CSCE 611
Verilog HDL

Instructor: Jason D. Bakos

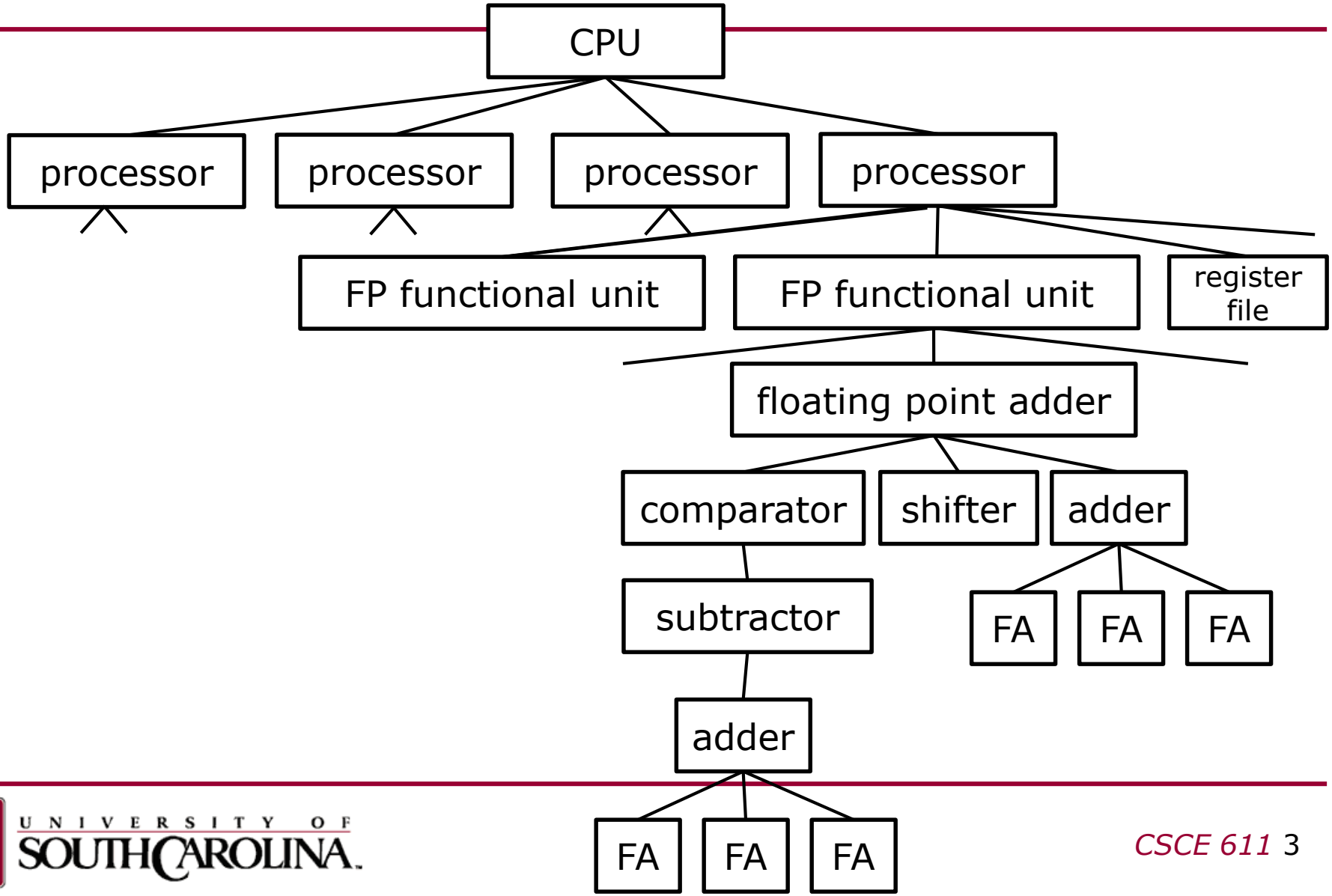
UNIVERSITY OF SOUTH CAROLINA

Review

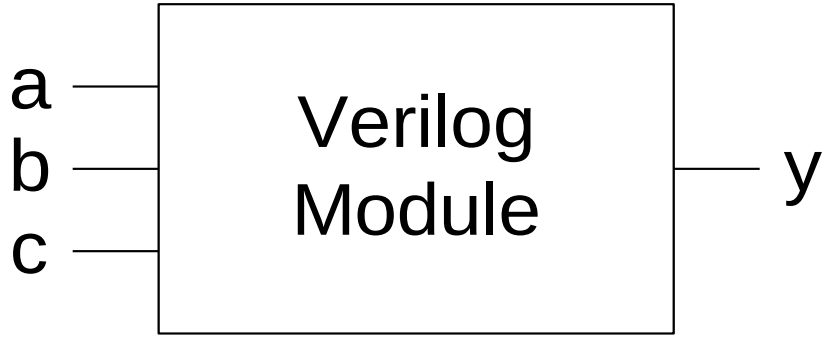
- Last lecture:
  - What is HDL? How is HDL different from other high-level languages?
  - What is an FPGA? How is it different from a PLA?
  - How are FPGAs programmed? Once programmed, how are they used?
  - What is the HDL design flow for FPGAs?
  - How is the design flow different between FPGAs and ASICs?

- Notes:
  - A 1 billion transistor CPU has the equivalent of ~71 million 7-input gates
  - Carry-in logics shortens routes for LEs forming a carry chain

Design Hierarchy



Verilog Modules



- Basic unit of design hierarchy
- Two types of modules:
  - Behavioral: describe what a module does
  - Structural: describe how a module is built from simpler modules
- One module per file
- File name is <module name>.v

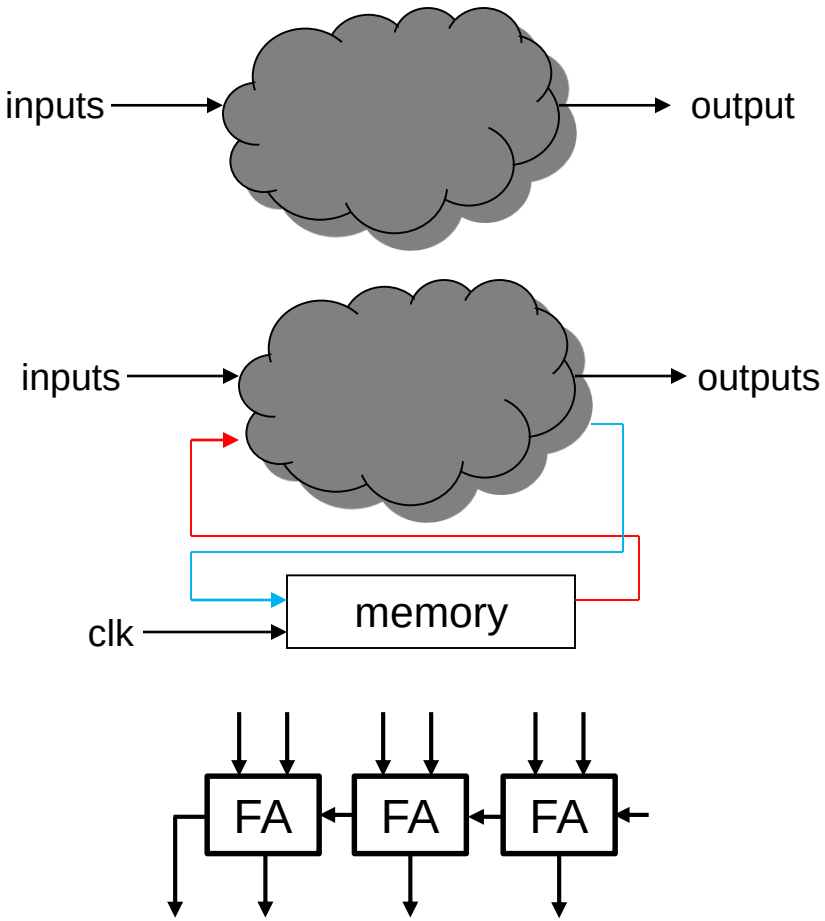
Hardware Description Language

- Digital logic design is comprised of:
  - Some arrangement of off-the-shelf components
    - "Structural HDL"
  - Combinational behavior
    - F(inputs) = output
    - Example:

```
assign a = b and c;
always @(*) a = b and c;
```

  - Sequential behavior
    - F(inputs,input history) = output
    - Example:

```
always @(posedge clk)
  a = up ? a + 8'b1 : a - 8'b1;
```

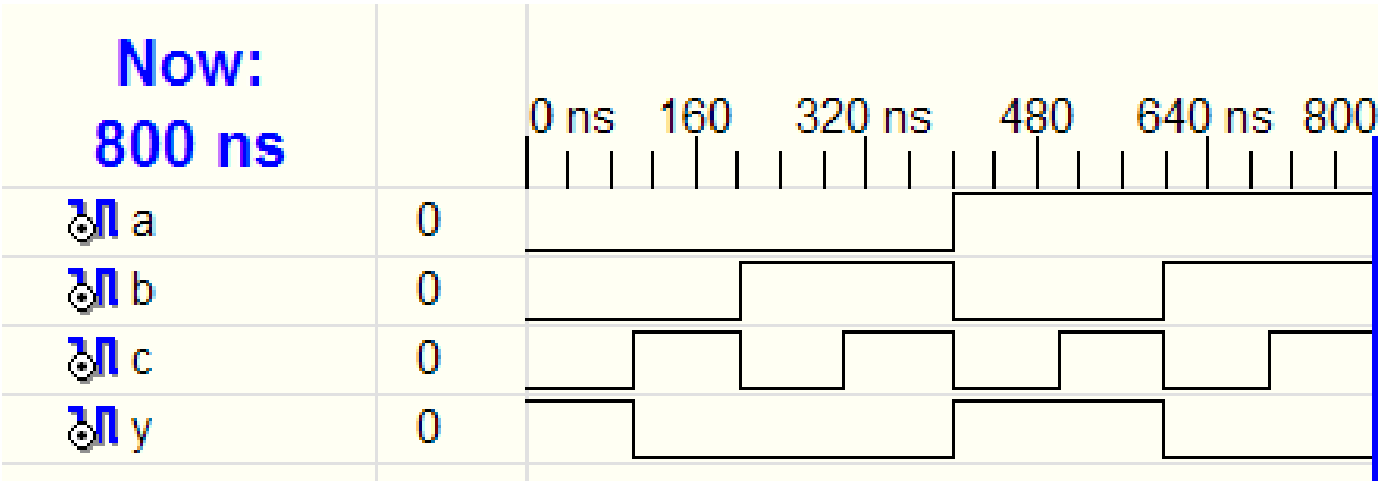


Behavioral Verilog Example

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

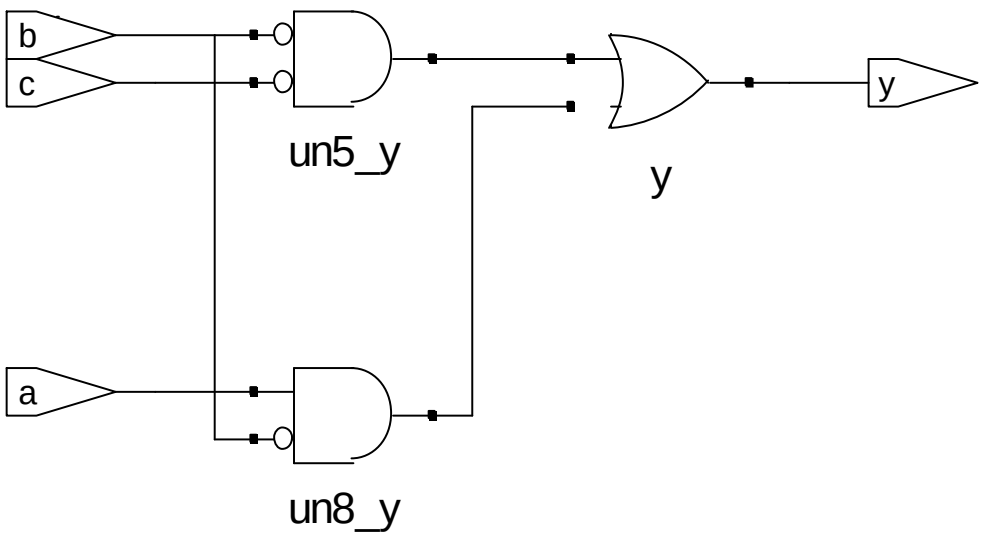
Behavioral Verilog Example

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



Behavioral Verilog Example

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



Verilog Syntax

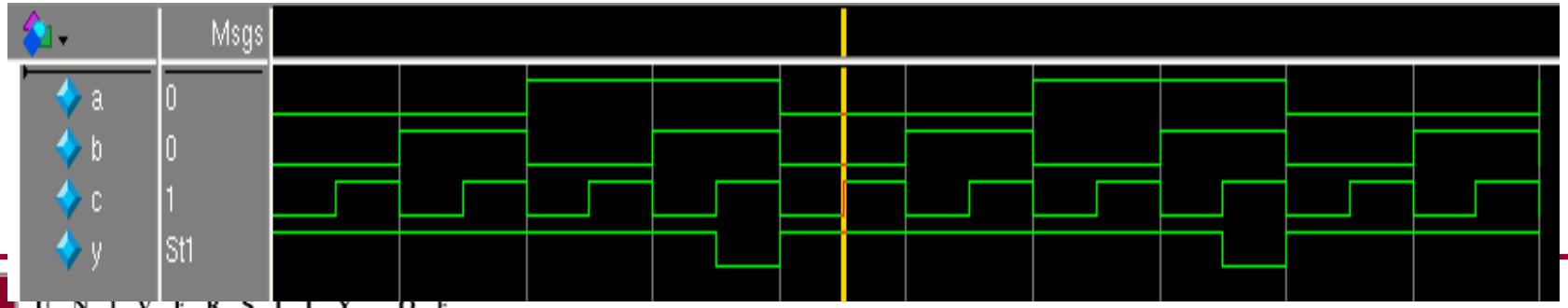
- Case sensitive
  - Example: reset and Reset are not the same signal
- No names that start with numbers
  - Example: 2mux is an invalid name
- Whitespace ignored
- Comments:
  - // single line comment
  - /* multiline
    comment */

Structural Modeling - Hierarchy

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

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

module nand3(input a, b, c, output y);
  wire n1;                      // internal signal
  and3 andgate(a, b, c, n1);    // instance of and3
  inv  inverter(n1, y);         // instance of inverter
endmodule
```

formal arguments

actual arguments



Arguments

definition:

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

instantiation:

```
and3 andgate(a, b, c, n1);
```

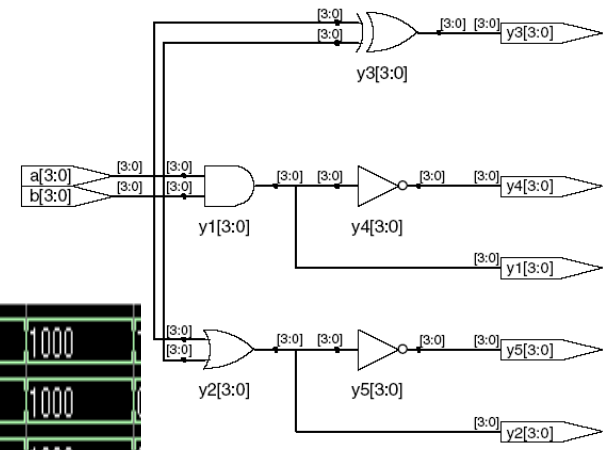
alternative instantiation:

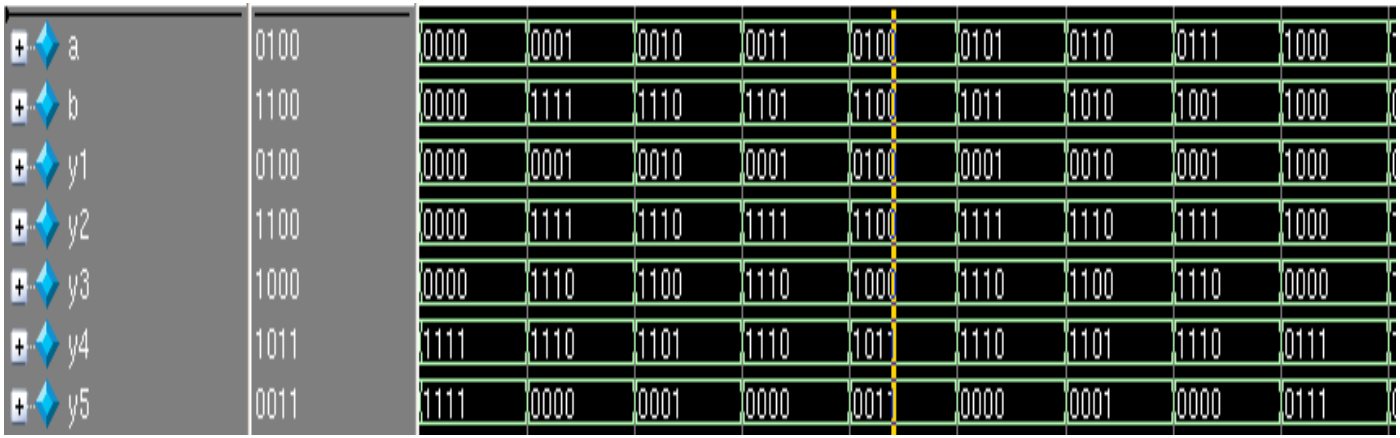
```
and3 andgate(.y(n1), .b(b), .a(a), .c(c))
```

Bitwise Operators

```
module gates(input  [3:0]  a, b,
             output [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
```

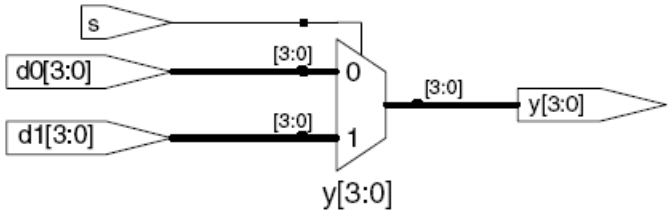
order of statements doesn't matter!!!

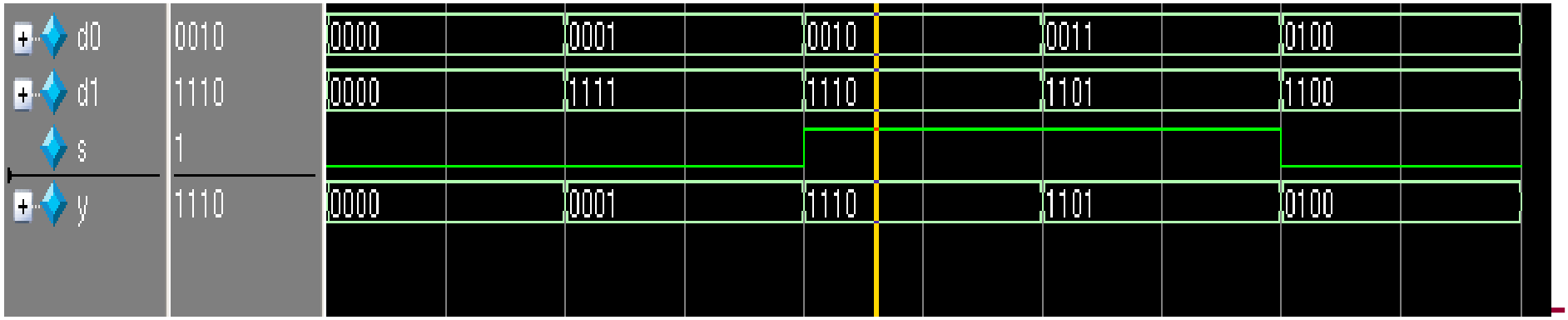




Conditional Assignment

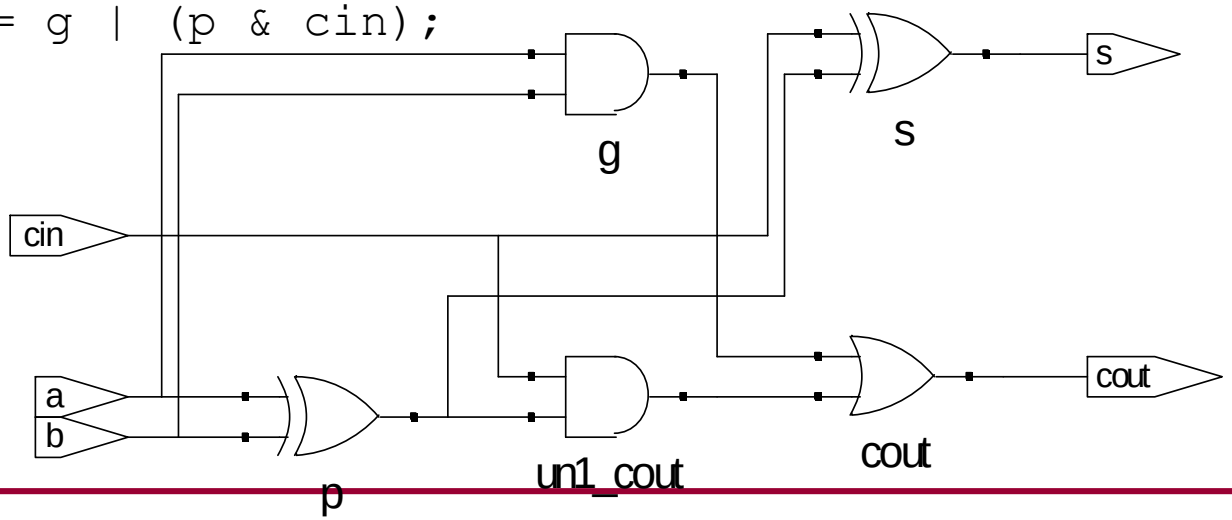
```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);
   assign y = s ? d1 : d0;
endmodule
```





Internal Signals

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

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

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



Precedence

Highest

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

Lowest

Numbers

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

Bit Manipulations: Example 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

or:  assign {a[2:1], b[0]} = y;

// underscores (_) are used for formatting only to make
   it easier to read.  Verilog ignores them.
```

- Example:
  - Sign extend b[15:0]

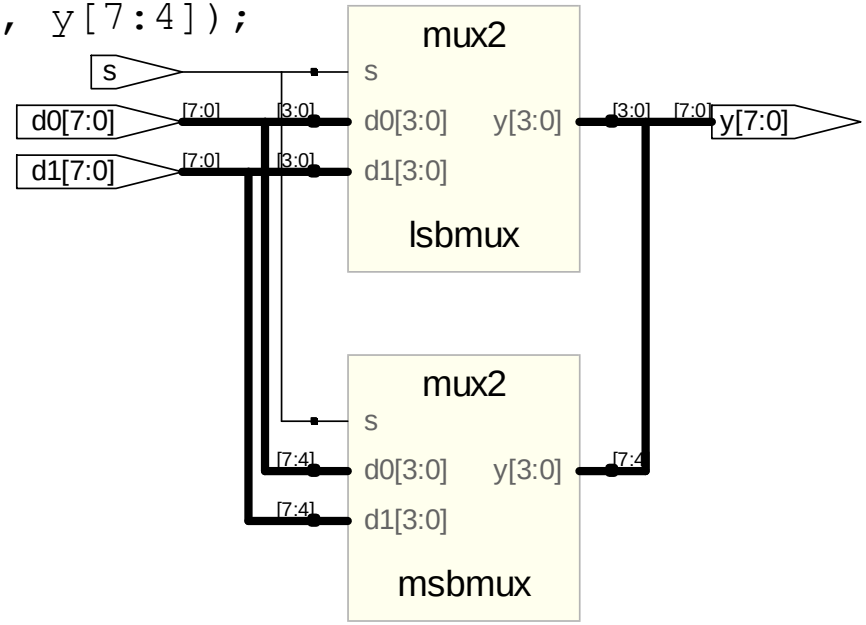
```
assign b_ext = {b[15], b[15], b[15], b[15], b[15], b[15], b[15], b[15],
b[15], b[15], b[15], b[15], b[15], b[15], b[15], b[15], b};
```

or

```
assign b_ext = { {16{b[15]}}, b };
```

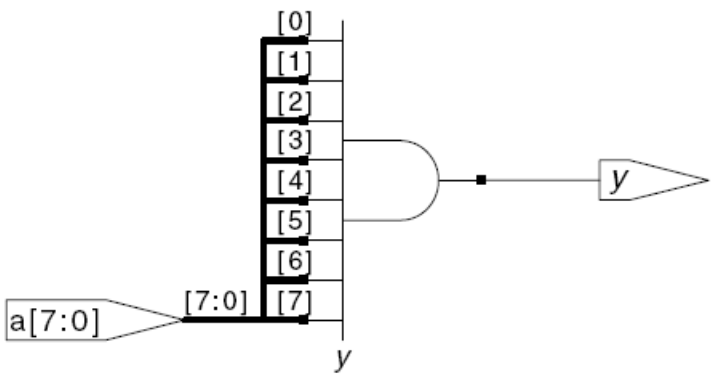
Bit Manipulations: Example 2

```
module mux2_8(input  [7:0] d0, d1,
              input        s,
              output [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
```



Reduction Operators

```
module and8(input  [7:0] a,
            output       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
```



Available reduction operators:
& (and)
~& (nand)
| (or)
~| (nor)
^ (xor)
~^ (xnor)

Always Statement

- Need a mechanism for creating:
  - Complex combinational logic
  - Sequential logic
  - Non synthesizable behaviors, e.g. clocks that repeat forever

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

Whenever the event in the sensitivity list occurs, the statement is executed

For multiple statements, use begin/end

Always Statement

- Always statements allow additional programming constructs not available with assign statement:
  - if statement
  - case statement
  - loops (probably won't use in this class)

```
wire out1;
assign out1 = sel ? foo : bar;
```

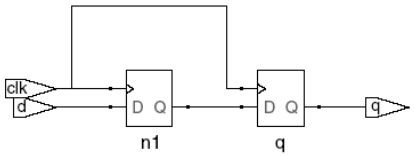
— or —

```
reg out1;
always @(*) begin
  if (sel) out1 = foo;
  else out1 = bar;
end
```

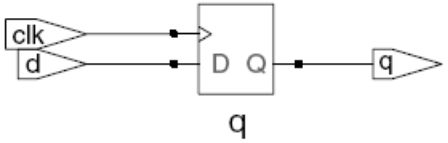
Blocking vs. Nonblocking Assignments

- <= is a "nonblocking assignment"
  - Occurs simultaneously with others (takes effect at end)
- = is a "blocking assignment"
  - Occurs in the order it appears in the file
  - Uses sequential semantics, use to build serialized paths

```
// Good synchronizer using
// nonblocking assignments
module syncgood(input      clk,
                input      d,
                output reg q);
  reg n1;
  always @(posedge clk)
    begin
      n1 <= d;  // nonblocking
      q  <= n1; // nonblocking
    end
endmodule
```




```
// Bad synchronizer using
// blocking assignments
module syncbad(input      clk,
               input      d,
               output reg q);
  reg n1;
  always @(posedge clk)
    begin
      n1 = d;  // blocking
      q  = n1; // blocking
    end
endmodule
```



Rules for Signal Assignment

- Use always @(posedge clk) and nonblocking assignments (<=) to model synchronous sequential logic

```
always @ (posedge clk)
    q <= d; // nonblocking
```

- Use continuous assignments (assign …) to model simple combinational logic.

```
assign y = a & b;
```

- Use always @ (*) and blocking assignments (=) to model more complicated combinational logic where the always statement is helpful.
- Do not make assignments to the same signal in more than one always statement or continuous assignment statement

Default Assignments

- For combinational logic, make sure all output signals are assigned on every control path
- Can use default signals to ensure this:

```
module default_assignment_test (input sel, foo, bar,
       output reg out1);

always @(*) begin
  out1 = bar;
  if (sel==1) out1 = foo;
end

endmodule
```

Combinational Logic using **always**

```
// combinational logic using an always statement
module gates(input       [3:0] a, b,
             output reg [3:0] y1, y2, y3, y4, y5);
  always @(*)        // 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

Combinational Logic using **case**

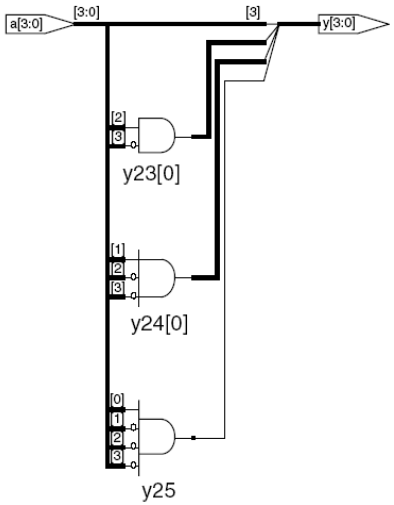
```
module sevenseg(input      [3:0] data,
                output reg [6:0] segments);
  always @(*)
    case (data)
      //                abc_defg
      0: segments = 7'b111_1110;
      1: segments = 7'b011_0000;
      2: segments = 7'b110_1101;
      3: segments = 7'b111_1001;
      4: segments = 7'b011_0011;
      5: segments = 7'b101_1011;
      6: segments = 7'b101_1111;
      7: segments = 7'b111_0000;
      8: segments = 7'b111_1111;
      9: segments = 7'b111_1011;
      default: segments = 7'b000_0000; // required
    endcase
endmodule
```

Combinational Logic using `case`

- In order for a case statement to imply combinational logic, all possible input combinations must have a corresponding output assignment in the HDL
- Remember to use a default statement when necessary
- Use begin/end for multiple statements

Combinational Logic using **casez**

```
module priority_casez(input       [3:0] a,
                      output reg  [3:0] y);

  always @(*)
    casez(a)
      4'b1???: y = 4'b1000;  // ? = don't care
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
   endcase

endmodule
```



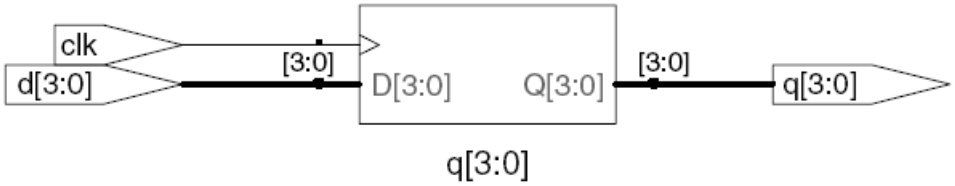
Sequential Logic

- Not listing all possible inputs in sensitivity list or not assigning all possible outputs on every control path will lead to memory being inferred

- When you intend to infer memory, include an explicit clk signal

Sequential Logic: D Flip-Flop

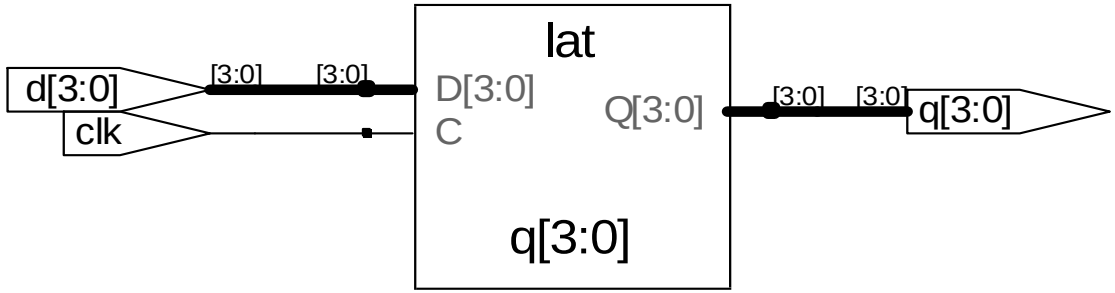
```
module flop(input              clk,
            input       [3:0]  d,
            output reg  [3:0]  q);

  always @ (posedge clk)
    q <= d;                    // pronounced "q gets d"

endmodule
```



- Any signal assigned in an always statement must be declared reg
- A variable declared reg is not necessarily a registered output

Latch

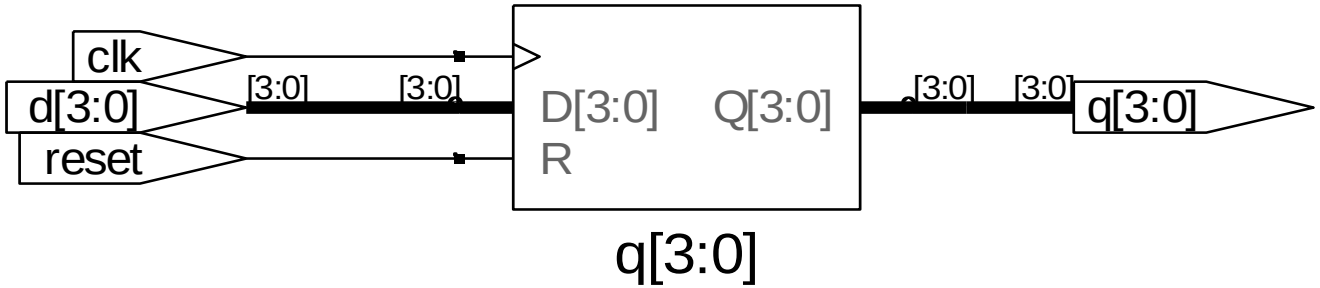
```
module latch(input             clk,
             input       [3:0] d,
             output reg  [3:0] q);

  always @ (clk, d)
    if (clk) q <= d;

endmodule
```



Warning: We won't use latches in this course, but you might write code that inadvertently implies a latch. So if your synthesized hardware has latches in it, this indicates an error

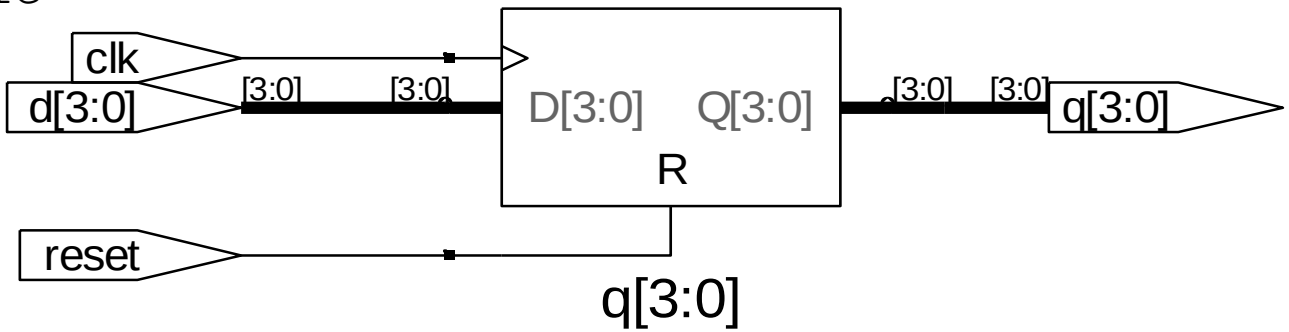
Resettable D Flip-Flop

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

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

endmodule
```



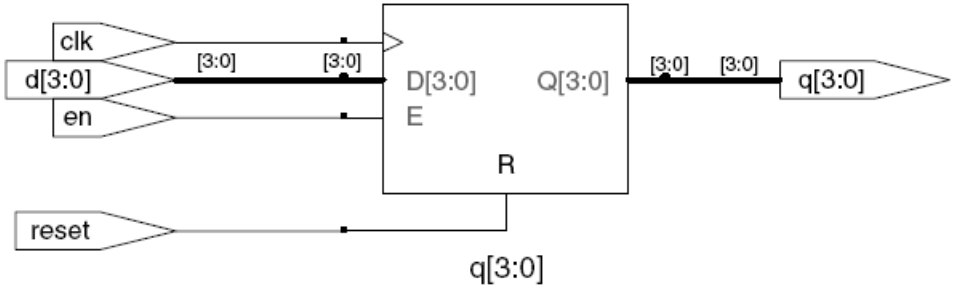
Resettable D Flip-Flop

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

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

endmodule
```



D Flip-Flop with Enable

```
module flopren(input            clk,
               input            reset,
               input            en,
               input      [3:0] d,
               output reg [3:0] q);

  // asynchronous reset and enable
  always @ (posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;

endmodule
```



Sequential Logic

- Create a 5-bit up/down counter
- Like register but deletes input d and increment count when enable is high on rising edge of clock

```
module cnt6 (input clk,rst,en_up,en_down,output reg
[5:0] cnt);

always @(posedge clk) begin
  if (rst) cnt <= 0;
  else if (en_up) cnt <= cnt+1;
  else if (en_down) cnt <= cnt-1;
end
endmodule
```

Parameterized Modules

- 2:1 mux

```
module mux2
  #(parameter width = 8)  // name and default value
   (input  [width-1:0] d0, d1,
    input              s,
    output [width-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

- Instance with 8-bit bus width (uses default)

```
mux2 mux1(d0, d1, s, out);
```

- Instance with 12-bit bus width

```
mux2 #(12) lowmux(d0, d1, s, out);
```

Parameters

- Another way of including parameters:

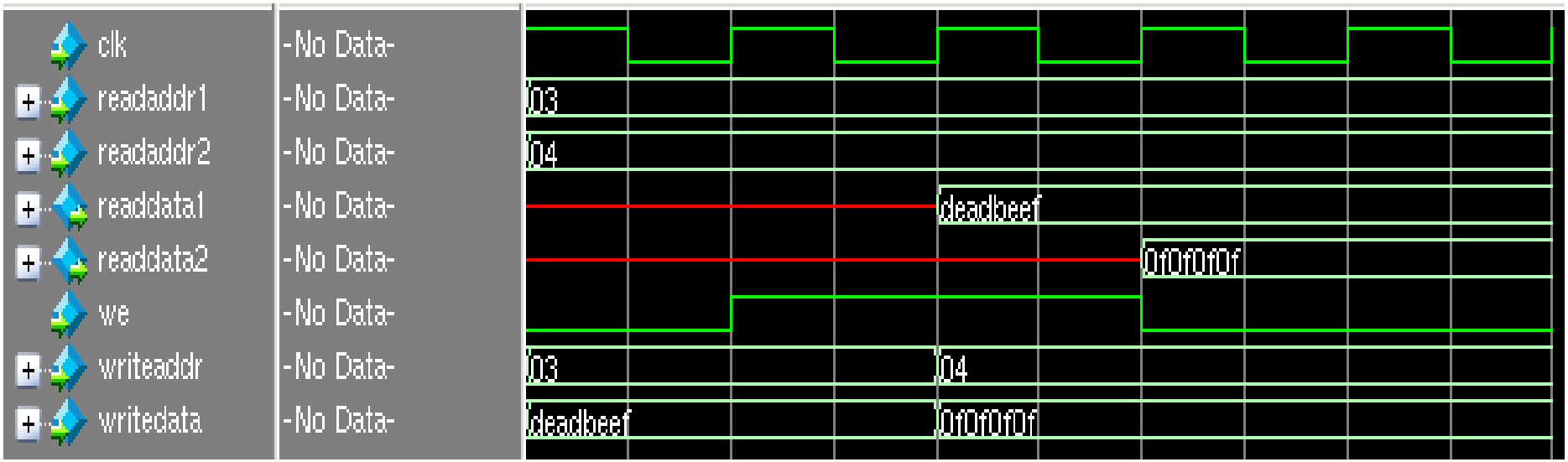
```
module mux2 (input  [width-1:0] d0, d1, input s, output
   [width-1:0] y);
  parameter width = 8;
  assign y = s ? d1 : d0;
endmodule

mux2 #(.width(16)) mymux (in0, in1, select,
mux_out);
```

RAM

- Example register file
  - 2 asynchronous read ports
  - 1 write port

```
module regfile32x32 (input clk,we,input [4:0] readaddr1, readaddr2,
writeaddr, input [31:0] writedata, output [31:0] readdata1, readdata2);

reg [31:0] mem[31:0];

always @(posedge clk) begin
  if (we) mem[writeaddr] <= writedata;
end

assign readdata1 = mem[readaddr1];
assign readdata2 = mem[readaddr2];

endmodule
```

RAM


clk
readaddr1
readaddr2
readdata1
readdata2
we
writeaddr
writedata
-No Data-
03
04
deadbeef
0f0f0f0f
03
04
deadbeef
0f0f0f0f

RAM

- To initialize, add

```
initial begin
  $readmemh("regs.dat",mem);
end
```

- regs.dat could contain (for example) two lines to initialize address locations 0 and 1:

```
DEADBEEF
0F0F0F0F
```

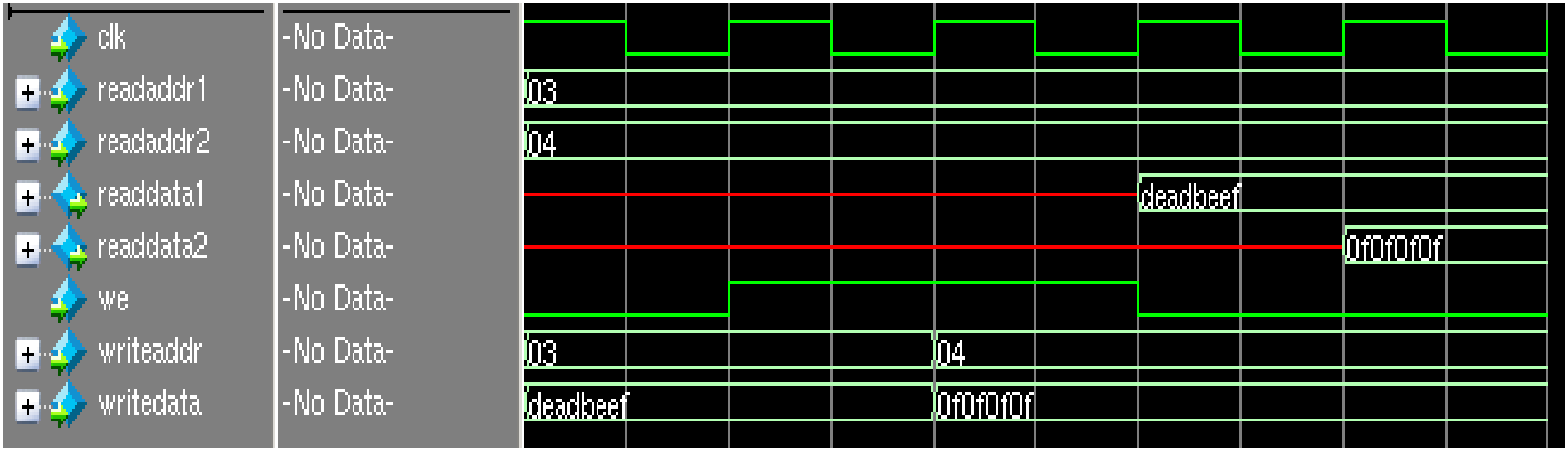
RAM

- On FPGA, this RAM uses 1024 LE registers
- In order to use dedicated SRAMs (M4Ks), read ports must be synchronous:

```
module regfile32x32 (input clk,we,input [4:0] readaddr1, readaddr2, writeaddr, input
[31:0] writedata,output reg [31:0] readdata1, readdata2);

reg [31:0] mem[31:0];

always @(posedge clk) begin
  if (we) mem[writeaddr] <= writedata;
  readdata1 <= mem[readaddr1];
  readdata2 <= mem[readaddr2];
end

endmodule
```

- Uses 0 LEs and 2/105 M4Ks

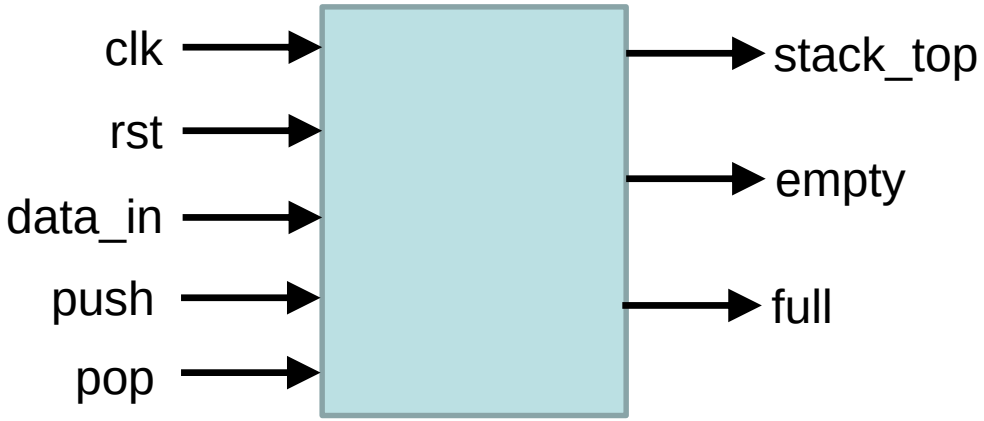
RAM


clk
readaddr1
readaddr2
readdata1
readdata2
we
writeaddr
writedata
-No Data-
03
04
deadbeef
0f0f0f0f

Structural Verilog

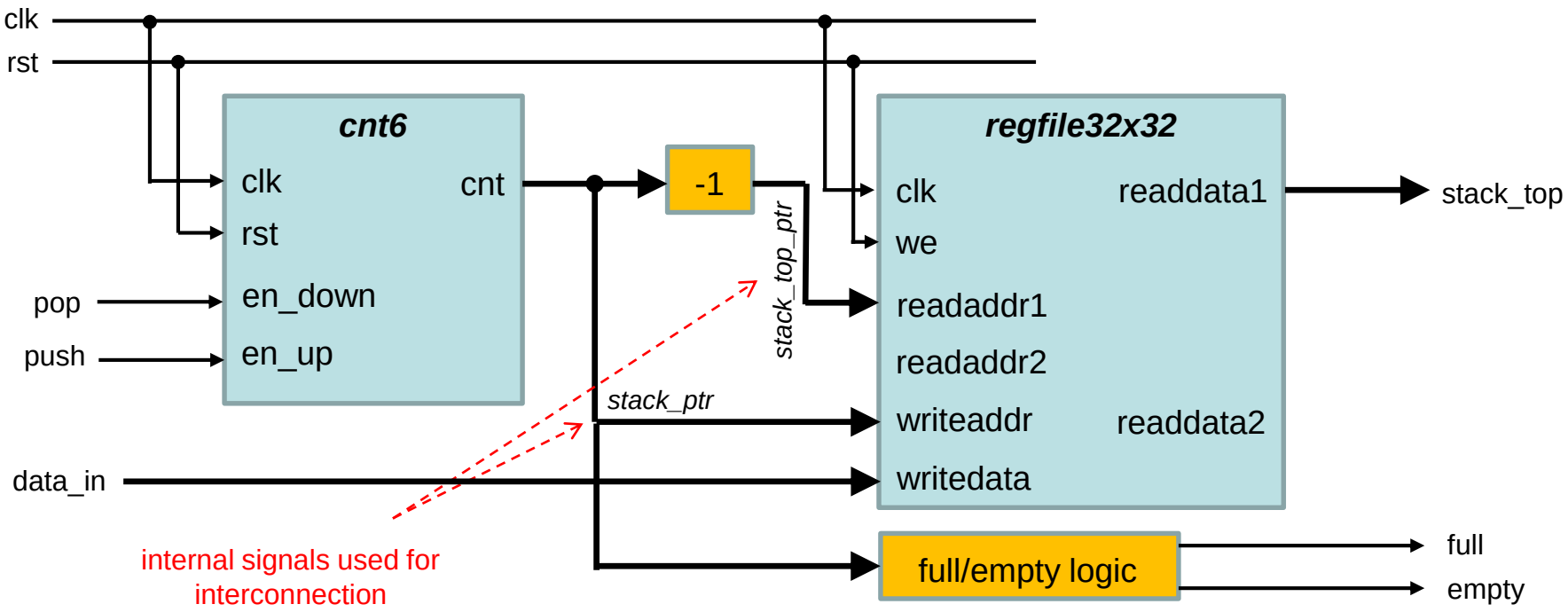
- Typically HDL files are built from the bottom-up
- Let's build a stack memory using our
  - 32x32 register file (asynchronous read)
  - 5-bit up/down counter

- Top-level interface:



Structural Verilog

- Internal structure:



Structural Verilog

```
module stack (input clk,rst,pop,push,input [31:0] data_in,
              output [31:0] stack_top,output full,empty);

wire [4:0] stack_top_ptr,stack_ptr;

cnt6 stackctr(clk,rst,push,pop,stack_ptr);

assign stack_top_ptr = stack_ptr-1;
assign full = stack_ptr==32 ? 1 : 0;
assign empty = stack_ptr==0 ? 1 : 0;

regfile32x32 stackregs(.clk(clk),
                       .we(push),
                       .readaddr1(stack_top_ptr),
                       .readaddr2(5'b0),
                       .writeaddr(stack_ptr),
                       .writedata(data_in),
                       .readdata1(stack_top),
                       .readdata2());
endmodule
```

Structural Verilog