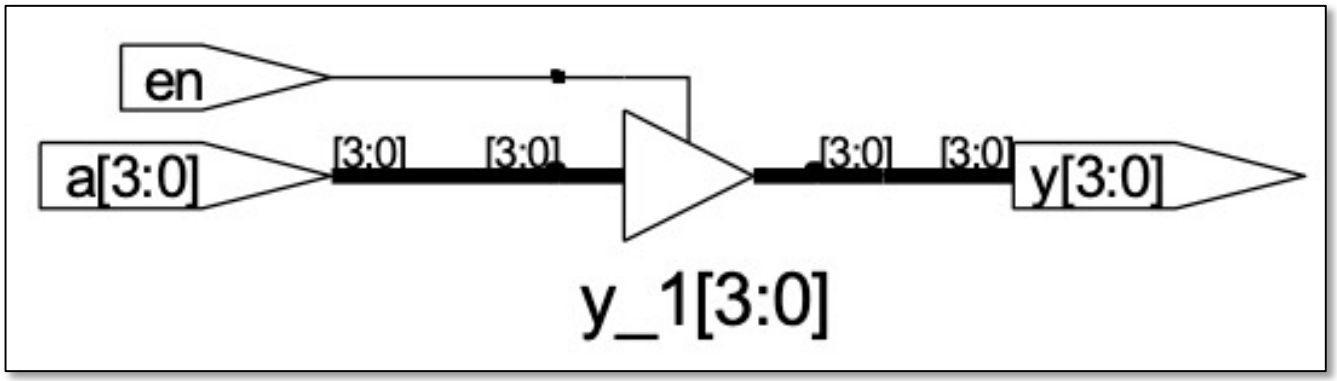
```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

System Verilog: Floating Output Z

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);
   assign y = en ? a : 4'bz;
endmodule
```

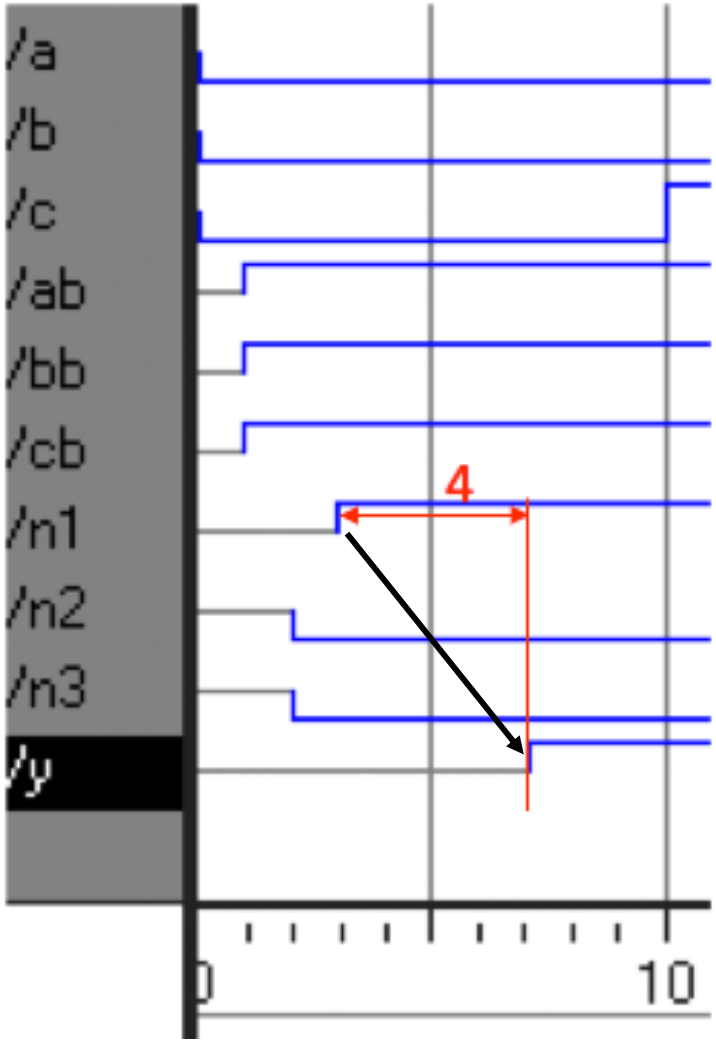


- Note that Verilator does not handle floating output Z

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*"
by Sarah Harris and David Harris (H&H),

System Verilog: Delays

```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



- Delays are for simulation only! They do not determine the delay of your hardware.
- **Verilator similator ignores delays** – it is cycle accurate without timing.

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

System Verilog: Sequential Logic

- System Verilog uses **idioms** (or special keywords or groups of words) to describe latches, flip-flops and FSMs
- Other coding styles may simulate correctly but produce incorrect hardware
- GENERAL STRUCTURE:

```
always @(sensitivity list)
        statement;
```

- Whenever the event in **`sensitivity list`** occurs, **`statement`** is executed

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

System Verilog: D Flip-Flop

```
module flop(input  logic        clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;                  // pronounced “q gets d”

endmodule
```

Based on: “*Digital Design and Computer Architecture (RISC-V Edition)*”
by Sarah Harris and David Harris (H&H),

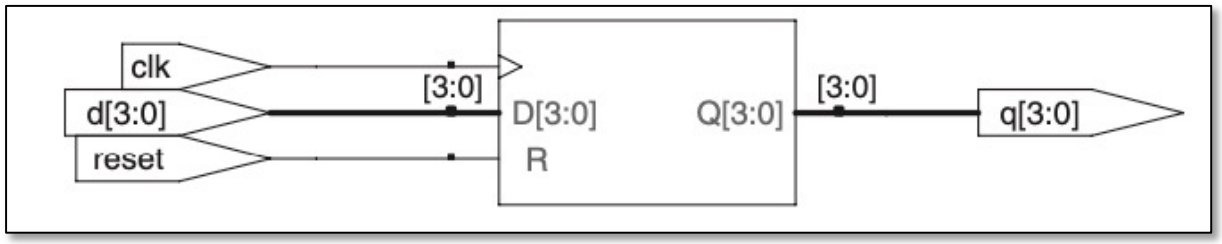
System Verilog: Resettable D Flip-Flop

Asynchronous reset

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

// asynchronous reset
always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else q <= d;
endmodule
```

Synchronous reset

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // synchronous reset
  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*"
by Sarah Harris and David Harris (H&H),

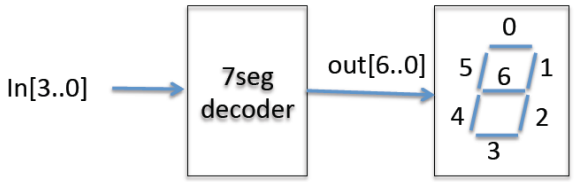
Combinational Logic using always

```
// combinational logic using an always statement
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
  always_comb          // need begin/end because there is
    begin              // more than one statement in always
      y1 = a & b;      // AND
      y2 = a | b;      // OR
      y3 = a ^ b;      // XOR
      y4 = ~(a & b); // NAND
      y5 = ~(a | b); // NOR
    end
endmodule
```

This hardware could be described with **`assign` statements using fewer lines** of code, so it's better to use **`assign`** statements in this case.

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*"
by Sarah Harris and David Harris (H&H),

Putting everything together – 7 seg decoder



in[3..0]	out[6:0]	Digit
0000	1000000	0
0001	1111001	1
0010	0100100	2
0011	0110000	3
0100	0011001	4
0101	0010010	5
0110	0000010	6
0111	1111000	7

in[3..0]	out[6:0]	Digit
1000	0000000	8
1001	0010000	9
1010	0001000	A
1011	0000011	b
1100	1000110	C
1101	0100001	d
1110	0000110	E
1111	0001110	F

In3 : in2

In1 : in0

out6	00	01	11	10
00	1	0	1	0
01	1	0	0	0
11	0	1	0	0
10	0	0	0	0

out6 = /in3*/in2*/in1 + in3*in2*/in1*/in0 + /in3*in2*in1*in0

out5 = /in3*/in2*in0 + /in3*/in2*in1 + /in3*in1*in0 + in3*in2*/in1*in0

out4 = /in3*in0 + /in3*in2*/in1 + in3*/in2*/in1*in0

out3 = /in3*in2*/in1*/in0 + /in3*/in2*/in1*in0 + in2*in1*in0 + /in2*in1*/in0

out2 = /in3*/in2*in1*/in0 + in3*in2*/in0 + in3*in2*in1

out1 = in3*in2*/in0 + /in3*in2*/in1*in0 + in3*in1*in0 + in2*in1*/in0

out0 = /in3*/in2*/in1*in0 + /in3*in2*/in1*/in0 + in3*in2*/in1*in0 + in3*/in2*in1*in0

Method 1: Schematic Entry Implementation

out6 = /in3*/in2*/in1 + in3*in2*/in1*/in0 + /in3*in2*in1*in0

out5 = /in3*/in2*in0 + /in3*/in2*in1 + /in3*in1*in0 + in3*in2*/in1*in0

out4 = /in3*in0 + /in3*in2*/in1 + in3*/in2*/in1*in0

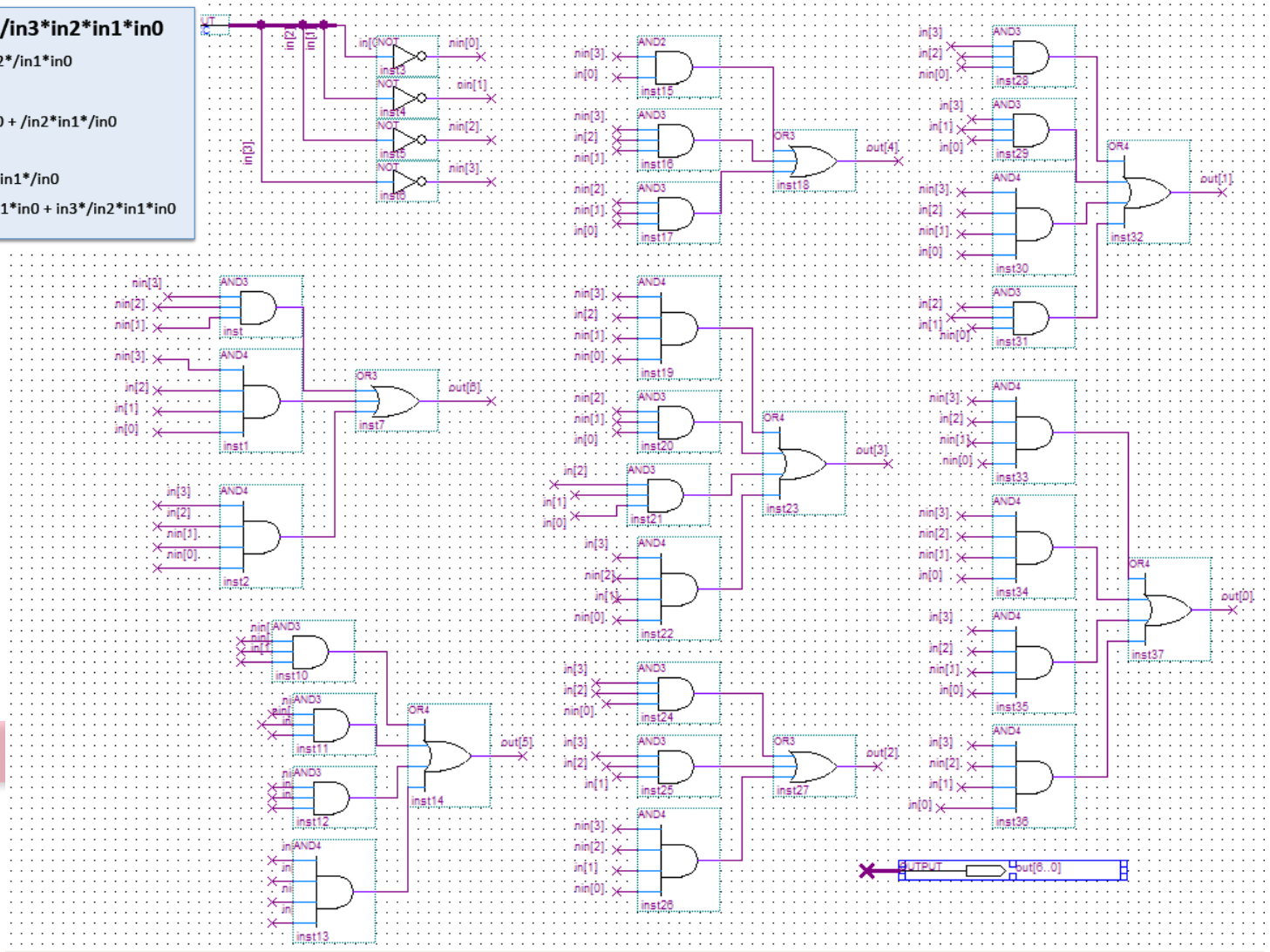
out3 = /in3*in2*/in1*/in0 + /in3*/in2*/in1*in0 + in2*in1*in0 + /in2*in1*/in0

out2 = /in3*/in2*in1*/in0 + in3*in2*/in0 + in3*in2*in1

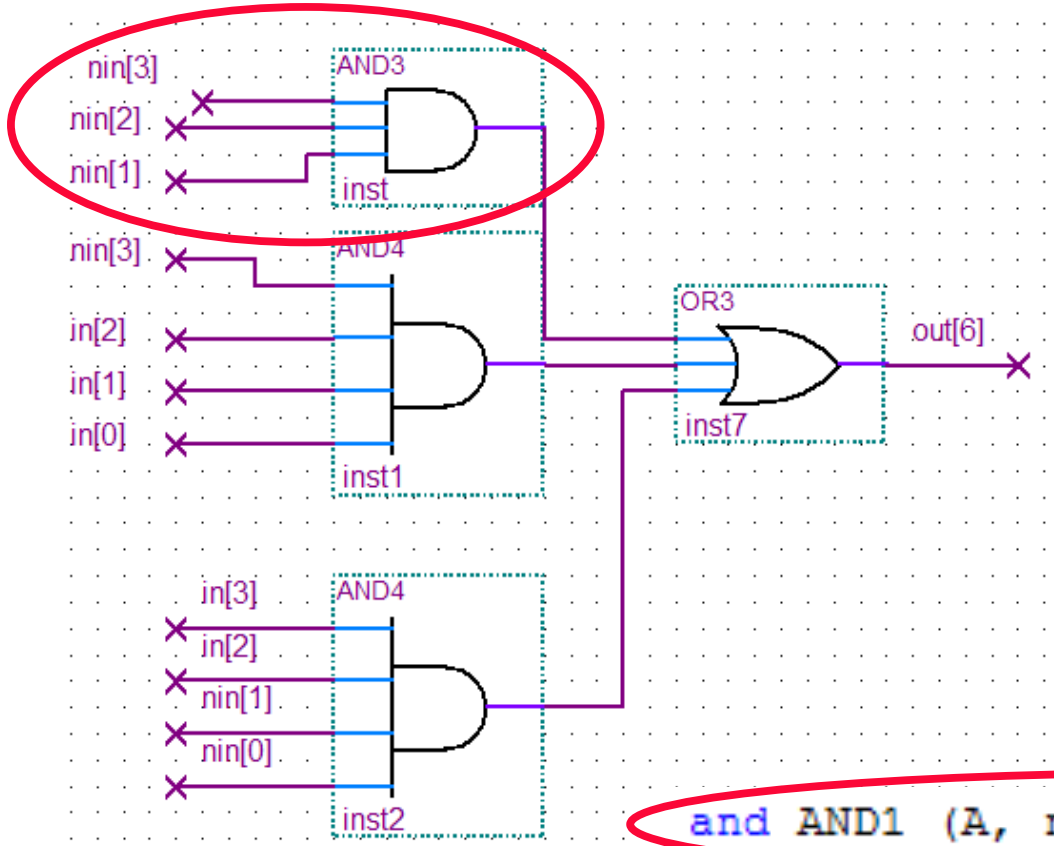
out1 = in3*in2*/in0 + /in3*in2*/in1*in0 + in3*in1*in0 + in2*in1*/in0

out0 = /in3*/in2*/in1*in0 + /in3*in2*/in1*/in0 + in3*in2*/in1*in0 + in3*/in2*in1*in0

TEDIOUS!!!!!



Method 2: Use primitive gates in Verilog



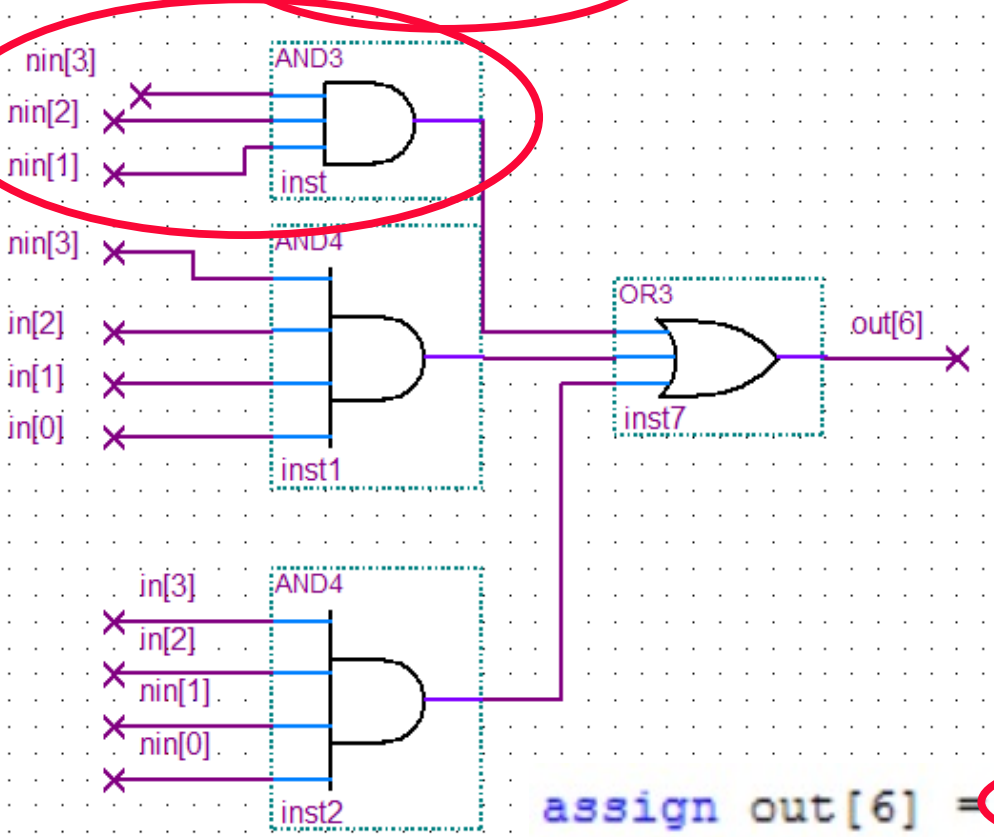
Equally TEDIOUS!!!!!

Direct mapping of gates to primitives

```
and AND1 (A, nin[3], nin[2], nin[1]);
and AND2 (B, nin[3], in[2], in[1], in[0]);
and AND3 (C, in[3], in[2], nin[1], nin[0]);
or  OR1  (out[6], A, B, C);
```

Method 3: Use continuous assignment in Verilog

out6 = /in3*/in2*/in1 + in3*in2*/in1*/in0 + /in3*in2*in1*in0



Much Better?

Direct mapping of Boolean equation using continuous assignment..

```
assign out[6] = ~in[3]&~in[2]&~in[1] |
                in[3]&in[2]&~in[1]&~in[0] |
                ~in[3]&in[2]&in[1]&in[0];
```

Hexto7seg.v (in Verilog)

module & **endmodule** sandwich the content of this hardware module

good header helps documenting your code

specify interface to this module as viewed from outside

specify a 7-bit output bus, out[6] … out[0]

declaration of input and output ports

assign used to specify combinational circuit

```
//------------------------------
// Module name: hex_to_7seg
// Function: convert 4-bit hex value to drive 7 segment display
//           output is low active
// Creator:  Peter Cheung
// Version:  1.0
// Date:     22 Oct 2011
//------------------------------

module hex_to_7seg  (out,in);

    output  [6:0]   out;    // low-active output to drive 7 segment display
    input   [3:0]   in;     // 4-bit binary input of a hexademical number

    assign out[6] = ~in[3]&~in[2]&~in[1] | in[3]&in[2]&~in[1]&~in[0] |
                    ~in[3]&in[2]&in[1]&in[0];
    assign out[5] = ~in[3]&~in[2]&in[0] | ~in[3]&~in[2]&in[1] |
                    ~in[3]&in[1]&in[0] | in[3]&in[2]&~in[1]&in[0];
    assign out[4] = ~in[3]&in[0] | ~in[3]&in[2]&~in[1] | in[3]&~in[2]&~in[1]&in[0];
    assign out[3] = ~in[3]&in[2]&~in[1]&~in[0] | ~in[3]&~in[2]&~in[1]&in[0] |
                    in[2]&in[1]&in[0] | ~in[2]&in[1]&~in[0];
    assign out[2] = ~in[3]&~in[2]&in[1]&~in[0] | in[3]&in[2]&~in[0] |
                    in[3]&in[2]&in[1];
    assign out[1] = in[3]&in[2]&~in[0] | ~in[3]&in[2]&~in[1]&in[0] |
                    in[3]&in[1]&in[0] | in[2]&in[1]&~in[0];
    assign out[0] = ~in[3]&~in[2]&~in[1]&in[0] | ~in[3]&in[2]&~in[1]&~in[0] |
                    in[3]&in[2]&~in[1]&in[0] | in[3]&~in[2]&in[1]&in[0];
endmodule
```

Method 4: Power of behavioural abstraction

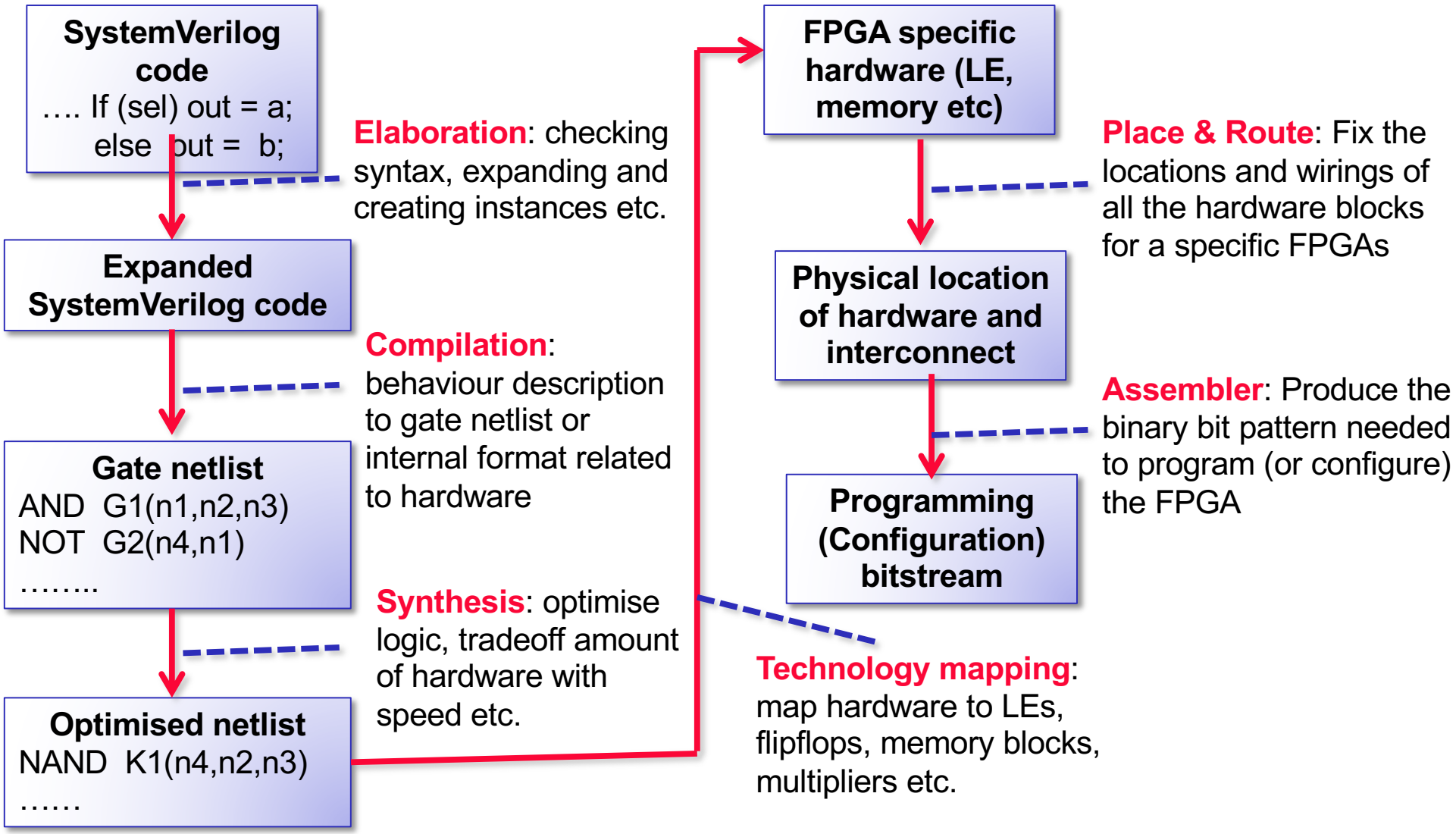
```
module hexto7seg    (
    output logic [6:0]  out,    // low-active
    input  logic [3:0]  in      // 4-bit binar
);
    always_comb
      case (in)
        4'h0: out = 7'b1000000;
        4'h1: out = 7'b1111001;     // -- 0 --
        4'h2: out = 7'b0100100;     // |
        4'h3: out = 7'b0110000;     // 5
        4'h4: out = 7'b0011001;     // |
        4'h5: out = 7'b0010010;     // -- 6 --
        4'h6: out = 7'b0000010;     // |
        4'h7: out = 7'b1111000;     // 4
        4'h8: out = 7'b0000000;     // |
        4'h9: out = 7'b0011000;     // -- 3 --
        4'ha: out = 7'b0001000;
        4'hb: out = 7'b0000011;
        4'hc: out = 7'b1000110;
        4'hd: out = 7'b0100001;
        4'he: out = 7'b0000110;
        4'hf: out = 7'b0001110;
        default: out = 7'b0000000; // def
      endcase
endmodule
```

BEAUTIFUL !!!

- Direct mapping of truth table to case statement
- Close to specification, not implementation

in[3..0]	out[6:0]	Digit
0000	1000000	0
0001	1111001	1
0010	0100100	2
0011	0110000	3
0100	0011001	4
0101	0010010	5
0110	0000010	6
0111	1111000	7
1000	0000000	8
1001	0010000	9
1010	0001000	A
1011	0000011	b
1100	1000110	C
1101	0100001	d
1110	0000110	E
1111	0001110	F

From SystemVerilog code to FPGA hardware



Power of SystemVerilog: Integer Arithmetic

- Arithmetic operations make computation easy:

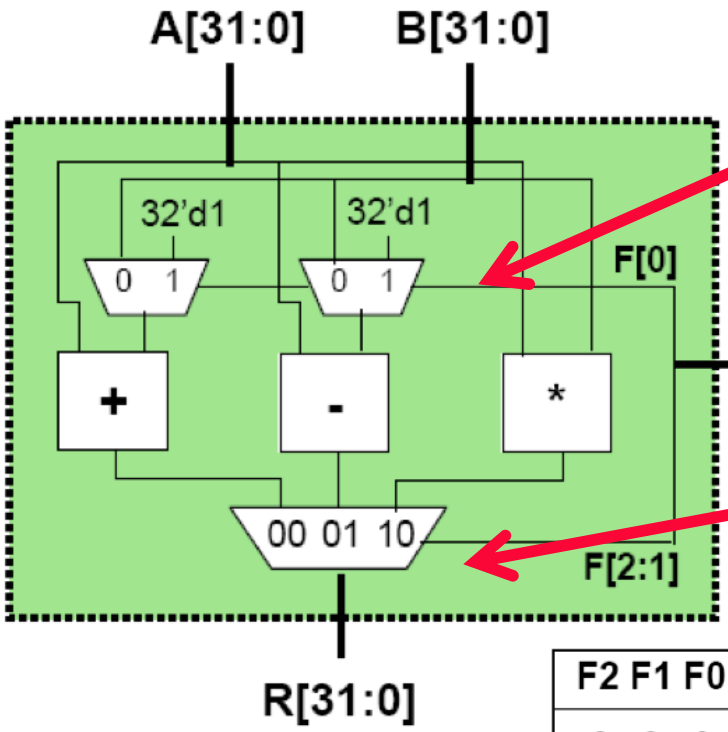
```
module  add32 (
    input  logic [31:0]     a,
    input  logic [31:0]     b,
    output logic [31:0]     sum
);
    assign sum = a + b;
endmodule
```

- Here is a 32-bit adder with carry-in and carry-out:

```
module  add32_carry (
    input  logic [31:0]     a,
    input  logic [31:0]     b,
    input  logic            cin,
    output logic [31:0]     sum,
    output logic            cout
);
    assign {cout, sum} = a + b + cin;
endmodule
```

A larger example – 32-bit ALU in SV

- Here is an 32-bit ALU with 5 simple instructions:



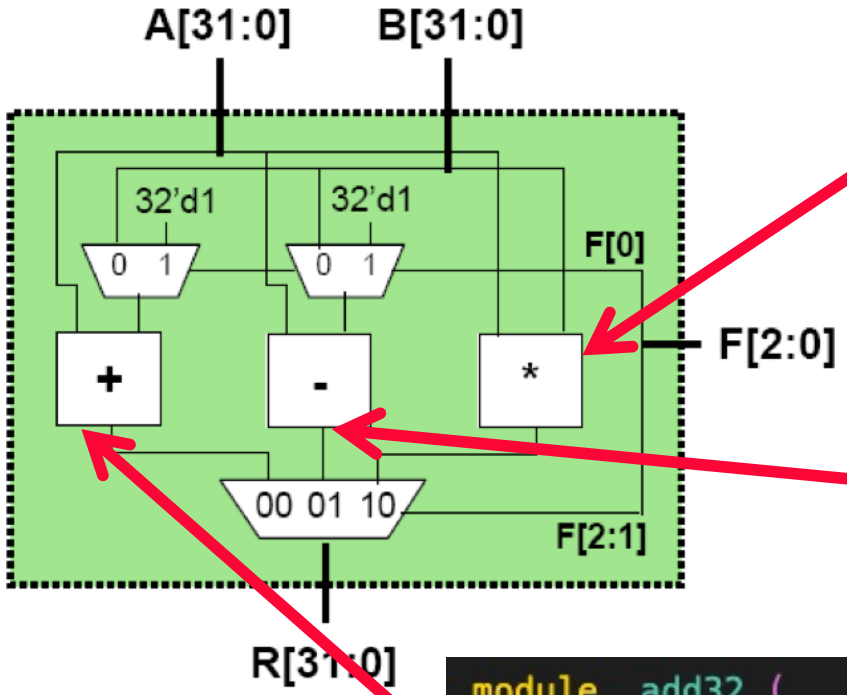
F2 F1 F0	Function
0 0 0	A + B
0 0 1	A + 1
0 1 0	A - B
0 1 1	A - 1
1 0 X	A * B

```
module  mux2to1 (
    input  logic [31:0]     i0,
    input  logic [31:0]     i1,
    input  logic            sel,
    output logic [31:0]     out
);
    assign out = sel ? i1 : i0;
endmodule
```

```
module  mux3to1 (
    input  logic [31:0]     i0,
    input  logic [31:0]     i1,
    input  logic [31:0]     i3,
    input  logic [1:0]      sel,
    output logic [31:0]     out
);
    always_comb
        case (sel)
            2'b00:  out = i0;
            2'b01:  out = i1;
            2'b10:  out = i2;
            default:  out = 32'bx;
        endcase
endmodule
```

The arithmetic modules

- Here is an 32-bit ALU with 5 simple instructions:




```
module  mul16 (
    input  logic [15:0]     i0,
    input  logic [15:0]     i1,
    output logic [31:0]     pro
);
    assign  prod = i0 * i1;
endmodule
```

```
module  sub32 (
    input  logic [31:0]     i0,
    input  logic [31:0]     i1,
    output logic [31:0]     diff
);
    assign diff = i0 - i1;
endmodule
```

```
module  add32 (
    input  logic [31:0]     i0,
    input  logic [31:0]     i1,
    output logic [31:0]     sum
);
    assign sum = i0 + i1;
endmodule
```

Top-level module – putting them together

- Given submodules:

```
module mux2to1 (i0, i1, sel, out);
module mux3to1 (i0, i1, i2, sel, out);
module add32 (i0, i1, sum);
module sub32 (i0, i1, diff);
module mul16 (i0, i1, prod);
```

```
module alu (
    input  logic [31:0]     a,
    input  logic [31:0]     b,
    input  logic [2:0]      f,
    output logic [31:0]     r
);
    logic [32:0]    addmux_out, submux_out;
    logic [32:0]    add_out, sub_out, mul_out;

    mux2to1 adder_mux (b, 32'd1, f[0], addmux_out);
    mux2to1 sub_mux   (b, 32'd1, f[0], submux_out);
    add32   our_adder (a, addmux_out, add_out);
    sub32   out_sub   (a, submux_out, sub_out);
    mul16   our_mult  (a[15:0], b[15:0], mul_out);
    mux3to1 output_mux(add_out, sub_out, mul_out, f[2:1], r);
endmodule
```