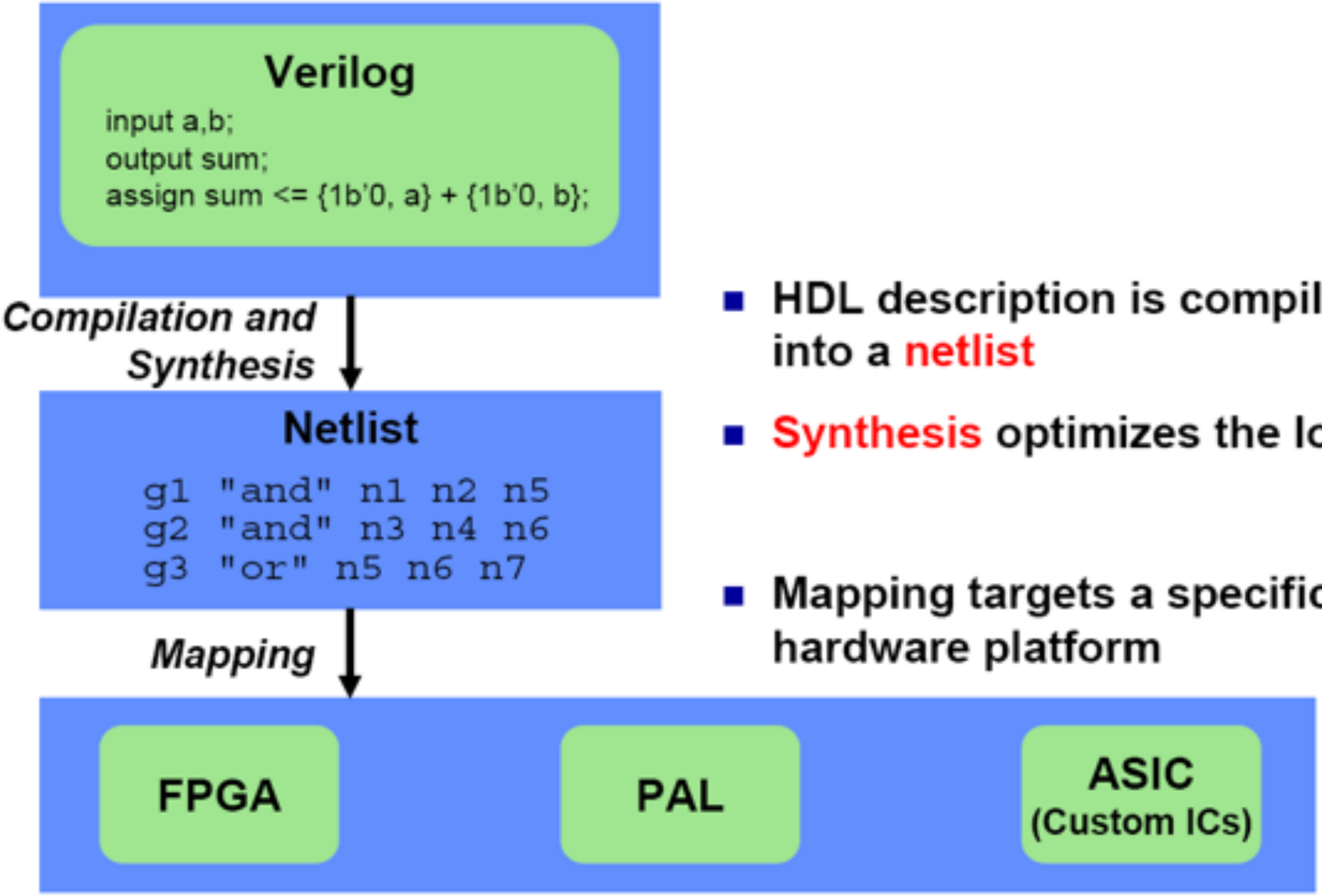
Introduction to Verilog

Synthesis and HDLs

- Hardware description language (HDL) is a convenient, device-independent representation of digital logic



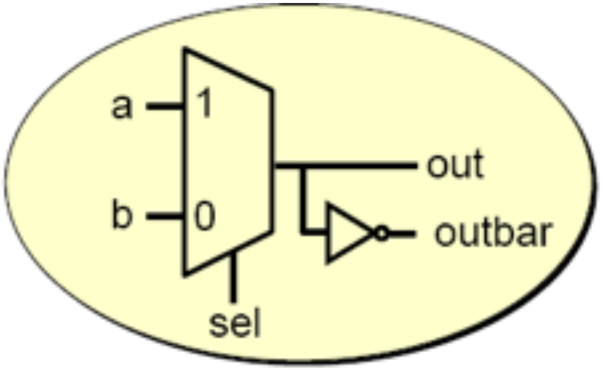
```
input a,b;
output sum;
assign sum <= {1b’0, a} + {1b’0, b};
```

```
g1 "and" n1 n2 n5
g2 "and" n3 n4 n6
g3 "or" n5 n6 n7
```

- HDL description is compiled into a netlist
- Synthesis optimizes the logic
- Mapping targets a specific hardware platform

Verilog: The Module

- Verilog designs consist of interconnected modules.
- A module can be an element or collection of lower level design blocks.
- A simple module with combinational logic might look like this:



$$\text{Out} = \text{sel} \bullet \text{a} + \overline{\text{sel}} \bullet \text{b}$$

2-to-1 multiplexer with inverted output

```
module mux_2_to_1(a, b, out,
                  outbar, sel);
```

Declare and name a module; list its ports. Don't forget that semicolon.

```
// This is 2:1 multiplexor
```

Comment starts with //
Verilog skips from // to end of the line

```
  input a, b, sel;

  output out, outbar;
```

Specify each port as input, output, or inout

```
  assign out = sel ? a : b;

  assign outbar = ~out;
```

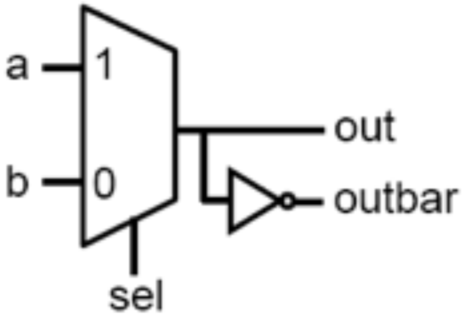
Express the module's behavior. Each statement executes in parallel; order does not matter.

```
endmodule
```

Conclude the module code.

Continuous (Dataflow) Assignment

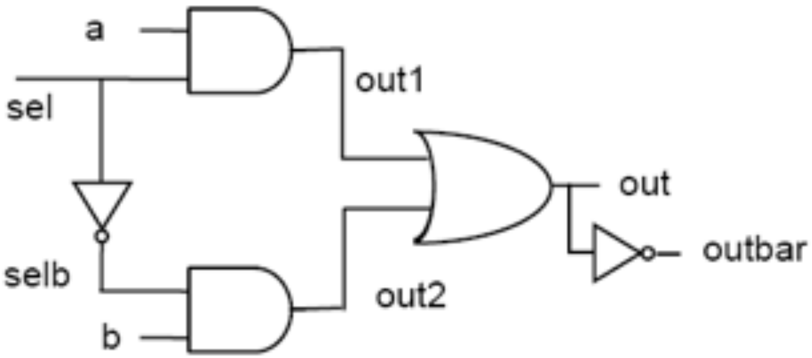
```
module mux_2_to_1(a, b, out,
                  outbar, sel);

  input a, b, sel;
  output out, outbar;

  assign out = sel ? a : b;
  assign outbar = ~out;

endmodule
```



- Continuous assignments use the `assign` keyword
- A simple and natural way to represent combinational logic
- Conceptually, the right-hand expression is continuously evaluated as a function of arbitrarily-changing inputs…just like dataflow
- The target of a continuous assignment is a net driven by combinational logic
- Left side of the assignment must be a scalar or vector net or a concatenation of scalar and vector nets. It can’t be a scalar or vector register (*discussed later*). Right side can be register or nets
- Dataflow operators are fairly low-level:
  - Conditional assignment: (conditional_expression) ? (value-if-true) : (value-if-false);
  - Boolean logic: ~, &, |
  - Arithmetic: +, -, *
- Nested conditional operator (4:1 mux)
  - `assign out = s1 ? (s0 ? i3 : i2) : (s0? i1 : i0);`

Gate Level Description

```
module muxgate (a, b, out,
outbar, sel);
input a, b, sel;
output out, outbar;
wire out1, out2, selb;
and a1 (out1, a, sel);
not i1 (selb, sel);
and a2 (out2, b , selb);
or o1 (out, out1, out2);
assign outbar = ~out;
endmodule
```



- Verilog supports basic logic gates as primitives
  - □ `and, nand, or, nor, xor, xnor, not, buf`
  - □ can be extended to multiple inputs: e.g., `nand` nand3in (out, in1, in2,in3);
  - □ `bufif1` and `bufif0` are tri-state buffers
- Net represents connections between hardware elements. Nets are declared with the keyword `wire`.

Procedural Assignment with always

- Procedural assignment allows an alternative, often higher-level, behavioral description of combinational logic
- Two structured procedure statements: `initial` and `always`
- Supports richer, C-like control structures such as `if`, `for`, `while`, `case`

```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input a, b, sel;
  output out, outbar;
```

Exactly the same as before.

```
  reg out, outbar;
```

Anything assigned in an `always` block must *also* be declared as type `reg (next slide)`

```
  always @ (a or b or sel)
```

Conceptually, the `always` block runs *once* whenever a signal in the sensitivity list changes value

```
  begin
    if (sel) out = a;
    else out = b;

    outbar = ~out;
```

Statements within the `always` block are executed sequentially. Order matters!

```
  end
```

Surround multiple statements in a single `always` block with `begin/end`.

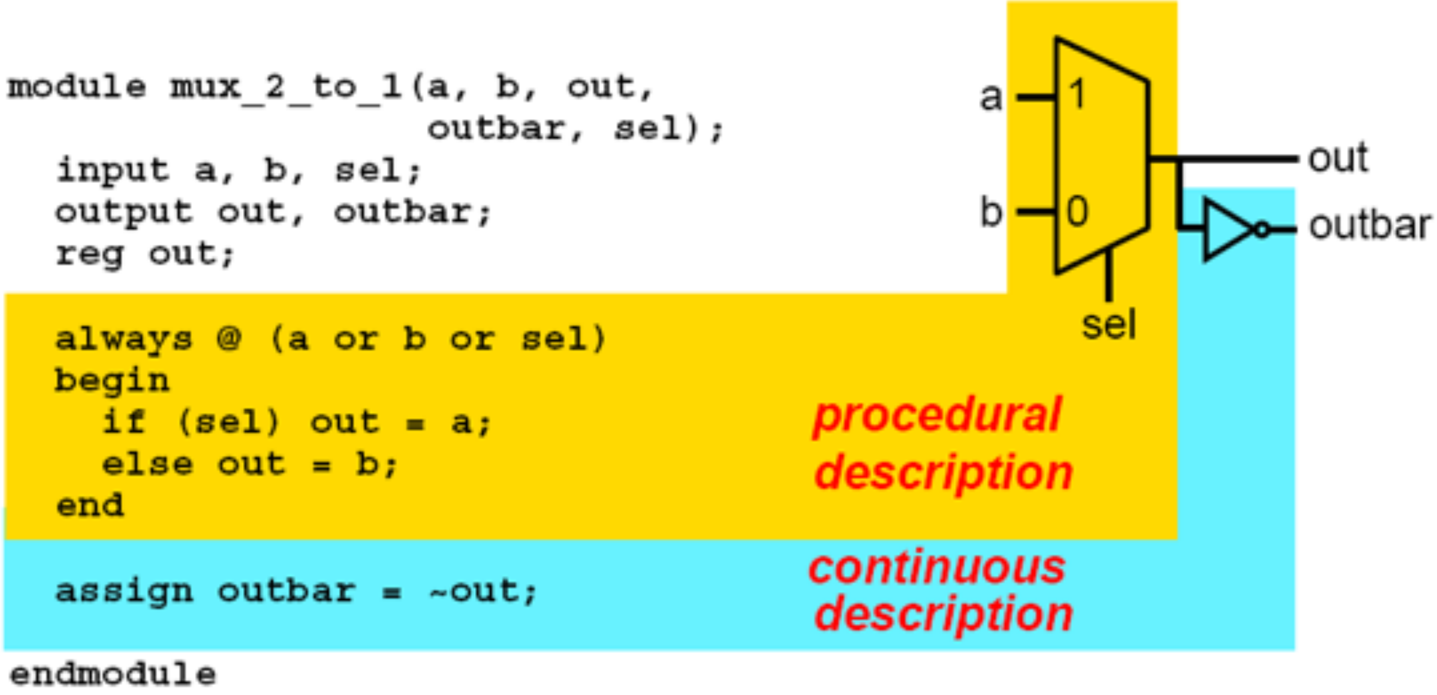
```
endmodule
```

Verilog Registers

- In digital design, registers represent memory elements
- Digital registers need a clock to operate and update their state on certain phase or edge
- Registers in Verilog should not be confused with hardware registers
- In Verilog, the term register (`reg`) simply means a variable that can hold a value
- Verilog registers don’t need a clock and don’t need to be driven like a net. Values of registers can be changed anytime in a simulation by assuming a new value to the register

Mix-and-Match Assignments

- Procedural and continuous assignments can (and often do) co-exist within a module
- Procedural assignments update the value of `reg`. The value will remain unchanged till another procedural assignment updates the variable. This is the main difference with continuous assignments in which the right hand expression is constantly placed on the left-side



```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input a, b, sel;
  output out, outbar;
  reg out;

  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end

  assign outbar = ~out;

endmodule
```

The case Statement

- `case` and `if` may be used interchangeably to implement conditional execution within `always` blocks
- `case` is easier to read than a long string of `if...else` statements

```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input a, b, sel;
  output out, outbar;
  reg out;

  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end

  assign outbar = ~out;

endmodule
```

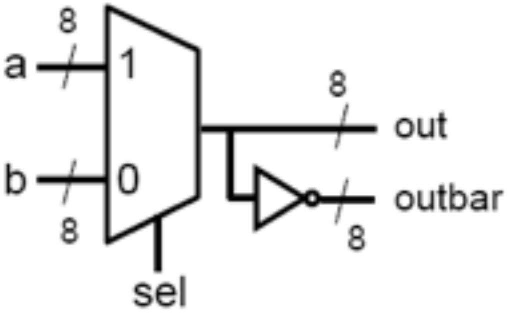
```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input a, b, sel;
  output out, outbar;
  reg out;

  always @ (a or b or sel)
  begin
    case (sel)
      1'b1: out = a;
      1'b0: out = b;
    endcase
  end

  assign outbar = ~out;

endmodule
```

Note: Number specification notation: <size>'<base><number>
(4'b1010 if a 4-bit binary value, 16'h6cda is a 16 bit hex number, and 8'd40 is an 8-bit decimal value)

The Power of Verilog: ***n*-bit Signals**

- **Multi-bit signals and buses are *easy* in Verilog.**
- **2-to-1 multiplexer with *8-bit operands*:**

```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input[7:0] a, b;
  input sel;
  output[7:0] out, outbar;
  reg[7:0] out;

  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end

  assign outbar = ~out;

endmodule
```



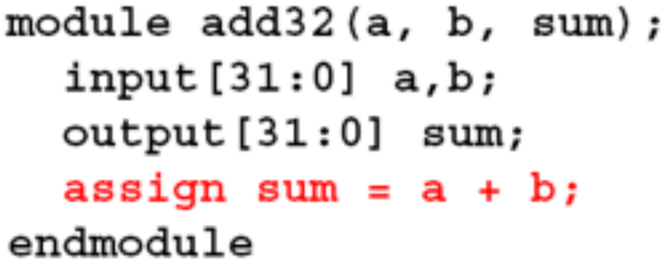
Concatenate signals using the { } operator

```
assign {b[7:0],b[15:8]} = {a[15:8],a[7:0]};
```

effects a byte swap

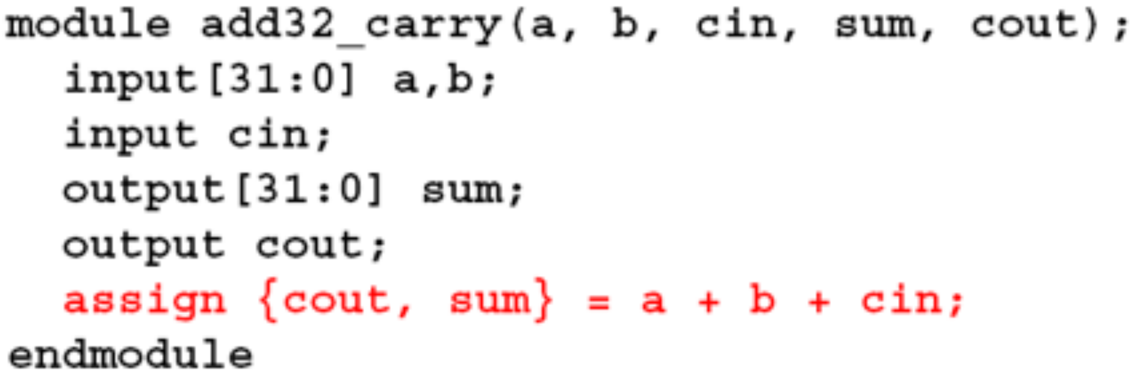
The Power of Verilog: Integer Arithmetic

- Verilog's built-in arithmetic makes a 32-bit adder easy:



```
module add32(a, b, sum);
  input[31:0] a,b;
  output[31:0] sum;
  assign sum = a + b;
endmodule
```

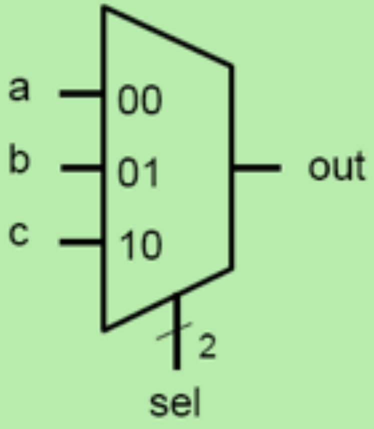
- A 32-bit adder with carry-in and carry-out:



```
module add32_carry(a, b, cin, sum, cout);
  input[31:0] a,b;
  input cin;
  output[31:0] sum;
  output cout;
  assign {cout, sum} = a + b + cin;
endmodule
```

Dangers of Verilog: Incomplete Specification

Goal:



3-to-1 MUX

('11' input is a don't-care)

Proposed Verilog Code:

```
module maybe_mux_3to1(a, b, c,
                            sel, out);
   input [1:0] sel;
   input a,b,c;
   output out;
   reg out;

   always @(a or b or c or sel)
   begin
      case (sel)
         2'b00: out = a;
         2'b01: out = b;
         2'b10: out = c;
      endcase
   end
endmodule
```

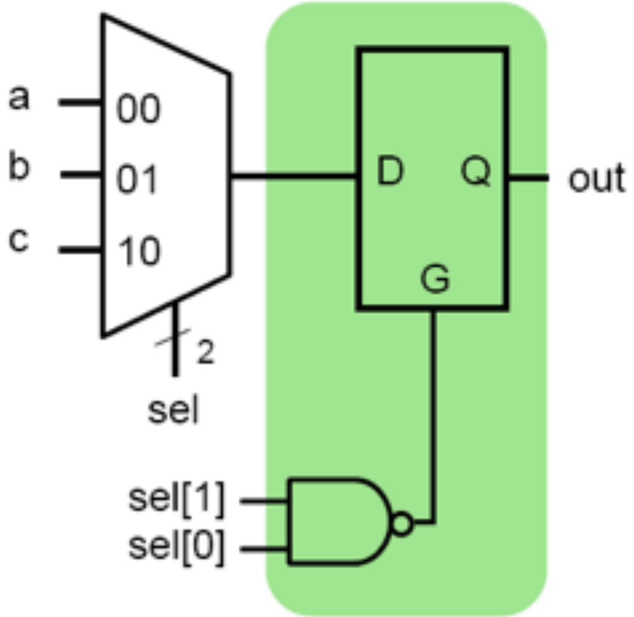
Is this a 3-to-1 multiplexer?

Incomplete Specification Infers Latches

```
module maybe_mux_3to1(a, b, c,
                      sel, out);
  input [1:0] sel;
  input a,b,c;
  output out;
  reg out;

  always @(a or b or c or sel)
  begin
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    endcase
  end
endmodule
```

if `out` is not assigned during any pass through the always block, then **the previous value must be retained!**

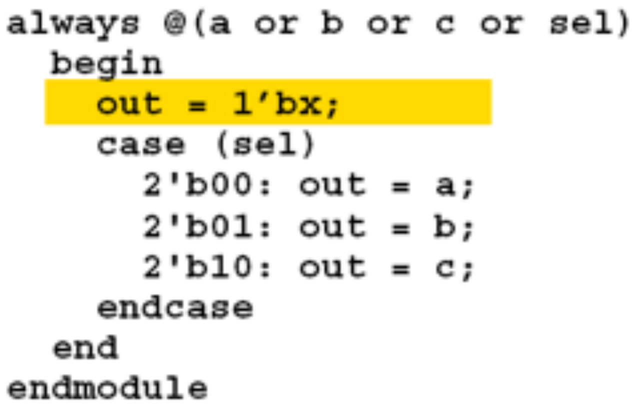
Synthesized Result:



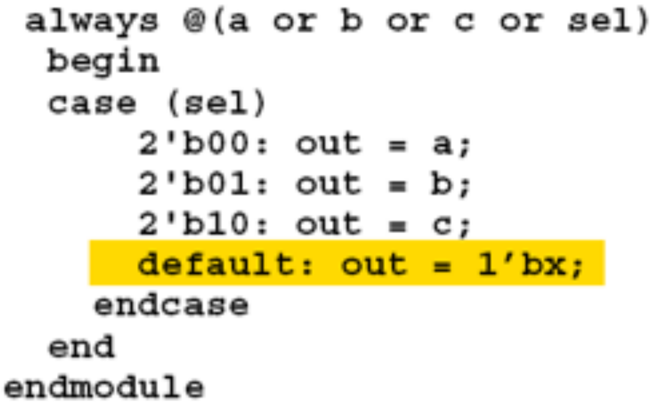
- Latch memory "latches" old data when G=0 (we will discuss latches later)
- In practice, we almost *never* intend this

Avoiding Incomplete Specification

- Precede all conditionals with a default assignment for all signals assigned within them…



```
always @(a or b or c or sel)
  begin
    out = 1’bx;
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    endcase
  end
endmodule
```



```
 always @(a or b or c or sel)
  begin
  case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
      default: out = 1’bx;
    endcase
  end
endmodule
```

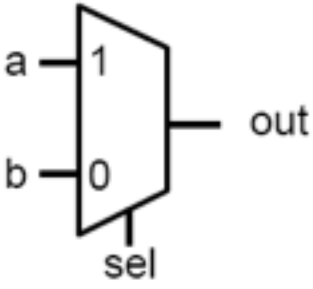
- …or, fully specify all branches of conditionals and assign all signals from all branches
  - For each `if`, include `else`
  - For each `case`, include `default`

The Sequential always Block

- Edge-triggered circuits are described using a sequential `always` block

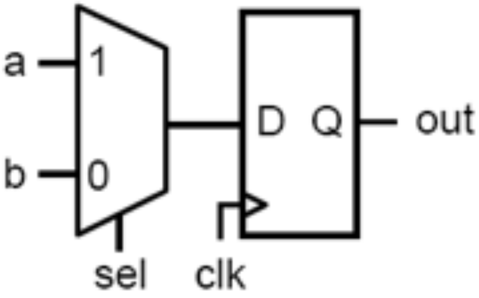
Combinational

```
module combinational(a, b, sel,
                     out);
  input a, b;
  input sel;
  output out;
  reg out;

  always @ (a or b or sel)
  begin
    if (sel) out = a;
    else out = b;
  end

endmodule
```



Sequential

```
module sequential(a, b, sel,
                  clk, out);
  input a, b;
  input sel, clk;
  output out;
  reg out;

  always @ (posedge clk)
  begin
    if (sel) out <= a;
    else out <= b;
  end

endmodule
```



Importance of the Sensitivity List

- **The use of `posedge` and `negedge` makes an `always` block sequential (edge-triggered)**
- **Unlike a combinational `always` block, the sensitivity list does determine behavior for synthesis!**

D Flip-flop with synchronous clear

```
module dff_sync_clear(d, clearb,
clock, q);
input d, clearb, clock;
output q;
reg q;
always @ (posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```

`always` block entered only at each positive clock edge

D Flip-flop with asynchronous clear

```
module dff_async_clear(d, clearb, clock, q);
input d, clearb, clock;
output q;
reg q;

always @ (negedge clearb or posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```

`always` block entered immediately when (active-low) clearb is asserted

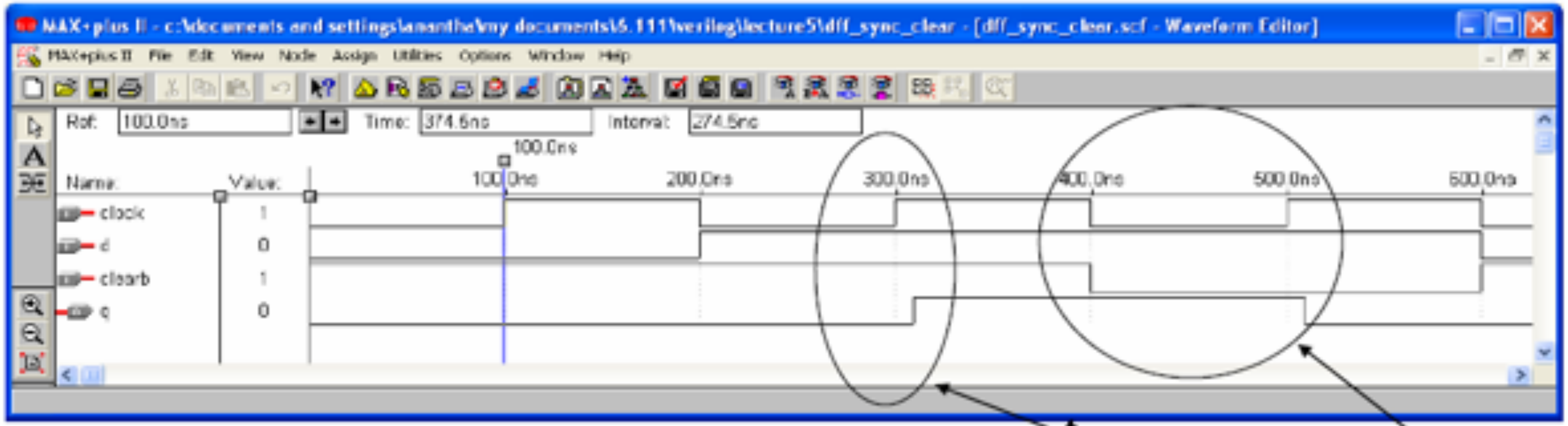
Note: The following is **incorrect** syntax: `always @ (clear or negedge clock)`

If one signal in the sensitivity list uses posedge/negedge, then all signals must.

- **Assign any signal or variable from <u>only one</u> always block, Be wary of race conditions: always blocks execute in parallel**

Simulation

- **DFF with Synchronous Clear**



MAX+plus II - c:\documents and settings\anantha\my documents\6.111\verilog\lecture5\dff_sync_clear - [dff_sync_clear.scf - Waveform Editor]

Ref: 100.0ns Time: 374.6ns Interval: 274.6ns

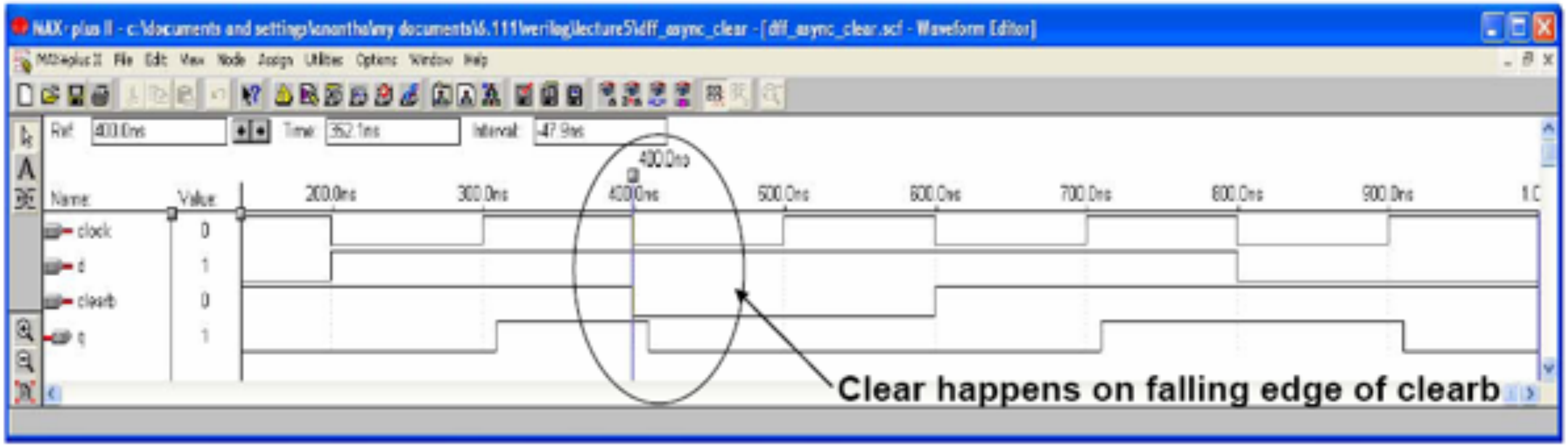
Name:	Value:
clock	1
d	0
clearb	1
q	0

MAX+plus II File Edit View Node Assign Utilities Options Window Help
100.0ns
100.0ns
200.0ns
300.0ns
400.0ns
500.0ns
600.0ns

$t_{c\text{-}q}$

Clear on Clock Edge

- **DFF with Asynchronous Clear**



MAX+plus II - c:\documents and settings\anantha\my documents\6.111\verilog\lecture5\dff_async_clear - [dff_async_clear.scf - Waveform Editor]

Ref: 400.0ns Time: 352.1ns Interval: -47.9ns

Name:	Value:
clock	0
d	1
clearb	0
q	1

MAX+plus II File Edit View Node Assign Utilities Options Window Help
400.0ns
200.0ns
300.0ns
400.0ns
500.0ns
600.0ns
700.0ns
800.0ns
900.0ns
1.0

Clear happens on falling edge of clearb

Blocking vs. Nonblocking Assignments

- Verilog supports two types of assignments within `always` blocks, with subtly different behaviors.
- *Blocking assignment:* evaluation and assignment are immediate

```
always @ (a or b or c)
begin
  x = a | b;                 1. Evaluate a | b, assign result to x
  y = a ^ b ^ c;             2. Evaluate a^b^c, assign result to y
  z = b & ~c;                3. Evaluate b&(~c), assign result to z
end
```

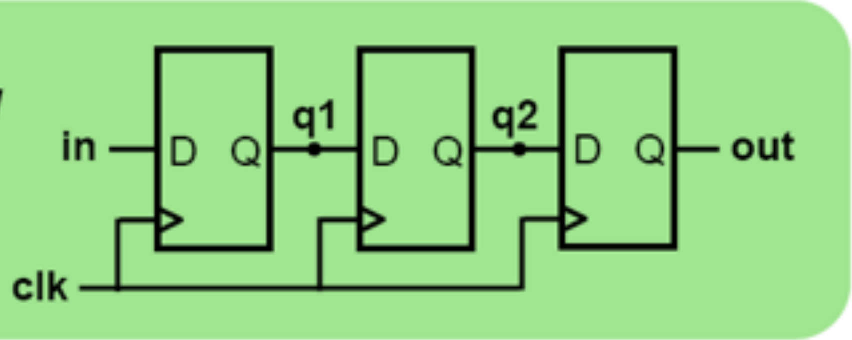
- *Nonblocking assignment:* all assignments deferred until all right-hand sides have been evaluated (end of simulation timestep)

```
always @ (a or b or c)
begin
  x <= a | b;                1. Evaluate a | b but defer assignment of x
  y <= a ^ b ^ c;            2. Evaluate a^b^c  but defer assignment of y
  z <= b & ~c;               3. Evaluate b&(~c) but defer assignment of z
end                          4. Assign x, y, and z with their new values
```

- Sometimes, as above, both produce the same result. Sometimes, not!

Assignment Styles for Sequential Logic

Flip-Flop Based Digital Delay Line



- **Will nonblocking and blocking assignments both produce the desired result?**

```
module nonblocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;
  always @ (posedge clk)
  begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end
endmodule
```

```
module blocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;
  always @ (posedge clk)
  begin
    q1 = in;
    q2 = q1;
    out = q2;
  end
endmodule
```

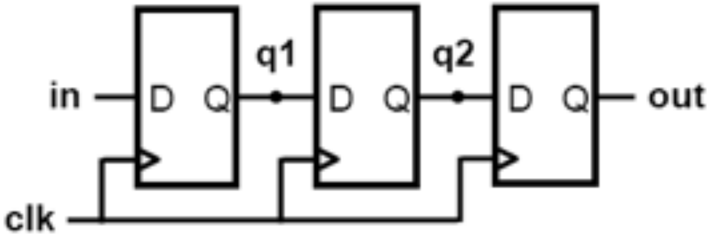
Use Nonblocking for Sequential Logic

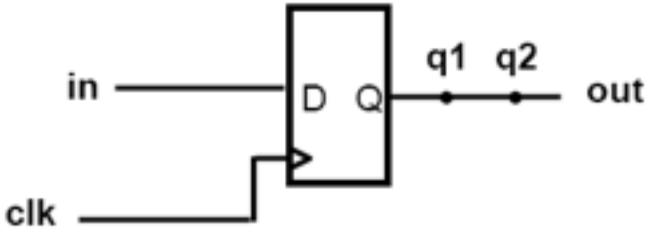
```
always @ (posedge clk)
begin
  q1 <= in;
  q2 <= q1;
  out <= q2;
end
```

```
always @ (posedge clk)
begin
  q1 = in;
  q2 = q1;
  out = q2;
end
```

"At each rising clock edge, *q1*, *q2*, and *out* simultaneously receive the old values of *in*, *q1*, and *q2*."

"At each rising clock edge, *q1* = *in*.
After that, *q2* = *q1* = *in*.
After that, *out* = *q2* = *q1* = *in*.
Therefore *out* = *in*."

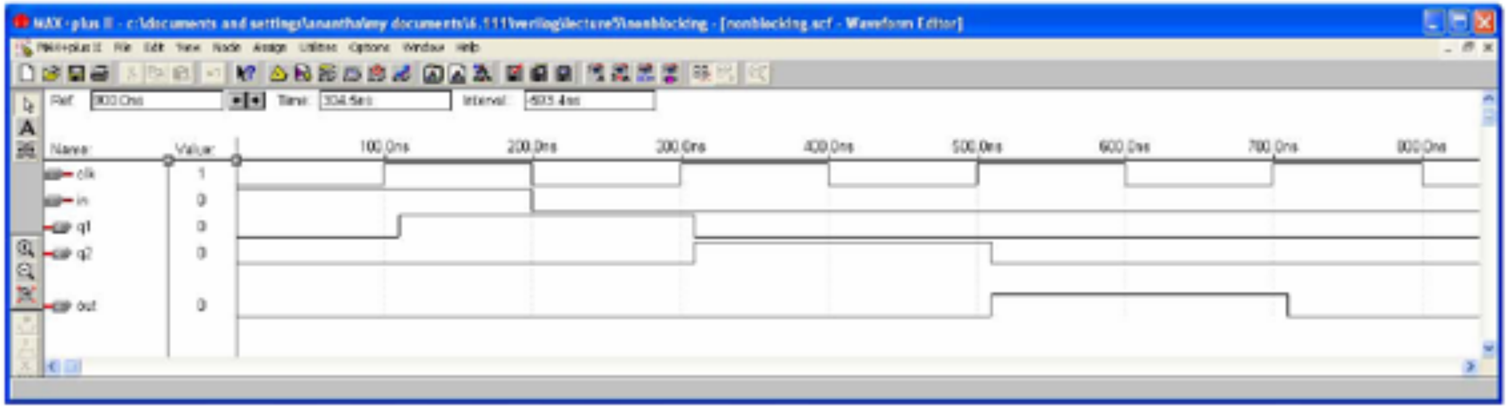




- Blocking assignments do not reflect the intrinsic behavior of multi-stage sequential logic
- **Guideline: use nonblocking assignments for sequential `always` blocks**

Simulation

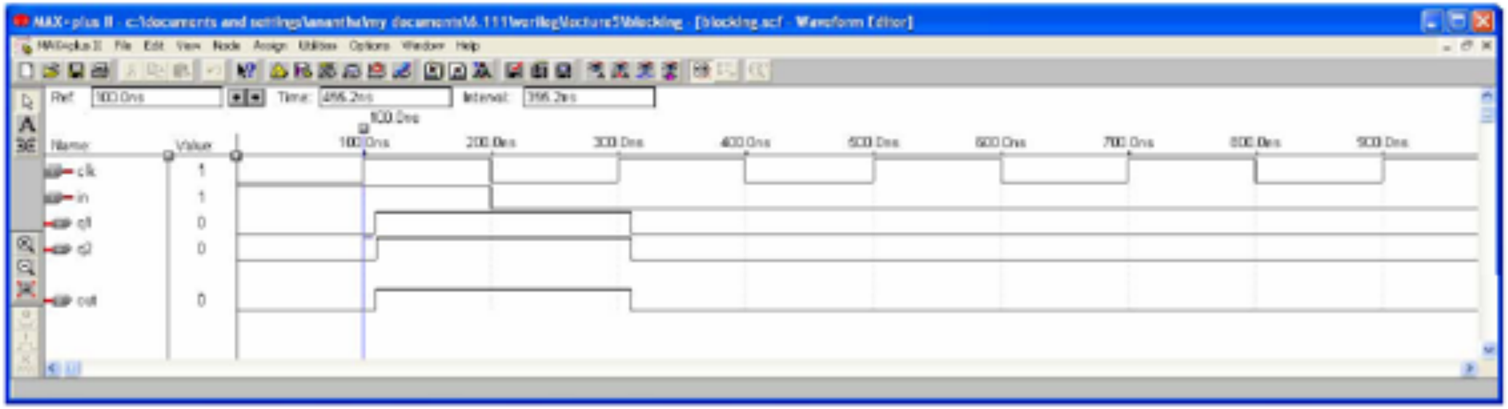
- **Non-blocking Simulation**



MAX+plus II - c:\documents and settings\anantha\my documents\6.111\verilog\lecture5\nonblocking - [nonblocking.scf - Waveform Editor]

Ref: 800.0ns Time: 304.5ns Interval: -495.5ns

- **Blocking Simulation**

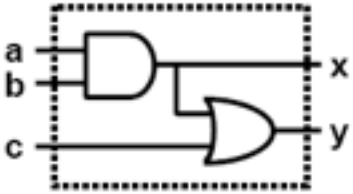


MAX+plus II - c:\documents and settings\anantha\my documents\6.111\verilog\lecture5\blocking - [blocking.scf - Waveform Editor]

Ref: 100.0ns Time: 496.2ns Interval: 396.2ns

Use Blocking for Combinational Logic

Blocking Behavior	a b c x y
(Given) Initial Condition	1 1 0 1 1
a changes; `always` block triggered	0 1 0 1 1
`x = a & b;`	0 1 0 0 1
`y = x | c;`	0 1 0 0 0



```
module blocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;

  always @ (a or b or c)
  begin
    x = a & b;
    y = x | c;
  end

endmodule
```

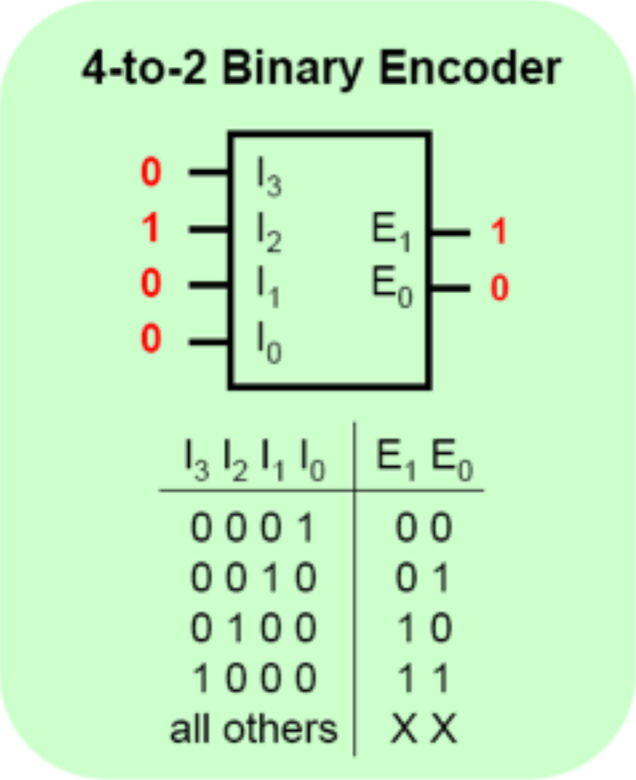
Nonblocking Behavior	a b c x y	*Deferred*
(Given) Initial Condition	1 1 0 1 1	
a changes; `always` block triggered	0 1 0 1 1	
`x <= a & b;`	0 1 0 1 1	x<=0
`y <= x | c;`	0 1 0 1 1	x<=0, y<=1
Assignment completion	0 1 0 0 1	

```
module nonblocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;

  always @ (a or b or c)
  begin
    x <= a & b;
    y <= x | c;
  end

endmodule
```

- Nonblocking and blocking assignments will synthesize correctly. Will both styles simulate correctly?
- Nonblocking assignments do not reflect the intrinsic behavior of multi-stage combinational logic
- While nonblocking assignments can be hacked to simulate correctly (expand the sensitivity list), it's not elegant
- Guideline: use blocking assignments for combinational `always` blocks

Dangers of Verilog : Priority Logic

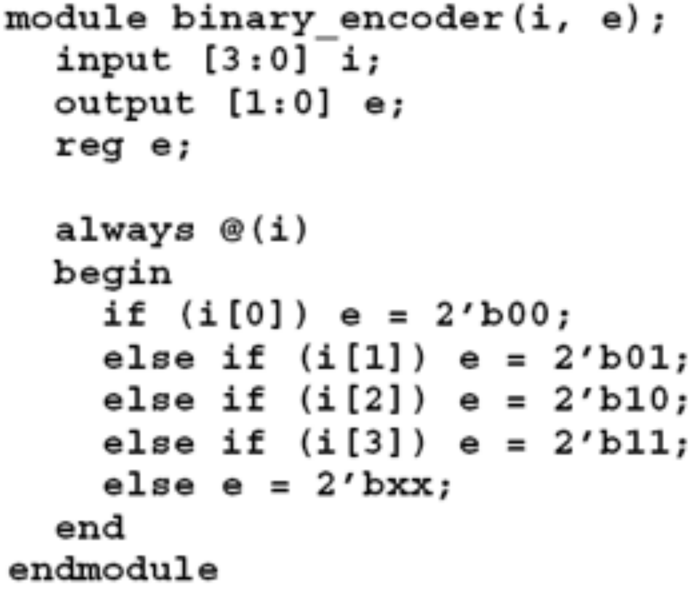
Goal:



4-to-2 Binary Encoder

$I_3\ I_2\ I_1\ I_0$	$E_1\ E_0$
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

Proposed Verilog Code:



```
module binary_encoder(i, e);
   input [3:0] i;
   output [1:0] e;
   reg e;

   always @(i)
   begin
     if (i[0]) e = 2'b00;
     else if (i[1]) e = 2'b01;
     else if (i[2]) e = 2'b10;
     else if (i[3]) e = 2'b11;
     else e = 2'bxx;
   end
endmodule
```

What is the resulting circuit?

Priority Logic

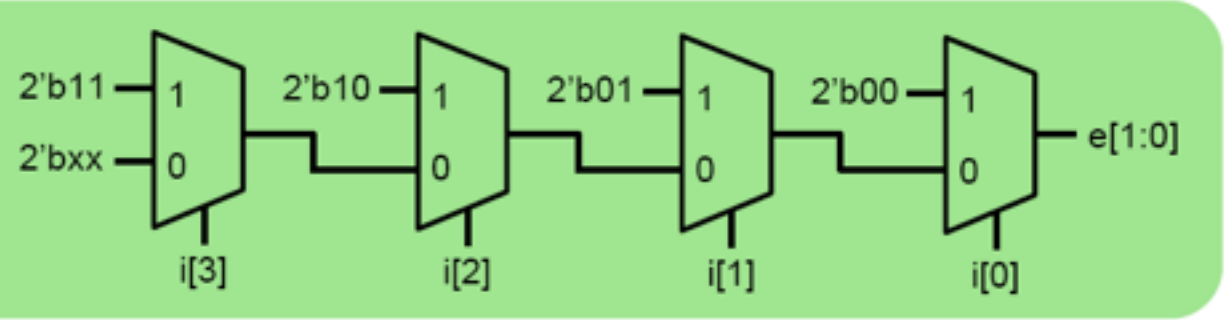
Intent: if more than one input is 1, the result is a don't-care.

I_3 I_2 I_1 I_0	E_1 E_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

Code: if i[0] is 1, the result is 00 regardless of the other inputs. ***i[0] takes the highest priority.***

```
if (i[0]) e = 2'b00;
else if (i[1]) e = 2'b01;
else if (i[2]) e = 2'b10;
else if (i[3]) e = 2'b11;
else e = 2'bxx;
end
```

Inferred Result:



- `if-else` and `case` statements are interpreted very literally! Beware of unintended priority logic.

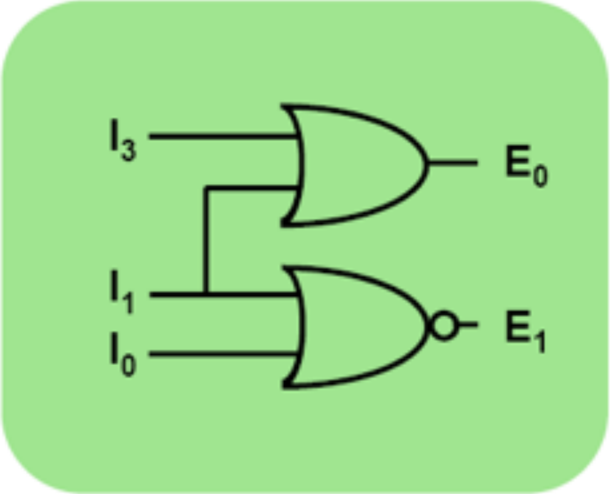
Avoiding (Unintended) Priority Logic

- Make sure that `if-else` and `case` statements are *parallel*
  - If mutually exclusive conditions are chosen for each branch...
  - ...then synthesis tool can generate a simpler circuit that evaluates the branches in parallel

Parallel Code:

```
module binary_encoder(i, e);
  input [3:0] i;
  output [1:0] e;
  reg e;

  always @(i)
  begin
    if (i == 4'b0001) e = 2'b00;
    else if (i == 4'b0010) e = 2'b01;
    else if (i == 4'b0100) e = 2'b10;
    else if (i == 4'b1000) e = 2'b11;
    else e = 2'bxx;
  end
endmodule
```

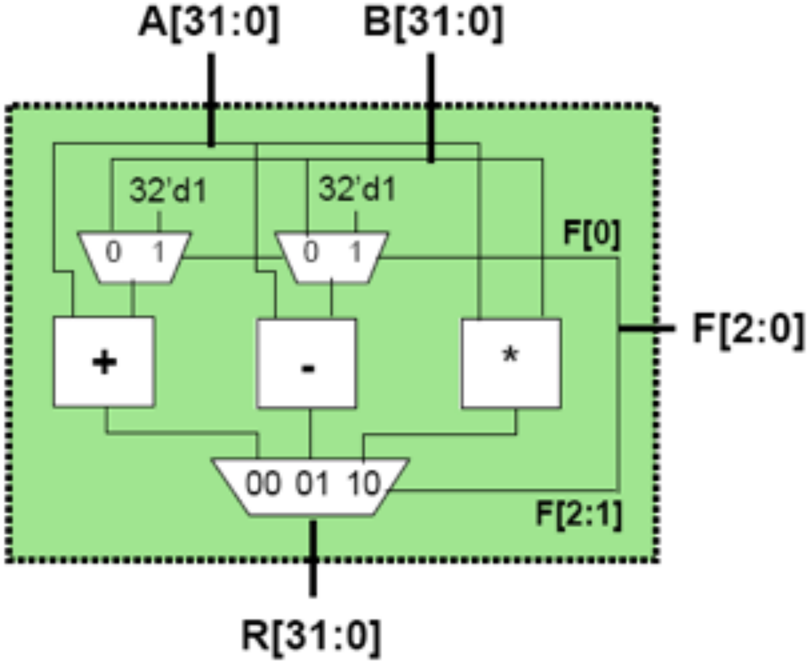
Minimized Result:



Interconnecting Modules

- Modularity is essential to the success of large designs
- A Verilog `module` may contain submodules that are "wired together"
- High-level primitives enable direct synthesis of behavioral descriptions (functions such as additions, subtractions, shifts (<< and >>), etc.

Example: A 32-bit ALU

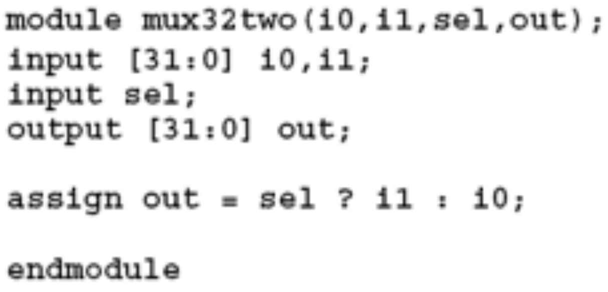


Function Table

F2	F1	F0	Function
0	0	0	A + B
0	0	1	A + 1
0	1	0	A - B
0	1	1	A - 1
1	0	X	A * B

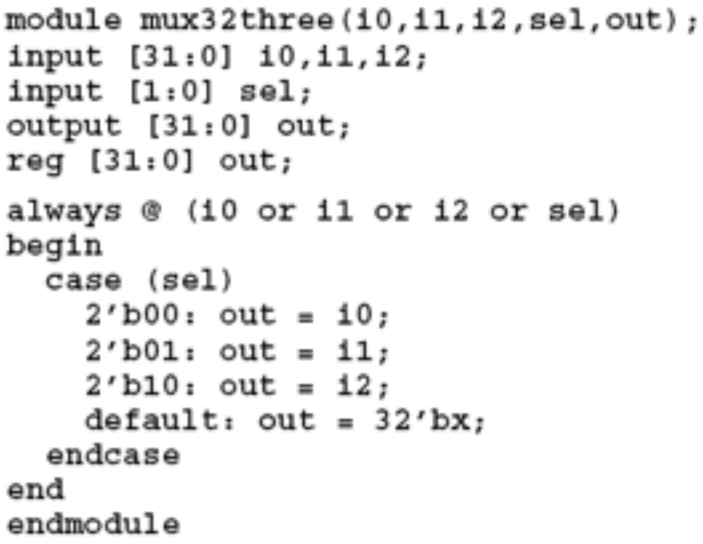
Module Definitions

2-to-1 MUX



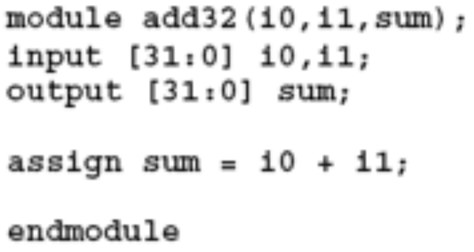
```
module mux32two(i0,i1,sel,out);
input [31:0] i0,i1;
input sel;
output [31:0] out;

assign out = sel ? i1 : i0;

endmodule
```

3-to-1 MUX



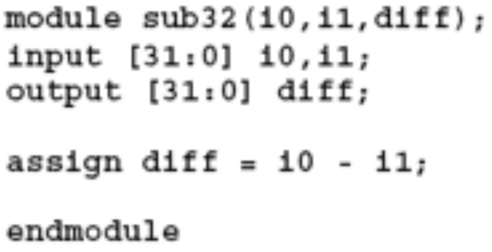
```
module mux32three(i0,i1,i2,sel,out);
input [31:0] i0,i1,i2;
input [1:0] sel;
output [31:0] out;
reg [31:0] out;

always @ (i0 or i1 or i2 or sel)
begin
  case (sel)
    2’b00: out = i0;
    2’b01: out = i1;
    2’b10: out = i2;
    default: out = 32’bx;
  endcase
end
endmodule
```

32-bit Adder



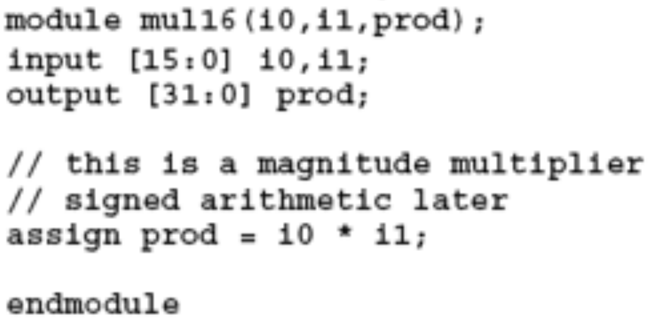
```
module add32(i0,i1,sum);
input [31:0] i0,i1;
output [31:0] sum;

assign sum = i0 + i1;

endmodule
```

32-bit Subtracter



```
module sub32(i0,i1,diff);
input [31:0] i0,i1;
output [31:0] diff;

assign diff = i0 - i1;

endmodule
```

16-bit Multiplier

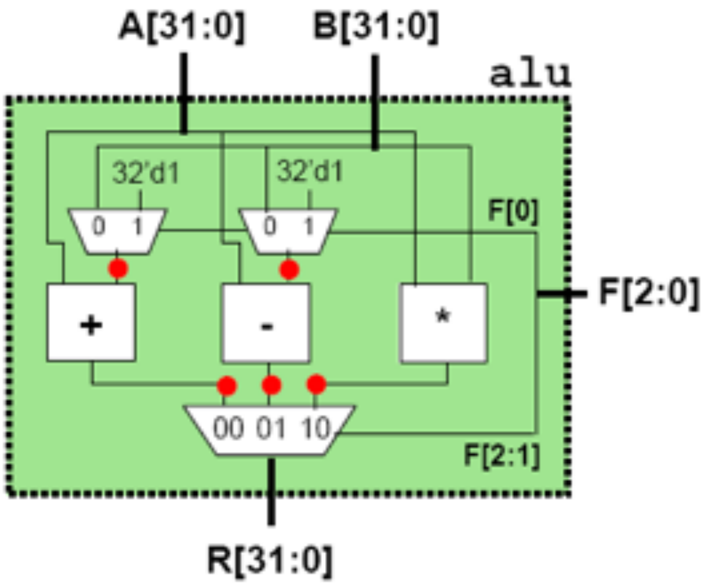


```
module mul16(i0,i1,prod);
input [15:0] i0,i1;
output [31:0] prod;

// this is a magnitude multiplier
// signed arithmetic later
assign prod = i0 * i1;

endmodule
```

Top-Level ALU Declaration

- **Given submodules:**

```
module mux32two(i0,i1,sel,out);
module mux32three(i0,i1,i2,sel,out);
module add32(i0,i1,sum);
module sub32(i0,i1,diff);
module mul16(i0,i1,prod);
```



- **Declaration of the ALU Module:**

```
module alu(a, b, f, r);
  input  [31:0] a, b;
  input  [2:0] f;
  output [31:0] r;

  wire [31:0] addmux_out, submux_out;
  wire [31:0] add_out, sub_out, mul_out;

   mux32two    adder_mux(b, 32'd1, f[0], addmux_out);
   mux32two    sub_mux(b, 32'd1, f[0], submux_out);
   add32       our_adder(a, addmux_out, add_out);
   sub32       our_subtracter(a, submux_out, sub_out);
   mul16       our_multiplier(a[15:0], b[15:0], mul_out);
   mux32three  output_mux(add_out, sub_out, mul_out, f[2:1], r);

endmodule
```

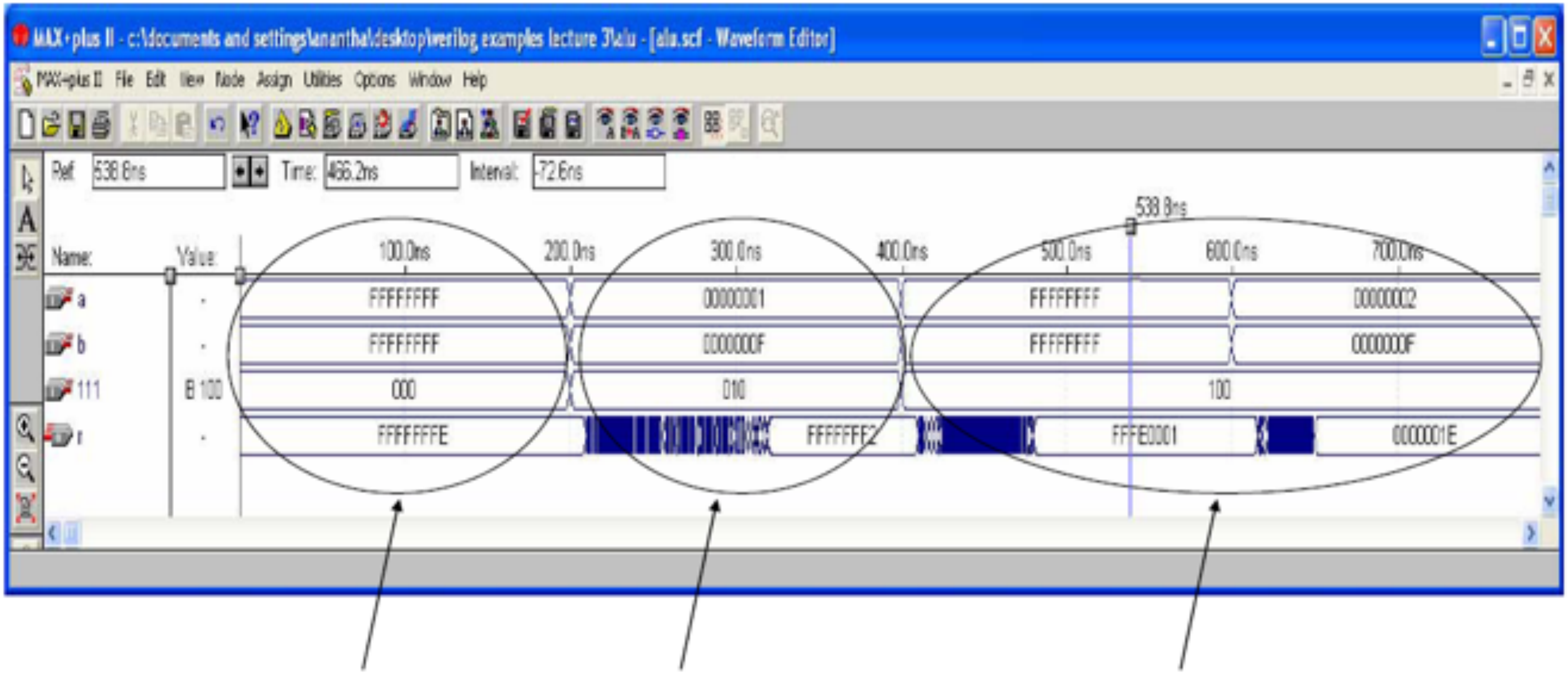
intermediate output nodes ●

module names

(unique) instance names

corresponding wires/regs in module alu

Simulation



MAX+plus II - c:\documents and settings\anantha\desktop\verilog examples lecture 3\alu - [alu.scf - Waveform Editor]

Ref: 538.8ns

Time: 466.2ns

Interval: -72.6ns

addition

subtraction

multiplier

More on Module Interconnection

- **Explicit port naming allows port mappings in arbitrary order: better scaling for large, evolving designs**

Given Submodule Declaration:

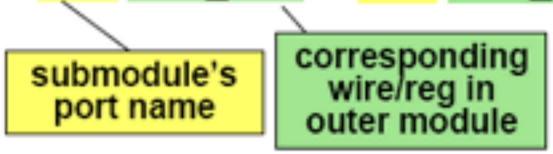
```
module mux32three(i0,i1,i2,sel,out);
```

Module Instantiation with Ordered Ports:

```
mux32three output_mux(add_out, sub_out, mul_out, f[2:1], r);
```

Module Instantiation with Named Ports:

```
mux32three output_mux(.sel(f[2:1]), .out(r), .i0(add_out),
                      .i1(sub_out), .i2(mul_out));
```



submodule's port name

corresponding wire/reg in outer module

- **Built-in Verilog gate primitives may be instantiated as well**
  - □ **Instantiations may omit instance name and must be ordered:**

```
buf(out1,out2,...,outN, in); and(in1,in2,...inN,out);
```

Useful Boolean Operators

- **Bitwise operators** perform bit-sliced operations on vectors
  - □ ~(4’b0101) = {~0,~1,~0,~1} = 4’b1010
  - □ 4’b0101 & 4’b0011 = 4’b0001
- **Logical operators** return one-bit (true/false) results
  - □ !(4’b0101) = ~1 = 1’b0
- **Reduction operators** act on each bit of a single input vector
  - □ &(4’b0101) = 0 & 1 & 0 & 1 = 1’b0
- **Comparison operators** perform a Boolean test on two arguments

Bitwise

~a	NOT
a & b	AND
a | b	OR
a ^ b	XOR
a ~^ b	XNOR

Logical

!a	NOT
a && b	AND
a || b	OR

Reduction

&a	AND
~&	NAND
|	OR
~|	NOR
^	XOR

Comparison

a < b a > b a <= b a >= b	Relational
a == b a != b	[in]equality returns x when x or z in bits. Else returns 0 or 1
a === b a !== b	case [in]equality returns 0 or 1 based on bit by bit comparison

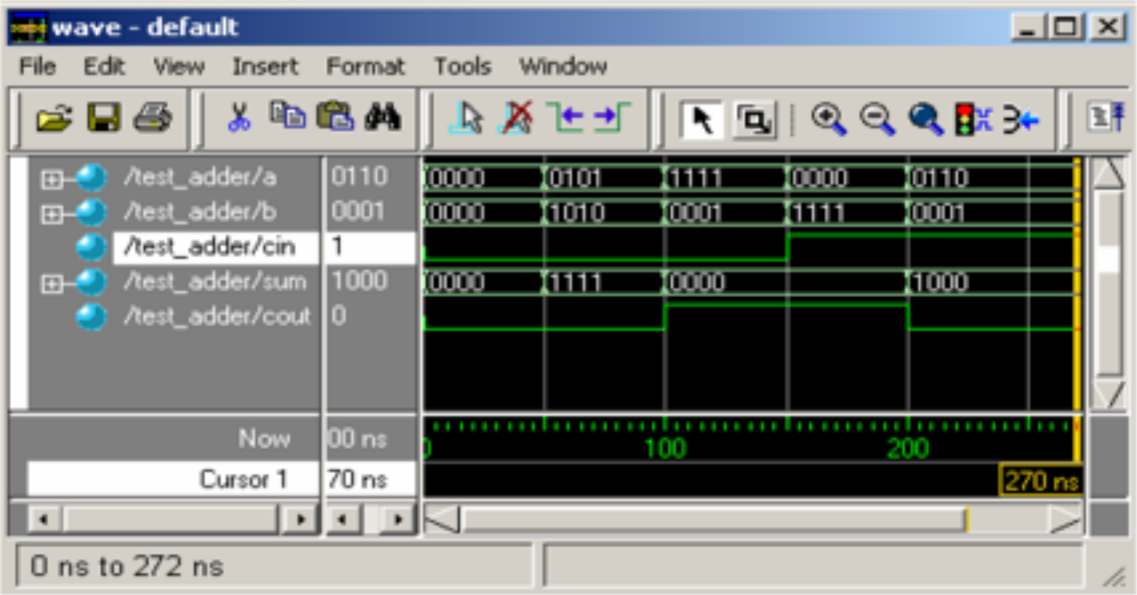
Note distinction between ~a and !a

Testbenches (ModelSim)

Full Adder (1-bit)

```
module full_adder (a, b, cin,
                   sum, cout);
  input   a, b, cin;
  output  sum, cout;
  reg     sum, cout;

  always @(a or b or cin)
   begin
     sum = a ^ b ^ cin;
     cout = (a & b) | (a & cin) | (b & cin);
   end
Endmodule
```

ModelSim Simulation



Full Adder (4-bit)

```
module full_adder_4bit (a, b, cin, sum,
cout);
  input[3:0]   a, b;
  input        cin;
  output [3:0] sum;
  output        cout;
  wire         c1, c2, c3;

  // instantiate 1-bit adders
  full_adder FA0(a[0],b[0], cin, sum[0], c1);
  full_adder FA1(a[1],b[1], c1, sum[1], c2);
  full_adder FA2(a[2],b[2], c2, sum[2], c3);
  full_adder FA3(a[3],b[3], c3, sum[3], cout);
endmodule
```

Testbench

```
module test_adder;
  reg [3:0]  a, b;
  reg         cin;
  wire [3:0] sum;
  wire        cout;

  full_adder_4bit dut(a, b, cin,
                      sum, cout);

initial
    begin
       a = 4'b0000;
       b = 4'b0000;
       cin = 1'b0;
       #50;
      a = 4'b0101;
      b = 4'b1010;
      // sum = 1111, cout = 0
      #50;
      a = 4'b1111;
      b = 4'b0001;
      // sum = 0000, cout = 1
#50;
      a = 4'b0000;
      b = 4'b1111;
      cin = 1'b1;
      // sum = 0000, cout = 1
      #50;
      a = 4'b0110;
      b = 4'b0001;
      // sum = 1000, cout = 0
   end // initial begin
endmodule // test_adder
```

Summary

- Multiple levels of description: behavior, dataflow, logic and switch
- Gate level is typically not used as it requires working out the interconnects
- Continuous assignment using `assign` allows specifying dataflow structures
- Procedural Assignment using `always` allows efficient behavioral description. Must carefully specify the sensitivity list
- Incomplete specification of `case` or `if` statements can result in non-combinational logic
- Verilog registers (`reg`) is not to be confused with a hardware memory element
- Modular design approach to manage complexity

Advance Verilog

Parameter

- Parameters are useful because they can be redefined on a module instance basis. That is, each different instance can have different parameter values. This is particularly useful for vector widths.
- For example, the following module implements a shifter:

```
module shift (shiftOut, dataIn, shiftCount);
    parameter width = 4;
    output [width-1:0] shiftOut;
    input [width-1:0] dataIn;
    input [31:0] shiftCount;
    assign shiftOut = dataIn << shiftCount;
endmodule
```

- This module can now be used for shifters of various sizes, simply by changing the width parameter.

Define Parameter Value

- There are two ways to change parameter values from their defaults, `defparam` statements and module instance parameter assignment.
  - The `defparam` statement allows you to change a module instance parameter directly from another module. This is usually used as follows:

    ```
     shift sh1 (shiftedVal, inVal, 7); //instantiation
    defparam sh1.width = 16; // parameter redefinition
    ```

  - Parameter values can be specified in the module instantiation directly. This is done as follows:

    ```
     shift #(16) sh1 (shiftedVal, inVal, 7);
     //instance of 16-bit shift module
    ```

Task and Function

- Tasks and functions are declared within modules. The declaration may occur anywhere within the module, but it may not be nested within procedural blocks. The declaration does not have to precede the task or function invocation.
- Tasks may only be used in procedural blocks. A task invocation, or task enable as it is called in Verilog, is a statement by itself. It may not be used as an operand in an expression.
- Functions are used as operands in expressions. A function may be used in either a procedural block or a continuous assignment, or indeed, any place where an expression may appear.

Task

- Tasks may have zero or more arguments, and they may be input, output, or inout arguments.

```
task do_read;
input [15:0] addr;
output [7:0] value;
begin
   adbus_reg = addr; // put address out
   adbus_en = 1; // drive address bus
   @(posedge clk) ; // wait for the next clock
         while (~ack)
   @(posedge clk); // wait for ack
   value = data_bus; // take returned value
   adbus_en = 0; // turn off address bus
   count = count + 1; // how many have we done
end
endtask
```

Function

- In contrast to tasks, no time or delay controls are allowed in a function. Function arguments are also *restricted to inputs only*. Output and inout arguments are not allowed. The output of a function is indicated by an assignment to the function name. For example,

```
function [15:0] relocate;
    input [11:0] addr;
    input [3:0] relocation_factor;
begin
    relocate = addr + (relocation_factor<<12);
    count = count + 1; // how many have we done
end
endfunction
```

- The above function might be used like this:

```
assign absolute_address = relocate(relative_address, rf);
```

System Task

- System tasks are used just like tasks which have been defined with the *task ... endtask* construct. They are distinguished by their first character, which is always a "$".
- There are many system tasks, but the most common are:
  - `$display, $write, $strobe`
  - `$monitor`
  - `$readmemh and $readmemb`
  - `$stop`
  - `$finish`

Example of System Task

- The `$write` system task is just like `$display`, except that it does not add a newline character to the output string.

- Example:

```
$write ($time," array:");
for (i=0; i<4; i=i+1) write(" %h", array[i]);
  $write("\n");
```

  This would produce the following output:

```
110 array: 5a5114b3 0870261a 0678448d 4e8a7776
```

System Function

- Likewise, system functions are used just like functions which have been defined with the *function ... endfunction* construct. Their first character is also always a "$".
- There are many system functions, with the most common being:
  - `$time ($stime)`
  - `$random`
  - `$bitstoreal`

Example of System Function

- The `$time` system function simply returns the current simulation time. Simulation time is a 64-bit unsigned quantity, and that is what `$time` is assumed to be when it is used in an expression.
- `$stime` (short time) is just like $time, except that it returns a 32-bit value of time.
- Example:

```
$display ("The current time is %d", $time);

$display ($time," now the value of x is %h", x);
```

Conversion Function

$rtoi(real_value)

Returns a signed integer, truncating the real value.

$itor(int_val)

Returns the integer converted to a real value.

$realtobits(real_value)

Returns a 64-bit vector with the bit representation of the real number.

$bitstoreal(bit_value)

Returns a real value obtained by interpreting the bit_value argument as an IEEE 754 floating point number.

```
module driver (net_r);
output net_r;
real r;
wire [64:1]
net_r = $realtobits(r);
endmodule
module receiver (net_r);
input net_r;
wire [64:1] net_r;
real r;
always @(net_r)
   r = $bitstoreal(net_r);
endmodule
```

XMR

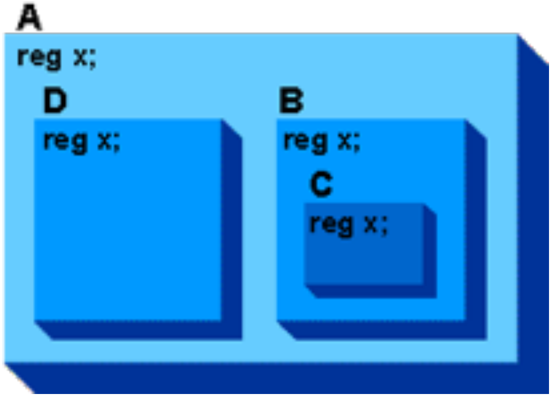
- Verilog has a mechanism for globally referencing nets, registers, events, tasks, and functions called the cross-module reference, or XMR. This is in marked contrast to VHDL, which rejected the concept.
- Cross-module references, or hierarchical references as they are sometimes called, can take several different forms:
  - References to a Different Scope within a Module
  - References between Modules
  - Downward Reference
  - Upward Reference

Hierarchical Module

- There is a static scope within each module definition with which one can locate any identifier. For example, in the following,

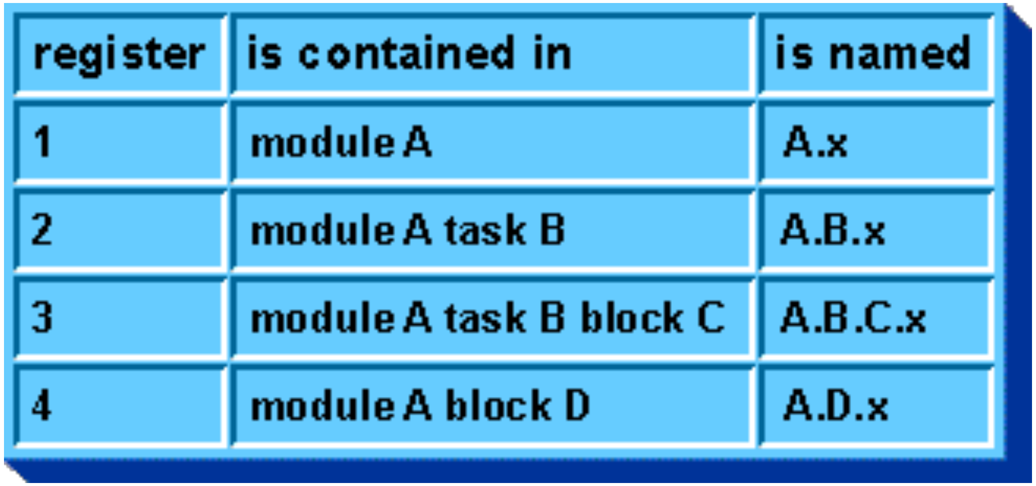
```
module A;
reg x; // 1
...
task B;
reg x; // 2
  begin
  ...
    begin: C
    reg x; // 3
    ...
    end
  end
endtask
```

```
initial
begin: D
   reg x; // 4
   ...
end
endmodule
```



Reference to Scopes within Module

- there is a module, a task, and two named blocks. There are four distinct registers, each named x within its local scope.



register	is contained in	is named
1	module A	A.x
2	module A task B	A.B.x
3	module A task B block C	A.B.C.x
4	module A block D	A.D.x

Coding Styles

Memory

- The following are examples of memory declarations.

```
reg [7:0] memdata[0:255];// 256 8-bit registers
reg [8*6:1] strings[1:10];// 10 6-byte strings
reg membits [1023:0];// 1024 1-bit registers
```

- The maximum size of a memory is implementation-dependent, but is at least 2^24 (16,777,216) elements.

Access to Memory

- A memory element is accessed by means of a memory index operation. A memory index looks just like a bit-select:

```
    mem[index]
```

- Another limitation on memory access is that you can't take a bit-select or part-select of a memory element. Thus, if you want to get the 3rd bit out of the 10th element of a memory, you need to do it in two steps:

```
 reg [0:31] temp, mem[1:1024];
   ...
   temp = mem[10];
   bit = temp[3];
```

Finite State Machine

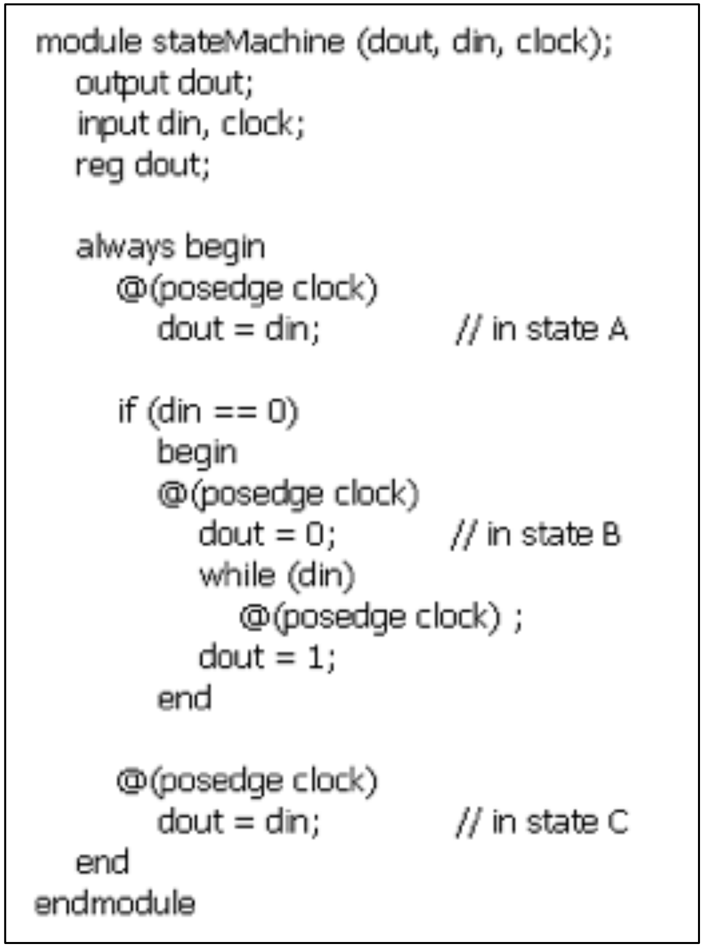
- There are two common variations of state machines, Mealy and Moore machines.
    - Mealy machines produce outputs based on both current state and input.
    - Moore machines produce outputs based only on the current state. As you would expect, the Verilog representation of the two types is very similar.
- Typically, the clock is used to change the state based on the inputs which have been seen up to that point. It is often convenient to think of all the activity of the state machine as taking place on the clock edge:
    - sample inputs
    - compute next state
    - compute outputs
    - change state
    - produce outputs

Finite State Machine

- Finite state machines are one of the common types of logic designed using Verilog. There are several ways to represent them:
    - Implicit
    - Explicit
- State machines always have inputs, a state variable or set of variables (sometimes called a state vector), and a clock. The clock does not have to be periodic, but there must be some strobe signal which indicates when the state transition decision should be made.

Implicit Coding

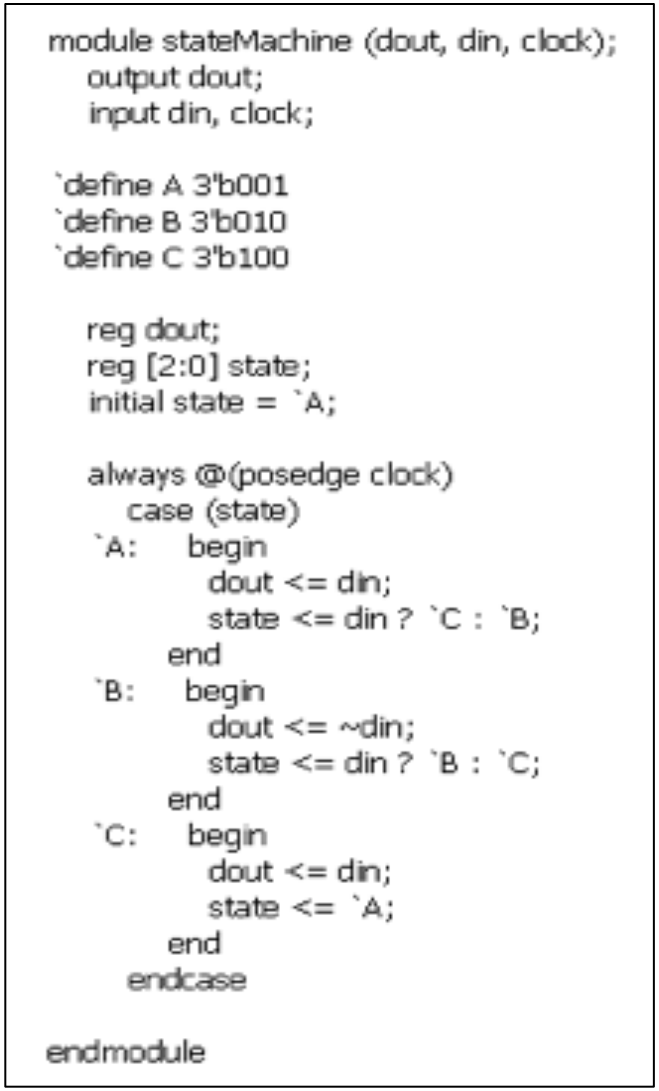
An implicit FSM is simply one whose state encoding is done by means of procedural code. In essence, the program counter is the current state variable.



```
module stateMachine (dout, din, clock);
  output dout;
  input din, clock;
  reg dout;

  always begin
    @(posedge clock)
      dout = din;          // in state A

    if (din == 0)
      begin
      @(posedge clock)
        dout = 0;          // in state B
        while (din)
          @(posedge clock) ;
        dout = 1;
      end

    @(posedge clock)
      dout = din;          // in state C
  end
endmodule
```

Explicit Coding

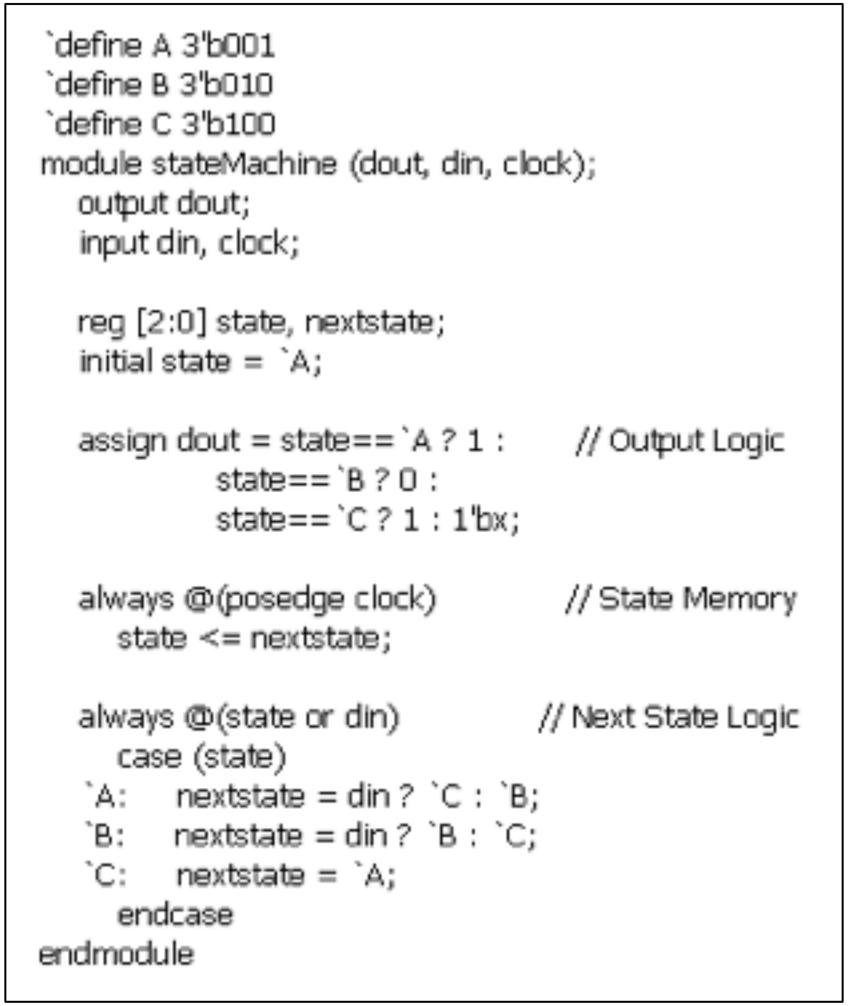
Representing FSMs explicitly is a better style than implicit coding, both because the code maps well to a state transition table and also because explicit representation is synthesizable.



```
module stateMachine (dout, din, clock);
   output dout;
   input din, clock;

`define A 3'b001
`define B 3'b010
`define C 3'b100

   reg dout;
   reg [2:0] state;
   initial state = `A;

   always @(posedge clock)
      case (state)
    `A:     begin
              dout <= din;
              state <= din ? `C : `B;
            end
    `B:     begin
              dout <= ~din;
              state <= din ? `B : `C;
            end
    `C:     begin
              dout <= din;
              state <= `A;
            end
      endcase

endmodule
```

Explicit Coding

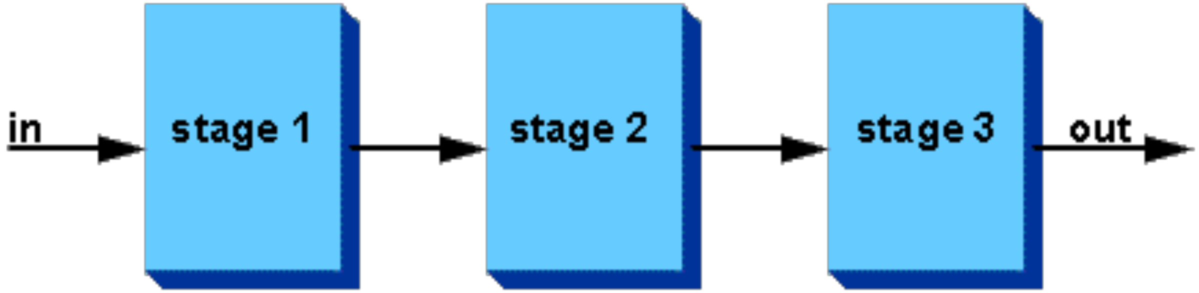
The following is an example of using an always block for next state logic. This style is probably more common, but it is really no different than the first version.



```
`define A 3'b001
`define B 3'b010
`define C 3'b100
module stateMachine (dout, din, clock);
   output dout;
   input din, clock;

   reg [2:0] state, nextstate;
   initial state = `A;

   assign dout = state==`A ? 1 :        // Output Logic
                 state==`B ? 0 :
                 state==`C ? 1 : 1'bx;

   always @(posedge clock)              // State Memory
      state <= nextstate;

   always @(state or din)               // Next State Logic
      case (state)
    `A:    nextstate = din ? `C : `B;
    `B:    nextstate = din ? `B : `C;
    `C:    nextstate = `A;
      endcase
endmodule
```

Pipeline

- Pipelines, queues, and FIFOs are common logic structures which are all related, in the sense that data moves from one storage location to another synchronously, based on a strobe signal, usually a clock.



Pipeline Coding

```
module pipeline (out, in, clock);
output out;
input in, clock;
reg out, pipe[1:2];
always @(posedge clock)
    begin
    out = pipe[2];
    pipe[2] = pipe[1];
    pipe[1] = in;
    end
endmodule
```

- This code works fine. The only potential problem is that out changes value on the clock edge, so whatever takes it as an input may get the wrong value.

Pipeline Coding

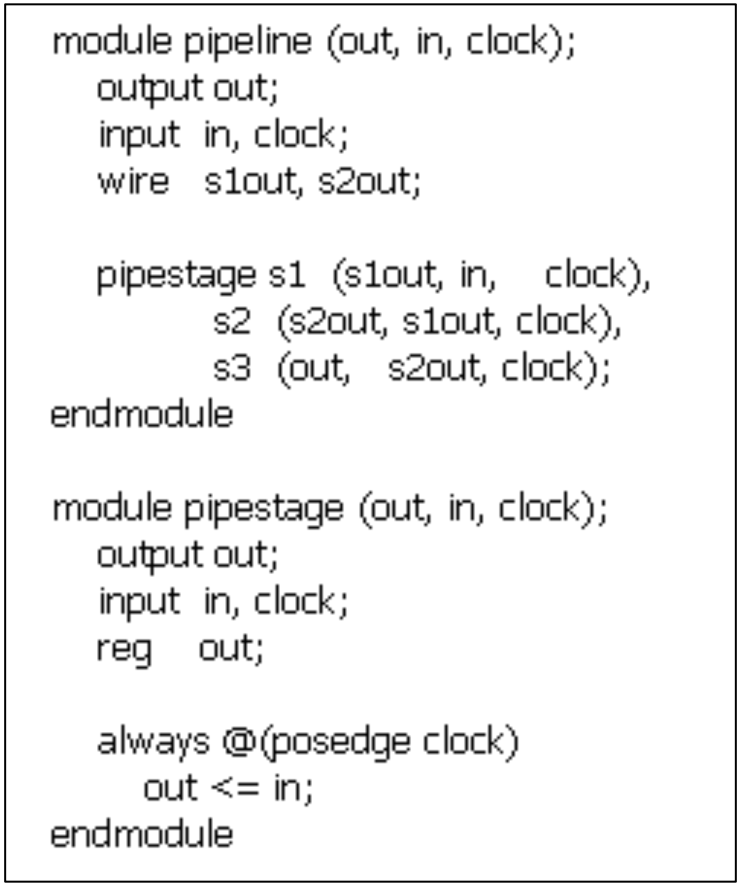
- A better version would be to use a non-blocking assign:

```
always @(posedge clock)
begin
out <= pipe[2];
pipe[2] <= pipe[1];
pipe[1] <= in;
end
```

- Note that with the non-blocking assign, the order of the assignment statements is irrelevent.

Pipe Stage as Separate Module

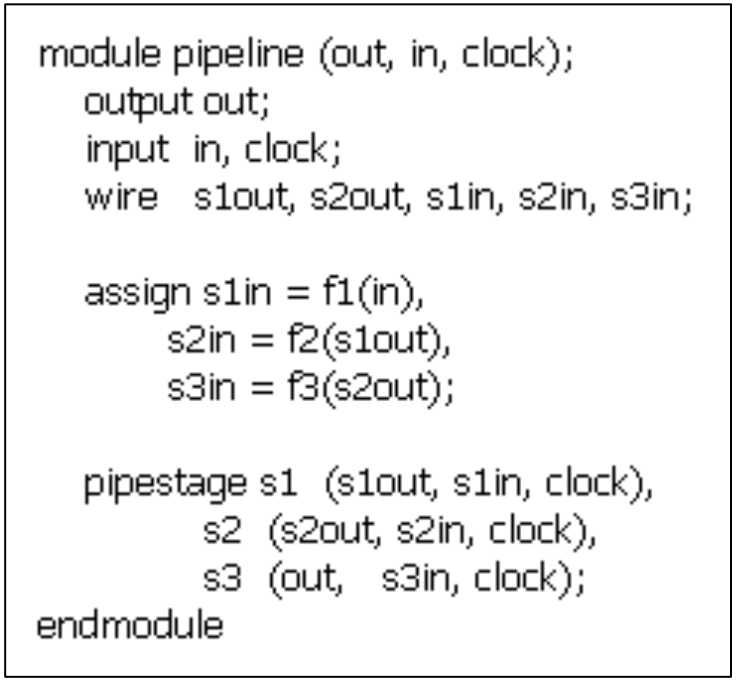
It is common to make a single pipe stage module and use it repetitively, as follows:



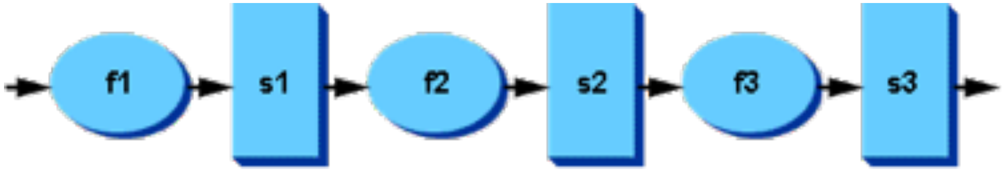
```
module pipeline (out, in, clock);
   output out;
   input  in, clock;
   wire   s1out, s2out;

   pipestage s1  (s1out, in,    clock),
             s2  (s2out, s1out, clock),
             s3  (out,   s2out, clock);
endmodule

module pipestage (out, in, clock);
   output out;
   input  in, clock;
   reg    out;

   always @(posedge clock)
      out <= in;
endmodule
```

Combinational Logic in Pipeline

It is more interesting if there is some combinational logic associated with each pipe stage. Suppose each stage has some logic represented by a function f1, f2, f3 which is applied to the input.



```
module pipeline (out, in, clock);
   output out;
   input  in, clock;
   wire   s1out, s2out, s1in, s2in, s3in;

   assign s1in = f1(in),
          s2in = f2(s1out),
          s3in = f3(s2out);

   pipestage s1  (s1out, s1in, clock),
             s2  (s2out, s2in, clock),
             s3  (out,   s3in, clock);
endmodule
```



Race Condition

- The implication of all this is that you had better not write Verilog code which has a different result depending on the order of execution of simultaneous, unordered events. This is known generally as a race condition, and it occurs when one event samples a data value, another event changes the data value, and the two events are unordered with respect to each other.
- Example:

```
always @(posedge clock) dff1 = f(x);
always @(posedge clock) dff2 = dff1;
```

- This attempt at a pipeline doesn't work, because the value of dff2 may be either the old or the new value of dff1