Introduction to SystemVerilog

Lecture 2 & 3

Reza Molavi
Dept. of ECE
University of British Columbia
reza@ece.ubc.ca

Slides Courtesy : Prof. Sudip Shekhar (UBC)

Hardware Description Language (HDL)

- **Shorthand for describing digital hardware**
- **Don't treat HDLs as programming languages**
  - Begin your design process by planning the hardware you want
  - Then, write the HDL code that implies that hardware to a synthesis tool.
- **Otherwise – surprises**
  - Can't synthesize
  - Extra latches appearing in places you didn't expect.
  - Much slower circuit
  - Far more gates
- **Focus on synthesizable HDL**
  - Many ways to write HDL code whose behavior in simulation and synthesis differ, resulting in improper chip operation or the need to fix bugs after synthesis is complete.

Different HDLs

- **VHDL - Very High Speed Integrated Circuits Hardware Description Language**
  - Developed in 1981 by the US Department of Defense, inspired by the Ada programming language
  - Still heavily used by U.S. military contractors and European companies.
- **Verilog**
  - Developed by Gateway Design Automation in 1984. Gateway was acquired by Cadence in 1989 and Verilog was made an open standard Projects.
- **SystemVerilog**
  - Introduced in 2005 to streamline many of the "annoyances" of Verilog and add high-level programming language features that have proven useful in verification
  - Based on OpenVera language donated by Synopsys
  - Both HDL & Hardware Verification Language
  - Uses object-oriented programming techniques

A 2007 user survey claims that 73% of respondents primarily used Verilog/SystemVerilog and 20% primarily used VHDL, but 41% needed to use both on their project because of legacy code, IP blocks, or because Verilog is better suited to netlists.

Modules

- **A Block of hardware with inputs and outputs**
  - E.g. an AND gate, a multiplexer, and a priority circuit
- **Describe a module using:**
  - Behavioral models – describe what a module does.
  - Structural models – describe how a module is built from simpler pieces; it is an application of hierarchy.

Behavioral Model

```
module sillyfunction(input  logic a, b, c,
                     output logic y);

  assign y = ~a & ~b & ~c |
              a & ~b & ~c |
              a & ~b &  c;
endmodule
```

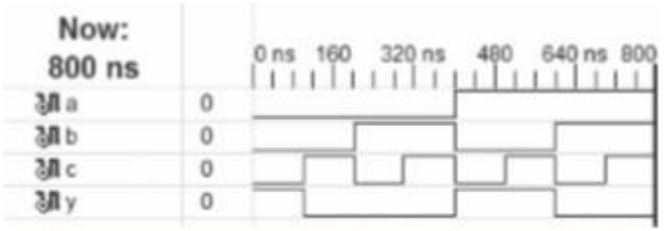
- **A module begins with a listing of the inputs and outputs.**
- **"assign" statement describes combinational logic.**
- **"logic" signals such as the inputs and outputs are Boolean variables (0 or 1).**
  - May also have floating and undefined values
  - The logic type was introduced in SystemVerilog. It supersedes the reg type, which was a perennial source of confusion in Verilog.
  - logic should be used everywhere except on nets with multiple drivers

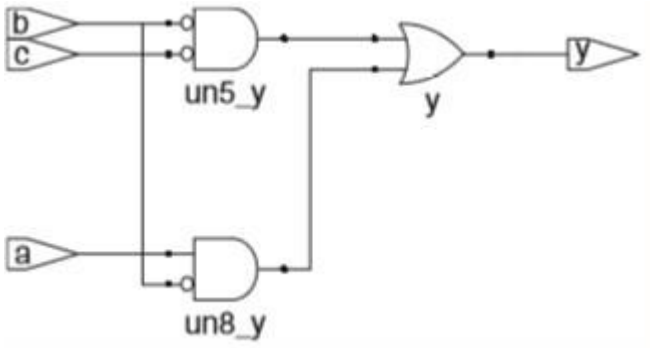
Higher level of abstraction vs. Sch

```
module adder(input  logic [31:0] a,
             input  logic [31:0] b,
             output logic [31:0] y);

  assign y = a + b;
endmodule
```

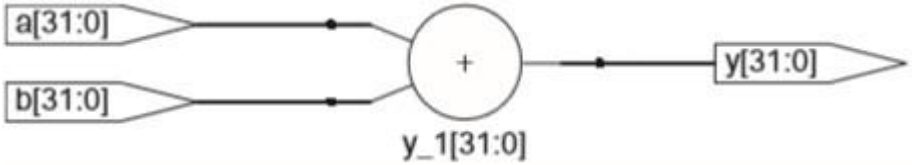
- Note that the inputs and outputs above are 32-bit busses
- **Schematic: A 32-bit adder schematic is a complicated structure. The designer must choose what type of adder architecture to use. A carry ripple adder has 32 full adder cells, each of which in turn contains half a dozen gates or a bucketful of transistors.**
- **HDL: Specify the adder with one line of behavioral HDL code!**

HDL – Logic Simulation & Synthesis





- **The logic synthesizer may perform optimizations to reduce the amount of hardware required.**



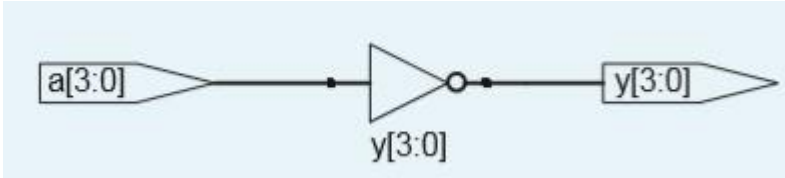
Combinational Logic

- **Outputs of combinational logic depend only on the current inputs; combinational logic has no memory.**
- **Outline**
  - Bitwise operators
  - Comments & White space
  - Reduction operators
  - Conditional assignment
  - Internal variables
  - Precedence & other operators
  - Numbers
  - Zs and Xs
  - Bit Swizzling
  - Delays

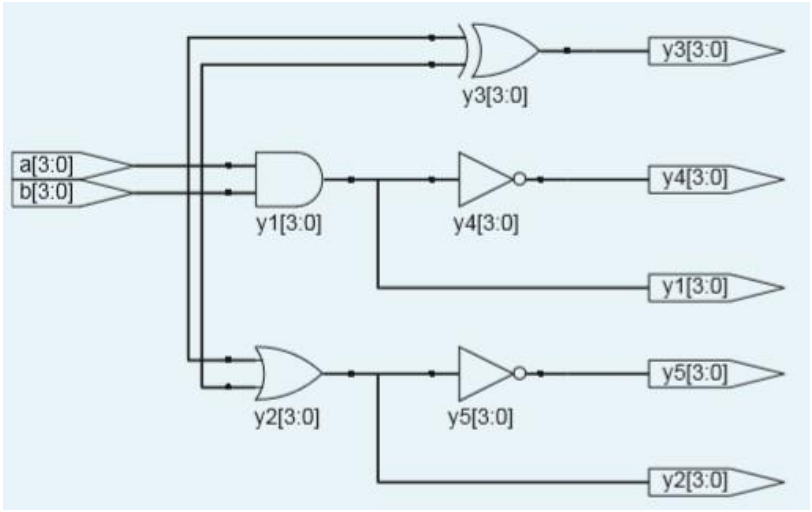
Bitwise Operators

- **Bitwise operators act on single-bit signals or on multibit busses (operands)**

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  assign y = ~a;
endmodule
```




```
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
```



- assign out = in1 op in2; is called a continuous assignment statement.
- Continuous assignment statements end with a semicolon.
- Any time the inputs on the RHS change, the output on the LHS is recomputed.
- Thus, continuous assignment statements describe combinational logic

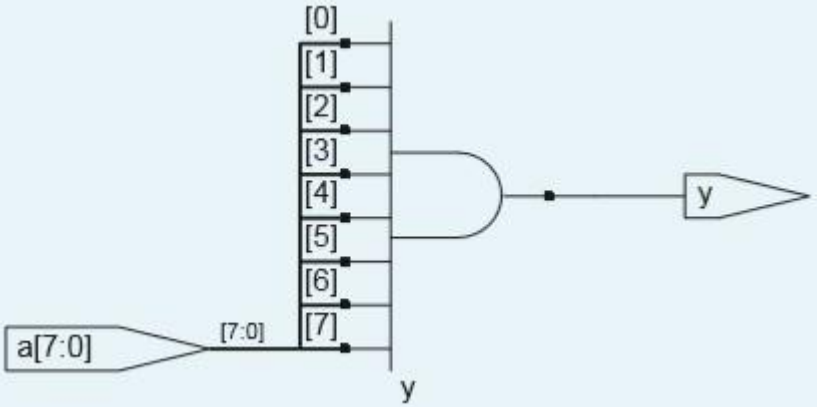
Comments & White Space

- **Freely use white spaces – spaces, tabs, and line breaks for proper indenting and to make nontrivial designs readable.**
  - Be consistent in your use of capitalization and underscores in signal and module names.
- **Comments just like C or Java.**
  - Comments beginning with /* continue, possibly across multiple lines, to the next */.
  - Comments beginning with // continue to the end of the line.
- **SystemVerilog is case-sensitive.**
  - y1 and Y1 are different signals in SystemVerilog.
  - However, using separate signals that only differ in their capitalization is a confusing and dangerous practice.

Reduction Operators

- **Reduction operators imply a multiple-input gate acting on a single bus**

```
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
```

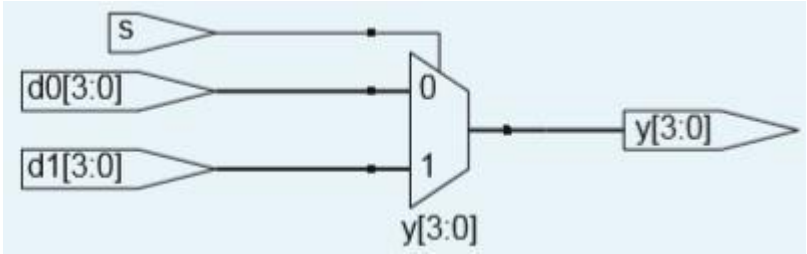


Conditional Assignment

- **The conditional operator ?: chooses, based on a first expression, between a second and third expression. The first expression is called the condition.**
  - If the condition is 1, the operator chooses the second expression.
  - If the condition is 0, the operator chooses the third expression.

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

   assign y = s ? d1 : d0;
endmodule
```

If `s = 1`, then `y = d1`. If `s = 0`, then `y = d0`.




```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

   assign y = s[1] ? (s[0] ? d3 : d2)
                   : (s[0] ? d1 : d0);
endmodule
```

If `s[1] = 1`, then the multiplexer chooses the first expression, `(s[0] ? d3 : d2)`. This expression in turn chooses either `d3` or `d2` based on `s[0]` (`y = d3` if `s[0] = 1` and `d2` if `s[0] = 0`). If `s[1] = 0`, then the multiplexer similarly chooses the second expression, which gives either `d1` or `d0` based on `s[0]`.

Internal Variables

- **Internal variables are neither inputs nor outputs but are only used internal to the module.**
  - Similar to local variables in programming languages

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

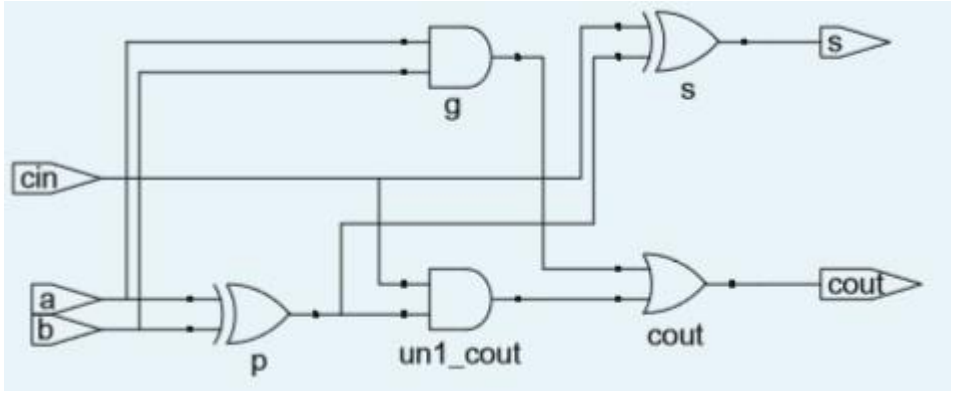
If we define intermediate signals P and G

$$P = A \oplus B$$
$$G = AB$$

we can rewrite the full adder as

$$S = P \oplus C_{in}$$
$$C_{out} = G + PC_{in}$$

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

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

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



- All “assign”s take place concurrently and the order does not matter.
  - Unlike C or Java in which statements are evaluated in the order they are written.
- Like hardware, assignment statements are evaluated any time the signals on the RHS change their value, regardless of the order in which they appear in a module.

Precedence and Other Operators

- **Similar to other programming languages.**

SystemVerilog operator precedence

	Op	Meaning
Highest	~	NOT
	*, /, %	MUL, DIV, MOD
	+, -	PLUS, MINUS
	<<, >>	Logical Left / Right Shift
	<<<, >>>	Arithmetic Left / Right Shift
	<, <=, >, >=	Relative Comparison
	==, !=	Equality Comparison
	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
Lowest	|, ~|	OR, NOR
	?:	Conditional

AND has precedence over OR.

```
assign cout = g | p & cin;
```

- Subtraction involves a two's complement and addition.
- Multipliers and shifters use substantially more area (unless they involve easy constants).
- Division and modulus in hardware is so costly that it may not be synthesizable.
- Equality comparisons imply *N* XOR2s to determine equality of each bit and an AND*N* to combine all of the bits.
- Relative comparison involves a subtraction.

Numbers

- SystemVerilog supports 'b for binary (base 2), 'o for octal (base 8), 'd for decimal (base 10), and 'h for hexadecimal (base 16). If the base is omitted, the base defaults to decimal.
- If the size is not given, the number is assumed to have as many bits as the expression in which it is being used. Zeros are automatically padded on the front of the number to bring it up to full size.

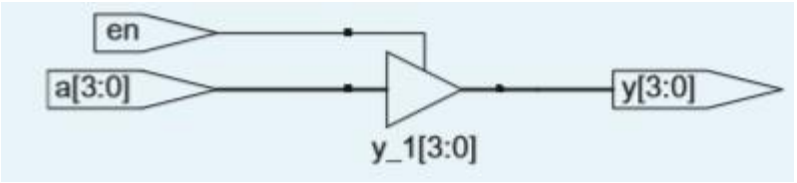
Numbers	Bits	Base	Val	Stored
3'b101	3	2	5	101
'b11	?	2	3	000...0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00...0101010
'1	?	n/a		11...111

- Good practice: Explicitly give the size of the bus
- Good practice: Break long numbers into more readable chunks using underscores (which are ignored).

Zs

- **Use z to indicate a floating value.**
  - z is particularly useful for describing a tristate buffer, whose output floats when the enable is 0.
  - A bus can be driven by several tristate buffers, exactly one of which should be enabled.
- **Must declare variable as a net - tri rather than logic.**
  - logic signals can only have a single driver.
  - Tristate busses can have multiple drivers, so they should be declared as a net.
- **Two types of nets – tri and trireg. Typically, exactly one driver on a net is active at a time, and the net takes on that value. If no driver is active, a tri floats (z), while a trireg retains the previous value.**
- **If no type is specified for an input or output, tri is assumed.**

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```



Xs

- **use x to indicate an invalid logic level.**
  - If a bus is simultaneously driven to 0 and 1 by two enabled tristate buffers (or other gates), the result is x, indicating contention.
  - If all the tristate buffers driving a bus are simultaneously OFF, the bus will float, indicated by z.
  - At the start of simulation, state nodes such as F/F outputs are initialized to an unknown state x. This is helpful to track errors caused by forgetting to reset a F/F before its output isused.
  - If a gate receives a floating input, it may produce an x output when it can't determine the correct output value.
  - Similarly, if a gate receives an illegal or uninitialized input, it may produce an x output.

SystemVerilog AND gate truth table with z and x

&		A			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

- **Seeing x in simulation? → Indication of a bug or bad coding practice.**
  - In the synthesized circuit, this corresponds to a floating gate input or uninitialized state. The x may randomly be interpreted by the circuit as 0 or 1, leading to unpredictable behavior..

Bit Swizzling

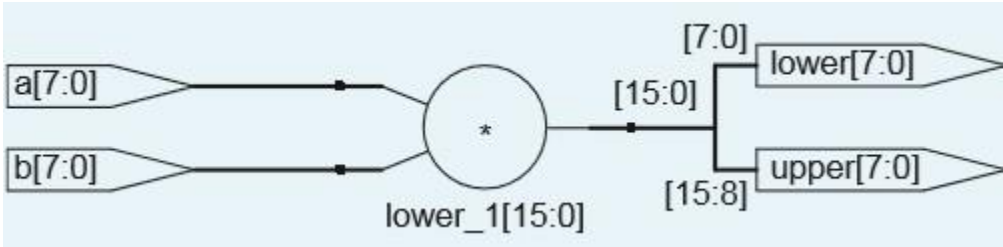
- **To operate on a subset of a bus or to concatenate, i.e., join together, signals to form busses.**
- **Use {} operator to concatenate busses.**

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

  - {3{d[0]}} indicates three copies of d[0].
  - Note that it was critical to specify the length of 3 bits in the constant; otherwise, it would have had an unknown number of leading zeros that might appear in the middle of y.
  - If y were wider than 9 bits, zeros would be placed in the MSBs.

- **Split an output into two pieces using bit swizzling**

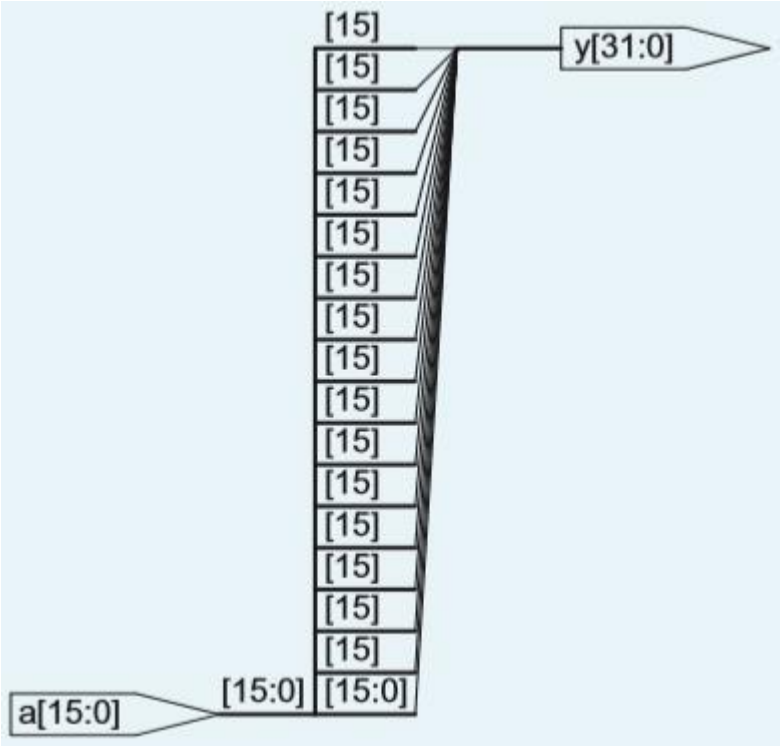
```
module mul(input  logic [7:0] a, b,
           output logic [7:0] upper, lower);

  assign {upper, lower} = a*b;
endmodule
```



Bit Swizzling – Sign Extension

- **Sign extend a 16b number to 32b by copying the MSB into the upper 16 positions.**

```
module signextend(input  logic [15:0] a,
                  output logic [31:0] y);

   assign y = {{16{a[15]}}, a[15:0]};
endmodule
```



Delays

- **Delays can be specified in arbitrary units.**
  - Helpful during simulation to predict how fast a circuit will work (if you specify meaningful delays)
  - Helpful for debugging purposes to understand cause and effect (deducing the source of a bad output is tricky if all signals change simultaneously in the simulation results).
- **Delays are ignored during synthesis**
  - the delay of a gate produced by the synthesizer depends on its tpd and tcd specifications, not on numbers in HDL code..
- **A # symbol is used to indicate the number of units of delay**

Delays

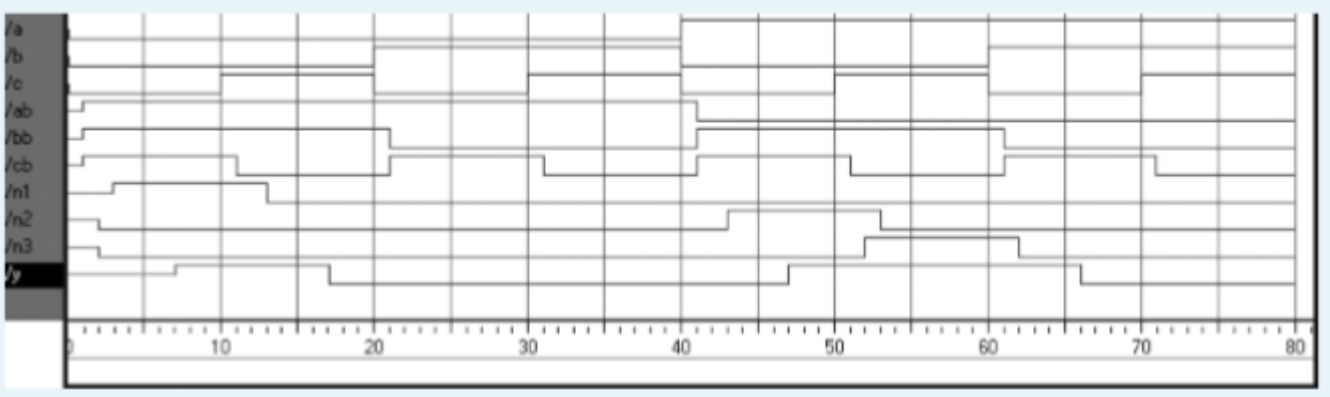
- SV files can include a timescale directive that indicates the value of each time unit.
- `timescale unit/step (1ns default unit for both if not specified)

```
`timescale 1ns/1ps

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
```

Each unit is 1ns & the simulation has 1 ps resolution.

Assumes inverters have a delay of 1 ns
AND3 gates have a delay of 2 ns

OR3 gates have a delay of 4 ns

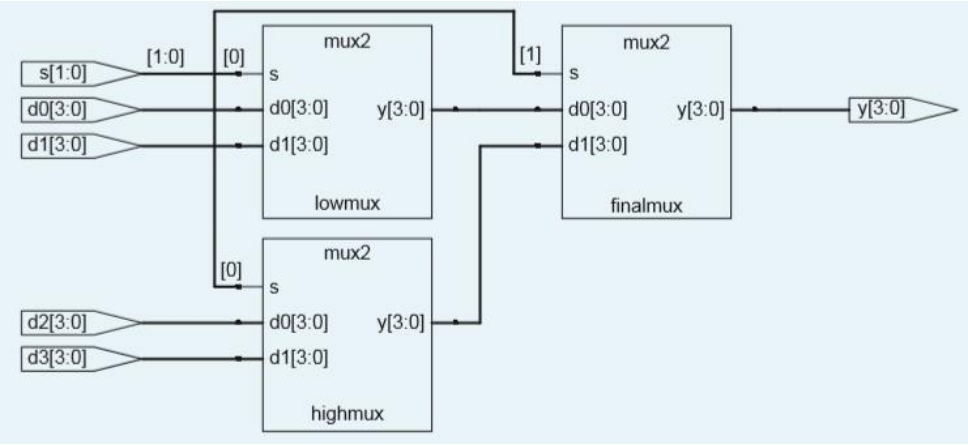


y lagging 7 ns after the inputs.
y is initially unknown at the beginning of the simulation.

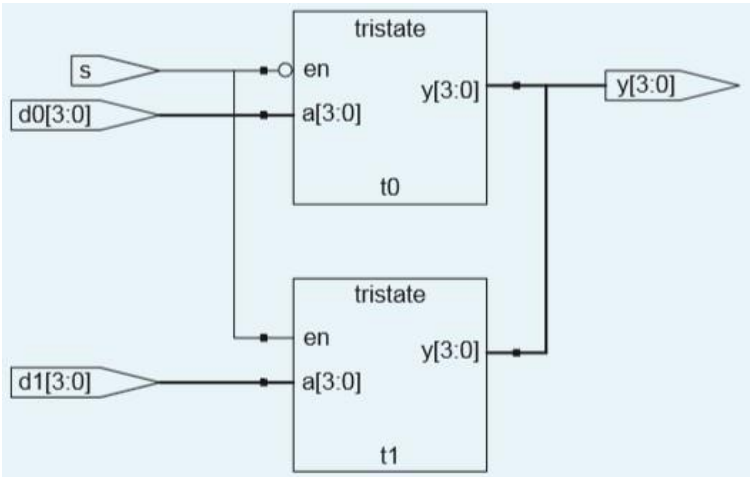
Structural Modeling

- **Behavioral modeling: describing a module in terms of the relationships between inputs and outputs.**
- **Structural modeling: describing a module in terms of how it is composed of simpler modules.**

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  logic [3:0] low, high;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 highmux(d2, d3, s[0], high);
  mux2 finalmux(low, high, s[1], y);
endmodule
```




```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output tri   [3:0] y);

  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

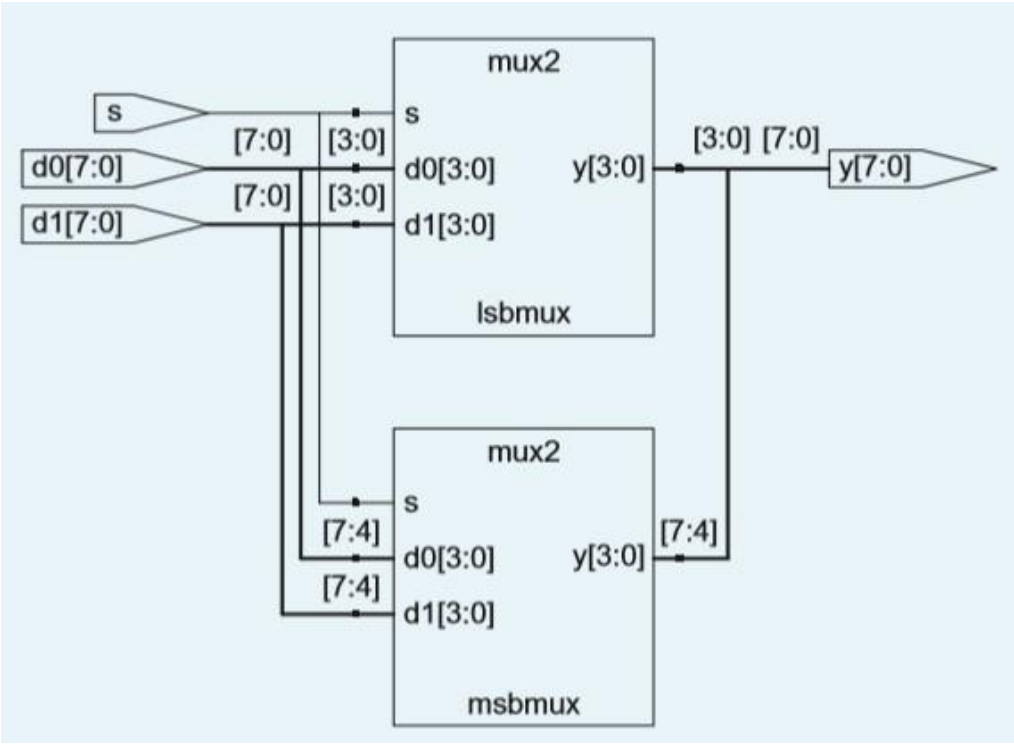


- Expressions such as ~s are permitted in the port list for an instance.
- Arbitrarily complicated expressions are legal, but discouraged because they make the code difficult to read.
- Note that y is declared as tri rather than logic because it has two drivers.

Accessing parts of Busses

- Modules can access parts of a bus

```
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
```



Sequential Logic

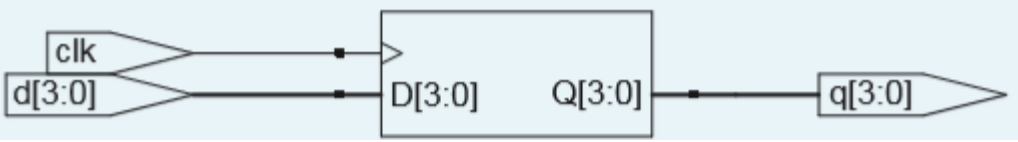
- **Outputs depend on the current inputs & previous inputs; sequential logic has memory.**
- **Outline**
  - Registers
  - Resettable registers
  - Enabled registers
  - Multiple registers
  - Latches
  - Counters
  - Shift registers

Registers

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

- The statement is executed only when the event specified in the sensitivity list occurs.
- Use always and *nonblocking assignment* to describe a register – +ve edge triggered D F/F

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

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



- In always statements, signals keep their old value (memory) until an event takes place that explicitly causes them to change.
- Do not use assign inside an always statement

Always

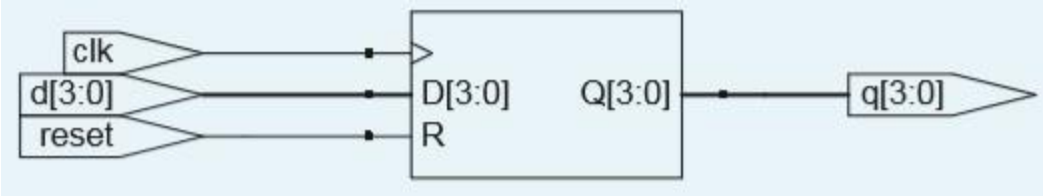
- **always statements can be used to imply F/F, latches, or combinational logic, depending on the sensitivity list and statement.**
  - Because of this flexibility, it is easy to produce the wrong hardware inadvertently.
- **SV introduces always_ff, always_latch, and always_comb to reduce the risk of common errors.**
  - always_ff behaves like always, but is used exclusively to imply flip-flops and allows tools to produce a warning if anything else is implied.

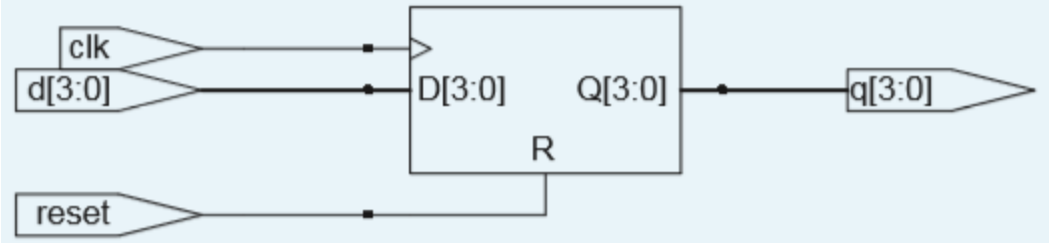
Resettable Registers

- When simulation begins or power is first applied to a circuit, the output of the flop is unknown (x)
- Should use resettable registers so that on power up you can put your system in a known state.
- Asynchronous reset occurs immediately.
- Synchronous reset occurs on the rising edge of the clock, takes fewer transistors and reduces the risk of timing problems on the trailing edge of reset.
  However, if clock gating is used, care must be taken that all F/Fs reset properly at startup.

Resettable Registers

```
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

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
```



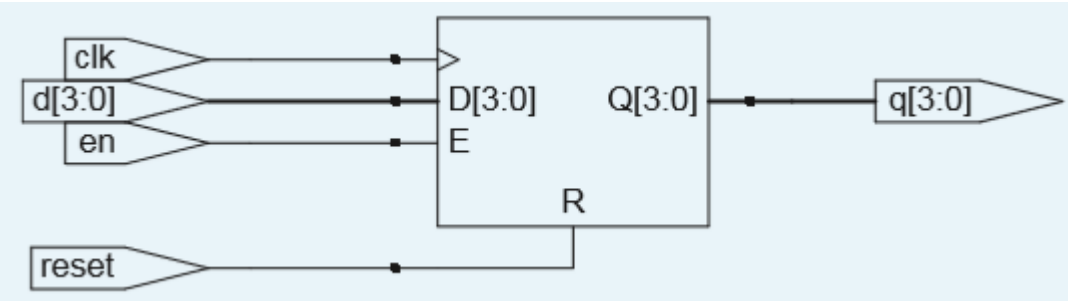


Multiple signals in an always statement sensitivity list are separated with a comma or the word or.

Enabled Registers

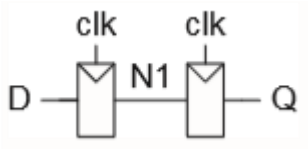
- Enabled registers only respond to the clock when the enable is asserted.
- A synchronously resettable enabled register retains its old value if both reset and en are FALSE.

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

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



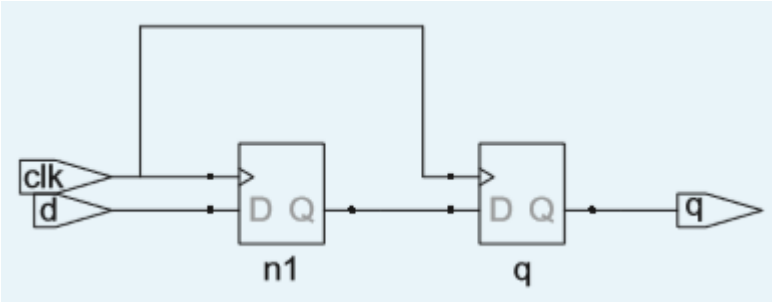
Multiple Registers

- **A single always / process statement can be used to describe multiple pieces of hardware.**
- **For e.g., consider describing a synchronizer made of two back-to-back F/Fs**



  - On the rising edge of clk, d is copied to n1. At the same time, n1 is copied to q.

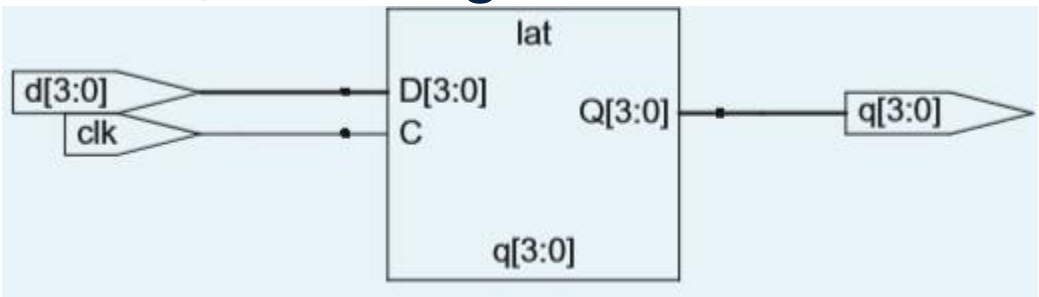
```
module sync(input  logic clk,
            input  logic d,
            output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
      n1 <= d;
      q <= n1;
    end
endmodule
```



Latches

- **A D latch is transparent when the clock is HIGH, allowing data to flow from input to output. The latch becomes opaque when the clock is LOW, retaining its old state.**

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

  always_latch
    if (clk) q <= d;
endmodule
```

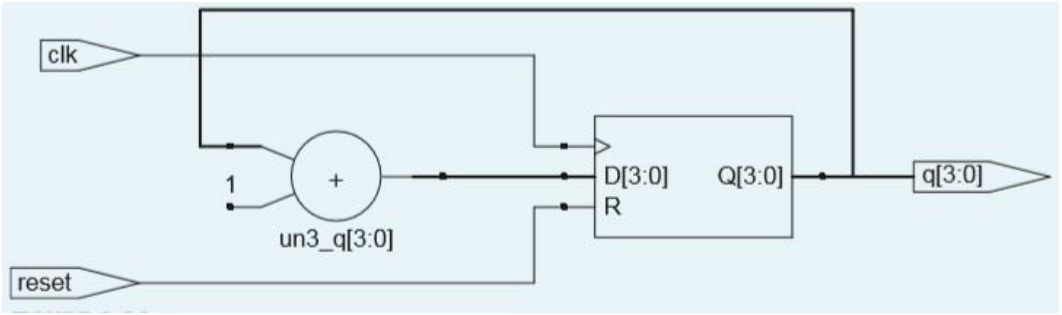


- **always_latch is equivalent to always @(clk, d) and is the preferred way of describing a latch in SV.**
  - It evaluates any time clk or d changes.
  - If clk is HIGH, d flows through to q, so this code describes a positive level sensitive latch. Otherwise, q keeps its old value.
- **SV can generate a warning if the always_latch block doesn't imply a latch.**
- **Avoid latches, use edge-triggered F/Fs instead.**

Counters

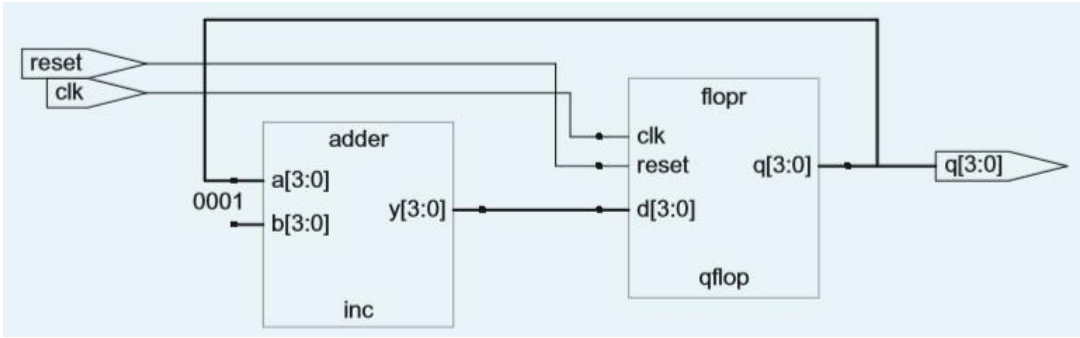
- **Behavioral style:**

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

  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= q+1;
endmodule
```



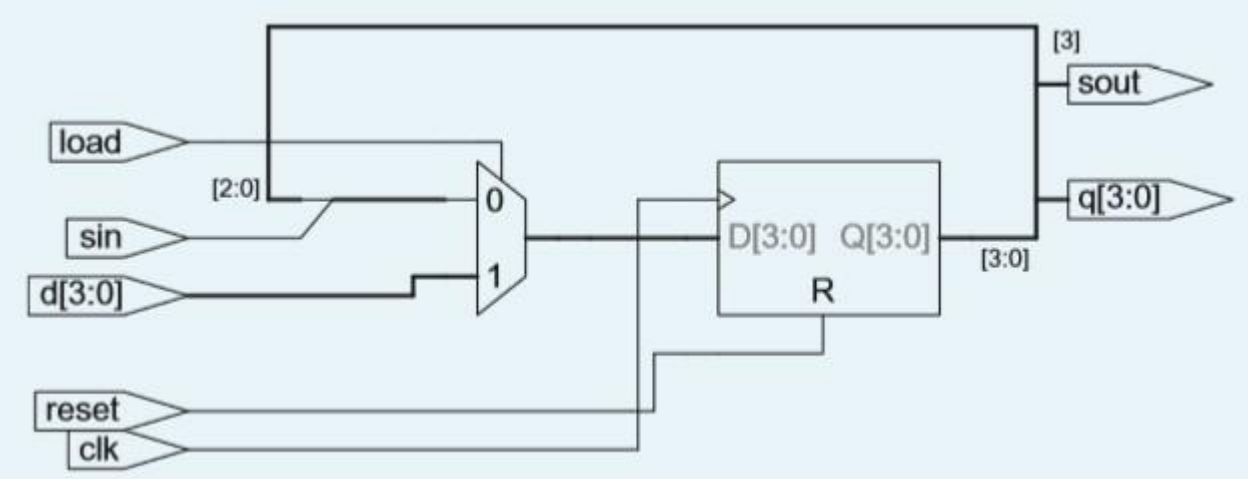
- **Structural style:**

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

   logic [3:0] nextq;

   flopr qflop(clk, reset, nextq, q);
   adder inc(q, 4'b0001, nextq);
endmodule
```



Shift Register with Parallel Load

```
module shiftreg(input  logic       clk,
                input  logic       reset, load,
                input  logic       sin,
                input  logic [3:0] d,
                output logic [3:0] q,
                output logic       sout);

  always_ff @(posedge clk)
    if (reset)     q <= 0;
    else if (load) q <= d;
    else           q <= {q[2:0], sin};

  assign sout = q[3];
endmodule
```



Combinational Logic with Always

- If possible, Use continuous assignments to model simple combinational logic

```
assign y = s ? d1 : d0;
```

- Use always_ff and always_latch for sequential circuits
- For modeling more complicated combinational logic, always statement can be help.
- But the sensitivity list must be written to respond to changes in *all* of the inputs and the body prescribes the output value for *every* possible input combination.
- Otherwise, may inadvertently imply sequential logic if the sensitivity list leaves out inputs!
- Preferred: use always_comb if you have to use always for combinational logic

always_comb

- always_comb is equivalent to always @(*).
- always_comb reevaluates the statements inside the always statement *any* time *any* of the signals on the RHS of <= or = inside the always statement change.
- always_comb is a safe way to model combinational logic.

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  always_comb
    y = ~a;
endmodule
```

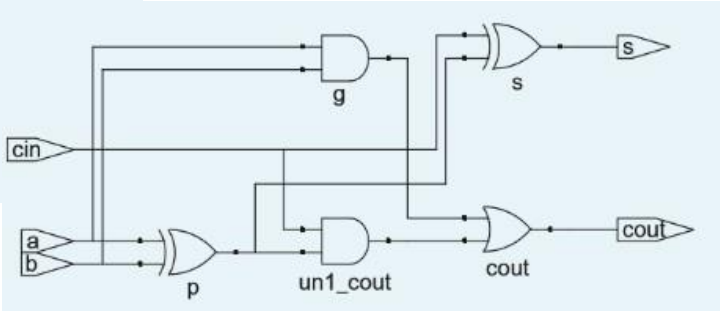
- In this particular example, always @(a) would also have sufficed.
- Use more lines of code than the equivalent approach with assign statements ☹

always_comb & blocking assignment

- = when used inside an always is blocking

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  always_comb
    begin
      p = a ^ b;   // blocking
      g = a & b;   // blocking

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

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

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

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



- always @(a, b, cin) or always @(*) would have been equivalent to always_comb.
- However, always_comb is preferred because it is succinct and allows SVtools to generate a warning if the block inadvertently describes sequential logic.
- The begin / end construct is necessary because multiple statements appear in the always statement. This is analogous to { } in C or Java.
- Use more lines of code than the equivalent approach with assign statements ☹

- **Use blocking (=) assignment with always_comb**

always_comb & blocking assignment

- **Use blocking (=) assignment with always_comb**

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  always_comb
    begin
      p = a ^ b;  // blocking
      g = a & b;  // blocking

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

Imagine that a, b, and cin are all initially 0. → p, g, s, and cout are thus 0 as well.

At some time, a changes to 1, triggering the always statement. The four blocking assignments evaluate in the order shown below.

1. $p \leftarrow 1 \oplus 0 = 1$
2. $g \leftarrow 1 \cdot 0 = 0$
3. $s \leftarrow 1 \oplus 0 = 1$
4. $cout \leftarrow 0 + 1 \cdot 0 = 0$

- **Blocking Assignments – "blocks concurrency"**
- **Assignments evaluated sequentially after activation**
  - s and cout use the new value of p and g

always_comb & non-blocking assignment

- **Don't use non-blocking (<=) assignment w/ always_comb**

```
// nonblocking assignments (not recommended)
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  always_comb
    begin
      p <= a ^ b; // nonblocking
      g <= a & b; // nonblocking

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

Imagine that a, b, and cin are all initially 0. → p, g, s, and cout are thus 0 as well.

At some time, a changes to 1, triggering the always statement. The four blocking assignments evaluate concurrently:

$$p \leftarrow 1 \oplus 0 = 1 \quad g \leftarrow 1 \cdot 0 = 0 \quad \boxed{s \leftarrow 0 \oplus 0 = 0} \quad cout \leftarrow 0 + 0 \cdot 0 = 0$$

However, p does change from 0 to 1. This change triggers the always statement to evaluate a 2nd time as follows:

$$p \leftarrow 1 \oplus 0 = 1 \quad g \leftarrow 1 \cdot 0 = 0 \quad s \leftarrow 1 \oplus 0 = 1 \quad cout \leftarrow 0 + 1 \cdot 0 = 0$$

- The nonblocking assignments eventually reached the right answer, but the always statement had to evaluate twice.
- This makes simulation more time consuming, although it synthesizes to the same hardware.

- **Non-Blocking Assignments – acts concurrently**
- **Assignments evaluated concurrently after activation**
  - s and cout use the old value of p and g

always_comb & non-blocking assignment

- **Don't use non-blocking (<=) assignment w/ always_comb**

```
// nonblocking assignments (not recommended)
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  always @(a, b, cin)
    begin
      p <= a ^ b; // nonblocking
      g <= a & b; // nonblocking

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

Imagine that a, b, and cin are all initially 0. → p, g, s, and cout are thus 0 as well.

At some time, a changes to 1, triggering the always statement. The four blocking assignments evaluate concurrently:

$$p \leftarrow 1 \oplus 0 = 1 \quad g \leftarrow 1 \cdot 0 = 0 \quad \boxed{s \leftarrow 0 \oplus 0 = 0} \quad cout \leftarrow 0 + 0 \cdot 0 = 0$$

However, the statement would not reevaluate when p or g change now ☹ .

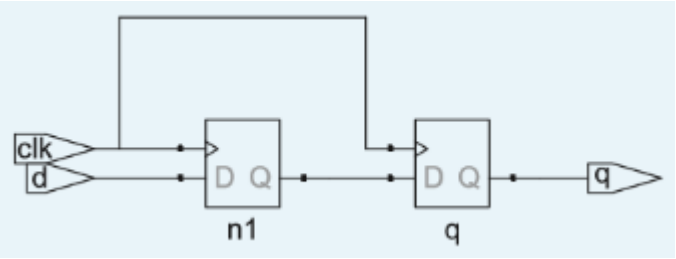
- Nonblocking assignments in modeling combinational logic can produce the wrong result if you forget to include the intermediate variables in the sensitivity list.

- **Non-Blocking Assignments – acts concurrently**
- **Assignments evaluated concurrently after activation**
  - s and cout use the old value of p and g

always_ff & non-blocking assignment

- **Use non-blocking (<=) assignment with sequential logic like always_ff**

```
module sync(input  logic clk,
            input  logic d,
            output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
      n1 <= d;
      q <= n1;
    end
endmodule
```



Imagine initially that d = 0, n1 = 1, and q = 0.

On the rising edge of the clock, the following two assignments occur concurrently, so that after the clock edge, n1 = 0 and q = 1.

$$n1 \leftarrow d = 0 \quad q \leftarrow n1 = 1$$

- **Non-Blocking Assignments – acts concurrently**
- **Assignments evaluated concurrently after activation**
  - q uses the old value of n1

always_ff & blocking assignment

- **Don't use blocking (=) assignment with sequential logic like always_ff**

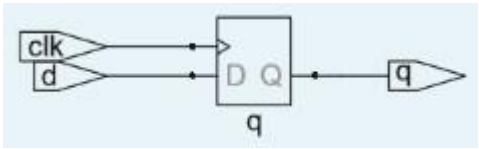
```
// Bad implementation using blocking assignments

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
```

Imagine initially that d = 0, n1 = 1, and q = 0.

On the rising edge of the clock, d is copied to n1.
This new value of n1 is then copied to q, resulting in d improperly appearing at both n1 and q.

1. $n1 \leftarrow d = 0$
2. $q \leftarrow n1 = 0$

Because n1 is invisible to the outside world and does not influence the behavior of q, the synthesizer optimizes it away entirely!



- **Blocking Assignments – 'blocks concurrency"**
- **Assignments evaluated sequentially after activation**
  - q uses the new value of n1

Good & Bad HDL Practices

- In an always statement, = indicates a blocking assignment and <= indicates a nonblocking assignment.
- Use blocking (=) assignment with always_comb
- Blocking Assignments – 'blocks concurrency" and evaluated sequentially
- Use non-blocking (<=) assignment with sequential logic like always_ff
- Non-Blocking Assignments – acts concurrently
- Do not confuse either type with continuous assignment using the assign statement.
- assign statements are normally used outside always statements and are also evaluated concurrently.

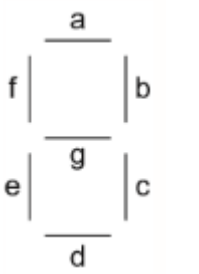
Case & If Statements

- case and if statements are convenient for modeling more complicated combinational logic.
- case and if statements can only appear within always statements.
- case/if statement implies combinational logic if all possible input combinations are considered;
- Otherwise it implies sequential logic because the output will keep its old value in the undefined cases!

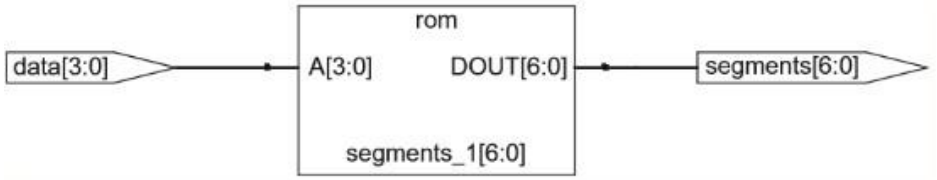
Case Statement Example

- The decoder for a 7-segment display takes a 4-bit number and displays its decimal value on the segments. For example, the number 0111 = 7 should turn on segments a, b, and c.

```
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_1011;
      default: segments = 7'b000_0000;
    endcase
endmodule
```



7-segment display



The default clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

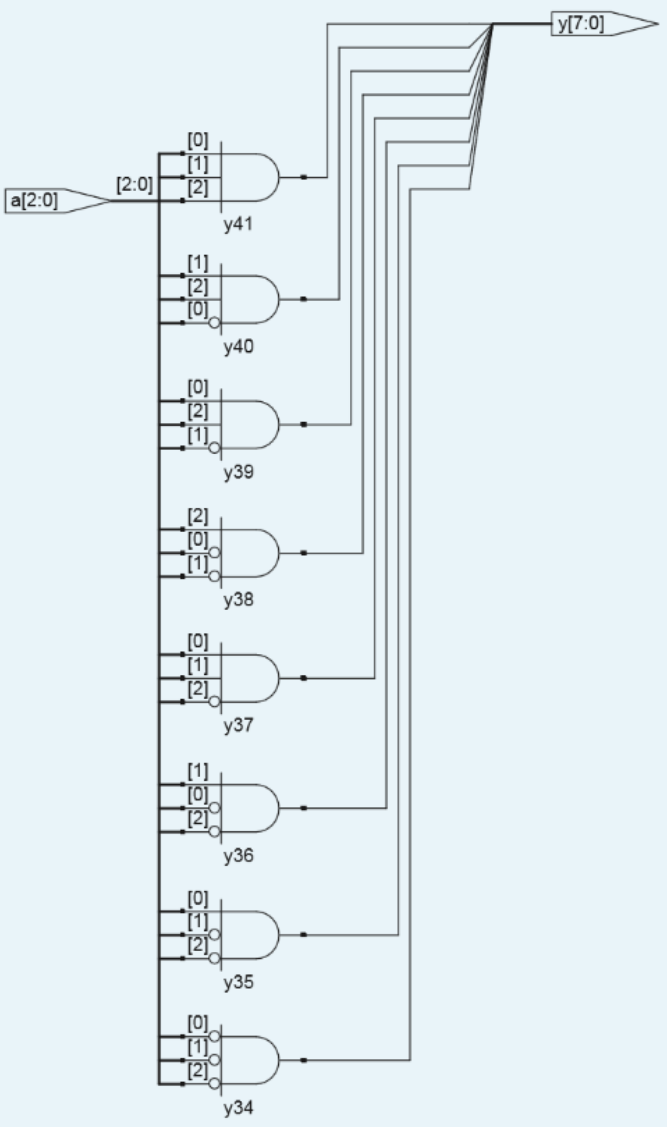
If the default or others clause were left out of the case statement, the decoder would have remembered its previous output whenever data were in the range of 10–15. This is strange behavior for hardware, and is not combinational logic.

Case Statement Example

- 3:8 decoder

```
module decoder3_8(input  logic [2:0] a,
                  output logic [7:0] y);

  always_comb
    case (a)
      3'b000: y = 8'b00000001;
      3'b001: y = 8'b00000010;
      3'b010: y = 8'b00000100;
      3'b011: y = 8'b00001000;
      3'b100: y = 8'b00010000;
      3'b101: y = 8'b00100000;
      3'b110: y = 8'b01000000;
      3'b111: y = 8'b10000000;
    endcase
endmodule
```

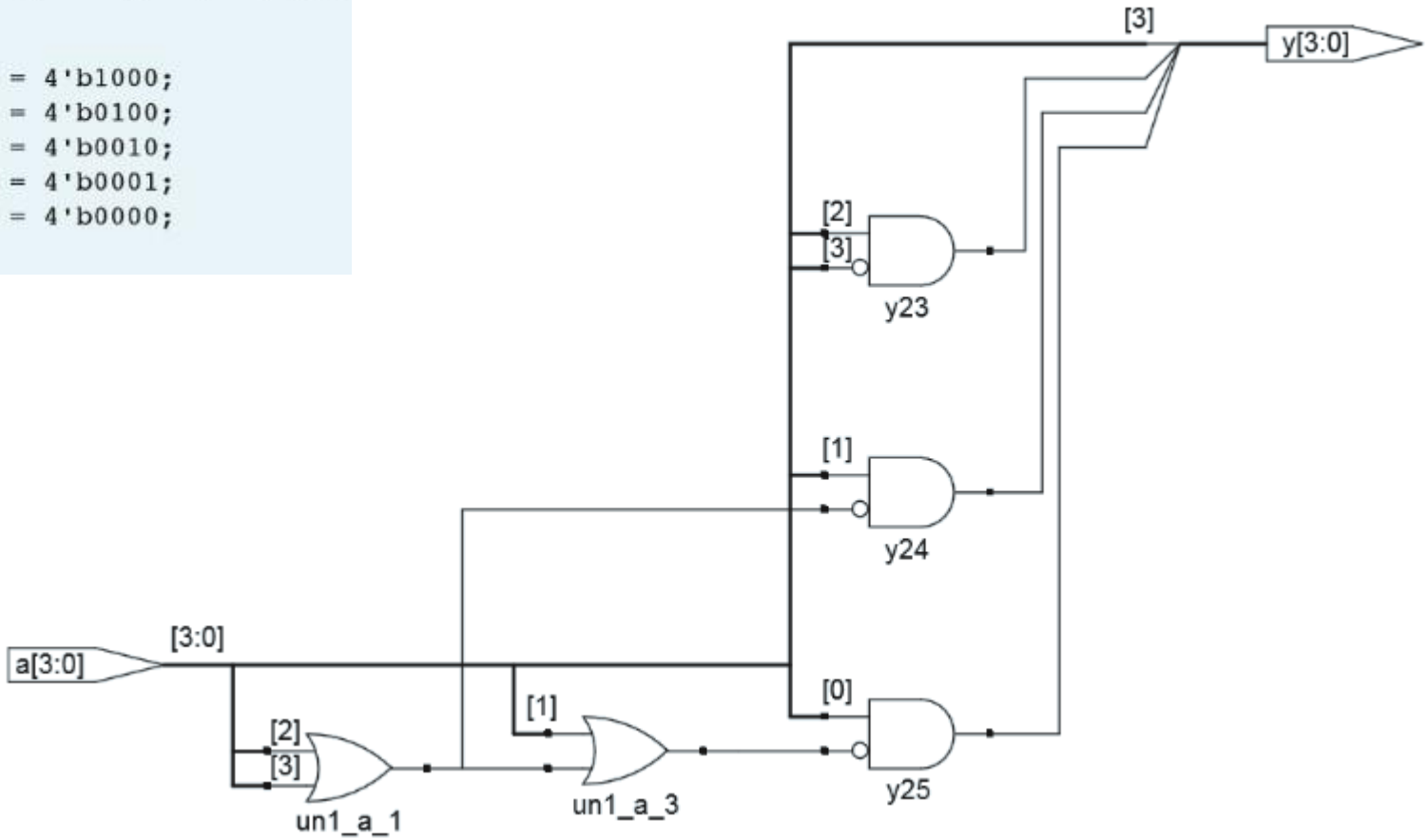
No default statement is needed because all cases are covered.



If Statement Example

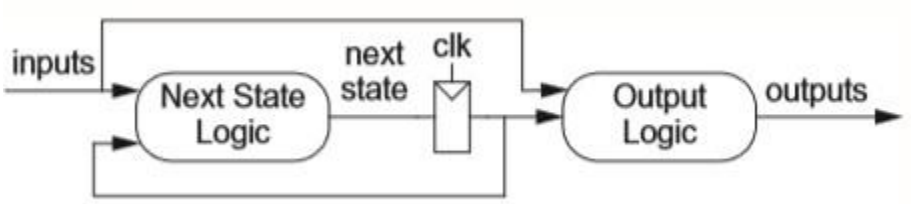
- A 4b priority circuit that sets one output TRUE corresponding to the most significant input that is TRUE

```
module priorityckt(input  logic [3:0] a,
                   output logic [3:0] y);

  always_comb
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
    else           y = 4'b0000;
endmodule
```

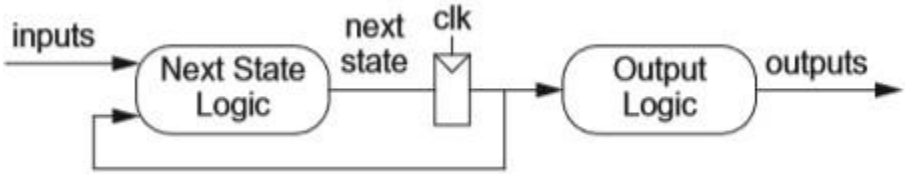


Finite State Machines

- **Mealy machines: o/p is a function of the current state and i/ps.**



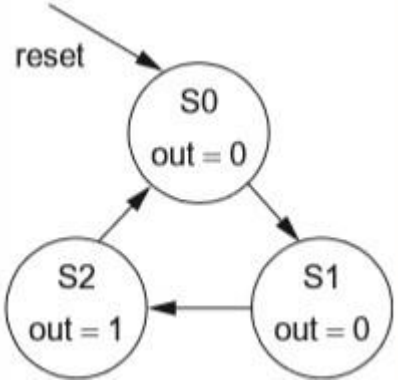
- **Moore machines: o/p is a function of the current state only.**



- **Just like the FSM, the FSM can be partitioned in HDL into a state register, next state logic, and o/p logic.**
  - Implement state register as F/F, and next stage logic as well as o/p logic as combinational.

FSM Example: Divide-by-3 Counter

- **Moore machine: o/p is a function of the current state only**
  → FSM has no inputs, only a clock & reset




```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);

  logic [1:0] state, nextstate;

  // State Register
  always_ff @(posedge clk)
    if (reset) state <= 2'b00;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      2'b00: nextstate = 2'b01;
      2'b01: nextstate = 2'b10;
      2'b10: nextstate = 2'b00;
      default: nextstate = 2'b00;
    endcase
// Output Logic
  assign y = (state == 2'b00);
endmodule
```

A case statement is used to define the state transition table.

Because the next state logic should be combinational, a default is necessary even though the state 11 should never arise.

Each always statement implies a separate block of logic. Therefore, a given signal can be assigned in only one always. Otherwise, two pieces of hardware with shorted outputs will be implied.

State Enumeration - typedef

- Name the states names using the enumeration type rather than referring to them by binary values.
- This makes the code more readable and easier to change

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);

  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;

  // State Register
  always_ff @(posedge clk)
    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 y = (state == S0);
endmodule
```

typedef statement defines statetype to be a 2bit logic value with one of three possibilities: S0, S1, or S2.

state and nextstate are statetype signals.

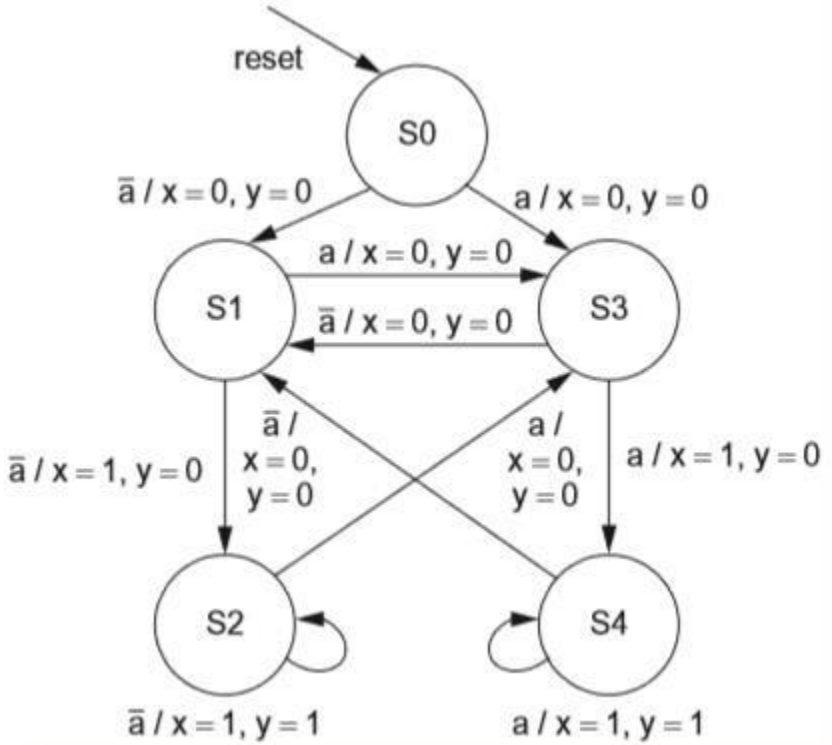
The enumerated encodings default to numerical order:
S0 = 00, S1 = 01, and S2 = 10.
The encodings can be explicitly set by the user too.

```
typedef enum logic [2:0] {S0 = 3'b001,
                          S1 = 3'b010,
                          S2 = 3'b100} statetype;
```

FSM Example: With inputs

- An FSM with an input a and 2 outputs
- Output x is true when the input is the same now as it was last cycle.
- Output y is true when the input is the same now as it was for the past two cycles.
- Mealy machine: o/p is a function of the current state and i/ps

```
module historyFSM(input  logic clk,
                  input  logic reset,
                  input  logic a,
                  output logic x, y);

  typedef enum logic [2:0]
    {S0, S1, S2, S3, S4} statetype;
  statetype state, nextstate;

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

  // Next State Logic
  always_comb
    case (state)
      S0: if (a) nextstate = S3;
          else   nextstate = S1;
      S1: if (a) nextstate = S3;
          else   nextstate = S2;
      S2: if (a) nextstate = S3;
          else   nextstate = S2;
      S3: if (a) nextstate = S4;
          else   nextstate = S1;
      S4: if (a) nextstate = S4;
          else   nextstate = S1;
      default:   nextstate = S0;
    endcase

// Output Logic
assign x = ((state == S1 | state == S2) & ~a) |
           ((state == S3 | state == S4) & a);
assign y = (state == S2 & ~a) | (state == S4 & a);
endmodule
```



System Verilog Testbenches

- A testbench is an HDL module used to test another module, called the device under test (DUT).
- The testbench contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced.
- The input and desired output patterns are called test vectors.
- Testbenches are not synthesizable.

Testbench Example – Exhaustive, Verification by inspection

- Ok for small modules, for tedious and error-prone.

```
module sillyfunction(input  logic a, b, c,
                     output logic y);

  assign y = ~a & ~b & ~c |
              a & ~b & ~c |
              a & ~b &  c;
endmodule
```

```
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
```

The initial statement executes the statements in its body at the start of simulation.

First applies the input pattern 000 and waits for 10 time units.

Then applies 001 and waits 10 more units, and so forth until all eight possible inputs have been applied.

Testbench Example – Exhaustive, Self-Checking

- **Ok for small modules, but writing code for each test vector also becomes tedious for modules that require a large number of vectors.**

```
module testbench2();
  logic a, b, c;
  logic y;

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

  // apply inputs one at a time
  // checking results

  initial begin
    a = 0; b = 0; c = 0; #10;
    assert (y === 1) else $error("000 failed.");
    c = 1;               #10;
    assert (y === 0) else $error("001 failed.");
    b = 1; c = 0;        #10;
    assert (y === 0) else $error("010 failed.");
    c = 1;               #10;
    assert (y === 0) else $error("011 failed.");
    a = 1; b = 0; c = 0; #10;
    assert (y === 1) else $error("100 failed.");
    c = 1;               #10;
    assert (y === 1) else $error("101 failed.");
    b = 1; c = 0;        #10;
    assert (y === 0) else $error("110 failed.");
    c = 1;               #10;
    assert (y === 0) else $error("111 failed.");
  end
endmodule
```

Assert checks if a specified condition is true. If it is not, it executes the else statement. $error system task prints an error message describing the assertion failure.

Assert is ignored during synthesis.

- **In SV, comparison using == or != spuriously indicates equality if one of the operands is x or z.**
- **The === and !== operators must be used instead for testbenches because they work correctly with x and z.**

Testbench Example – w/ Test Vector File

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

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

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // 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

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {a, b, c, yexpected} =
            testvectors[vectornum];
    end
// 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
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 'bx) begin
        $display("%d tests completed with %d
                  errors", vectornum, errors);
        $finish;
      end
    end
endmodule
```

- **The testbench simply reads the test vectors, applies the i/p test vector, waits, checks that the o/p values match the o/p vector, and repeats until it reaches the end of the file.**

The testbench generates a clock using an always / process statement with no stimulus list so that it is continuously reevaluated.

$readmemb reads a file of binary numbers into an array.
$readmemh is similar, but it reads a file of hexadecimal numbers.
example.tv is a text file containing the inputs and expected output written in binary:

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

This block waits 1 time unit after the rising edge of the clock (to avoid any confusion of clock and data changing simultaneously), then sets the 3 inputs and the expected output based on the 4 bits in the current test vector.

$display is a system task to print in the simulator window.

$finish terminates the simulation.

At the end of the simulation, the testbench prints the total number of test vectors applied and the number of errors detected.