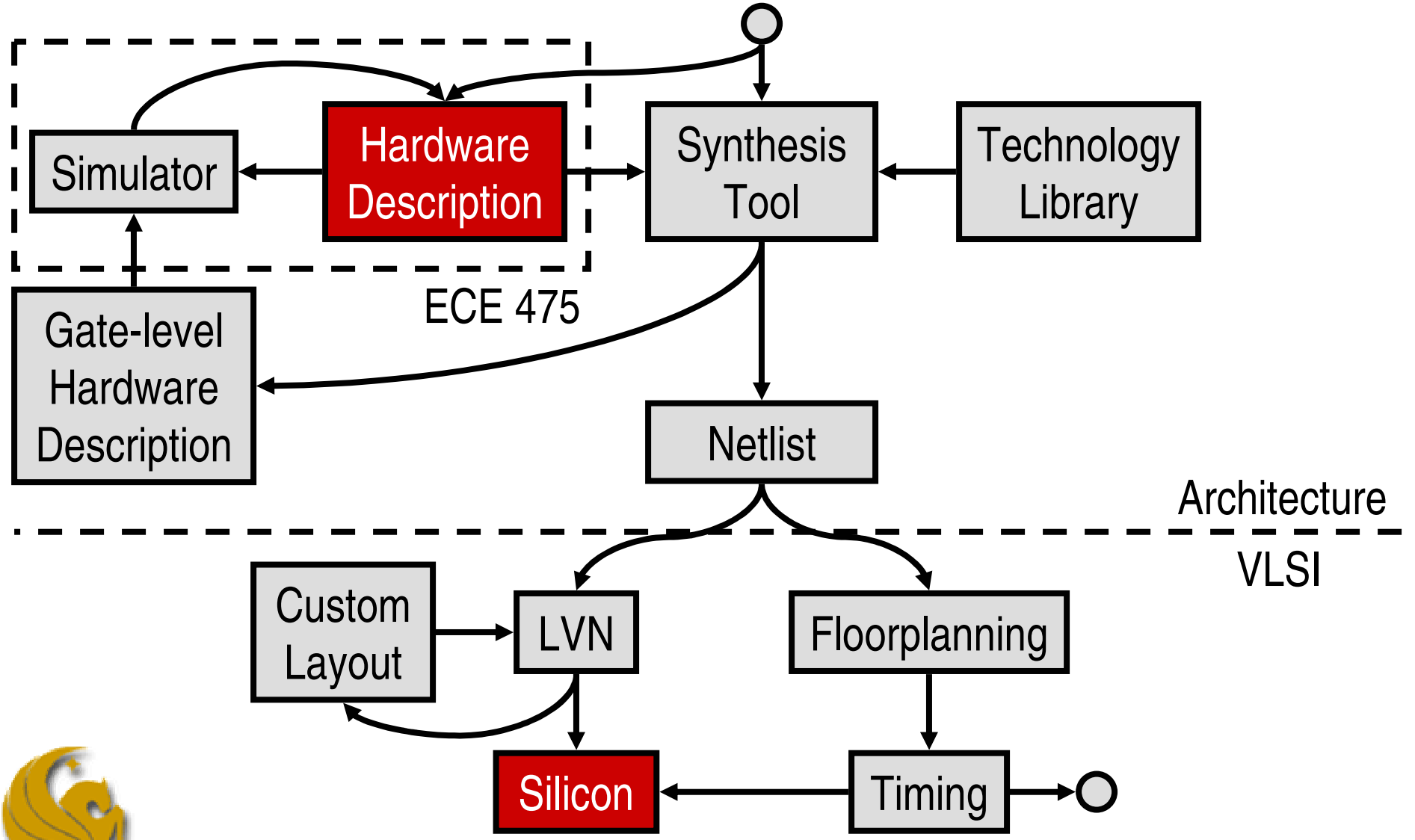
What Is Verilog?

- Hardware Description Language (HDL)
  - Not a *programming* language! (more on this later)
- Describes digital systems
  - Behavioral
  - Structural
- How is this useful?
  - Can't draw gate-level schematics of complex systems – big mess
  - Gate-level simulation unnecessarily slow
  - HDLs faster to simulate, and still provide:
    - Synthesizable low-level implementation
    - Hardware concurrency
    - Ease of use

Synthesizable?



Verilog vs. VHDL

- VHDL (VHSIC HDL)
  - ADA-like syntax (ADA anyone?)
  - Older, less expressive
- Verilog
  - C-like syntax (C anyone?)
  - Larger user community
  - Not VHDL

HDL vs. Programming Language

NOT A PROGRAMMING LANGUAGE

- Repeat on every keystroke: “I’m… designing… hardware…”
- No variables (outlawed) – signals!
  - Regs (containers)
  - Wires (connections)
- HDLs concurrent
  - Which happens first?

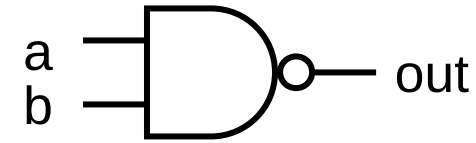
```
assign a = ~b;
assign c = d;
```

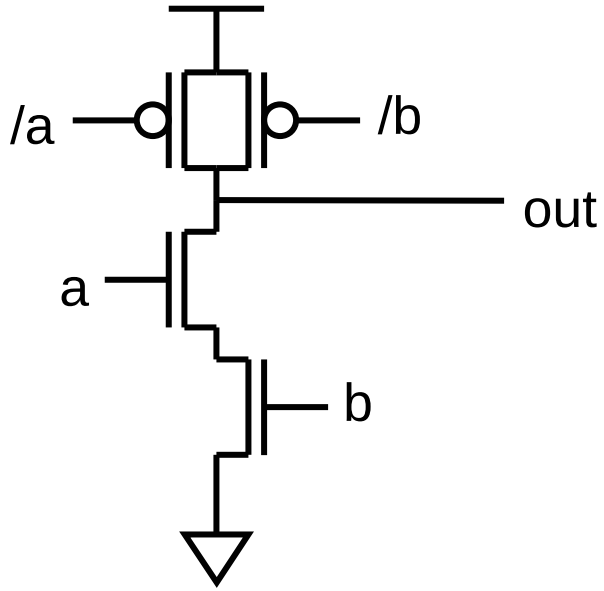
- Operators do not come for free – actual hardware!
  - Use ‘**+**’, ‘**-**’, ‘**<<**‘ sparingly; never use ‘*’, ‘/’

NAND Gate, Behavioral

```
module NAND(a,b,out);
input a;
input b;
output out;

  assign out = ~(a&b);
endmodule
```

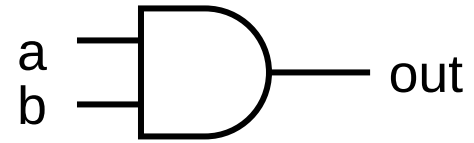


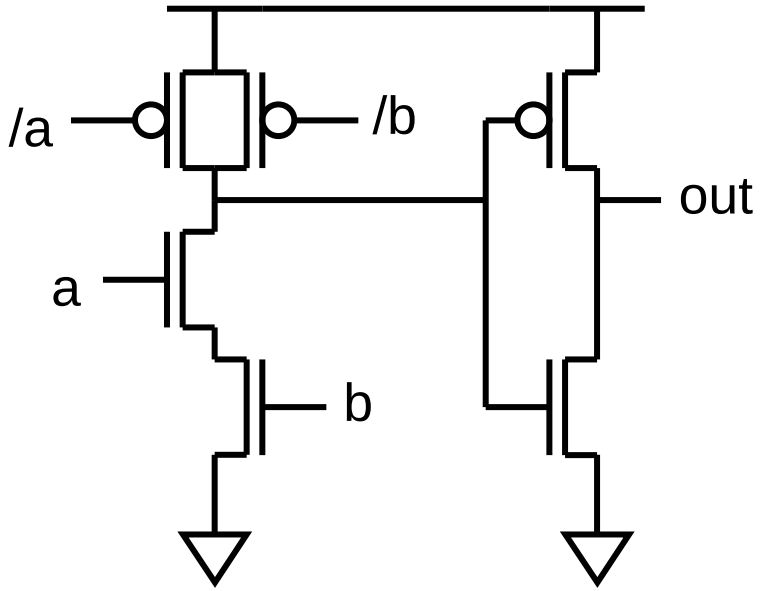


AND Gate, Behavioral

```
module AND(x,y,out);
input x;
input y;
output out;

  assign out = x&y;
endmodule
```

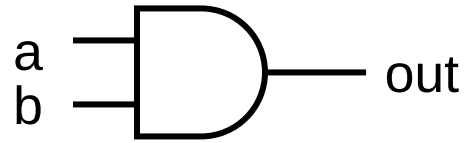


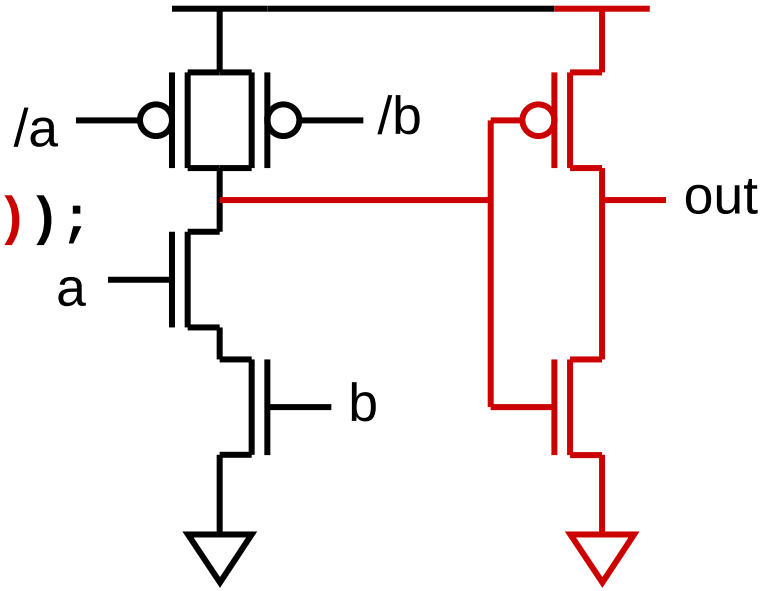


AND Gate, Structural

```
module AND(x,y,out);
input x;
input y;
output out;
wire z;

  NAND MyNAND(.a(x),.b(y),.out(z));
  assign out = ~z;
endmodule
```





Combinational Logic

- Done using *assign* statements
- LHS must be declared *wire*
  - Cannot feed into *reg* – it's combinational!
- Typical operators
  - '&', '|', '^', '~', instantiate corresponding gates
  - '==', '!=', instantiate comparators, return one bit
  - Physical data types: '0', '1', 'x' ("don't care"), 'z' ("high impedance")

```
assign s = a^b^ci;
assign co = a&b|a&ci|b&ci;
```

(What does this do?)

Buses

- Can actually operate on multiple bits in parallel
  - Correspondingly more hardware, of course
  - Default bit width is 1

```
module AND(x,y,out);
input x;
input y;
output out;

  assign out = x&y;
endmodule
```

```
module AND8(x,y,out);
input [7:0] x;
input [7:0] y;
output [7:0] out;

  assign out = x&y;
endmodule
```

Concatenation, Repetition

- Syntax: *R{E1,E2,…,En}*
  - *R* repetitions (default 1) of the concatenation of *E1, E2, …, En*

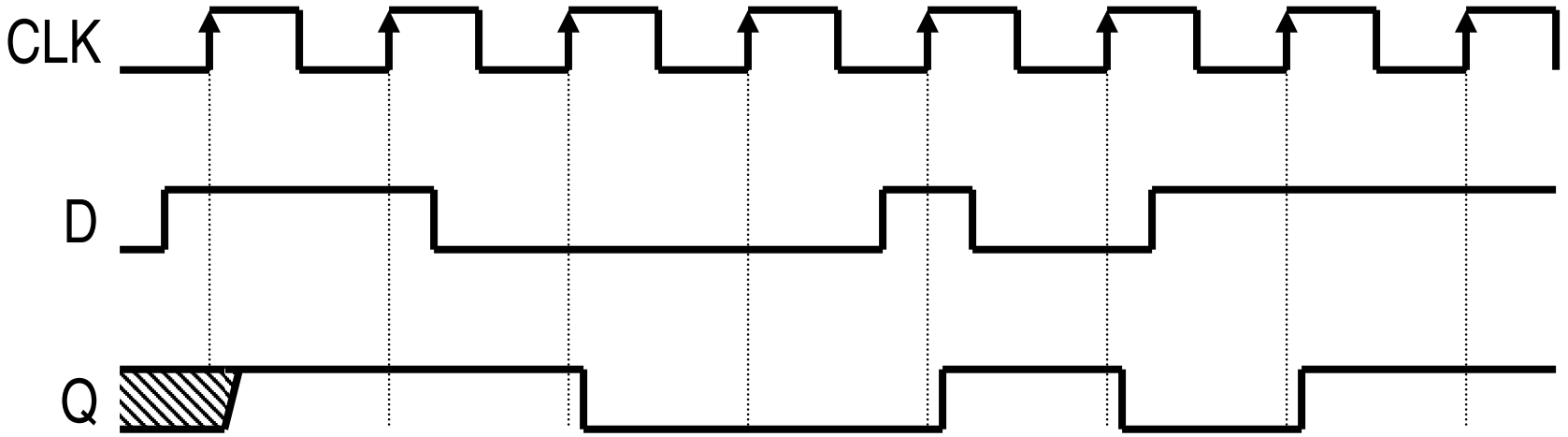
```
reg[15:0] a;
reg[31:0] b;
wire[31:0] out;

assign out = {16{a[15]},a}+b;
```

(What does this do?)

Sequential Logic

- Finite State Machines (CDA 3103 anyone?)



- Need event-driven simulation capability
- Need to trigger on edge – not value

(What is this?)

Sequential Logic

```
module DFF(d,q,clk);
input clk;
input d;
output q;
reg q;

  always @(posedge clk) begin
    q <= `TICK d;
  end
endmodule
```

- Can be *negedge* as well (and *clk* any other name)
- `*define TICK #2* (two Verilog time units) – *clk* period should be >> 2
- `*TICK q <= d;* – Legal! Wrong!

Sequential Logic

- Always use nonblocking assignment '<=' in sequential *always* blocks
- Always use *'TICK* before RHS in sequential always blocks
- Clock only signal in sensitivity list
- LHS must be declared *reg*
  - cannot use *wire* – it's sequential logic!
- Hoist combinational logic outside of *always* blocks as much as possible

```
…
always @(posedge clk) begin
    q <= `TICK a&(32{b==c});
  end
```

Legal

```
wire[31:0] d;
assign d = a&(32{b==c});
…
always @(posedge clk) begin
    q <= `TICK d;
  end
```

Preferred

Control Flow

- Can be used in *always* blocks
- Instantiates actual mux – not programming!

```
module DFF(d,r,q,clk);
input clk; input d; input r;
output q;
reg q;

  always @(posedge clk) begin
    if(r == 1'b1) begin
      q <= `TICK 1'b0;
    end
    else begin
      q <= `TICK d;
    end
  end
endmodule
```

(What does this do?)

Combinational always *Blocks*

- Useful for complex combinational logic
- All RHS signals must appear on sensitivity list
- LHS must be assigned in every possible case
  - otherwise implied sequential logic!

```
always @(sel or a) begin
  if(sel == 2'b0) begin
    z = 1'b0;
  end
  else if(sel == 2'b1) begin
    z = a;
  end
end
```

(What does this do? Is it correct?)

Extra Hardware?

- Watch out for “programming” too much hardware
  - Fortunately synthesis tool (somewhat) smart – but don’t count on it

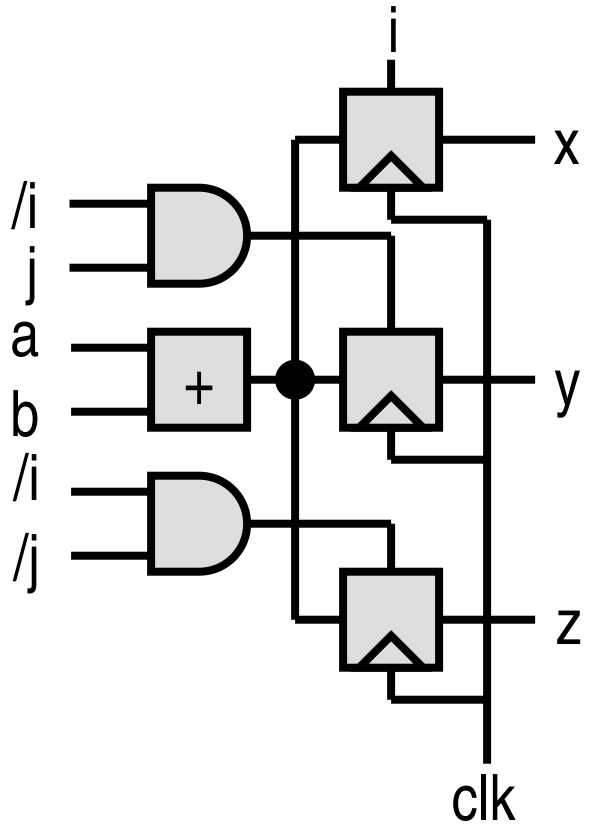
```
always @(posedge clk) begin
  if(i) begin
    x <= `TICK a+b;
  end
  else if(j) begin
    y <= `TICK a+b;
  end
  else begin
    z <= `TICK a+b;
end
```

(What is the generated hardware?)

Extra Hardware?

- Watch out for “programming” too much hardware
  - Fortunately synthesis tool (somewhat) smart – but don’t count on it

```
always @(posedge clk) begin
  if(i) begin
    x <= `TICK a+b;
  end
  else if(j) begin
    y <= `TICK a+b;
  end
  else begin
    z <= `TICK a+b;
end
```



Verilog Is Not C!

- Verilog is concurrent, C is not

```
initial begin
  a = 1'b0;
  b = 1'b0;
end

always @(posedge clk) begin
  a <= `TICK 1'b1;
  b <= `TICK a;
end
```

(Value of *a* and *b* after clock tick?)

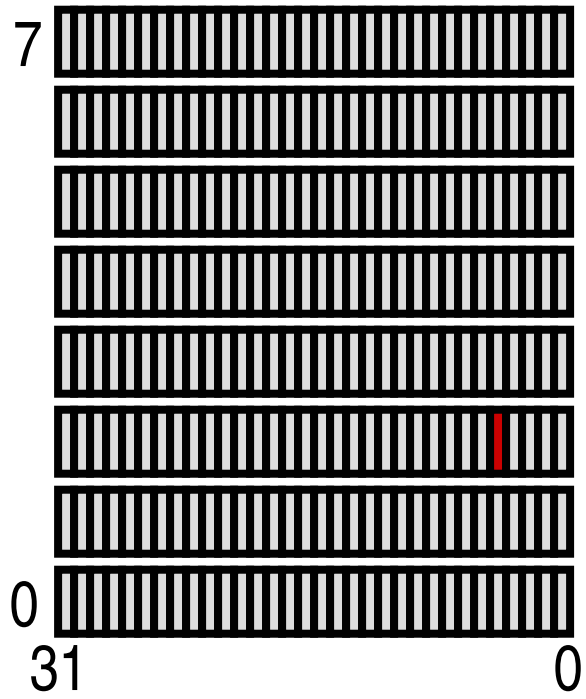
Other Useful Hardware Structures

- Register files/memories

```
reg[31:0] regfile[0:7];

wire[31:0] reg2;
wire r2b4;

assign reg2 = regfile[2];
assign r2b4 = reg2[4];
```

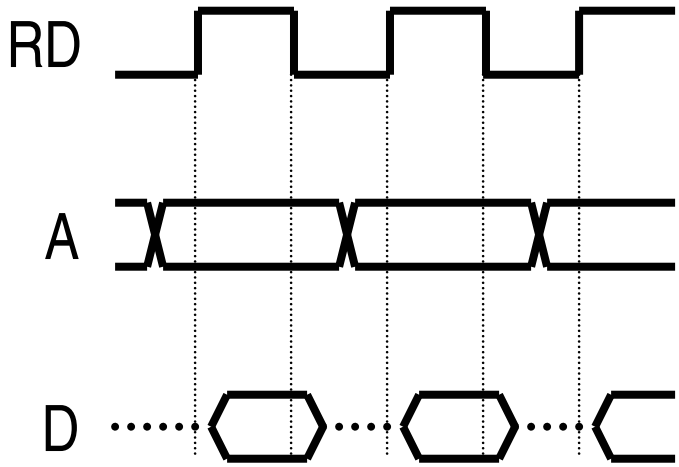


Other Useful Hardware Structures

- Tri-state devices

```
reg[31:0] mem[0:7];
wire[31:0] a;
wire[31:0] d;
wire rd;

assign d = rd?mem[a]:32'bz;
```



What is this?

Last Remarks

- It often helps to draw hardware diagrams first
- If stuck, think about what hardware does
- Use `make clobber` to clean up, or force a re-compile
- Use `vcheck`! (`vcheck *.v`)