Digital Logic Design

Lecture 8

More Verilog

Verilog is not a computer program.

It is a shorthand for describing digital hardware.

Think of your system in terms of blocks of combinational logic, registers, and finite state machines.

Sketch these blocks on paper and show how they are connected before you start writing code.

HDL assignment statements (`assign` in Verilog) take place concurrently.

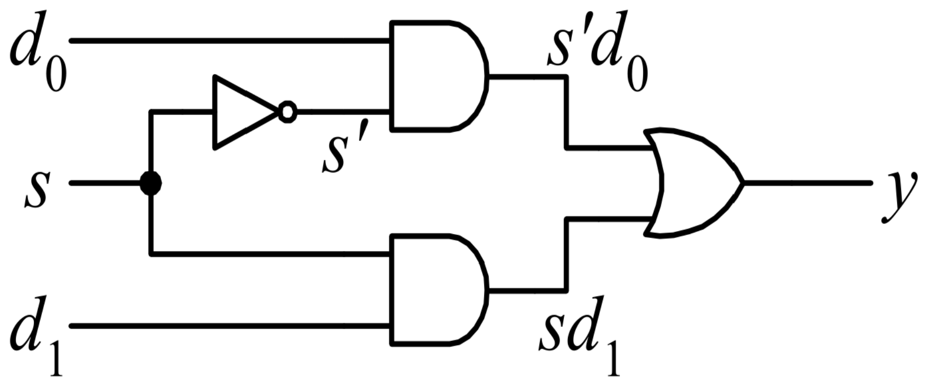
This is different from conventional programming languages such as C or Java, in which statements are evaluated in the order in which they are written.

Implicit net declarations

```
module mux2 (input d0, d1, s,
             output y);
    not #(1) (s1,s);
    and #(1) (s1d0,d0,s1);
    and #(1) (s1d1,d1,s);
    or  #(1) (y,s1d0, s1d1);

endmodule
```

A **net** represents a node in a circuit.



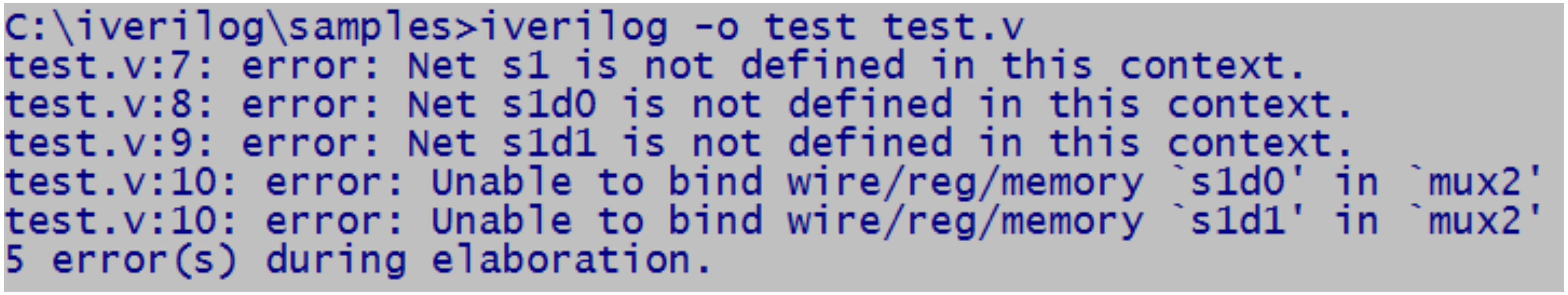
If an undeclared identifier is used on the left-hand side of a continuous assignment, then an implicit net declaration is inferred, and no error or warning is reported.

If an undeclared identifier is used as a connection to an instance of a module, interface, program, or primitive, then an implicit net is inferred, and no error or warning is reported.

```
`default_nettype none // turn off implicit data types
module mux2 (input d0, d1, s,
             output y);
    not #(1) U1(s1,s);
    and #(1) U2(s1d0,d0,s1);
    and #(1) U3(s1d1,d1,s);
    or  #(1) U4(y,s1d0, s1d1);

endmodule

// turn implicit nets on again to avoid side-effects
`default_nettype wire
```



```
C:\iverilog\samples>iverilog -o test test.v
test.v:7: error: Net s1 is not defined in this context.
test.v:8: error: Net s1d0 is not defined in this context.
test.v:9: error: Net s1d1 is not defined in this context.
test.v:10: error: Unable to bind wire/reg/memory `s1d0' in `mux2'
test.v:10: error: Unable to bind wire/reg/memory `s1d1' in `mux2'
5 error(s) during elaboration.
```

```
/* Verilog comments
   can span
   across multiple lines
*/

// This is a single-line comment
```

Verilog is case-sensitive: `y1` and `Y1` are different signals.

White space characters, such as SPACE and TAB, and blank lines are ignored by the Verilog compiler.

Multiple statements can be written on a single line:

```
f0 = w0; f1 = w1;
```

The name of a net or variable must not begin with a digit and it should not be a Verilog keyword.

The name of a net or variable can contain any letter or digit as well as the _ underscore and $ characters.

A signal can have four possible values: `1, 0, z, x`

The `z` and `x` values can also be denoted by the capital letters `Z` and `X`.

`z` – floating, high impedance value

`x` – unknown or illegal value

The value `x` can be used to denote a don’t-care condition; the symbol `?` can also be used for this purpose.

A `parameter` associates an identifier name with a constant.

```
parameter n = 4;
parameter S0 = 2’b00, S1 = 2’b01, S2 = 2’b10;
```

Variables

Nets provide a means for interconnecting logic elements, but they do not allow a circuit to be described in terms of its **behavior**.

For this purpose, Verilog provides **variables**.

There are two types of variables, `reg` and `integer`

```
Count = 0;
for (k = 0; k < 4; k = k + 1)
  if (S[k])
    Count = Count + 1;
```

This code stores in `Count` the number of bits in `S` that have the value 1.

Since it models the behavior of a circuit, `Count` has to be declared as a variable, rather than a `wire`.

```
Count = 0;
for (k = 0; k < 4; k = k + 1)
  if (S[k])
    Count = Count + 1;
```

If `Count` has three bits, then the declaration is

```
     reg [2:0] Count;
```

The keyword `reg` does not denote a storage element, or register.

reg variables can be used to model either combinational or sequential parts of a circuit.

In our example, the variable `k` serves as a loop index.

Such variables are declared as type `integer`

```
     integer k;
```

Integer variables do not directly correspond to nodes in a circuit.

Reduction operators

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

    assign y = &a;

endmodule
```

`&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];
```

`|` , `^` , `~&` , `~|` reduction operators are available for OR, XOR, NAND, and NOR as well.

Bit swizzling

The `{}` operator is used to concatenate busses.

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

`{3{d[0]}}` – three copies of `d[0]`

y is given the 9-bit value $c_2c_1d_0d_0d_0c_0101$

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

Accessing parts of busses

```
module mux2_8 (input [7:0] d0, d1,
               input s,
               output [7:0] y);
    mux2 lsbmux (d0[3:0], d1[3:0], s, y[3:0]);
    mux2 msbmux (d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

An 8-bit wide 2:1 multiplexer is built using two of the 4-bit 2:1 multiplexers already defined, operating on the low and high nibbles of the byte.

Category	**Examples**	**Bit Length**
Bitwise	$\sim A$, $+A$, $-A$	L(A)
	A & B, $A \mid B$, $A \sim\wedge B$, $A \wedge\sim B$	MAX (L(A), L(B))
Logical	$!A$, $A \&\& B$, $A \parallel B$	1 bit
Reduction	$\&A$, $\sim\&A$, $\mid A$, $\sim\mid A$, $\wedge\sim A$, $\sim\wedge A$	1 bit
Relational	$A == B$, $A != B$, $A > B$, $A < B$ $A >= B$, $A <= B$ $A === B$, $A !== B$	1 bit
Arithmetic	$A + B$, $A - B$, $A * B$, A/B $A \% B$	MAX (L(A), L(B))
Shift	$A << B$, $A >> B$	L(A)
Concatenate	$\{A, \ldots, B\}$	L(A) $+ \cdots +$ L(B)
Replication	$\{B\{A\}\}$	$B *$ L(A)
Condition	A ? B : C	MAX (L(B), L(C))

The modulus operator (%) is usually not supported for synthesis, except for use in calculating a compile-time constant.

Arithmetic operands of type wire and reg are treated as unsigned numbers.

If the two operands are of unequal size, zero digits are padded on the left, and bits are truncated if the result has fewer digits than the largest operand.

Integer variables are considered as 2’s complement numbers.

The << and >> operators perform logical shifts to the left with 0 and right with 0, respectively.

For synthesis, the operand B should be a constant.

We used `assign` statements to describe combinational logic behaviorally.

The assignments contaning the `assign` keyword are called **continuous assignments**.

Verilog also provides **procedural statements** (also called **sequential statements**) contained inside an always block

Verilog `always` statements are used to describe sequential circuits, because they remember the old state when no new state is prescribed.

However, `always` statements can also be used to describe combinational logic behaviorally if the sensitivity list is written to respond to changes in all of the inputs and the body prescribes the output value for every possible input combination.

```
module inv (input      [3:0] a,
            output reg [3:0] y);
    always @ (*)
        y = ~a;
endmodule
```

`always @ (*)` reevaluates the statements inside the `always` statement any time any of the signals on the right hand side of <= or = inside the always statement change.

`@ (*)` can be used to model combinational logic.

In this example, `@ (a)` would also have sufficed.

The = in the `always` statement is called a **blocking assignment**

<= is called a **nonblocking assignment**

In Verilog, it is good practice to use blocking assignments for combinational logic and nonblocking assignments for sequential logic.

Verilog supports blocking and nonblocking assignments in an `always` statement.

A group of blocking assignments are evaluated in the order in which they appear in the code.

A group of nonblocking assignments are evaluated concurrently; all of the statements are evaluated before any of the signals on the left hand sides are updated.

```
module fulladder (input a, b, cin,
                  output reg s, cout);
    reg p, g;
    always @ (*)
      begin
        p = a ^ b; // blocking
        g = a & b; // blocking
        s = p ^ cin; // blocking
        cout = g | (p & cin); // blocking
      end
endmodule
```

This example uses blocking assignments, first computing p, then g, then s, and finally cout.

Because p and g appear on the left hand side of an assignment in an always statement, they must be declared to be reg

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

In Verilog, `case` must be inside `always` statements.

```
module decoder3_8 (input [2:0] a,
                   output reg [7:0] y);
    always @ (*)
      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.

```
module priority (input [3:0] a,
       output reg [3:0] y);
    always @ (*)
      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
```

In Verilog, `if` statements must appear inside of `always` statements.

```
module priority_casez(input [3:0] a,
                      output reg [3:0] y);
    always @ (*)
      casez (a)
        4'b1???: y = 4'b1000;
        4'b01??: y = 4'b0100;
        4'b001?: y = 4'b0010;
        4'b0001: y = 4'b0001;
        default: y = 4'b0000;
      endcase
endmodule
```

`casez` describes truth tables with don’t cares (indicated with ? in the `casez` statement).

A `case` or `if` statement imply combinational logic if all possible input combinations are defined.

Otherwise they imply sequential logic, because the output will keep its old value in the undefined cases.

```
if (condition) begin
  //statements
end else begin
  //statements
end
```

```
while (condition)
begin
  //statements
end
```

For loops in Verilog are almost exactly like for loops in C

The `++` and `--` operators are not supported in Verilog

```
for (i = 0; i < 16; i = i + 1) begin
  $display ("Current value of i is %d", i);
end
```

Make sure your code is actually sanely implementable in hardware, and that your loop is not infinite.

Repeat is similar to the for loop:

```
repeat (16) begin
  $display ("Current value of i is %d", i);
  i = i + 1;
end
```

Blocking and nonblocking assignment guidelines

1. Use `always @ (posedge clk)` and nonblocking assignments to model synchronous sequential logic.

2. Use continuous assignments to model simple combinational logic.

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

3. Use `always @ (*)` and blocking assignments to model more complicated combinational logic where the `always` statement is helpful.

4. Do not make assignments to the same signal in more than one `always` statement or continuous assignment statement.

```
// nonblocking assignments (not recommended)
module fulladder (input a, b, cin,
                  output reg s, cout);
    reg p, g;
    always @ (*)
      begin
        p <= a ^ b; // nonblocking
        g <= a & b; // nonblocking
        s <= p ^ cin;
        cout <= g | (p & cin);
      end
endmodule
```

Because `p` and `g` appear on the left hand side of an assignment in an `always` statement, they must be declared to be `reg`

Let a, b, cin are all initially 0.

p, g, s, 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 here.

1. $\texttt{p} \leftarrow 1 \oplus 0 = 1$
2. $\texttt{g} \leftarrow 1 \bullet 0 = 0$
3. $\texttt{s} \leftarrow 1 \oplus 0 = 1$
4. $\texttt{cout} \leftarrow 0 + 1 \bullet 0 = 0$

Note that p and g get their new values before s and cout are computed because of the blocking assignments.

Now consider the same case of a rising from 0 to 1 while b and cin are 0, and all assignments are nonblocking, so they evaluate concurrently:

$$\mathtt{p} \leftarrow 1 \oplus 0 = 1 \quad \mathtt{g} \leftarrow 1 \bullet 0 = 0 \quad \mathtt{s} \leftarrow 0 \oplus 0 = 0 \quad \mathtt{cout} \leftarrow 0 + 0 \bullet 0 = 0$$

s is computed concurrently with p and hence uses the old value of p, not the new value

Therefore, s remains 0 rather than becoming 1.

However, p does change from 0 to 1.

This change triggers the always statement to evaluate a second time, as follows:

$$\mathtt{p} \leftarrow 1 \oplus 0 = 1 \quad \mathtt{g} \leftarrow 1 \bullet 0 = 0 \quad \mathtt{s} \leftarrow 1 \oplus 0 = 1 \quad \mathtt{cout} \leftarrow 0 + 1 \bullet 0 = 0$$

This time, p is already 1, so s correctly changes to 1.

The nonblocking assignments eventually reach the right answer, but the always statement had to evaluate twice.

This makes simulation slower, though it synthesizes to the same hardware.

Another drawback of nonblocking assignments in modeling combinational logic is that Verilog will produce the wrong result if you forget to include the intermediate variables in the sensitivity list.

If the sensitivity list of the always statement were written as `always @ (a, b, cin)` rather than `always @ (*)` then the statement would not reevaluate when p or g changes.

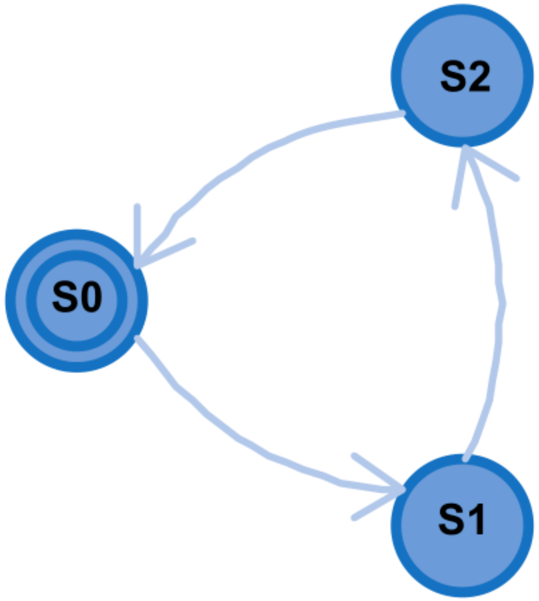
Thus, s would be incorrectly left at 0, not 1.

Finite state machines (FSM) in Verilog

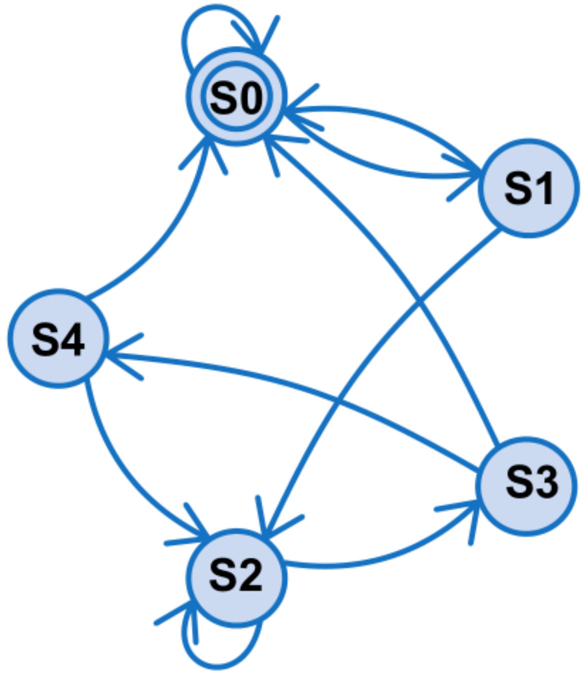
HDL descriptions of state machines are divided into three parts to model the state register, the next state logic, and the output logic.

We consider 3 examples:

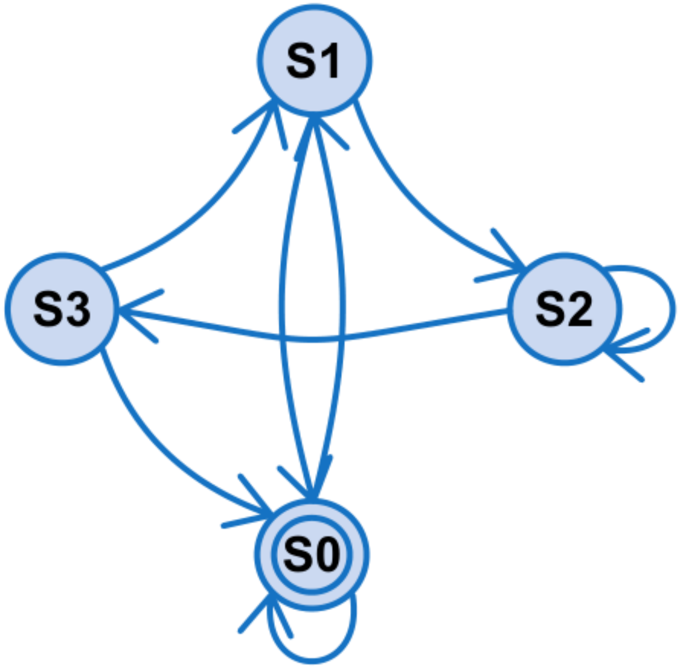
divideby3fsm



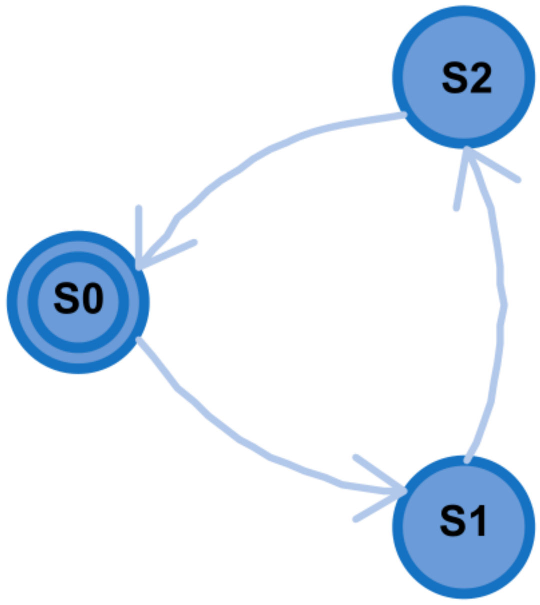
patternMoore



patternMealy



```verilog
module divideby3FSM (input clk, reset,
                     output y);
    reg [1:0] state, nextstate;
    parameter S0 = 2'b00;
    parameter S1 = 2'b01;
    parameter S2 = 2'b10;
    // state register
    always @ (posedge clk, posedge reset)
      if (reset) state <= S0;
      else state <= nextstate;
    // next state logic
    always @ (*)
      case (state)
        S0: nextstate = S1;
        S1: nextstate = S2;
        S2: nextstate = S0;
        default: nextstate = S0;
      endcase
    // output logic
    assign y = (state == S0);
endmodule
```



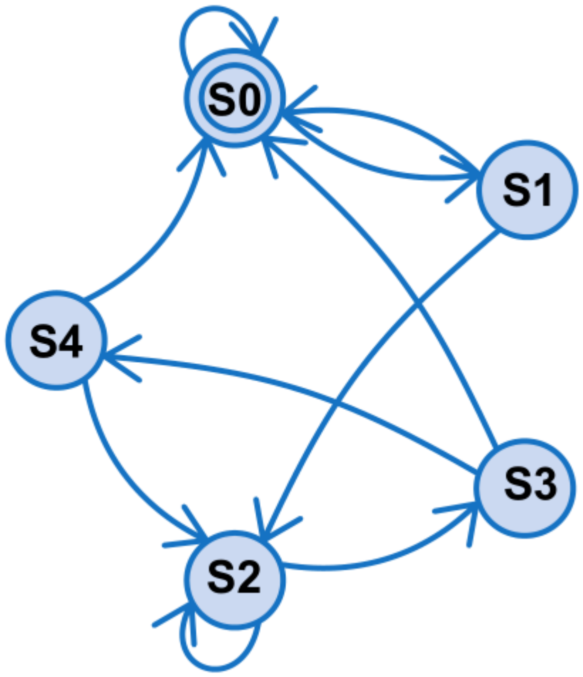
The `parameter` statement is used to define constants

Because the next state logic should be combinational, a `default` is necessary even though the state `2'b11` should never arise.

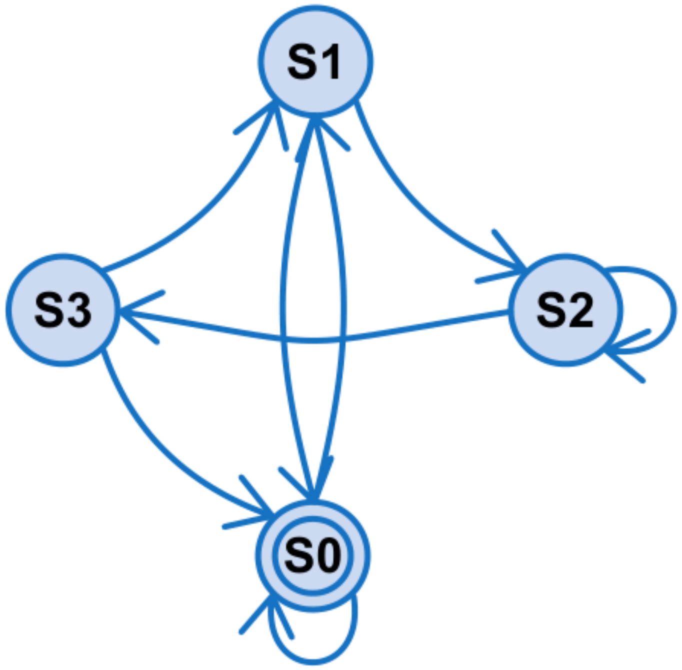
If, we had wanted the output to be HIGH in states S0 and S1, the output logic would be modified as follows.

```
// output logic
assign y = (state == S0 | state == S1);
```

```
module patternMoore (input clk, reset, a,
                     output y);
    reg [2:0] state, nextstate;
    parameter S0 = 3'b000;
    parameter S1 = 3'b001;
    parameter S2 = 3'b010;
    parameter S3 = 3'b011;
    parameter S4 = 3'b100;
    // state register
    always @ (posedge clk, posedge reset)
        if (reset) state <= S0;
        else state <= nextstate;
    // next state logic
    always @ (*)
      case (state)
        S0: if (a) nextstate = S1;
            else nextstate = S0;
        S1: if (a) nextstate = S2;
            else nextstate = S0;
        S2: if (a) nextstate = S2;
            else nextstate = S3;
        S3: if (a) nextstate = S4;
            else nextstate = S0;
        S4: if (a) nextstate = S2;
            else nextstate = S0;
        default: nextstate = S0;
      endcase
    // output logic
        assign y = (state == S4);
endmodule
```



```
module patternMealy (input clk, reset, a,
                     output y);
    reg [1:0] state, nextstate;
    parameter S0 = 2'b00;
    parameter S1 = 2'b01;
    parameter S2 = 2'b10;
    parameter S3 = 2'b11;
    // state register
    always @ (posedge clk, posedge reset)
        if (reset) state <= S0;
        else state <= nextstate;
    // next state logic
    always @ (*)
      case (state)
        S0: if (a) nextstate = S1;
            else nextstate = S0;
        S1: if (a) nextstate = S2;
            else nextstate = S0;
        S2: if (a) nextstate = S2;
            else nextstate = S3;
        S3: if (a) nextstate = S1;
            else nextstate = S0;
        default: nextstate = S0;
      endcase
    // output logic
    assign y = (a & state == S3);
endmodule
```



Memories

A memory is a two-dimensional array of bits.

Verilog allows such a structure to be declared as a variable (reg or integer) that is an array of vectors

```
reg [7:0] R [3:0];
```

This statement defines `R` as four eight-bit variables named `R[3], R[2], R[1], R[0]`.

An example of a three-dimensional array declaration:

```
reg [7:0]R[3:0][1:0];
```

Logic gates in Verilog

Name	Description	Usage
and	$f = (a \cdot b \cdot \cdots)$	**and** $(f, a, b, \ldots)$
nand	$f = \overline{(a \cdot b \cdot \cdots)}$	**nand** $(f, a, b, \ldots)$
or	$f = (a + b + \cdots)$	**or** $(f, a, b, \ldots)$
nor	$f = \overline{(a + b + \cdots)}$	**nor** $(f, a, b, \ldots)$
xor	$f = (a \oplus b \oplus \cdots)$	**xor** $(f, a, b, \ldots)$
xnor	$f = (a \odot b \odot \cdots)$	**xnor** $(f, a, b, \ldots)$
not	$f = \overline{a}$	**not** (f, a)
buf	$f = a$	**buf** (f, a)
notif0	$f = (!e\ ?\ \overline{a} : 'bz)$	**notif0** (f, a, e)
notif1	$f = (e\ ?\ \overline{a} : 'bz)$	**notif1** (f, a, e)
bufif0	$f = (!e\ ?\ a : 'bz)$	**bufif0** (f, a, e)
bufif1	$f = (e\ ?\ a : 'bz)$	**bufif1** (f, a, e)

Tri-state buffers

!e – active-low enable

Testbenches

A **testbench** is an HDL module that is 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 synthesizeable.

A self-checking testbench

```
module testbench2 ();
    reg a, b, c;
    wire 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;
      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
```

This module checks y against expectations after each input test vector is applied.

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

`initial` statements should be used only in testbenches for simulation, not in modules intended to be synthesized into actual hardware.

```
initial begin
  clk = 0;
  reset = 0;
  req_0 = 0;
  req_1 = 0;
end
```

Like signals in `always` statements, signals in initial statements must be declared to be `reg`

An always block executes always, unlike initial blocks which execute only once (at the beginning of simulation).

An always block should have a sensitive list or a delay associated with it.

```
always
  begin
    clk = 0; #1; clk = 1; #1;
end


always  begin
  #1  clk = ~clk;
end
```

`$finish` terminates the simulation.

```
if ($time>25) $finish;
```

Bi-directional ports:

```
inout read_enable;
```

Comparison using == or != is effective between signals that do not take on the values of x and z.

Testbenches use the === and !== operators for comparisons of equality and inequality, respectively, because these operators work correctly with operands that could be x or z.

Subcircuits

```
module group (Digits, Lights);
  input [11:0] Digits;
  output [1:21] Lights;
  seg7 digit0 (Digits[3:0], Lights[1:7]);
  seg7 digit1 (Digits[7:4], Lights[8:14]);
  seg7 digit2 (Digits[11:8], Lights[15:21]);
endmodule
```

```
module seg7(bcd, leds);
  input [3:0] bcd;
  output [1:7] leds;
  reg [1:7] leds;
  always @(bcd)
    case (bcd) //abcdefg
      0: leds = 7’b1111110;
      1: leds = 7’b0110000;
      2: leds = 7’b1101101;
      3: leds = 7’b1111001;
      4: leds = 7’b0110011;
      5: leds = 7’b1011011;
      6: leds = 7’b1011111;
      7: leds = 7’b1110000;
      8: leds = 7’b1111111;
      9: leds = 7’b1111011;
      default: leds = 7’bx;
    endcase
endmodule
```

Three copies of the seven-segment decoder are instantiated in the top module, `group`.