RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

ECE-332:437
DIGITAL SYSTEMS DESIGN (DSD)

Fall 2016 – Lecture 7 – Hardware Description Language (HDL)

Nagi Naganathan
October 6, 2016

Topics to cover today – October 6, 2016

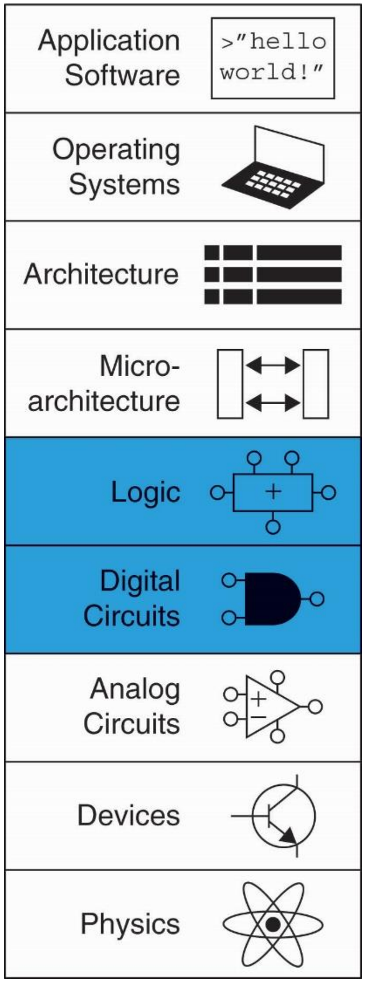
- Lecture 7 – Chapter 4 – HDL – Verilog, SystemVerilog

Announcements – October 6, 2016

- Mid Term 1 – Oct 13, 2016 – Closed Book - 15%
- Topics for Mid Term 1
  - Chapters 1 through 4
  - Lecture Slides 1-7
    - Basics of Digital Logic
    - Number Systems
    - Combinational Circuits
    - K-Map Minimization
    - Sequential Circuits
    - Finite State Machines
    - Digital Building Blocks
    - SystemVerilog

Chapter 4 :: Topics

- Introduction
- Combinational Logic
- Structural Modeling
- Sequential Logic
- More Combinational Logic
- Finite State Machines
- Parameterized Modules
- Testbenches



Digital Design

- *Digital:* circuits that use two voltage levels to represent information
  - *Logic:* use truth values and logic to analyze circuits
- *Design:* meeting functional requirements while satisfying constraints
  - Constraints: performance, size, power, cost, etc.

Design using Abstraction

- Circuits contain millions of transistors
  - How can we manage this complexity?
- Abstraction
  - Focus on aspects relevant aspects, ignoring other aspects
  - Don’t break assumptions that allow aspect to be ignored!
- Examples:
  - Transistors are on or off
  - Voltages are low or high

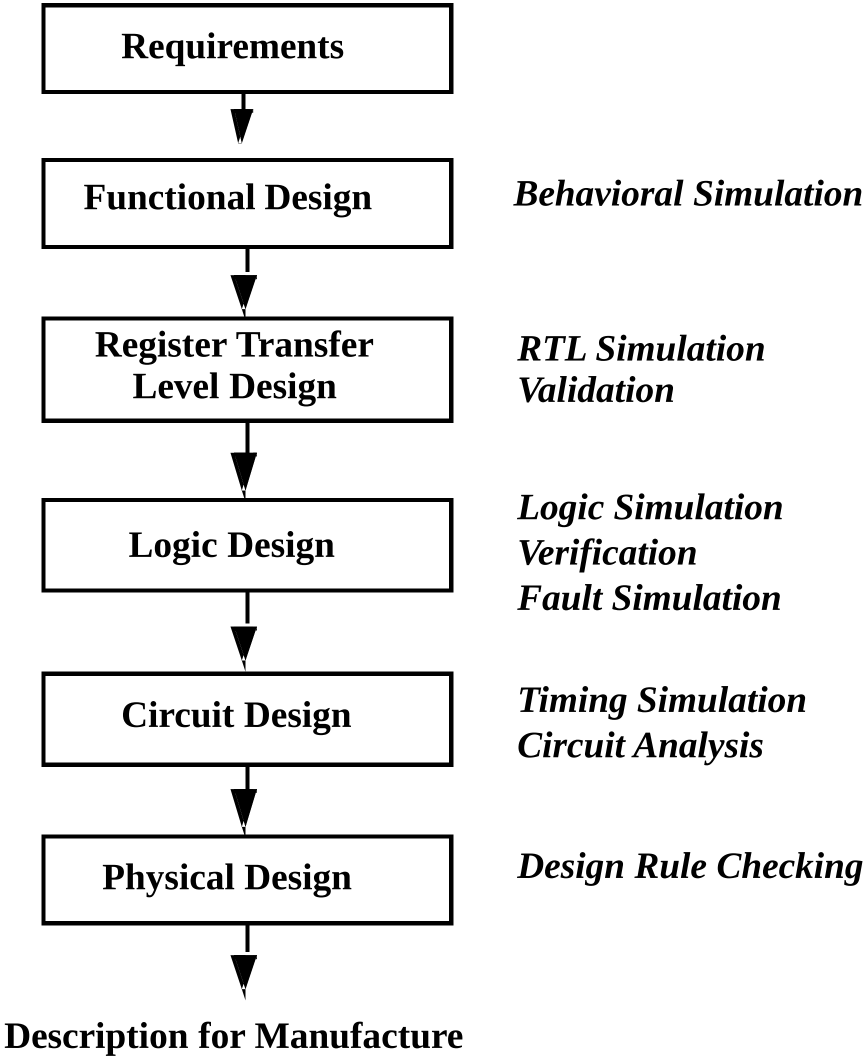
Binary Representation

- Basic representation for simplest form of information, with only two states
  - a switch: open or closed
  - a light: on or off
  - a microphone: active or muted
  - a logical proposition: false or true
  - a binary (base 2) digit, or bit: 0 or 1

Design Methodology

- Simple systems can be designed by one person using *ad hoc* methods
- Real-world systems are designed by teams
  - Require a systematic design methodology
- Specifies
  - Tasks to be undertaken
  - Information needed and produced
  - Relationships between tasks
    - dependencies, sequences
  - EDA tools used

Digital System Design



Requirements

Functional Design — *Behavioral Simulation*

Register Transfer Level Design — *RTL Simulation Validation*

Logic Design — *Logic Simulation Verification Fault Simulation*

Circuit Design — *Timing Simulation Circuit Analysis*

Physical Design — *Design Rule Checking*

Description for Manufacture

- Design flows operate at multiple levels of abstraction
- Need a uniform description to translate between levels
- Increasing costs of design and fabrication necessitate greater reliance on automation via CAD tools
  - $5M - $100M to design new chips
  - Increasing time to market pressures

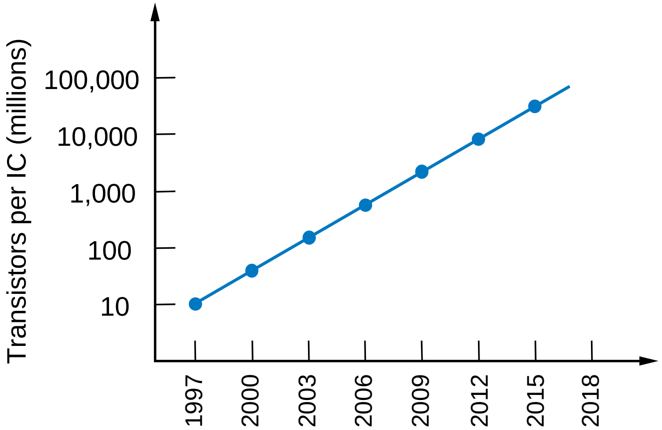
Top-down vs. bottom-up

- Top-down design:
  - start from most abstract description;
  - work to most detailed.
- Bottom-up design:
  - work from small components to big system.
- Real design uses both techniques.

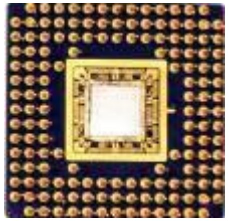
Hierarchical Design

- Circuits are too complex for us to design all the detail at once
- Design subsystems for simple functions
- Compose subsystems to form the system
  - Treating subcircuits as “black box” components
  - Verify independently, then verify the composition
- Top-down/bottom-up design

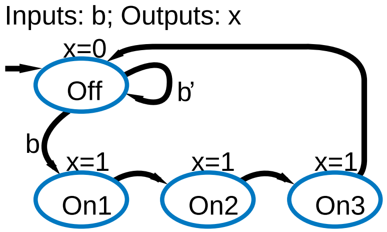
Digital Systems and HDLs



- Typical digital components per IC
  - 1960s/1970s: 10-1,000
  - 1980s: 1,000-100,000
  - 1990s: Millions
  - 2000s: Billions
- 1970s
  - IC behavior documented using combination of schematics, diagrams, and natural language (e.g., English)
- 1980s
  - Documentation was hundreds of pages for large ICs
    - Imprecise
  - Need for better documentation



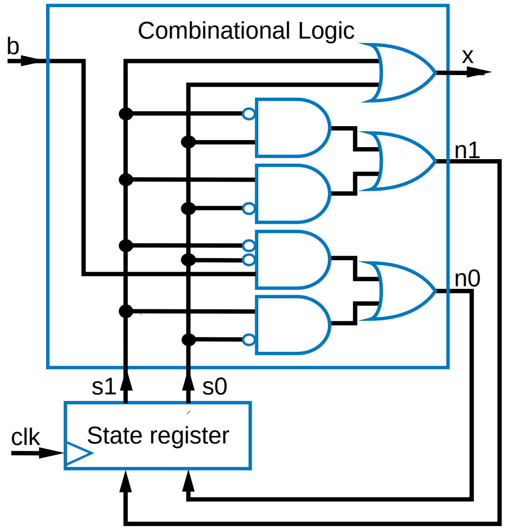
diagrams



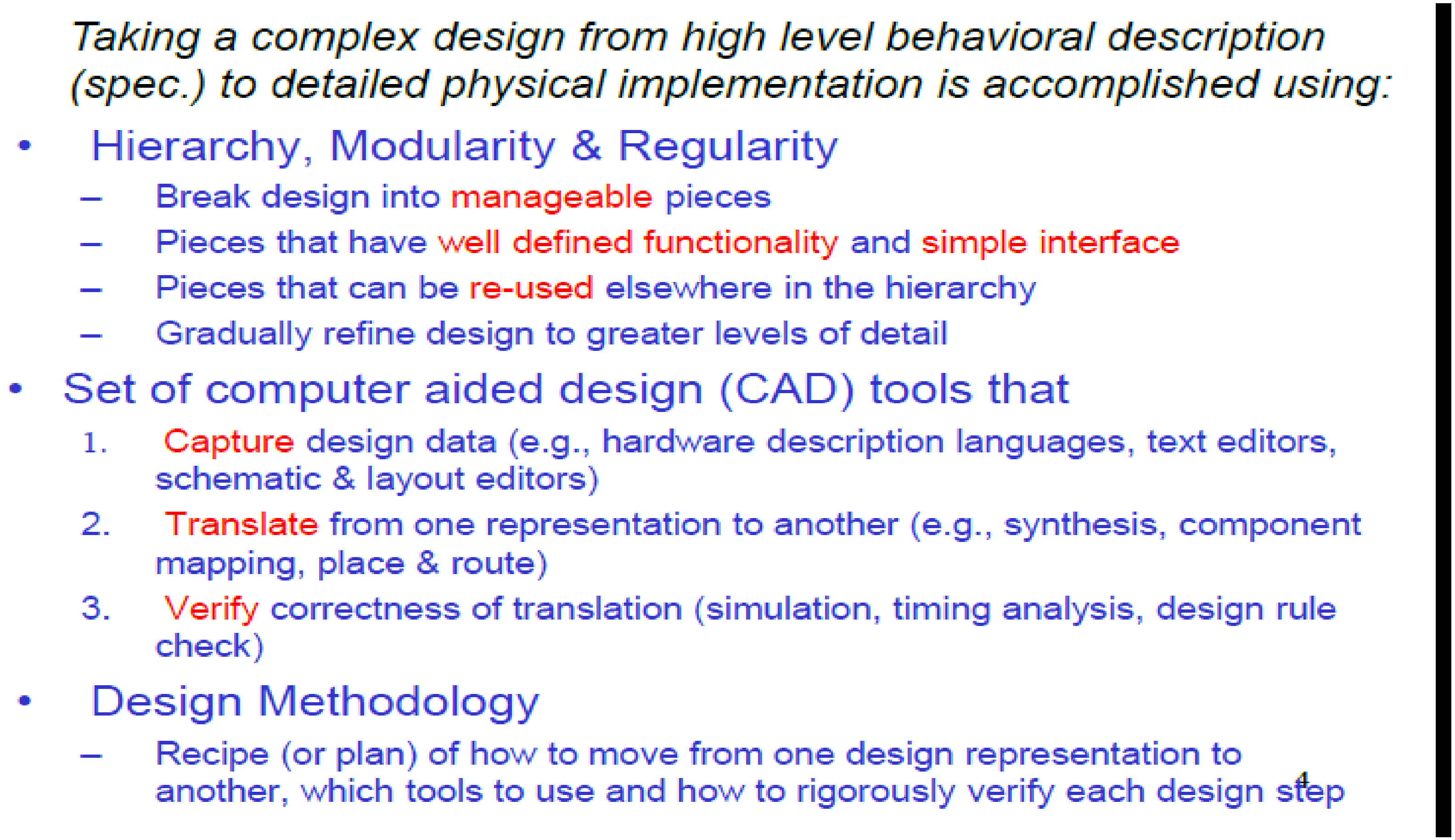
natural language

schematics

"The system has four states. When in state Off, the system outputs 0 and stays in state Off until the input becomes 1. In that case, the system enters state On1, followed by On2, and then On3, in which the system outputs 1. The system then returns to state Off."



Managing a design process

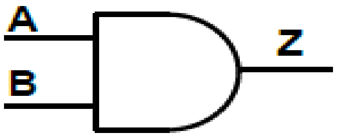


Taking a complex design from high level behavioral description (spec.) to detailed physical implementation is accomplished using:

- Hierarchy, Modularity & Regularity
  - Break design into manageable pieces
  - Pieces that have well defined functionality and simple interface
  - Pieces that can be re-used elsewhere in the hierarchy
  - Gradually refine design to greater levels of detail
- Set of computer aided design (CAD) tools that
  1. Capture design data (e.g., hardware description languages, text editors, schematic & layout editors)
  2. Translate from one representation to another (e.g., synthesis, component mapping, place & route)
  3. Verify correctness of translation (simulation, timing analysis, design rule check)
- Design Methodology
  - Recipe (or plan) of how to move from one design representation to another, which tools to use and how to rigorously verify each design step

Managing a design process – Boolean Equations

- Function represented by truth tables & logic equations



Z = A.B

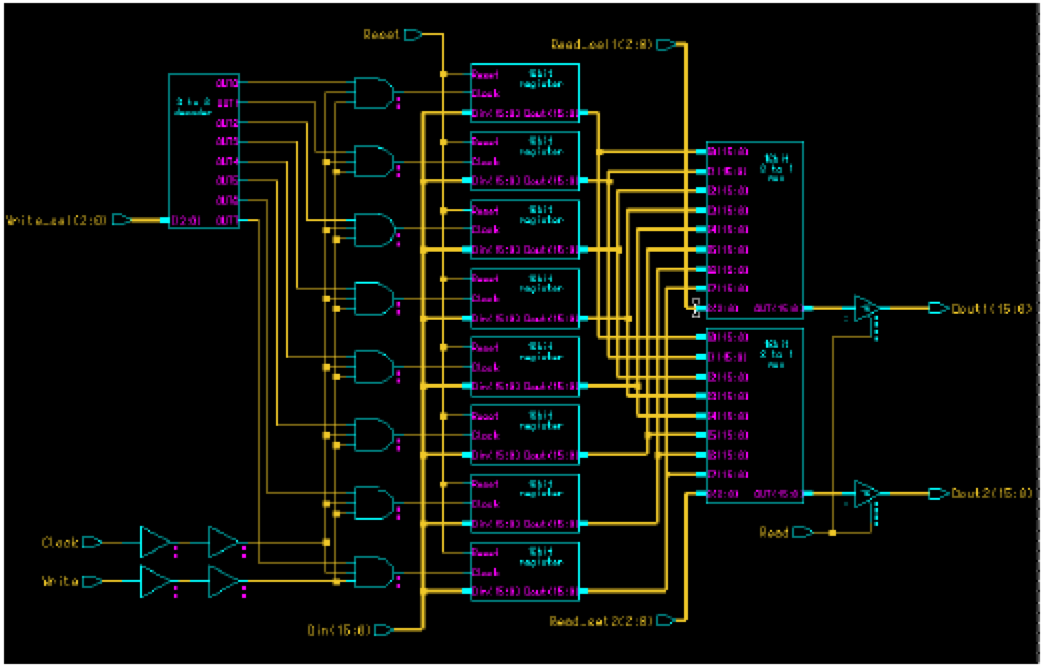
A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

- Function simplified using boolean arithmetic

$$
\begin{aligned}
Z &= \overline{B}.(A.(\overline{B} + C) + \overline{(A + B.\overline{C}}))\\
&= \overline{B}.(A.(\overline{B} + C) + \overline{A}.(\overline{B} + C)\\
&= \overline{B}.((A + \overline{A}).(\overline{B} + C))\\
&= \overline{B}.(\overline{B} + C)\\
&= \overline{B}
\end{aligned}
$$

- Captures behavior but impractical for more than few hundred gates

Managing a design process – Boolean Equations



- Graphical entry supporting hierarchy, regularity, higher level functions (register, multiplexer, ALU)
- Only captures structure – behavior must be inferred
- Limited to few thousand primitives (gates, registers etc.)

7

Motivation for HDL

- Digital System **Complexity** Continues to Increase
  - No longer able to breadboard systems
    - Number of chips
    - Number of components
    - Length of interconnects
  - Need to simulate before committing to hardware
    - Not just logic, but timing

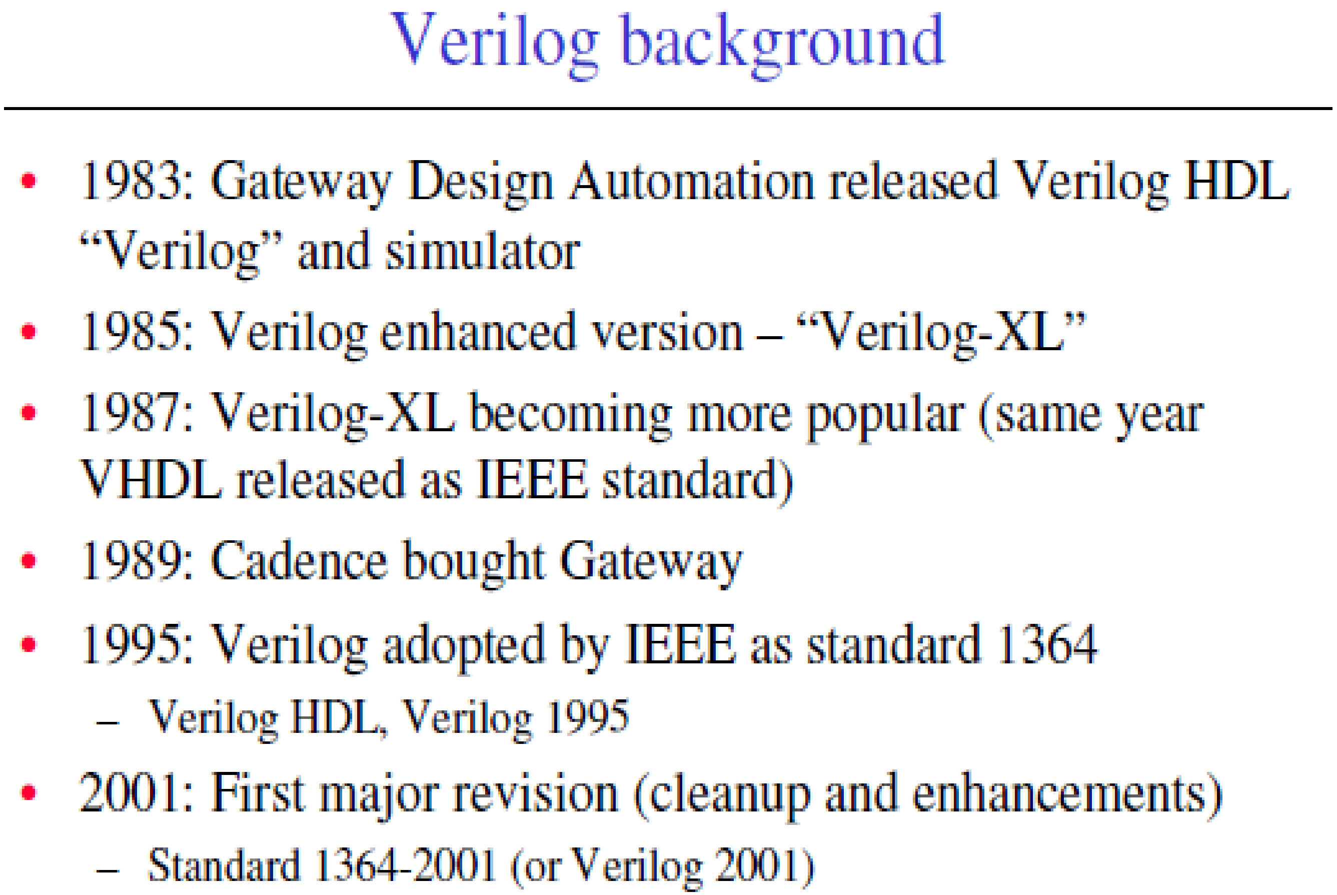
Motivation: different models

- Different Types of Models are Required at Various Development Stages
  - Logic models
  - Performance models
  - Timing models
  - System Models

Traditional vs. Hardware Description Languages

- Procedural programming languages provide the *how* or recipes
  - for computation
  - for data manipulation
  - for execution on a specific hardware model
- Hardware description languages *describe* a system
  - Systems can be described from many different points of view
    - Behavior: what does it do?
    - Structure: what is it composed of?
    - Functional properties: how do I interface to it?
    - Physical properties: how fast is it?

Verilog



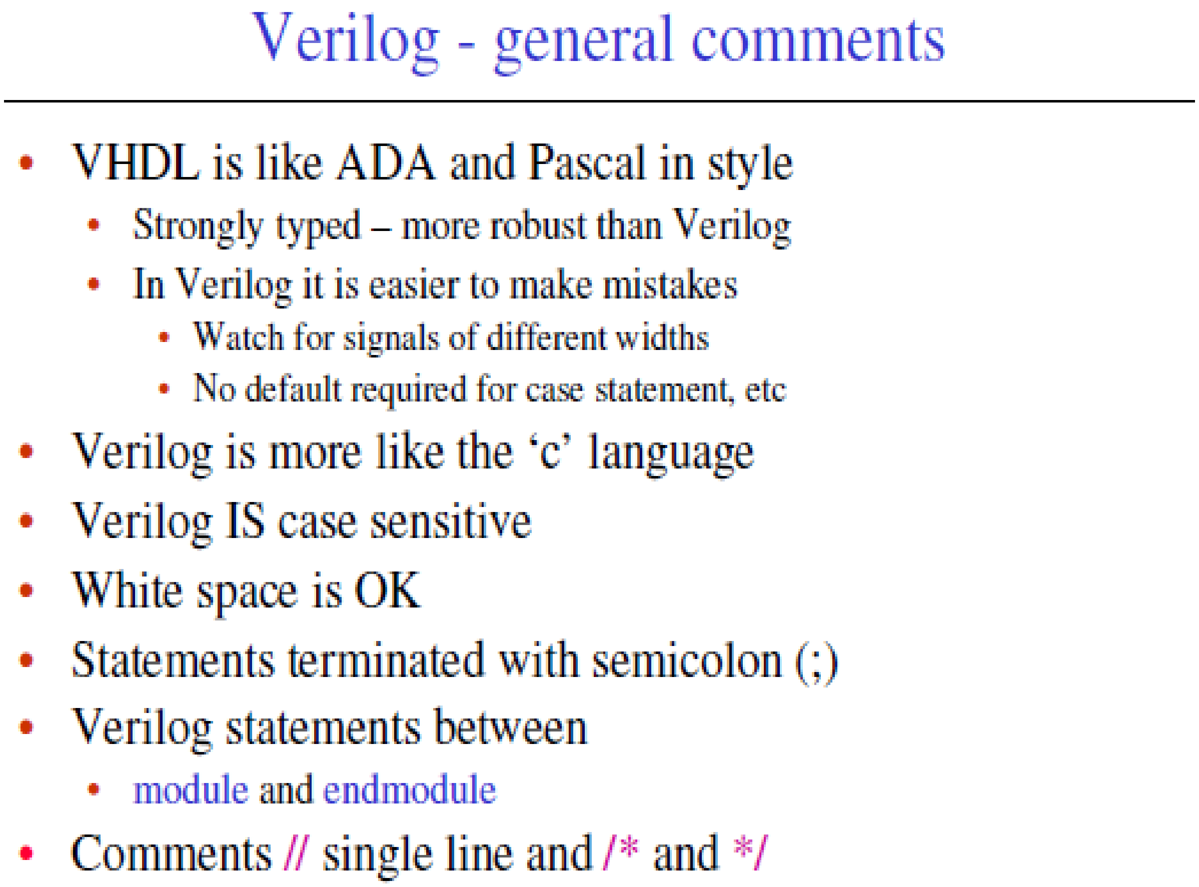
Verilog background

- 1983: Gateway Design Automation released Verilog HDL "Verilog" and simulator
- 1985: Verilog enhanced version – "Verilog-XL"
- 1987: Verilog-XL becoming more popular (same year VHDL released as IEEE standard)
- 1989: Cadence bought Gateway
- 1995: Verilog adopted by IEEE as standard 1364
    - Verilog HDL, Verilog 1995
- 2001: First major revision (cleanup and enhancements)
    - Standard 1364-2001 (or Verilog 2001)

SystemVerilog

- Enriched language derived from best features of other design and verification languages
- IEEE standard and unified language for design, specification and verification
- Contains many features of VHDL, C++ (classes)
- New Operators
- Includes assertion, coverage language
- Testbench features such as randomization

Verilog



Verilog - general comments

- VHDL is like ADA and Pascal in style
  - Strongly typed – more robust than Verilog
  - In Verilog it is easier to make mistakes
    - Watch for signals of different widths
    - No default required for case statement, etc
- Verilog is more like the ‘c’ language
- Verilog IS case sensitive
- White space is OK
- Statements terminated with semicolon (;)
- Verilog statements between
  - module and endmodule
- Comments // single line and /* and */

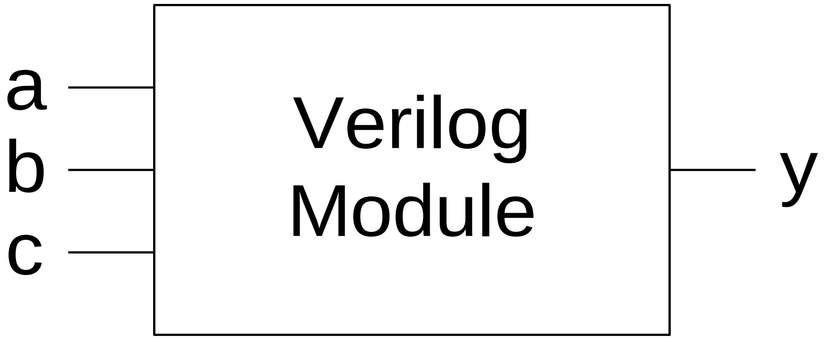
HDL to Gates

- **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)

IMPORTANT:

When using an HDL, think of the **hardware** the HDL should produce

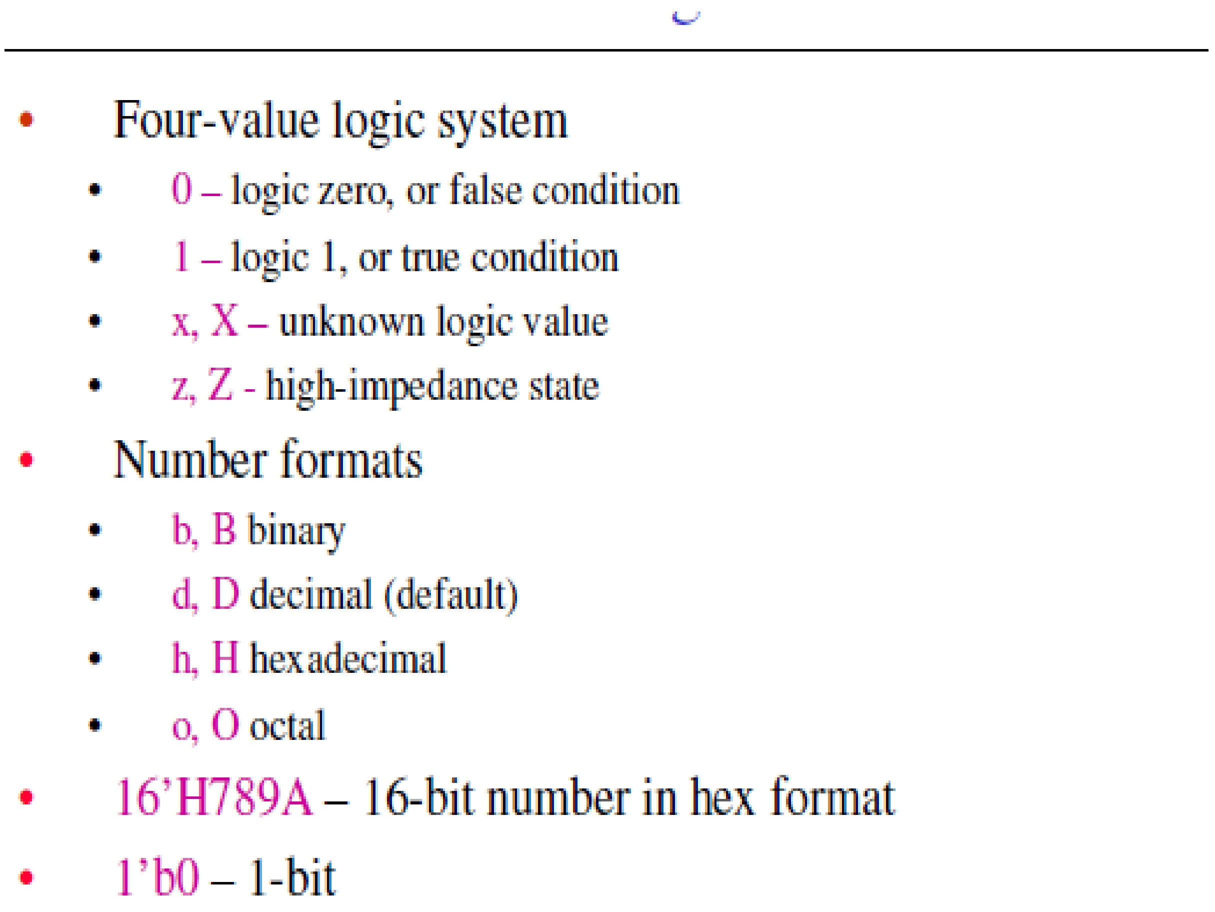
SystemVerilog Modules



Two types of Modules:

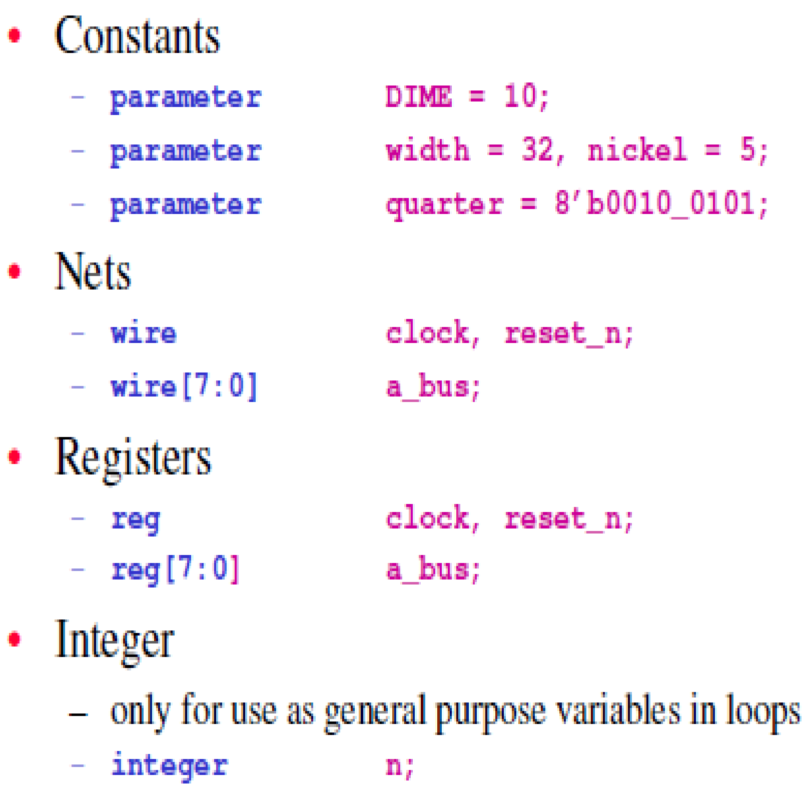
- **Behavioral:** describe what a module does
- **Structural:** describe how it is built from simpler modules

Verilog



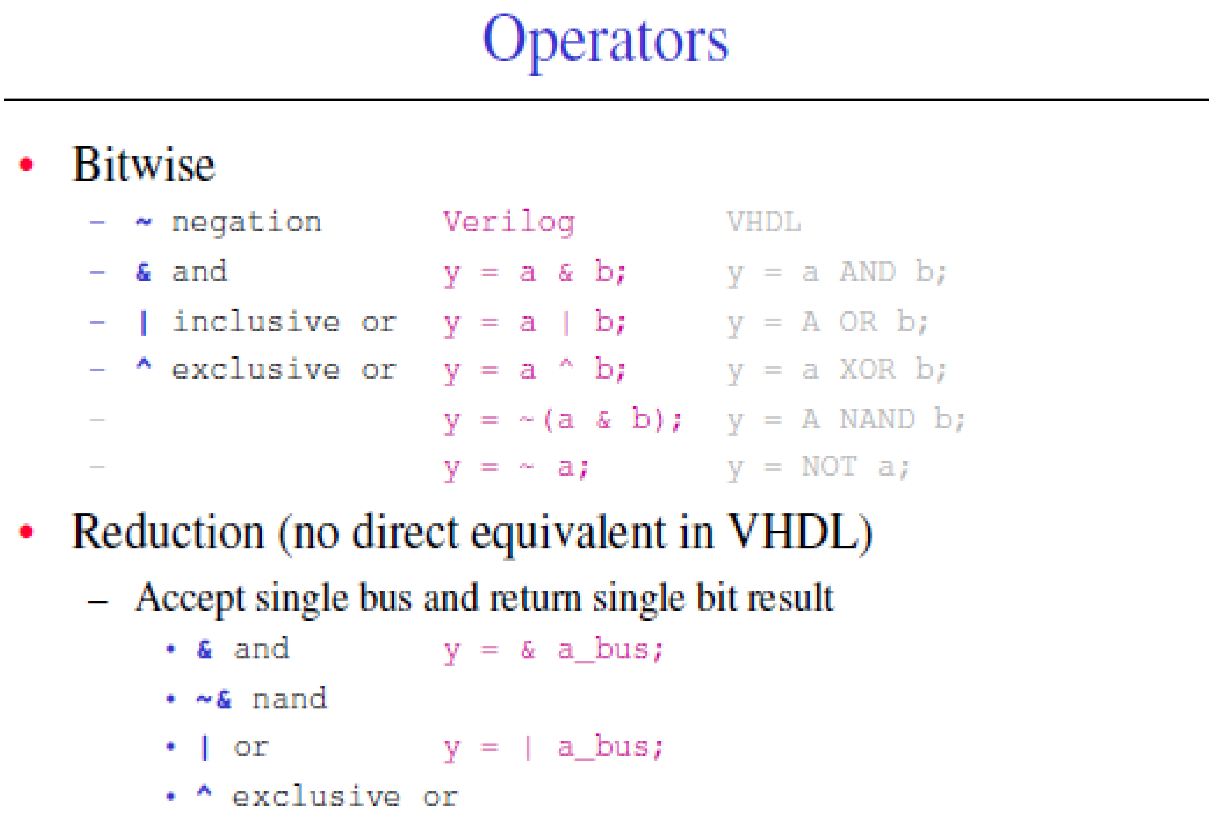
- Four-value logic system
  - 0 – logic zero, or false condition
  - 1 – logic 1, or true condition
  - x, X – unknown logic value
  - z, Z - high-impedance state
- Number formats
  - b, B binary
  - d, D decimal (default)
  - h, H hexadecimal
  - o, O octal
- 16'H789A – 16-bit number in hex format
- 1'b0 – 1-bit

Verilog



- Constants
  - parameter DIME = 10;
  - parameter width = 32, nickel = 5;
  - parameter quarter = 8'b0010_0101;
- Nets
  - wire clock, reset_n;
  - wire[7:0] a_bus;
- Registers
  - reg clock, reset_n;
  - reg[7:0] a_bus;
- Integer
  - only for use as general purpose variables in loops
  - integer n;

Verilog



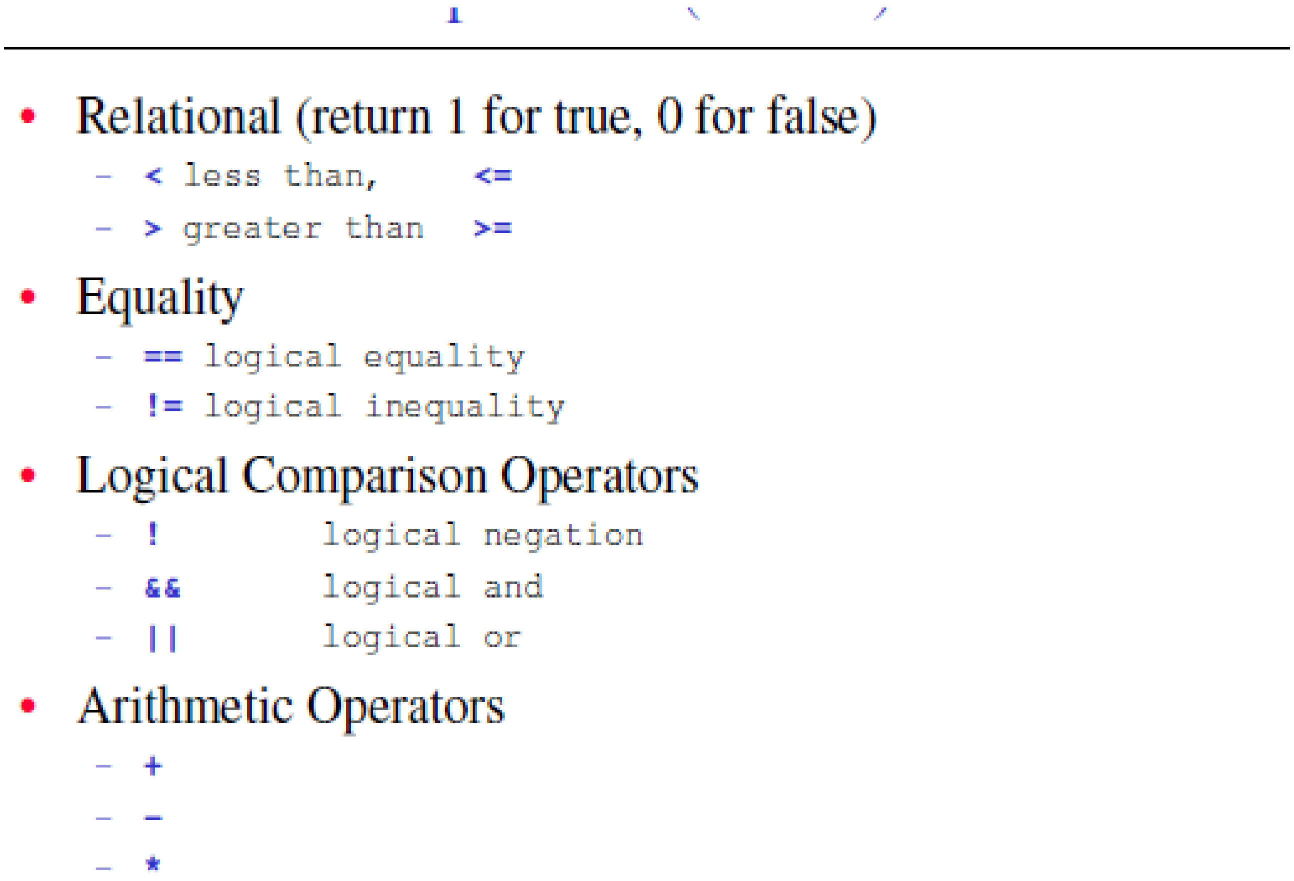
Operators

- Bitwise

– ~ negation	Verilog	VHDL
– & and	y = a & b;	y = a AND b;
– | inclusive or	y = a | b;	y = A OR b;
– ^ exclusive or	y = a ^ b;	y = a XOR b;
–	y = ~(a & b);	y = A NAND b;
–	y = ~ a;	y = NOT a;

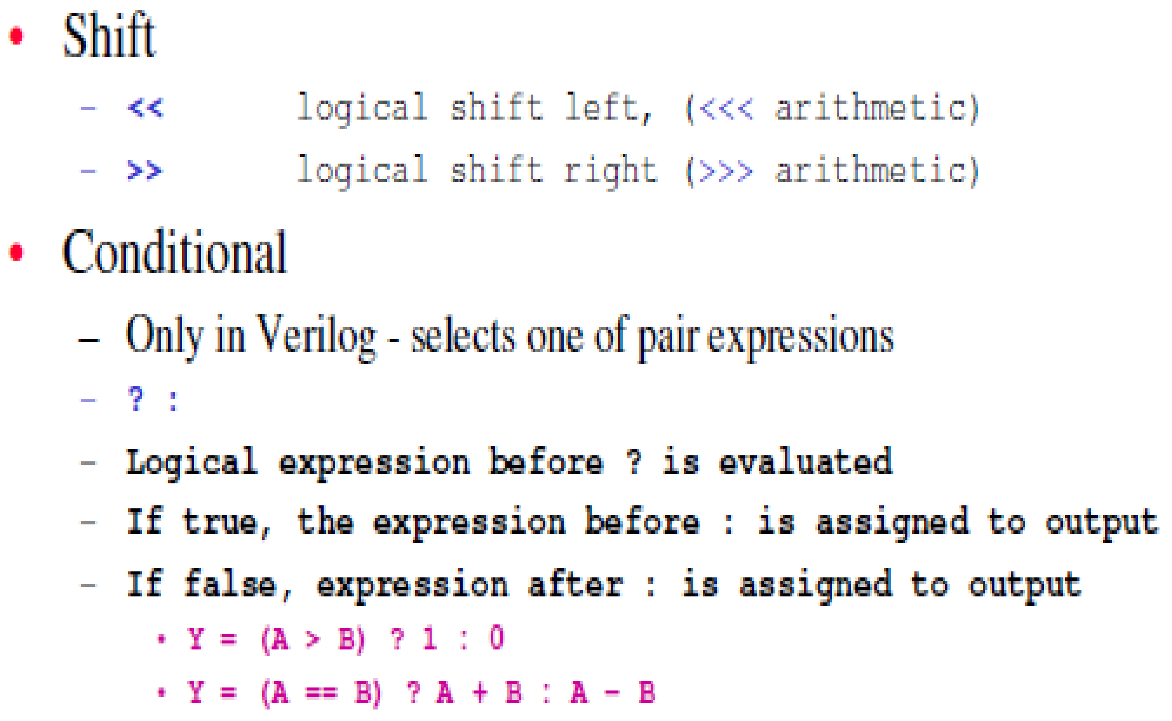
- Reduction (no direct equivalent in VHDL)
  - Accept single bus and return single bit result
    - & and `y = & a_bus;`
    - ~& nand
    - | or `y = | a_bus;`
    - ^ exclusive or

Verilog



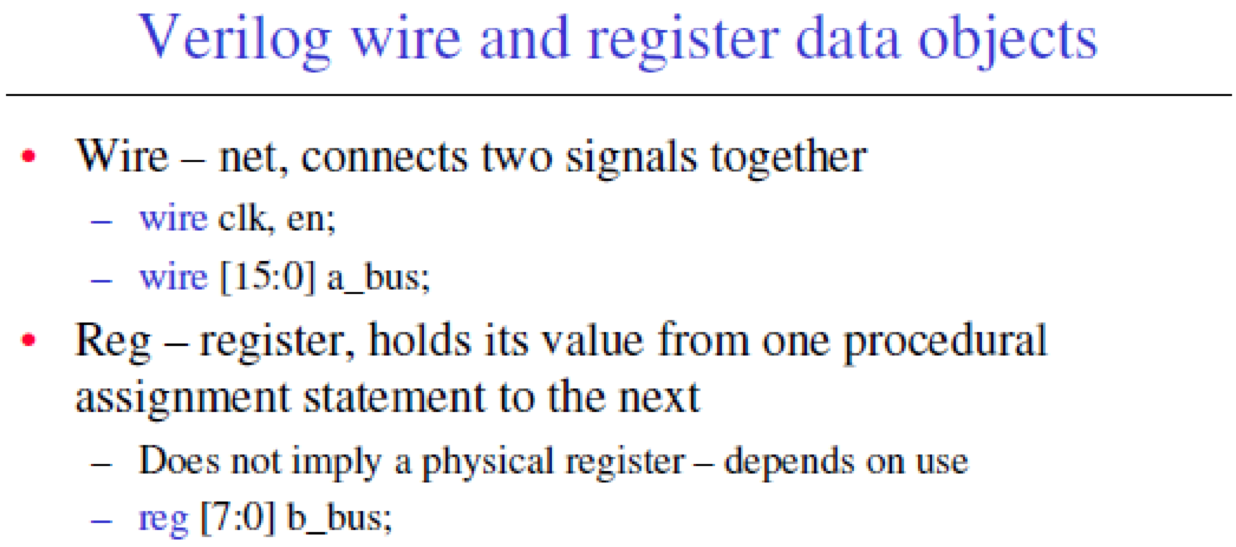
- Relational (return 1 for true, 0 for false)
  - < less than, <=
  - > greater than >=
- Equality
  - == logical equality
  - != logical inequality
- Logical Comparison Operators
  - ! logical negation
  - && logical and
  - || logical or
- Arithmetic Operators
  - +
  - -
  - *

Verilog



- Shift
  - `<<`   logical shift left, (<<< arithmetic)
  - `>>`   logical shift right (>>> arithmetic)
- Conditional
  - Only in Verilog - selects one of pair expressions
  - `? :`
  - Logical expression before ? is evaluated
  - If true, the expression before : is assigned to output
  - If false, expression after : is assigned to output
    - `Y = (A > B) ? 1 : 0`
    - `Y = (A == B) ? A + B : A - B`

Verilog



Verilog wire and register data objects

- Wire – net, connects two signals together
  - wire clk, en;
  - wire [15:0] a_bus;
- Reg – register, holds its value from one procedural assignment statement to the next
  - Does not imply a physical register – depends on use
  - reg [7:0] b_bus;

Verilog

Verilog

Part 2 –
Basics of Verilog

The Verilog Hardware Description Language

- These slides were created by Prof. Don Thomas at Carnegie Mellon University, and are adapted here with permission.

- The Verilog Hardware Description Language, Fifth Edition, by Donald Thomas and Phillip Moorby is available from Springer, http://www.springer.com.

Verilog Overview

- Verilog is a concurrent language
  - Aimed at modeling hardware — optimized for it!
  - Typical of hardware description languages (HDLs), it:
    - provides for the specification of concurrent activities
    - stands on its head to make the activities look like they happened at the same time
      - Why?
    - allows for intricate timing specifications
- A concurrent language allows for:
  - Multiple concurrent “elements”
  - An event in one element to cause activity in another. (An *event* is an output or state change at a given time)
    - based on interconnection of the element’s ports
  - Further execution to be delayed
    - until a specific event occurs

Simulation of Digital Systems

- **Simulation —**
  - **What do you do to test a software program you write?**
    - **Give it some inputs, and see if it does what you expect**
    - **When done testing, is there any assurance the program is bug free? — NO!**
    - **But, to the extent possible, you have determined that the program does what you want it to do**
  - **Simulation tests a model of the system you wish to build**
    - **Is the design correct? Does it implement the intended function correctly? For instance, is it a UART**
      - **Stick in a byte and see if the UART model shifts it out correctly**
    - **Also, is it the correct design?**
      - **Might there be some other functions the UART could do?**

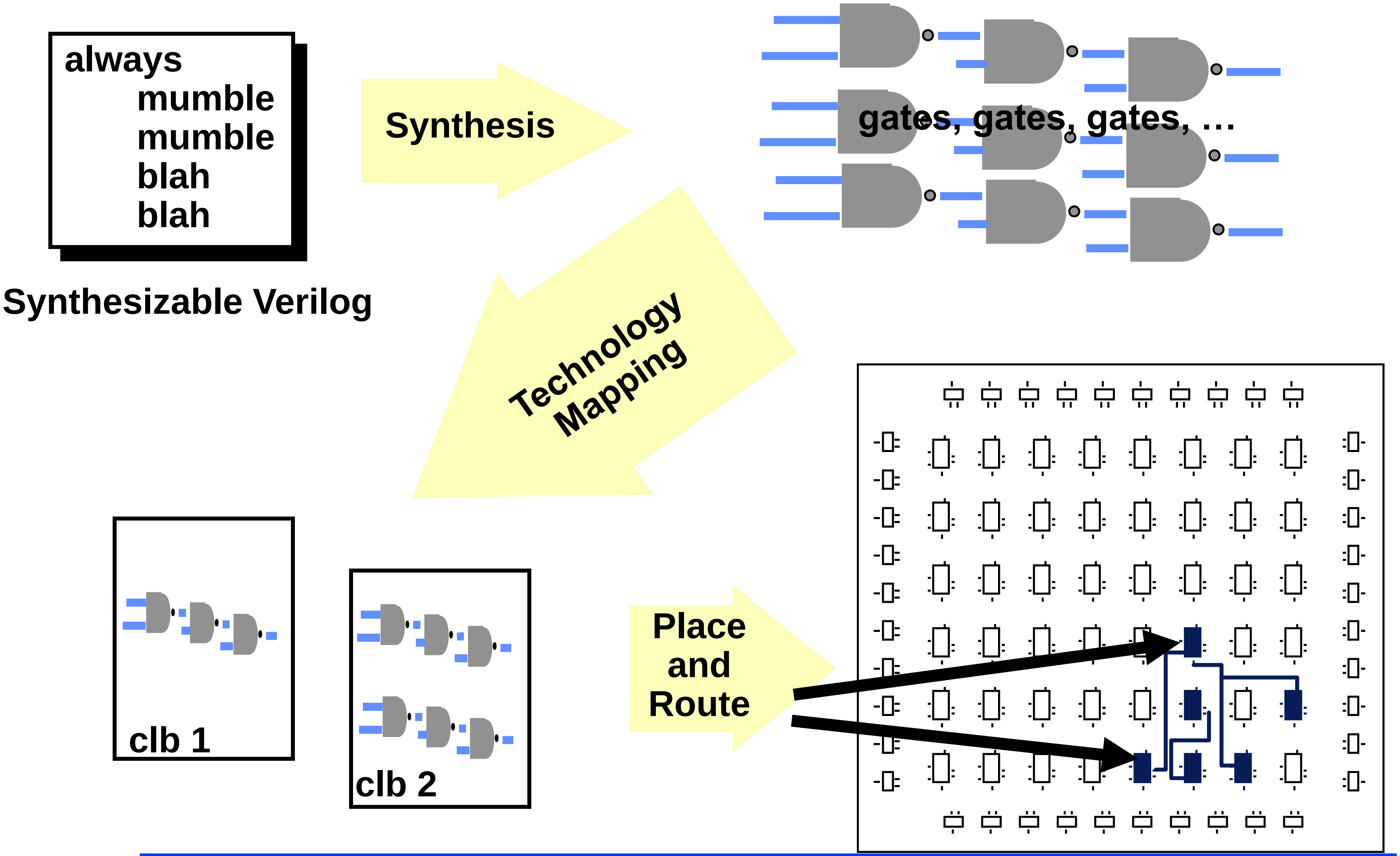
Simulation of Digital Systems

- Simulation checks two properties
  - functional correctness — is the logic correct
    - correct design, and design correct
  - timing correctness — is the logic/interconnect timing correct
    - e.g. are the set-up times met?

- It has all the limitations of software testing
  - Have I tried all the cases?
  - Have I exercised every path? Every option?

Modern Design Methodology

Simulation and Synthesis are components of a design methodology



```
always
      mumble
      mumble
      blah
      blah
```

Synthesizable Verilog

Synthesis

gates, gates, gates, …

Technology Mapping

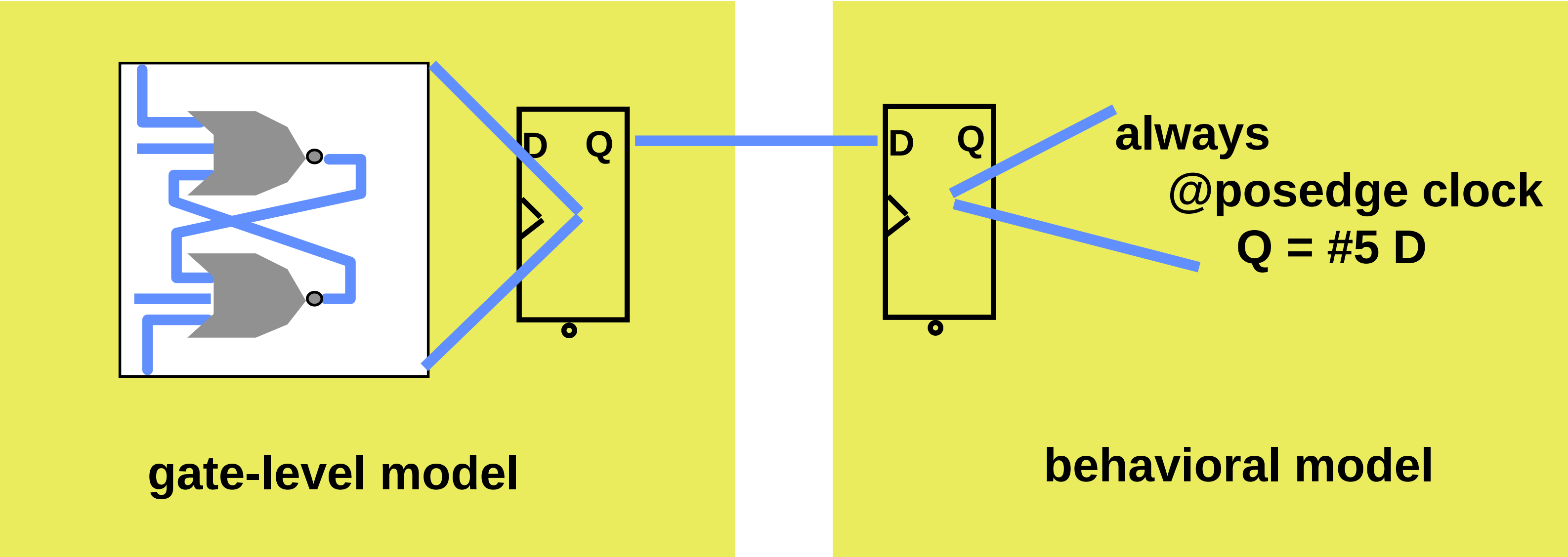
clb 1

clb 2

Place and Route

Verilog Levels of Abstraction

- Gate modeling (Structural modeling)
  - the system is represented in terms of primitive gates and their interconections
    - NANDs, NORs, …
- Behavioral modeling
  - the system is represented by a program-like language



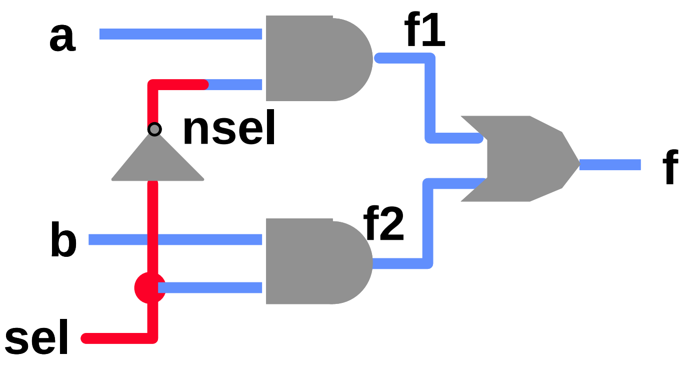
```
always
    @posedge clock
        Q = #5 D
```

gate-level model

behavioral model

Representation: Structural Models

- Structural models
  - Are built from gate primitives and/or other modules
  - They describe the circuit using logic gates — much as you would see in an implementation of a circuit.
- Identify:
  - Gate instances, wire names, delay from *a* or *b* to *f*.
  - This is a multiplexor — it selects one of n inputs (2 here) and passes it on to the output



f = a • sel’ + b • sel

```
module MUX (f, a, b, sel);
output          f;
input           a, b, sel;

       and #5   g1 (f1, a, nsel),
                g2 (f2, b, sel);
       or   #5  g3 (f, f1, f2);
       not      g4 (nsel, sel);
endmodule
```

Representation: Gate-Level Models

- **Need to model the gate’s:**
  - **Function**
  - **Delay**
- **Function**
  - **Generally, HDLs have built-in gate-level primitives**
    - **Verilog has NAND, NOR, AND, OR, XOR, XNOR, BUF, NOT, and some others**
  - **The gates operate on input values producing an output value**
    - **typical Verilog gate instantiation is:**

optional

“many”

```
and #delay  instance-name (out, in1, in2, in3, …);

and #5   g1 (f1, a, nsel);
```

a comma here let’s you list other instance names and their port lists.

Four-Valued Logic

- Verilog Logic Values
  - The underlying data representation allows for any bit to have one of four values
  - 1, 0, x (unknown), z (high impedance)
  - x — one of: 1, 0, z, or in the state of change
  - z — the high impedance output of a tri-state gate.
- What basis do these have in reality?
  - 0, 1 … no question
  - z … A *tri-state* gate drives either a zero or one on its output. If it's not doing that, its output is high impedance (z). Tri-state gates are real devices and z is a real electrical affect.
  - x … not a real value. There is no *real* gate that drives an x on to a wire. x is used as a debugging aid. x means the simulator can't determine the answer and so maybe you should worry!
- BTW …
  - some simulators keep track of more values than these. Verilog will in some situations.

How to build and test a module

- Construct a “test bench” for your design
  - Develop your hierarchical system within a module that has input and output ports (called “design” here)
  - Develop a separate module to generate tests for the module (“test”)
  - Connect these together within another module (“testbench”)

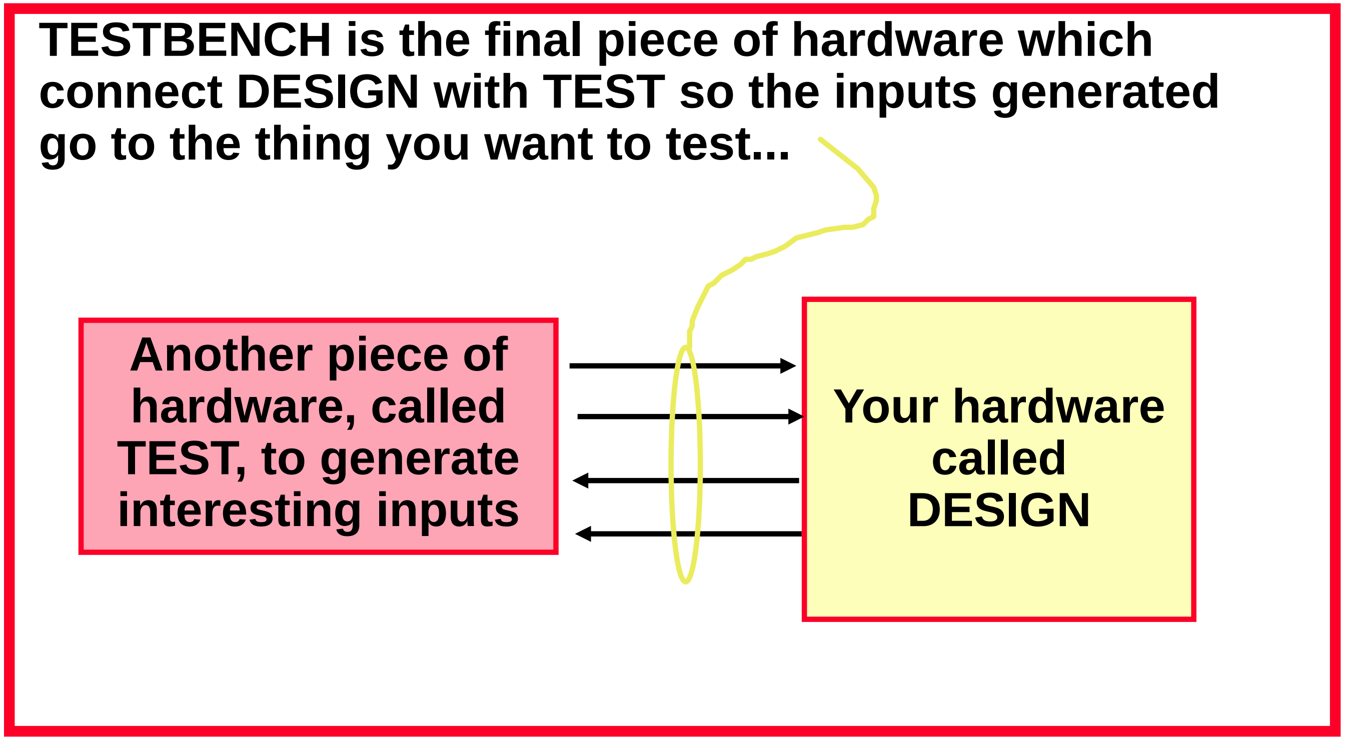
```
module testbench ();
        wire        l, m, n;

        design   d (l, m, n);
        test        t (l, m);

        initial begin
                //monitor and display
                …
```

```
module design (a, b, c);
        input      a, b;
        output   c;
        …
```

```
module test (q, r);
        output  q, r;

        initial begin
                //drive the outputs with signals
                …
```

Another view of this

- □ 3 chunks of Verilog, one for each of:



TESTBENCH is the final piece of hardware which connect DESIGN with TEST so the inputs generated go to the thing you want to test...

Another piece of hardware, called TEST, to generate interesting inputs

Your hardware called DESIGN

An Example

Module testAdd generates inputs for module halfAdd and displays changes. Module halfAdd is the *design*

```
module tBench;
      wire      su, co, a, b;

      halfAdd        ad(su, co, a, b);
      testAdd        tb(a, b, su, co);
endmodule
```

```
module halfAdd (sum, cOut, a, b);
      output     sum, cOut;
      input      a, b;

      xor  #2    (sum, a, b);
      and #2     (cOut, a, b);
endmodule
```

```
module testAdd(a, b, sum, cOut);
      input     sum, cOut;
      output  a, b;
      reg        a, b;

      initial begin
            $monitor ($time,,
               "a=%b, b=%b, sum=%b, cOut=%b",
                 a, b, sum, cOut);
            a = 0; b = 0;
            #10 b = 1;
            #10 a = 1;
            #10 b = 0;
            #10 $finish;
      end
endmodule
```

The test module

- It’s the test generator
- $monitor
  - prints its string when executed.
  - after that, the string is printed when one of the listed values changes.
  - only one monitor can be active at any time
  - prints at end of current simulation time
- Function of this tester
  - at time zero, print values and set a=b=0
  - after 10 time units, set b=1
  - after another 10, set a=1
  - after another 10 set b=0
  - then another 10 and finish

```
module testAdd(a, b, sum, cOut);
    input     sum, cOut;
    output  a, b;
    reg        a, b;

    initial begin
        $monitor ($time,,
          “a=%b, b=%b, sum=%b, cOut=%b”,
          a, b, sum, cOut);
        a = 0; b = 0;
        #10 b = 1;
        #10 a = 1;
        #10 b = 0;
        #10 $finish;
    end
endmodule
```

Another Version of a Test Module

- **Multi-bit constructs**
  - test is a two-bit register and output
  - It acts as a two-bit number (counts 00-01-10-11-00…)
  - Module tBench needs to connect it correctly — mod halfAdd has 1-bit ports.

```
module testAdd (test, sum, cOut);
      input              sum, cOut;
      output  [1:0]   test;
      reg        [1:0]   test;

      initial begin
            $monitor ($time,,
               "test=%b, sum=%b, cOut=%b",
               test, sum, cOut);
            test = 0;
            #10 test = test + 1;
            #10 test = test + 1;
            #10 test = test + 1;
            #10 $finish;
      end
endmodule
```

```
module tBench;
      wire    su, co;
      wire [1:0] t;

      halfAdd     ad (su, co, t[1], t[0]);
      testAdd     tb (t, su, co);
endmodule
```

Connects bit 0 of wire t to this port (b of the module halfAdder)

Yet Another Version of testAdd

- Other procedural statements
  - You can use “for”, “while”, “if-then-else” and others here.
  - This makes it easier to write if you have lots of input bits.

```
module testAdd (test, sum, cOut);
      input              sum, cOut;
      output  [1:0]          test;
      reg     [1:0]          test;

      initial begin
           $monitor ($time,,
             "test=%b, sum=%b, cOut=%b",
             test, sum, cOut);
           for (test = 0; test < 3; test = test + 1)
                 #10;
           #10 $finish;
      end
endmodule
```

```
module tBench;
     wire    su, co;
     wire [1:0] t;

     halfAdd    ad (su, co, t[1], t[0]);
     testAdd    tb (t, su, co);
endmodule
```

hmm… “<3” … ?

Other things you can do

- More than modeling hardware
  - $monitor — give it a list of variables. When one of them changes, it prints the information. Can only have one of these active at a time.
    e.g. …
    - $monitor ($time,,, "a=%b, b=%b, sum=%b, cOut=%b",a, b, sum, cOut);

extra commas print as spaces

%b is binary (also, %h, %d and others)

What if what you print has the value x or z?

    - The above will print:
      2   a=0, b=0, sum=0, cOut=0<return>

newline automatically included

  - $display() — sort of like printf()
    - $display ("Hello, world — %h", hexvalue)

display contents of data item called "hexvalue" using hex digits (0-9,A-F)

Structural vs Behavioral Models

- Structural model
  - Just specifies primitive gates and wires
  - i.e., the structure of a logical netlist
  - You basically know how to do this now.

- Behavioral model
  - More like a procedure in a programming language
  - Still specify a module in Verilog with inputs and outputs...
  - ...but inside the module you write code to tell what you want to have happen, NOT what gates to connect to make it happen
  - i.e., you specify the behavior you want, not the structure to do it

- Why use behavioral models
  - For testbench modules to test structural designs
  - For high-level specs to drive logic synthesis tools

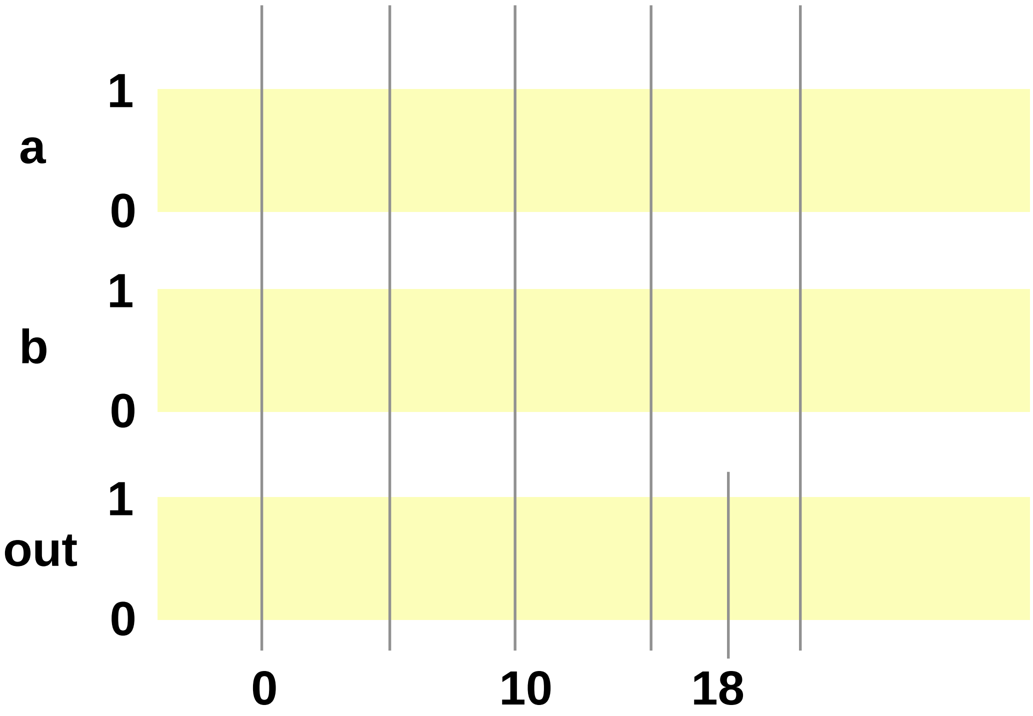
How do behavioral models fit in?

- How do they work with the event list and scheduler?
  - Initial (and always) begin executing at time 0 in arbitrary order
  - They execute until they come to a “#delay” operator
  - They then suspend, putting themselves in the event list 10 time units in the future (for the case at the right)
  - At 10 time units in the future, they resume executing where they left off.
- Some details omitted
  - ...more to come

```
module testAdd(a, b, sum, cOut);
        input     sum, cOut;
        output    a, b;
        reg       a, b;

        initial begin
                  $monitor ($time,,
                    “a=%b, b=%b,
                    sum=%b, cOut=%b”,
                    a, b, sum, cOut);
                  a = 0; b = 0;
                  #10 b = 1;
                  #10 a = 1;
                  #10 b = 0;
                  #10 $finish;
        end
endmodule
```

Two initial statements?

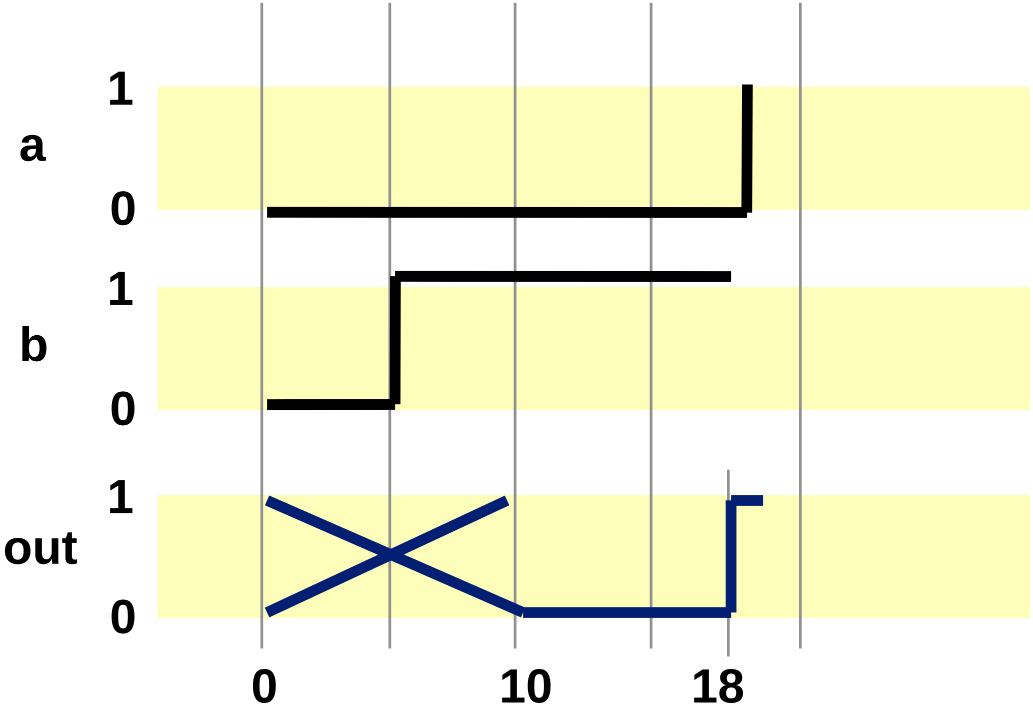
```
…
initial begin
        a = 0; b = 0;
        #5 b = 1;
        #13 a = 1;
end
…
initial begin
        #10 out = 0;
        #8 out = 1;
end
…
```



- **Things to note**
  - **Which initial statement starts first?**
  - **What are the values of a, b, and out when the simulation starts?**
  - **These appear to be executing concurrently (at the same time). Are they?**

Two initial statements?

```
…
initial begin
        a = 0; b = 0;
        #5 b = 1;
        #13 a = 1;
end
…
initial begin
        #10 out = 0;
        #8 out = 1;
end
…
```



- **Things to note**
  - Which initial statement starts first?
  - What are the initial values of a, b, and out when the simulation starts?
  - These appear to be executing concurrently (at the same time). Are they?

They start at same time

Undefined (x)

Not necessarily

Behavioral Modeling

- ***Procedural*** **statements are used**
  - **Statements using “initial” and “always” Verilog constructs**
  - **Can specify both combinational and sequential circuits**

- **Normally don’t think of procedural stuff as “logic”**
  - **They look like C: mix of ifs, case statements, assignments …**
  - **… but there is a semantic interpretation to put on them to allow them to be used for simulation and synthesis (giving equivalent results)**

Behavioral Constructs

- Behavioral descriptions are introduced by initial and always statements

Statement	Looks like	Starts	How it works	Use in Synthesis?

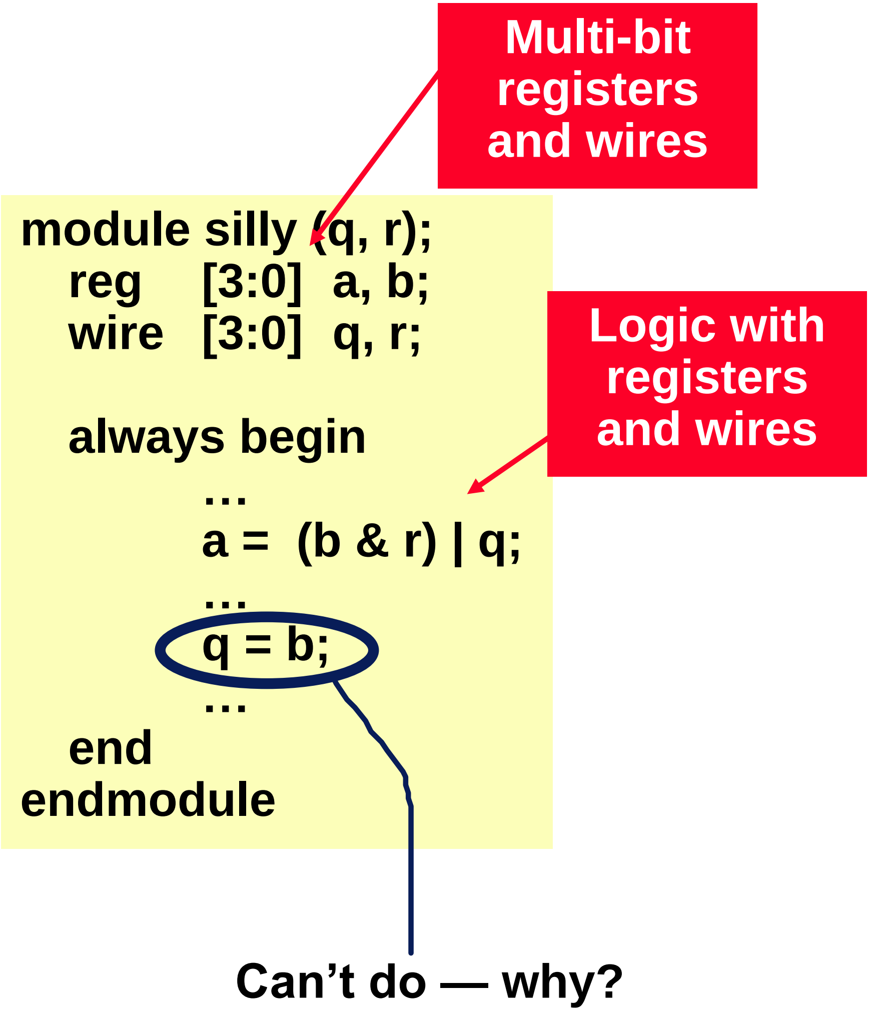
initial	initial
	begin
	…
	end
		Starts when simulation starts	Execute once and stop	Not used in synthesis

always	always
	begin
	…
	end
			Continually loop—while (sim. active) do statements;	Used in synthesis

- Points:
  - They all execute concurrently
  - They contain behavioral statements like if-then-else, case, loops, functions, …

Statements, Registers and Wires

- Registers
  - Define storage, can be more than one bit
  - Can only be changed by assigning value to them on the left-hand side of a behavioral expression.
- Wires (actually "nets")
  - Electrically connect things together
  - Can be used on the right-hand side of an expression
    - Thus we can tie primitive gates and behavioral blocks together!
- Statements
  - left-hand side = right-hand side
  - left-hand side must be a register
  - Four-valued logic



Multi-bit registers and wires

```
module silly (q, r);
   reg    [3:0]  a, b;
   wire   [3:0]  q, r;

   always begin
          …
          a =  (b & r) | q;
          …
          q = b;
          …
   end
endmodule
```

Logic with registers and wires

Can't do — why?

Behavioral Statements

- **if-then-else**
  - **What you would expect, except that it’s doing 4-valued logic. 1 is interpreted as True; 0, x, and z are interpreted as False**
- **case**
  - **What you would expect, except that it’s doing 4-valued logic**
  - **If “selector” is 2 bits, there are 4^2 possible case-items to select between**
  - **There is no *break* statement — it is assumed.**
- **Funny constants?**
  - **Verilog allows for sized, 4-valued constants**
  - **The first number is the number of bits, the letter is the base of the following number that will be converted into the bits.**

8’b00x0zx10

```
if (select == 1)
         f = in1;
else     f = in0;
```

```
case (selector)
    2’b00: a = b + c;
    2’b01: q = r + s;
    2’bx1: r = 5;
    default: r = 0;
endcase
```

assume f, a, q, and r are registers for this slide

Behavioral Statements

- Loops
  - There are restrictions on using these for synthesis — don’t.
  - They are mentioned here for use in test modules and behavioral models not intended for synthesis
- Two main ones — for and while
  - Just like in C
  - There is also repeat and forever

```
reg   [3:0]   testOutput, i;
…
for (i = 0; i <= 15; i = i + 1) begin
        testOutput = i;
        #20;
end
```

```
reg   [3:0]   testOutput, i;
…
i = 0;
while (i <= 15)) begin
        testOutput = i;
        #20 i = i + 1;
end
```

Important: Be careful with loops. Its easy to create infinite loop situations. More on this later.

Test Module, continued

- **Bit Selects and Part Selects**
  - **This expression extracts bits or ranges of bits or a wire or register**

The individual bits of register i are made available on the ports. These are later connected to individual input wires in module design.

```
module testgen (i[3], i[2], i[1], i[0]);
reg   [3:0]   i; output i;
always
        for (i = 0; i <= 15; i = i + 1)
                  #20;
endmodule
```

```
module top;
wire  w0, w1, w2, w3;

testgen t (w0, w1, w2, w3);
design d (w0, w1, w2, w3);
end
```

```
module design (a, b, c, d);
input a, b, c, d;

mumble, mumble, blah, blah;
end
```

Alternate:

```
module testgen (i);
reg   [3:0]   i; output i;
always
        for (i = 0; i <= 15; i = i + 1)
                  #20;
endmodule
```

```
module top;
wire  [3:0] w;

testgen t (w);
design d (w[0], w[1], w[2], w[3]);
end
```

Concurrent Constructs

- **We already saw #delay**
- **Others**
  - **@ … Waiting for a *change* in a value — used in synthesis**
    - **@ (var) w = 4;**
    - **This says wait for var to change from its current value. When it does, resume execution of the statement by setting w = 4.**
  - **Wait … Waiting for a value to be a certain level — not used in synthesis**
    - **wait (f == 0) q = 3;**
    - **This says that if f is equal to zero, then continue executing and set q = 3.**
    - **But if f is not equal to zero, then suspend execution until it does. When it does, this statement resumes by setting q = 3.**
- **Why are these concurrent?**
  - **Because the event being waited for can only occur as a result of the concurrent execution of some other always/initial block or gate.**
  - **They’re happening concurrently**

FAQs: behavioral model execution

- How does an *always* or *initial* statement start
  - That just happens at the start of simulation — arbitrary order
- Once executing, what stops it?
  - Executing either a #delay, @event, or wait(FALSE).
  - All always blocks need to have at least one of these. Otherwise, the simulator will never stop running the model -- (it’s an infinite loop!)
- How long will it stay stopped?
  - Until the condition that stopped it has been resolved
    - #delay … until the delay time has been reached
    - @(var) … until var changes
    - wait(var) … until var becomes TRUE
- Does time pass when a behavioral model is executing?
  - No. The statements (if, case, etc) execute in zero time.
  - Time passes when the model stops for #, @, or wait.
- Will an *always* stop looping?
  - No. But an initial will only execute once.

Using a case statement

- Truth table method
  - List each input combination
  - Assign to output(s) in each case item.
- Concatenation
  - {a, b, c} concatenates *a*, *b*, and c together, considering them as a single item
  - Example

    a = 4’b0111

    b = 6’b 1x0001

    c = 2’bzx

  then {a, b, c} = 12’b01111x0001zx

```
module fred (f, a, b, c);
    output   f;
    input    a, b, c;
    reg      f;

    always @ (a or b or c)
        case ({a, b, c})
            3’b000: f = 1’b0;
            3’b001: f = 1’b1;
            3’b010: f = 1’b1;
            3’b011: f = 1’b1;
            3’b100: f = 1’b1;
            3’b101: f = 1’b0;
            3’b110: f = 1’b0;
            3’b111: f = 1’b1;
        endcase
endmodule
```

Check the rules …

How about a Case Statement Ex?

- **Here’s another version ...**

```
module fred (f, a, b, c);
    output   f;
    input    a, b, c;
    reg      f;

    always @ (a or b or c)
        case ({a, b, c})
            3’b000: f = 1’b0;
            3’b001: f = 1’b1;
            3’b010: f = 1’b1;
            3’b011: f = 1’b1;
            3’b100: f = 1’b1;
            3’b101: f = 1’b0;
            3’b110: f = 1’b0;
            3’b111: f = 1’b1;
        endcase
endmodule
```

check the rules…

Could put a function here too

```
module fred (f, a, b, c);
    output   f;
    input    a, b, c;
    reg      f;

    always @ (a or b or c)
        case ({a, b, c})
            3’b000: f = 1’b0;
            3’b101: f = 1’b0;
            3’b110: f = 1’b0;
            default: f = 1’b1;
        endcase
endmodule
```

Important: *every* control path is specified

Two inputs, Three outputs

```
reg   [1:0]    newJ;
reg            out;
input          i, j;
always @(i or j)
     case (j)
     2'b00:    begin
                    newJ = (i == 0) ? 2'b00 : 2'b01;
                    out = 0;
               end
      2'b01 :  begin
                    newJ = (i == 0) ? 2'b10 : 2'b01;
                    out = 1;
               end
      2'b10 :  begin
                    newJ = 2'b00;
                    out = 0;
               end
     default: begin
                    newJ = 2'b00;
                    out = 1'bx;
               end
endcase
```

Works like the C conditional operator.

(expr) ? a : b;

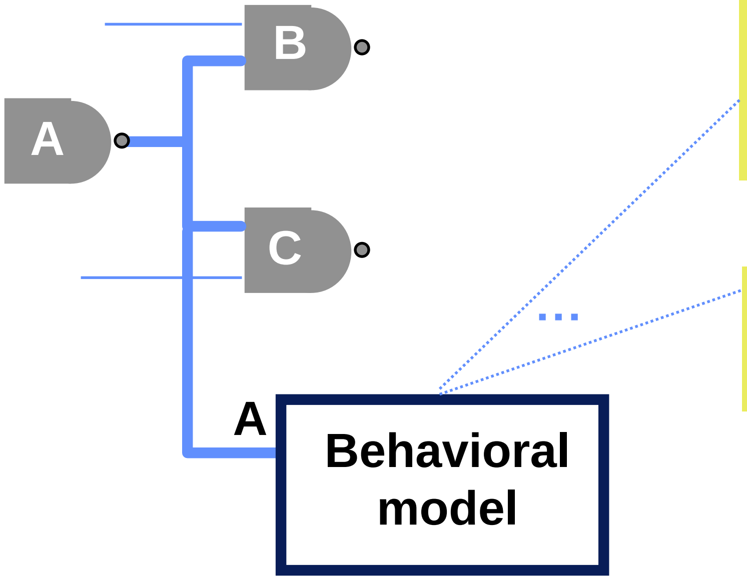
If the expr is true, then the resulting value is a, else it's b.

Behavioral Timing Model *(Not fully detailed here)*

- How does the behavioral model advance time?
  - # — delaying a specific amount of time
  - @ — delaying until an event occurs (“posedge”, “negedge”, or any change)
    - this is edge-sensitive behavior
  - wait — delaying until an event occurs (“wait (f == 0)”)
    - this is level sensitive behavior
- What is a behavioral model sensitive to?
  - any change on any input? — No
  - any event that follows, say, a “posedge” keyword
    - e.g. @posedge clock
    - Actually “no” here too. — not always

What are behavioral models sensitive to?

- Quick example
  - Gate A changes its output, gates B and C are evaluated to see if their outputs will change, if so, their fanouts are also followed…
  - The behavioral model will only execute if it was waiting for a change on the A input



```
always @(A)
     begin
          B = ~A;
      end
```

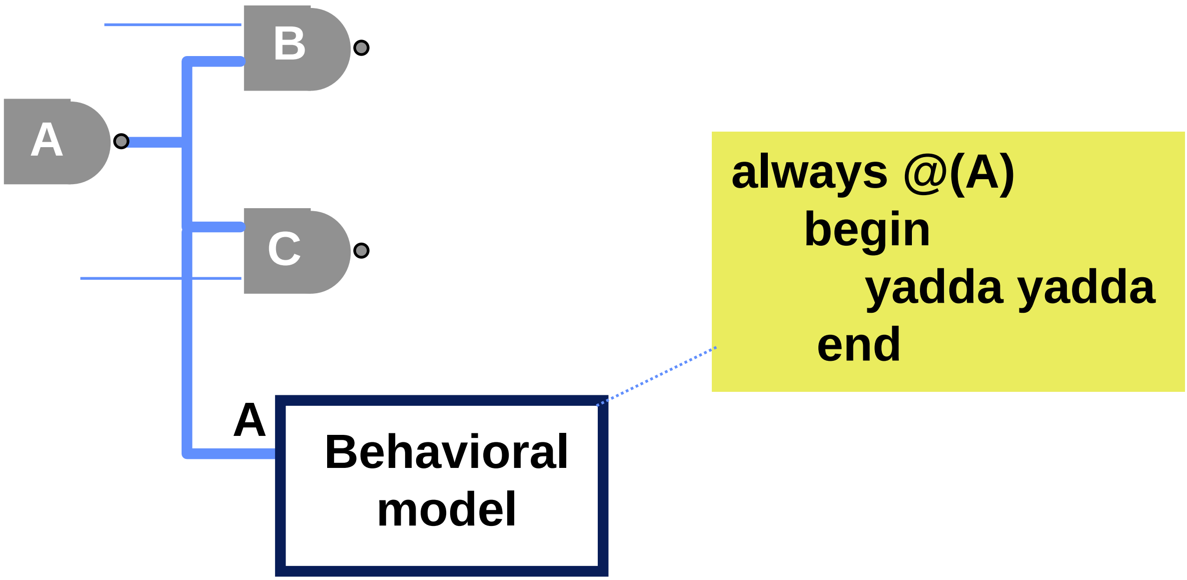
This would execute

```
always @(posedge clock)
          Q <= A;
```

This wouldn't

Order of Execution

- In what order do these models execute?
  - Assume A changes. Is B, C, or the behavioral model executed first?
    - Answer: the order is *defined* to be arbitrary
  - All events that are to occur at a certain time will execute in an arbitrary order.
  - The simulator will try to make them look like they all occur at the same time — but we know better.



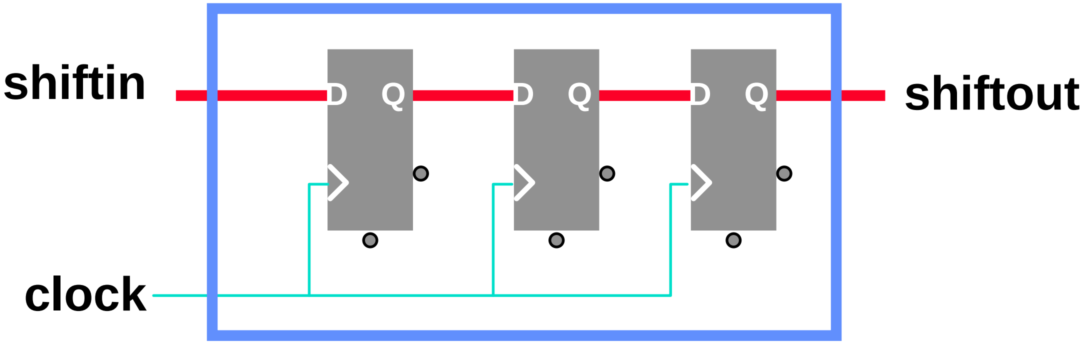
```
always @(A)
    begin
        yadda yadda
    end
```

Arbitrary Order? Oops!

- Sometimes you need to exert some control
  - Consider the interconnections of this D-FF
  - At the positive edge of c, what models are ready to execute?
  - Which one is done first?

```
module dff(q, d, c);
    …
    always @(posedge c)
            q = d;
endmodule

module sreg (…);
    …
    dff     a (q0, shiftin, clock),
            b (q1, q0, clock),
            c (shiftout, q1, clock);
endmodule
```

Oops — The order of execution can matter!



Behavioral Timing Model

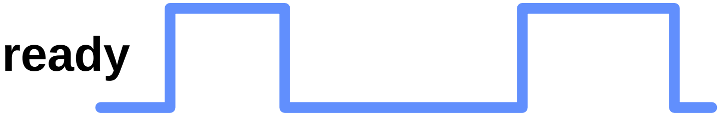
- How does the behavioral model advance time?
  - # — delaying a specific amount of time
  - @ — delaying until an event occurs — e.g. @v
    - “posedge”, “negedge”, or any change
    - this is edge-sensitive behavior
    - When the statement is encountered, the value v is sampled. When v changes in the specified way, execution continues.
  - wait — delaying until an event occurs (“wait (f == 0)”)
    - this is level sensitive behavior
  - While one model is waiting for one of the above reasons, other models execute — time marches on

Wait

- Wait — waits for a level on a line
  - How is this different from an "@" ?
- Semantics
  - wait (expression) statement;
    - e.g. wait (a == 35) q = q + 4;
  - if the expression is FALSE, the process is stopped
    - when *a* becomes 35, it resumes with q = q + 4
  - if the expression is TRUE, the process is <u>not</u> stopped
    - it continues executing
- Partial comparison to @ and #
  - @ and # always "block" the process from continuing
  - wait blocks only if the condition is FALSE

An example of wait

```
module handshake (ready, dataOut, …)
    input             ready;
    output  [7:0]  dataOut;
    reg       [7:0]  someValueWeCalculated;

    always begin
        wait (ready);
        dataOut = someValueWeCalculated;
        …
        wait (~ready)
        …
    end
endmodule
```



Do you always get the value right when ready goes from 0 to 1? Isn't this edge behavior?

Wait vs. While

- **Are these equivalent?**
  - **No: The left example is correct, the right one isn’t — it won’t work**
  - ***Wait* is used to wait for an expression to become TRUE**
    - **the expression eventually becomes TRUE because a variable in the expression is changed by another process**
  - ***While* is used in the normal programming sense**
    - **in the case shown, if the expression is TRUE, the simulator will continuously execute the loop. Another process will never have the chance to change “in”. Infinite loop!**
    - **while can’t be used to wait for a change on an input to the process. Need other variable in loop, or # or @ in loop.**

```
module yes (in, …);
input     in;
…
          wait (in == 1);
          …
endmodule
```

```
module no (in, …);
input     in;
…
          while (in != 1);
          …
endmodule
```

Blocking procedural assignments and #

- We've seen blocking assignments — they use =
  - Options for specifying delay

```
#10 a = b + c;
a = #10 b + c;
```

The difference?

  - The differences:

Note the action of the second one:

- an *intra-assignment* time delay
- execution of the always statement is blocked (suspended) in the middle of the assignment for 10 time units.
- how is this done?

Events — @something

- Action
  - when first encountered, sample the expression
  - wait for expression to change in the indicated fashion

  - This always blocks
- Examples

```
always @(posedge ck)
        q <= d;
```

```
always @(hello or goodbye)
        a = b;
```

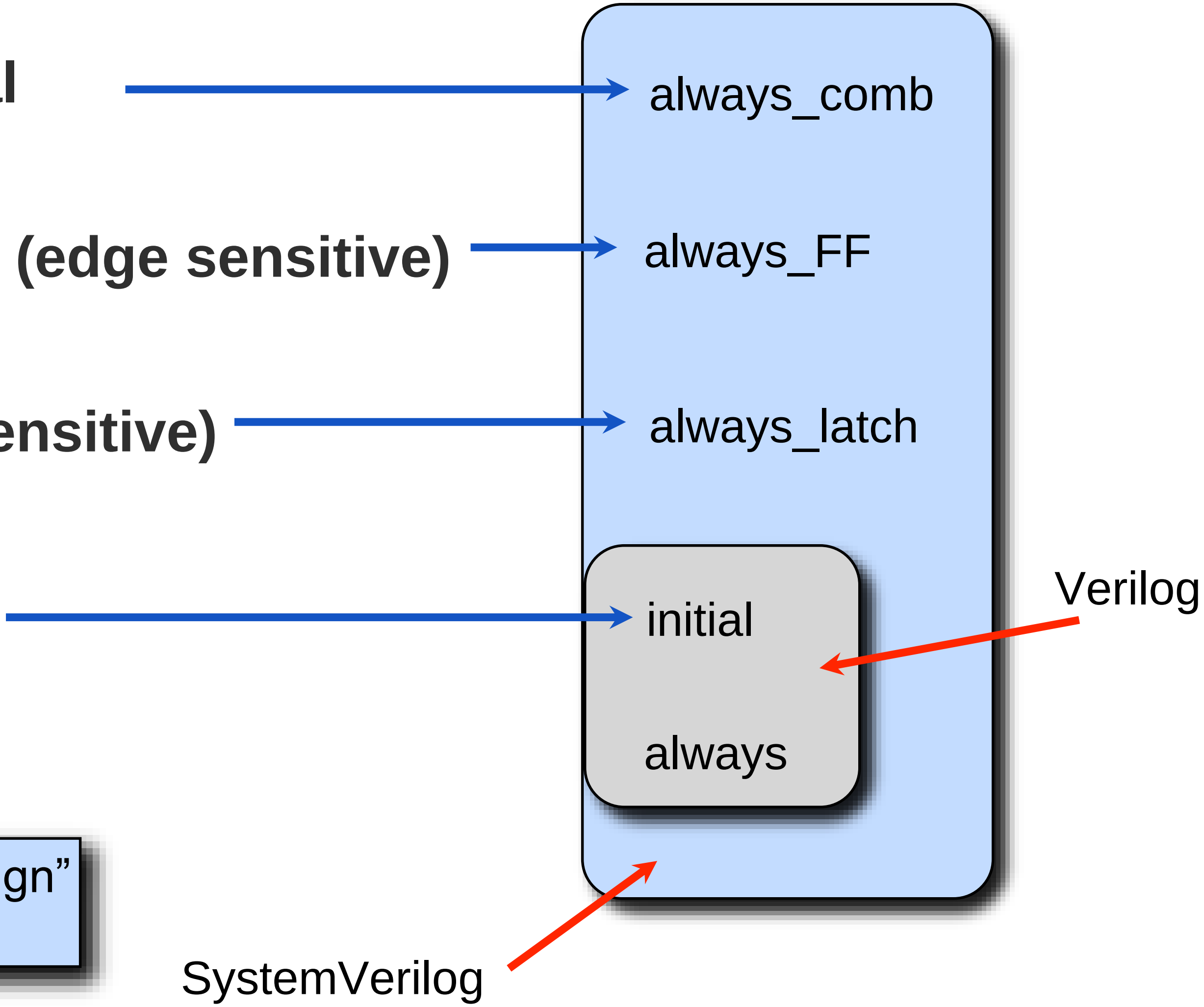
```
always @(hello)
        a = b;
```

```
always begin
        yadda = yadda;
        @(posedge hello or negedge goodbye)
        a = b;
        …
end
```

Part 3 –
Basics of SystemVerilog

Summary of what we're building

- ❑ **Combinational**
- ❑ **FSM/registers (edge sensitive)**
- ❑ **Latch (level sensitive)**
- ❑ **Testbench**
- ❑ **Other**



always_comb

always_FF

always_latch

initial

always

Verilog

SystemVerilog

What is the "how to design" view of this?

There are other things to build, but we'll get to them. They're built up from these basic modeling blocks.

Two types of modeling

- ❑ ***Structural*** **modeling**
  - ◆ focuses on interconnections of things
  - ◆ gate level designs — interconnections of gates
  - ◆ module hierarchy and interconnection
    - ❑ regardless of what's inside each module

```
        module mux1
        (output logic   f,
    input              a, b, sel);

     and #5   g1 (f1, a, nsel),
              g2 (f2, b, sel);
      or   #5  g3 (f, f1, f2);
     not         g4 (nsel, sel);
     endmodule: mux1
```

- ❑ ***Procedural*** **modeling**
  - ◆ modeling the functionality of logic blocks with statements that *look like* a programming language
    - ❑ look like, and execute like…
  - ◆ using always_i, initial, assign

```
        module mux2
         (output logic  f,
     input              a, b, sel);

            always_comb
                 if (sel)
                   f = b;
                 else
                   f = a;
     endmodule: mux2
```

Behavioral SystemVerilog

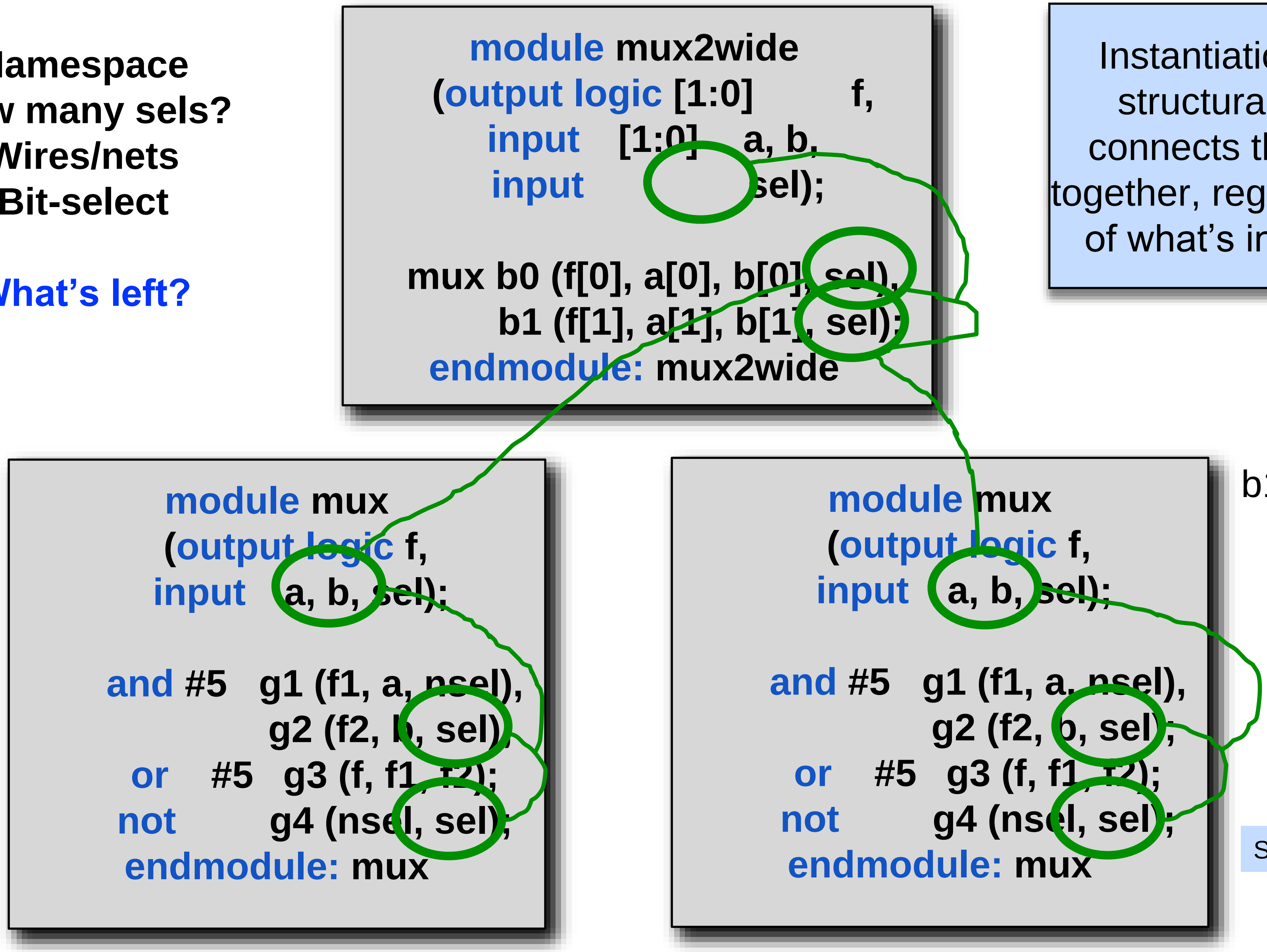
SystemVerilog:

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

Instantiation and hierarchy

Namespace
How many sels?
Wires/nets
Bit-select

What’s left?



```
module mux2wide
(output logic [1:0]        f,
 input        [1:0]    a, b,
 input                  sel);

 mux b0 (f[0], a[0], b[0], sel),
     b1 (f[1], a[1], b[1], sel);
endmodule: mux2wide
```

b0

```
module mux
(output logic f,
 input   a, b, sel);

 and #5   g1 (f1, a, nsel),
          g2 (f2, b, sel);
 or   #5  g3 (f, f1, f2);
 not      g4 (nsel, sel);
endmodule: mux
```

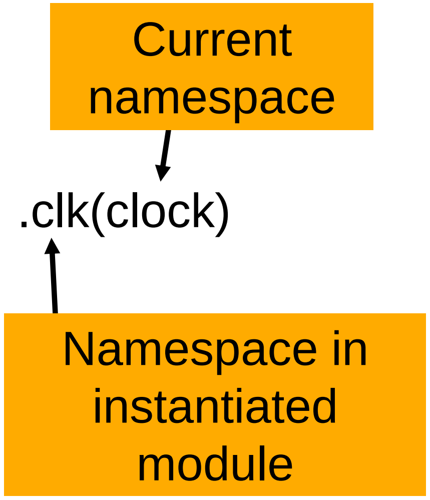
b1

```
module mux
(output logic f,
 input   a, b, sel);

 and #5   g1 (f1, a, nsel),
          g2 (f2, b, sel);
 or   #5  g3 (f, f1, f2);
 not      g4 (nsel, sel);
endmodule: mux
```

Instantiation is structural — connects things together, regardless of what’s inside!

SVbook 1.3

Port naming convention



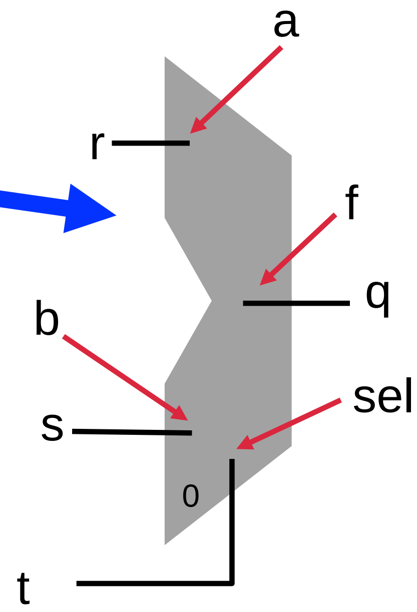
Current namespace

.clk(clock)

Namespace in instantiated module

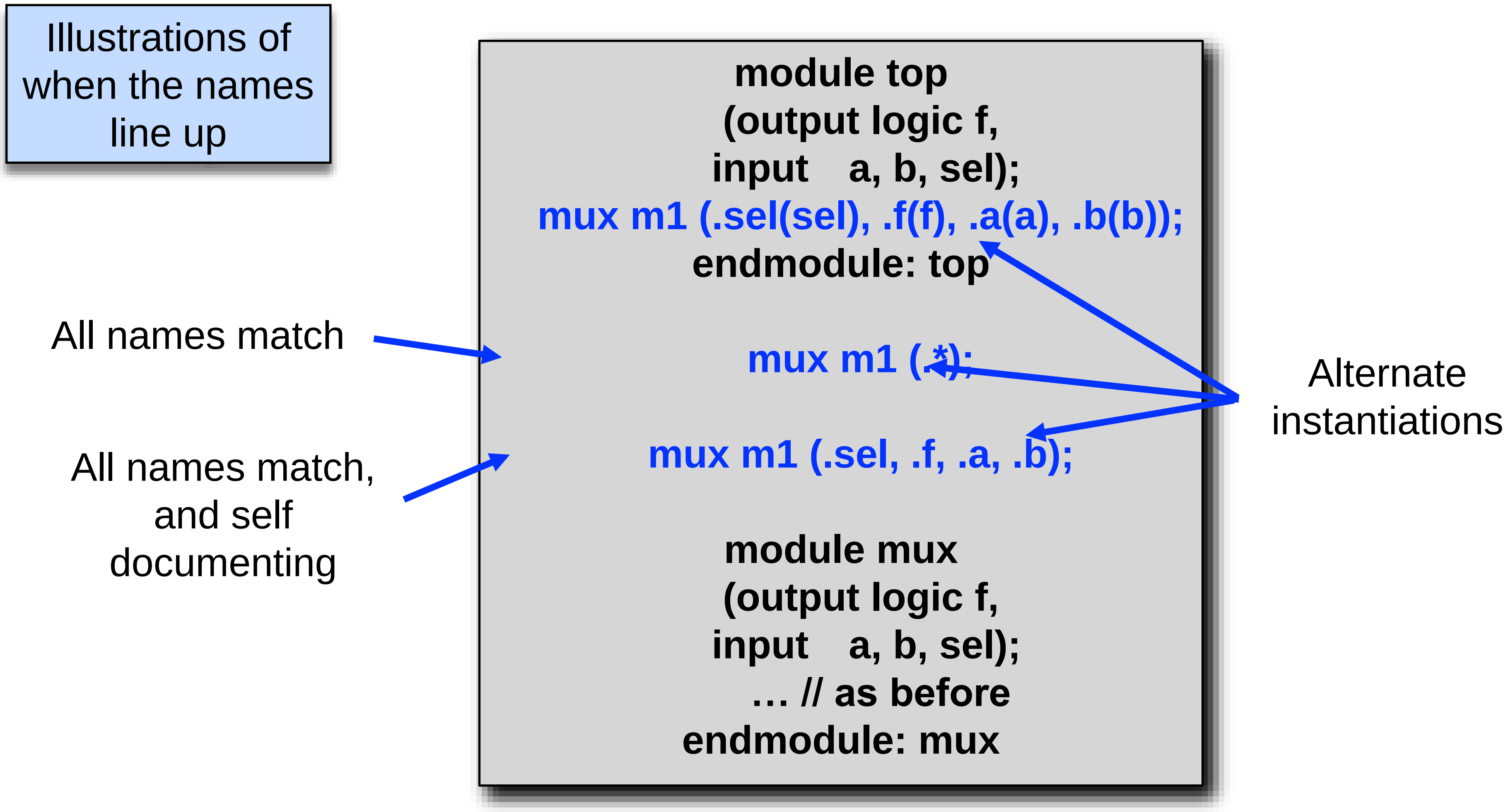
- ❑ Ordered-port connections
  - ◆ The port connections are given in the order that they're defined in the module being instantiated — most of our examples have used this
- ❑ Named-port connection — below
  - ◆ The port connections are specified by naming the element in the module that is being connected to
  - ◆ Many feel this is easier to follow/debug in a large description
  - ◆ Connections can be given in any order

```
module top
 (output logic q,
  input    r, s, t);
mux m1 (.sel(t), .f(q), .a(r), .b(s));
endmodule: top

module mux
 (output logic f,
  input    a, b, sel);
    … // as before
endmodule: mux
```



SVbook 2.5, 12.4.2

Alternate naming approaches



Illustrations of when the names line up

```
module top
 (output logic f,
 input   a, b, sel);
mux m1 (.sel(sel), .f(f), .a(a), .b(b));
endmodule: top

mux m1 (.*);

mux m1 (.sel, .f, .a, .b);

module mux
 (output logic f,
 input   a, b, sel);
   … // as before
endmodule: mux
```

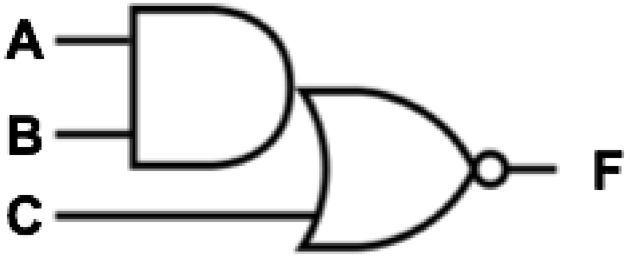
All names match

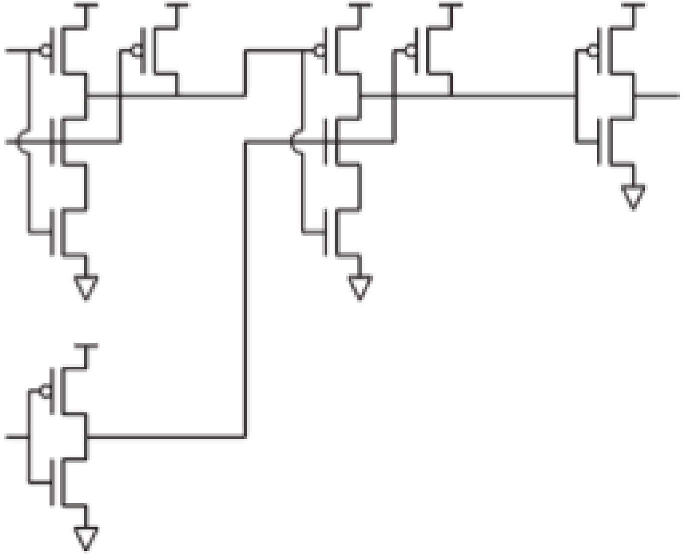
All names match, and self documenting

Alternate instantiations

Combinational ckts: What's underneath?

- How does it work?





Notice the natural flow through the circuitry — assign and always_comb model that

How to design a combinational ckt

- ❑ Two choices for most design
  - ◆ assign statements
  - ◆ always_comb statements (blocks)
  - ◆ No one uses gates (gates are not often used in higher level design except as outputs of synthesis tools, or tri-state drivers)
  - ◆ Use assign for “one-liners”
  - ◆ Use always_comb for more complex statements (if, case, …)

```
module  adder
(output logic  carryOut, sum,
 input               a, b, cIn);

assign   sum =  a ^ b ^ cIn,
         carryOut = (a & b) | (b & cIn)
                  | (a & cIn);
endmodule: adder
```

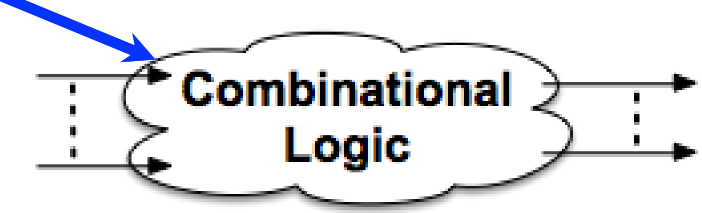
```
module adder
(output logic  f,
 input             sel, b, c);

always_comb begin
     sum =  a ^ b ^ cIn;
 carryOut = (a & b) | (b & cIn)
          | (a & cIn);
    end
endmodule: mux
```

SVbook 2.2

What do you mean, “no gates”

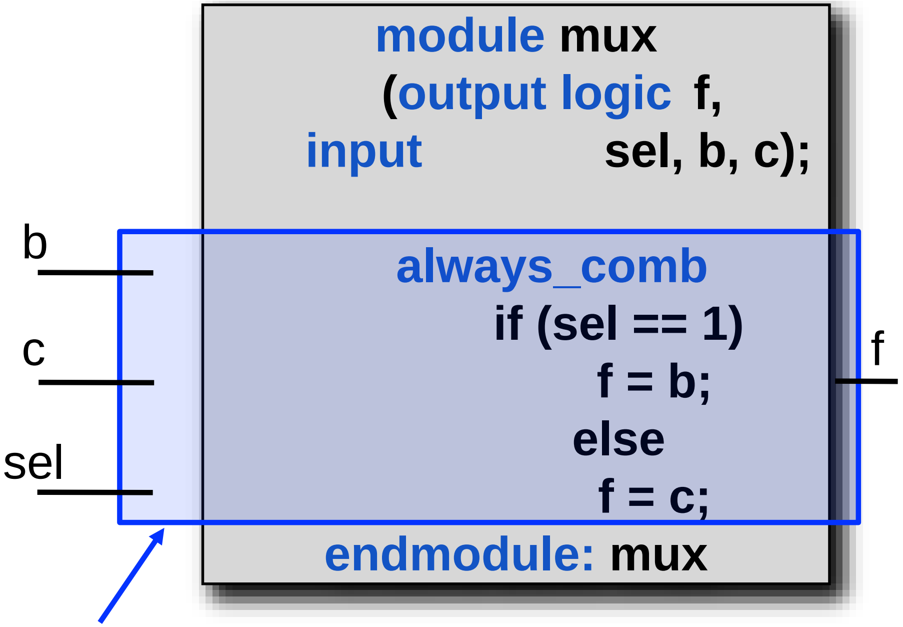
- ❑ In higher level logic design
  - ◆ People use synthesis tools to generate gate level logic
    - ❑ OK, they show up now and then
  - ◆ But no one cranks Boolean algebra, Kmaps, QM, or Shannon Expansion by hand for sizable designs
- ❑ We deal with blobs of combinational logic
  - ◆ Gate details are left to the synthesis tools and library builders

```
          module adder
           (output logic  f,
        input            sel, b, c);

        always_comb begin
           sum =  a ^ b ^ cIn;
     carryOut = (a & b) | (b & cIn)
                    | (a & cIn);
                    end
          endmodule: mux
```



Rules for Writing Combinational Logic

- It all boils down to writing a stateless function
- The rules for modeling combinational logic using procedural statements
  1. A change on any input of the function will cause it to execute
  2. The combinational output(s) must be assigned in any execution (thus in every control path)
  3. Nothing is remembered from execution to execution



```
module mux
   (output logic  f,
input             sel, b, c);

   always_comb
      if (sel == 1)
         f = b;
      else
         f = c;
endmodule: mux
```

virtual module — a blob of combinational logic

Issue 1, handled by always_comb or assign

Issue 2, handled by you!

Issue 3, (another way of stating issue 2) the loop is *stateless*. Nothing remembered internally. Handled by you!

SVbook 2.1.2

What about sequential elements?

- What's a sequential element?
  - a flip flop —
  - a latch —
  - Cross-coupled Nand or Nor gates
    - Huh? but they're just combinational logic…?
- How to do it procedurally
  - sequential elements are ***not explicitly specified***
  - rather, they are ***inferred***
  - ***hmm, a stateless function was used to infer combinational logic***
- That is,
  - synthesis tools infer — figure out — deduce — that you need a sequential element
  - need to write a loop that requires state to implement it
  - and maybe clock edges, reset and preset

To infer a flip-flop

❑ Flip-flops are edge-triggered

- ◆ need to use @ with posedge or negedge
- ◆ side effect — every variable on the left-hand side assigned with a “<=” will be an edge triggered flip-flop (or vector of flip-flops)
- ◆ Some combinational logic can be specified in certain places

```
            module Dff
           (output   logic Q,
      input              clk, d, resetN);

      always_ff @(posedge clk,
                  negedge resetN)
                     if (~resetN)
                        Q <= 0;
                     else
                        Q <= d;
            endmodule
```

What happens on the clock edge is inferred — it’s the last (default) action.

The assignment is made to a logic (or bit …) variable, not a reg as in old Verilog. Synthesis infers an FF

Q, d, clk, and resetN are not keywords

<= needed for synthesis and simulation

SVbook 3.1

How reset works

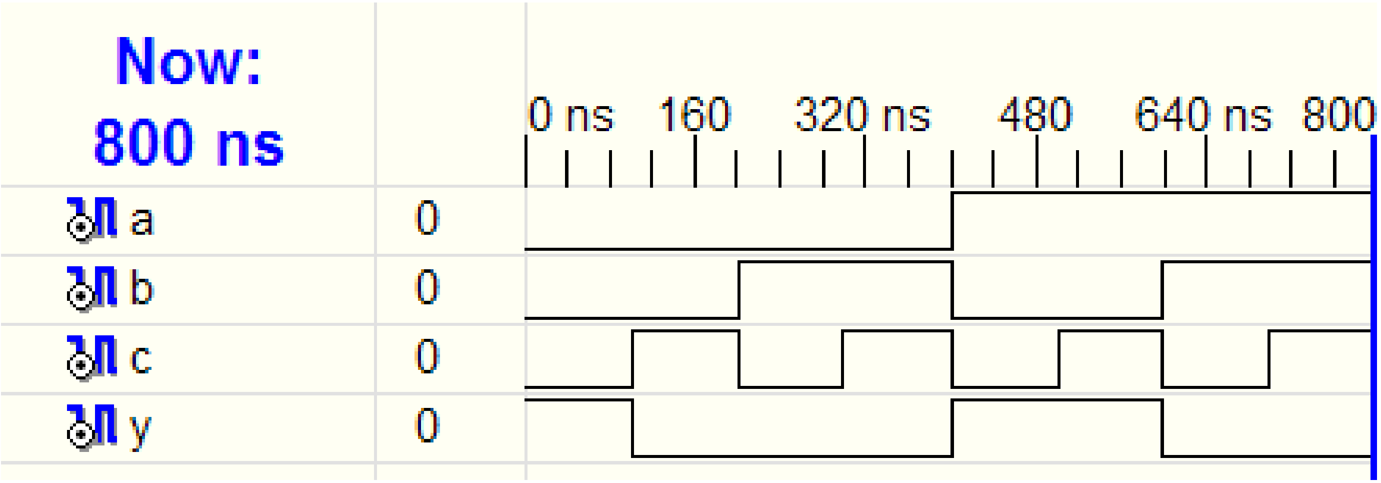
❑ reset is asynchronous … hmm, but there’s an edge

```
module Dff
    (output  logic Q,
  input      clk, d, resetN);

always_ff @(posedge clk, negedge resetN)
         if (~resetN)
            Q <= 0;
           else
            Q <= d;
endmodule
```

HDL Simulation

SystemVerilog:

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

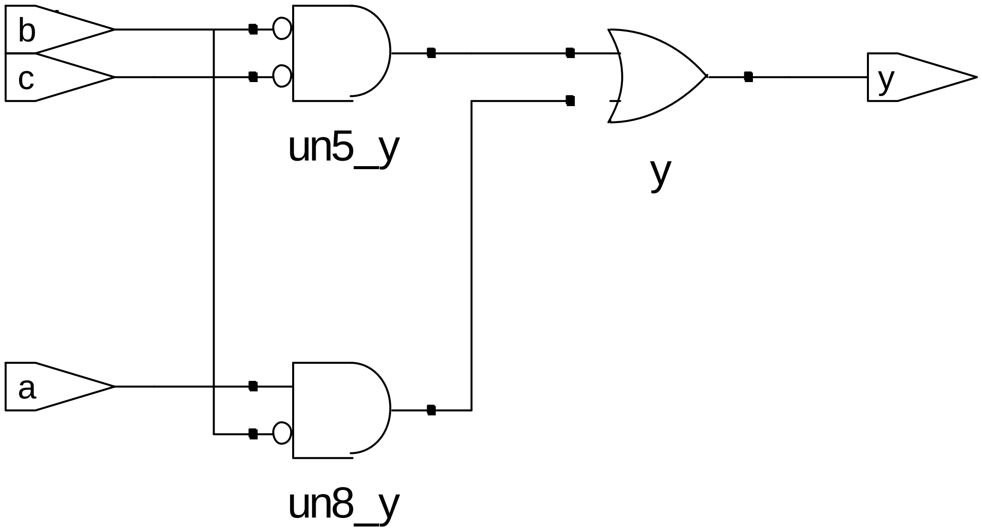


HDL Synthesis

SystemVerilog:

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

Synthesis:



SystemVerilog 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:

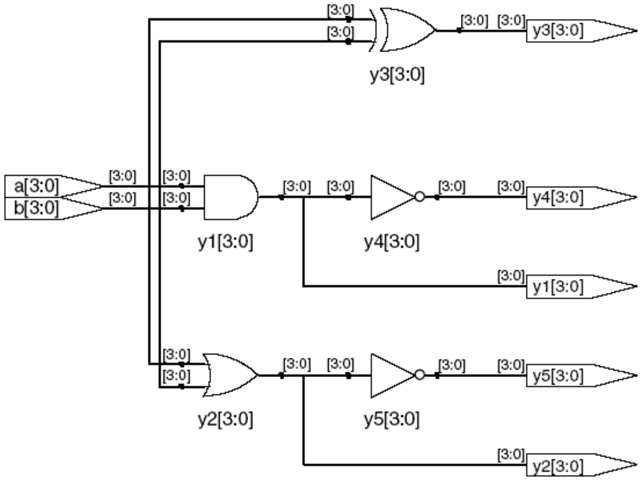
```
– // single line comment
– /* multiline
         comment */
```

Structural Modeling - Hierarchy

```
module and3(input  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


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 inverter
endmodule
```

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
```

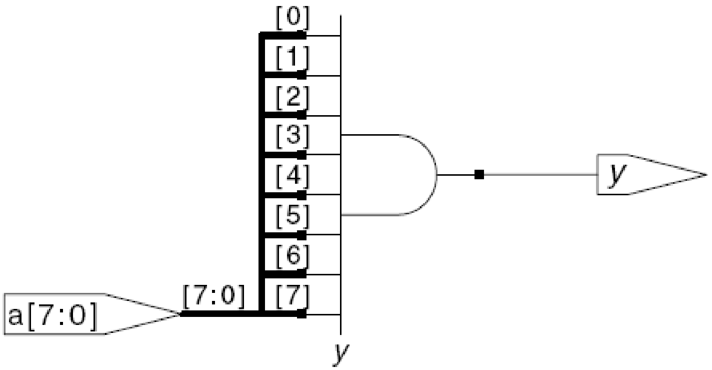


// single line comment

/*…*/ multiline comment

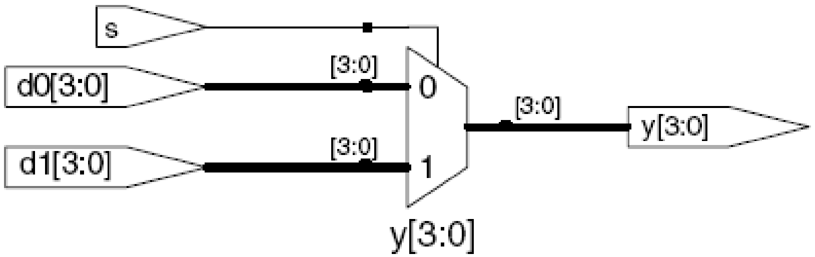
Reduction Operators

```
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
```



Conditional Assignment

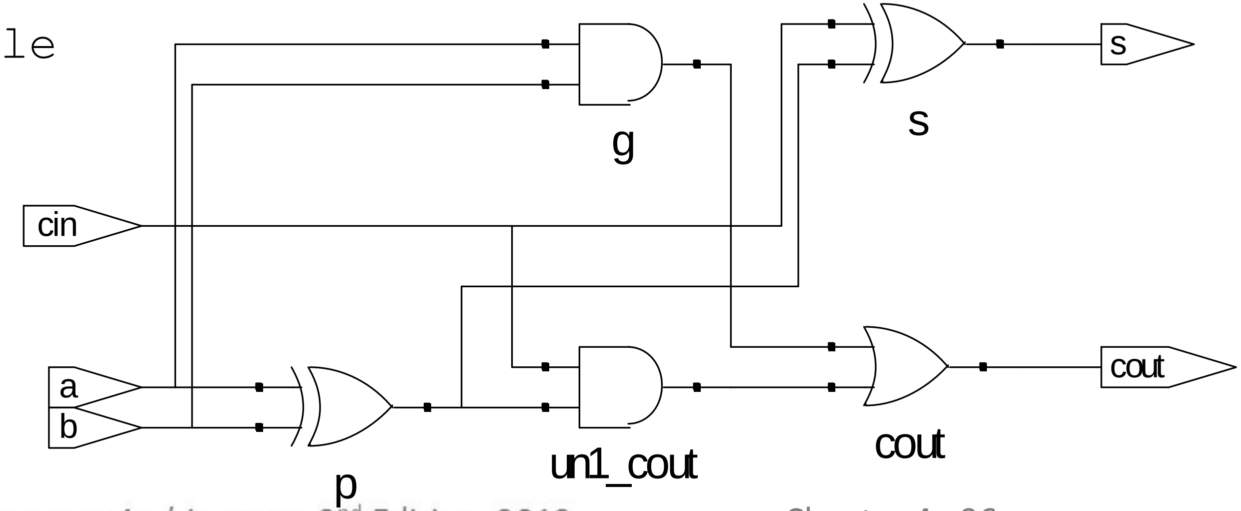
```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);
   assign y = s ? d1 : d0;
endmodule
```



`? :` is also called a *ternary operator* because it operates on 3 inputs: `s`, `d1`, and `d0`.

Internal Variables

```
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
```



Precedence

Order of operations

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

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
```

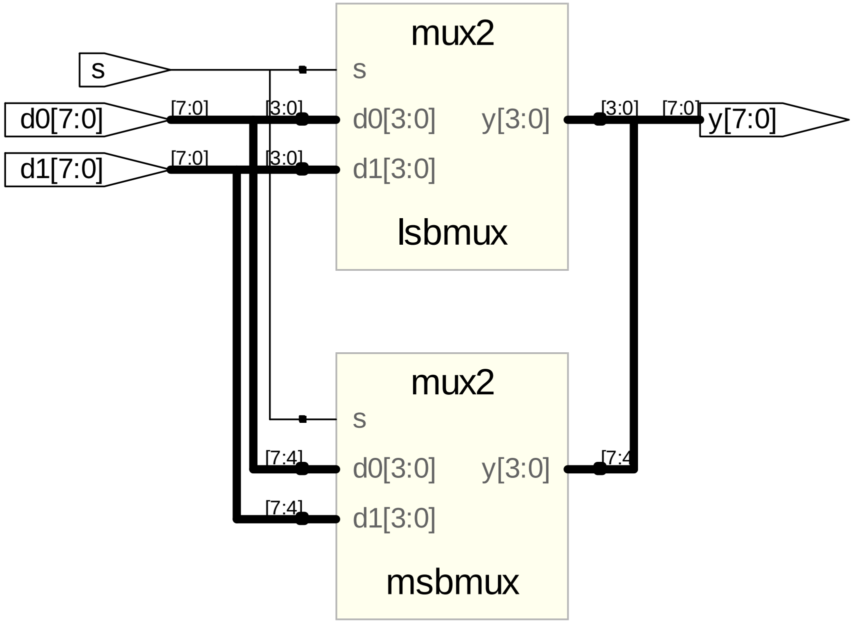
```
// underscores (_) are used for formatting only to make
   it easier to read. SystemVerilog ignores them.
```

ELSEVIER

Bit Manipulations: Example 2

SystemVerilog:

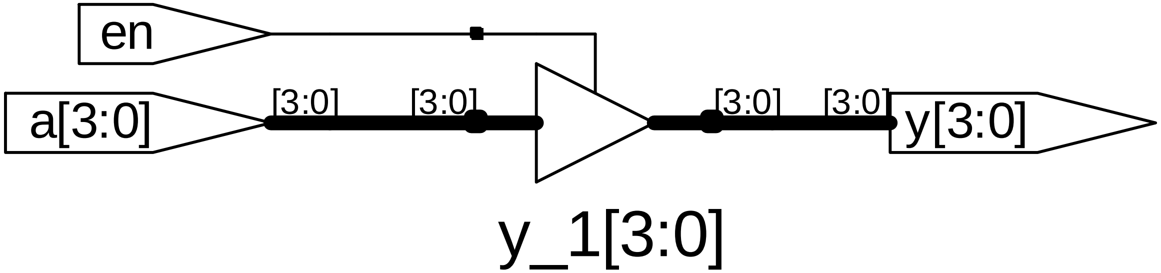
```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  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
```



Z: Floating Output

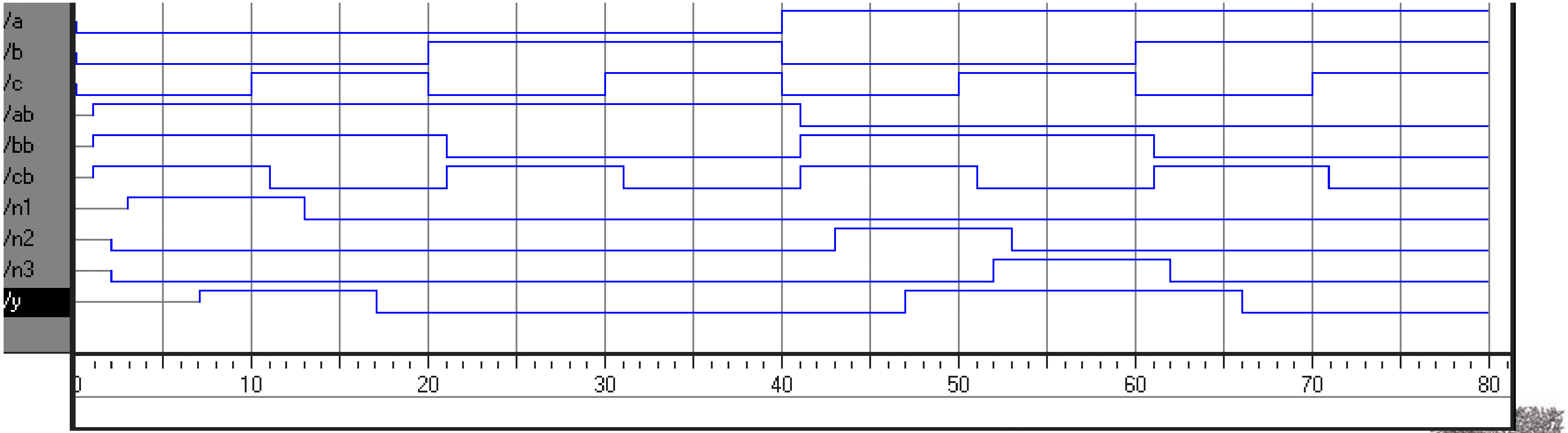
SystemVerilog:

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



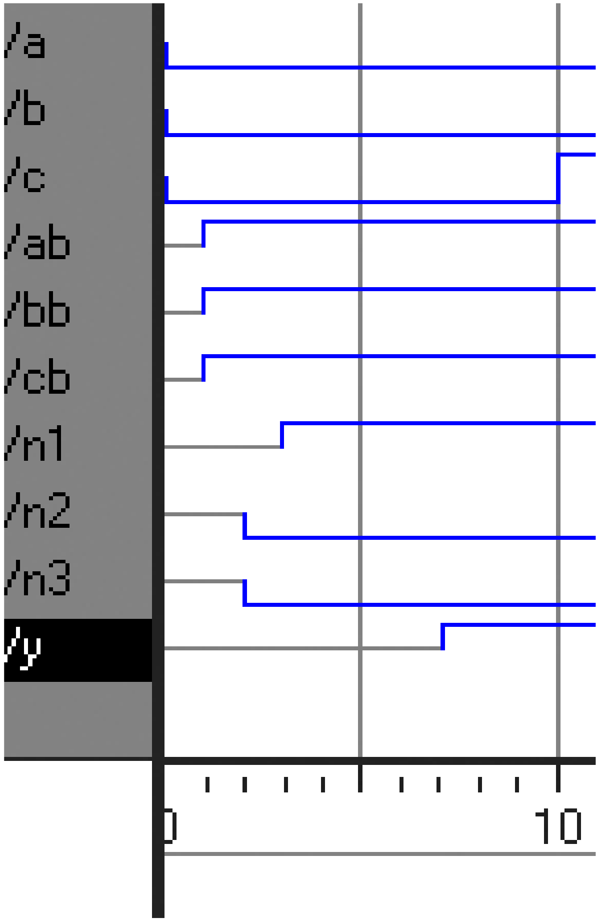
Delays

```
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

```
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
```



Sequential Logic

- SystemVerilog uses **Idioms** to describe latches, flip-flops and FSMs
- Other coding styles may simulate correctly but produce incorrect hardware

Always Statement

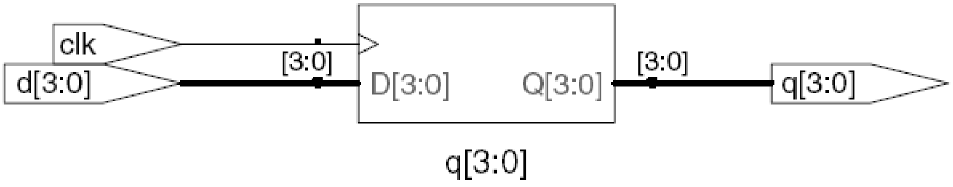
General Structure:

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

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

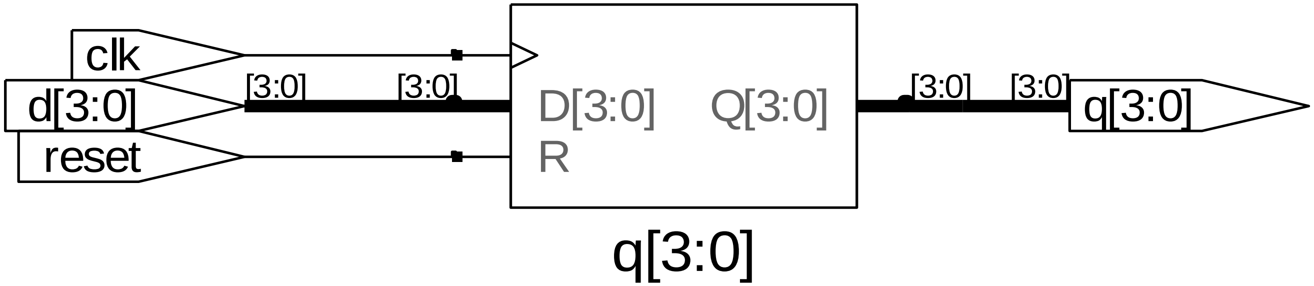
D Flip-Flop

```
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
```



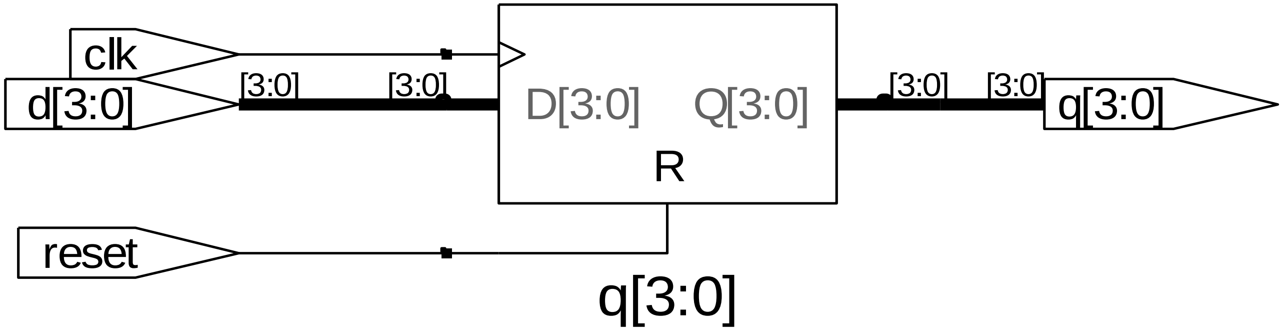
Resettable D Flip-Flop

```
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
```



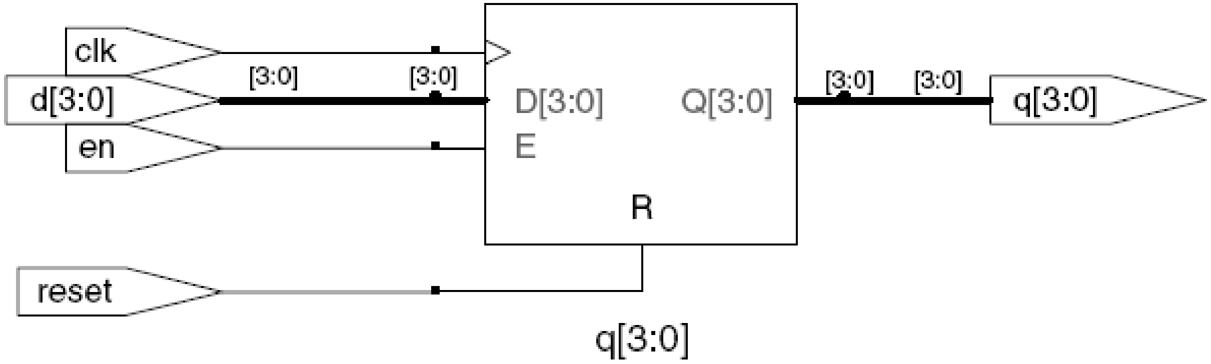
Resettable D Flip-Flop

```
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
```



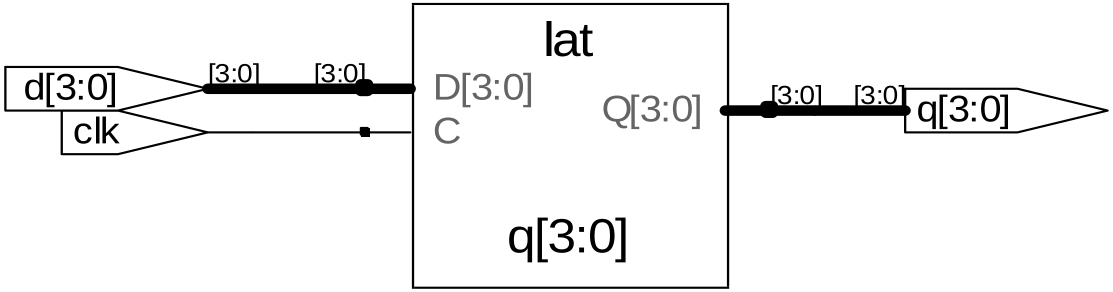
D Flip-Flop with Enable

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

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

endmodule
```



Latch

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

  always_latch
    if (clk) q <= d;

endmodule
```



Warning: We don’t use latches in this text. But you might write code that inadvertently implies a latch. Check synthesized hardware – if it has latches in it, there’s an error.

Other Behavioral Statements

- Statements that must be inside `always` statements:
  - `if/else`
  - `case, casez`

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.

Combinational Logic using case

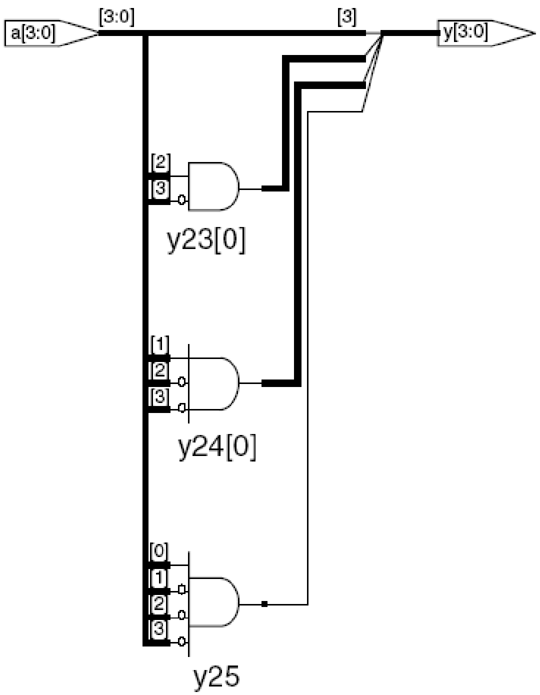
```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case (data)
      //                       abc_defg
      0: segments =       7'b111_1110;
      1: segments =       7'b011_0000;
      2: segments =       7'b110_1101;
      3: segments =       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_0011;
      default: segments = 7'b000_0000; // required
    endcase
endmodule
```

Combinational Logic using `case`

- `case` statement implies combinational logic **only if** all possible input combinations described
- Remember to use **`default`** statement

Combinational Logic using casez

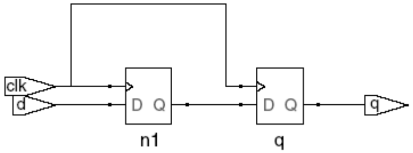
```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);
  always_comb
    casez(a)
      4'b1???: y = 4'b1000;  // ? = don’t care
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
   endcase
endmodule
```



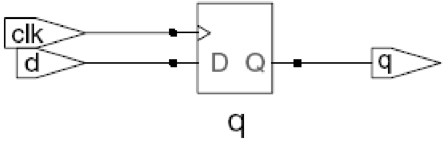
Blocking vs. Nonblocking Assignment

- <= is **nonblocking** assignment
  - Occurs simultaneously with others
- = is **blocking** assignment
  - Occurs in order it appears in file

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



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



Rules for Signal Assignment

- **Synchronous sequential logic:** use **`always_ff @(posedge clk)`** and nonblocking assignments (**`<=`**)

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

- **Simple combinational logic:** use continuous assignments (**`assign`**…)

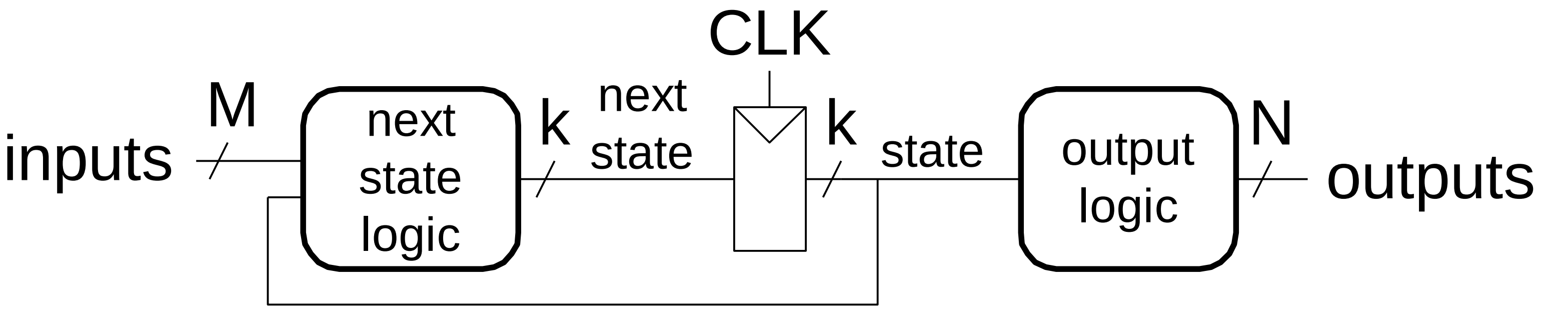
```
assign y = a & b;
```

- **More complicated combinational logic:** use **`always_comb`** and blocking assignments (**`=`**)

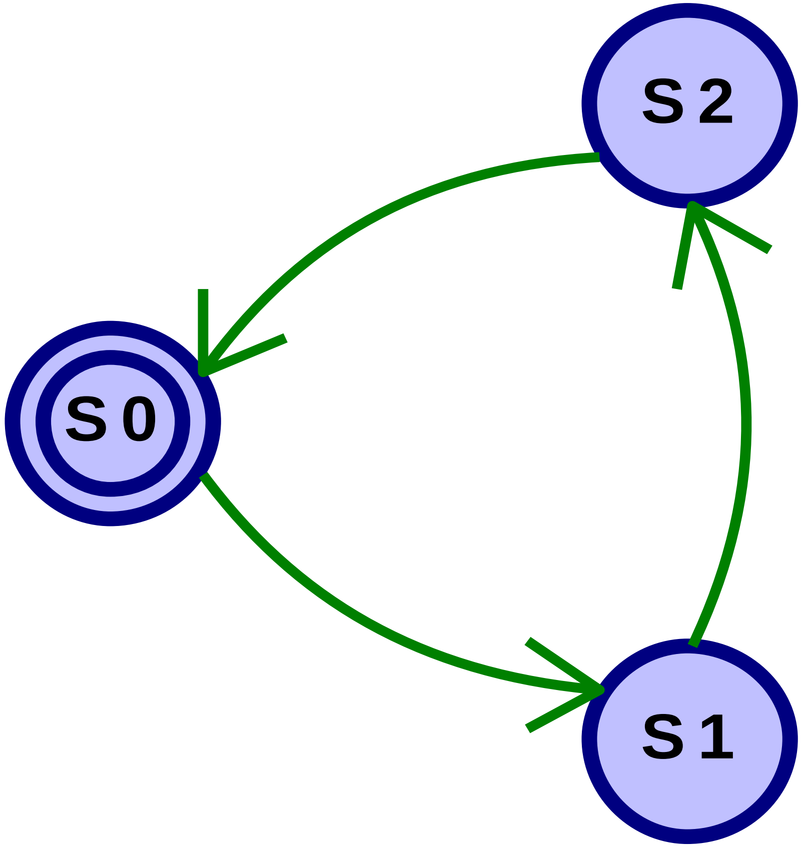
- Assign a signal in **only one** `always` statement or continuous assignment statement.

Finite State Machines (FSMs)

- **Three blocks:**
  - next state logic
  - state register
  - output logic



FSM Example: Divide by 3



The double circle indicates the reset state

FSM in SystemVerilog

```
module divideby3FSM (input  logic clk,
                     input  logic reset,
                     output logic q);
   typedef enum logic [1:0] {S0, S1, S2} statetype;
   statetype [1:0] state, nextstate;

   // state register
   always_ff @ (posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // next state logic
   always_comb
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase

   // output logic
   assign q = (state == S0);
endmodule
```

Parameterized Modules

2:1 mux:

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

Testbenches

- HDL that tests another module: *device under test* (dut)
- Not synthesizeable
- Types:
  - Simple
  - Self-checking
  - Self-checking with testvectors

Testbench Example

- Write SystemVerilog code to implement the following function in hardware:

$$y = \bar{b}\bar{c} + a\bar{b}$$

- Name the module `sillyfunction`

Testbench Example

- Write SystemVerilog code to implement the following function in hardware:

$$y = \overline{b}\,\overline{c} + a\overline{b}$$

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

Simple Testbench

```
module testbench1();
  logic a, b, c;
  logic y;
  // instantiate device under test
  sillyfunction dut(a, b, c, y);
  // apply inputs one at a time
  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
    a = 1; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
  end
endmodule
```

Self-checking Testbench

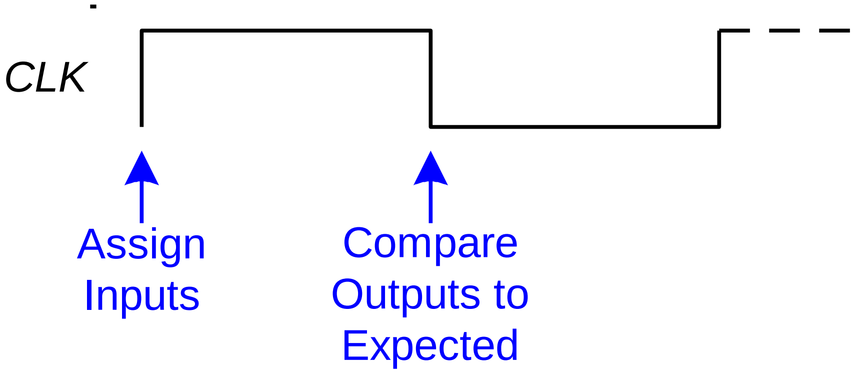
```
module testbench2();
  logic  a, b, c;
  logic y;
  sillyfunction dut(a, b, c, y);  // instantiate dut
  initial begin // apply inputs, check results one at a time
    a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display("000 failed.");
    c = 1; #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("010 failed.");
    c = 1; #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if (y !== 1) $display("100 failed.");
    c = 1; #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("110 failed.");
    c = 1; #10;
    if (y !== 0) $display("111 failed.");
  end
endmodule
```

Testbench with Testvectors

- Testvector file: inputs and expected outputs
- Testbench:
  1. Generate clock for assigning inputs, reading outputs
  2. Read testvectors file into array
  3. Assign inputs, expected outputs
  4. Compare outputs with expected outputs and report errors

Testbench with Testvectors

- Testbench clock:
  - assign inputs (on rising edge)
  - compare outputs with expected outputs (on falling edge).



- Testbench clock also used as clock for synchronous sequential circuits

Testvectors File

- File: `example.tv`
- contains vectors of abc_yexpected

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

1. Generate Clock

```
module testbench3();
  logic        clk, reset;
  logic        a, b, c, yexpected;
  logic        y;
  logic [31:0] vectornum, errors;    // bookkeeping variables
  logic [3:0]  testvectors[10000:0]; // array of testvectors

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always     // no sensitivity list, so it always executes
    begin
      clk = 1; #5; clk = 0; #5;
    end
```

2. Read Testvectors into Array

```
// at start of test, load vectors and pulse reset

initial
   begin
     $readmemb("example.tv", testvectors);
     vectornum = 0; errors = 0;
     reset = 1; #27; reset = 0;
   end
```

```
// Note: $readmemh reads testvector files written in
// hexadecimal
```

3. Assign Inputs & Expected Outputs

```
// apply test vectors on rising edge of clk
always @(posedge clk)
  begin
    #1; {a, b, c, yexpected} = testvectors[vectornum];
  end
```

4. Compare with Expected Outputs

```
// check results on falling edge of clk
   always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin
        $display("Error: inputs = %b", {a, b, c});
        $display("  outputs = %b (%b expected)",y,yexpected);
        errors = errors + 1;
      end

// Note: to print in hexadecimal, use %h. For example,
//       $display("Error: inputs = %h", {a, b, c});
```

4. Compare with Expected Outputs

```
// increment array index and read next testvector
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
          $display("%d tests completed with %d errors",
                vectornum, errors);
        $finish;
      end
    end
endmodule


// === and !== can compare values that are 1, 0, x, or z.
```