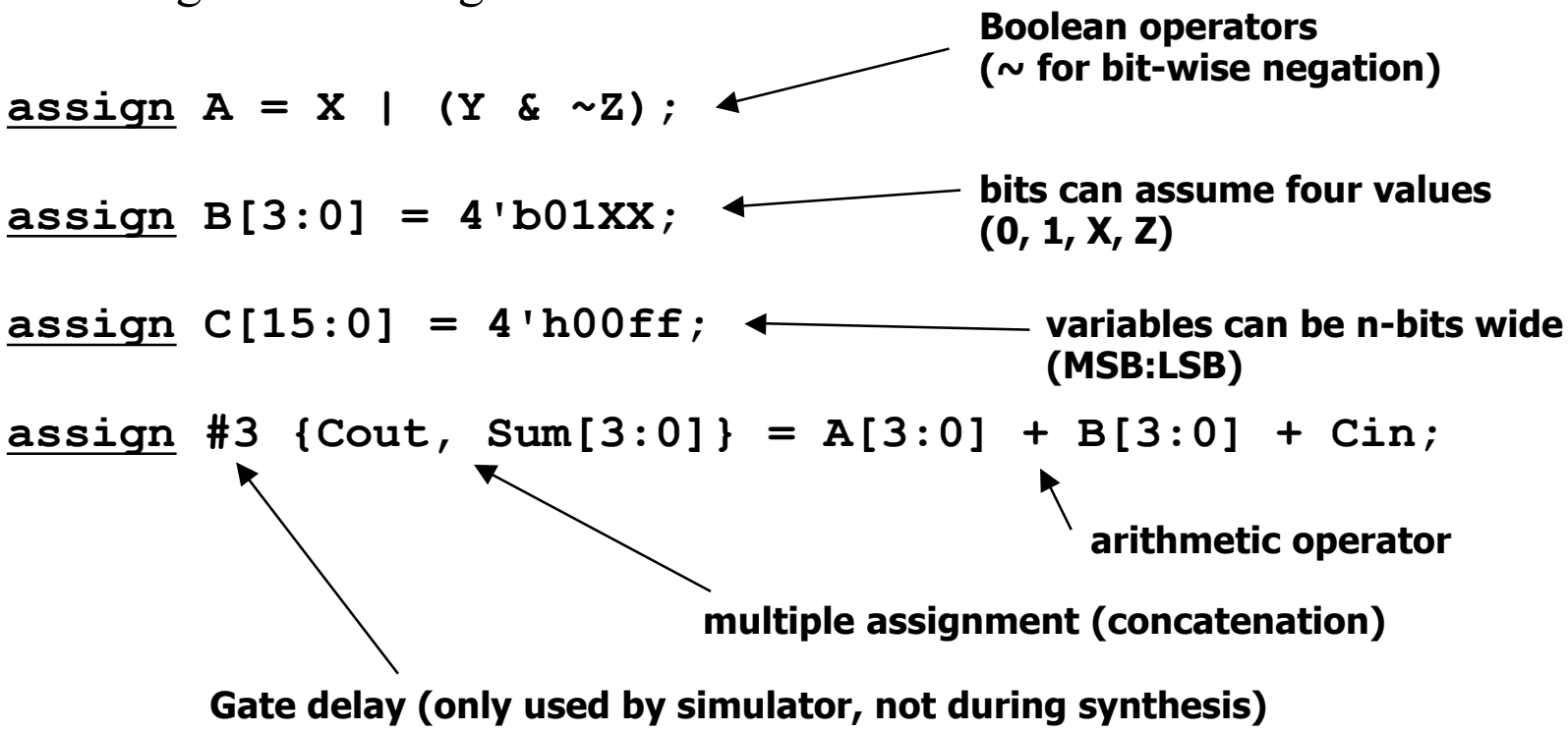
Variables

- wire
  - Connects components together
- reg
  - Saves a value
    - Part of a behavioral description
  - Does *NOT* necessarily become a register when you synthesize
    - May become a wire
- The rule
  - Declare a variable as reg if it is a target of an assignment statement
    - Continuous assign doesn't count

Continuous assignment

- Assignment is continuously evaluated
  - Corresponds to a connection or a simple component
  - Target is not a reg variable



Example: A comparator

```
module Compare1 (A, B, Equal, Alarger, Blarger);
  input     A, B;
  output    Equal, Alarger, Blarger;

  assign Equal = (A & B) | (~A & ~B);
  assign Alarger = (A & ~B);
  assign Blarger = (~A & B);
endmodule
```

Comparator example (con't)

```
// Make a 4-bit comparator from 4 1-bit comparators

module Compare4(A4, B4, Equal, Alarger, Blarger);
  input [3:0] A4, B4;
  output Equal, Alarger, Blarger;
  wire e0, e1, e2, e3, Al0, Al1, Al2, Al3, Bl0, Bl1, Bl2, Bl3;

  Compare1 cp0(A4[0], B4[0], e0, Al0, Bl0);
  Compare1 cp1(A4[1], B4[1], e1, Al1, Bl1);
  Compare1 cp2(A4[2], B4[2], e2, Al2, Bl2);
  Compare1 cp3(A4[3], B4[3], e3, Al3, Bl3);

  assign Equal = (e0 & e1 & e2 & e3);
  assign Alarger = (Al3 | (Al2 & e3) |
                    (Al1 & e3 & e2) |
                    (Al0 & e3 & e2 & e1));
  assign Blarger = (~Alarger & ~Equal);
endmodule
```

always block

- A procedure that describes a circuit's function
  - Can contain multiple statements
  - Can contain *if, for, case*
  - Statements execute sequentially
    - Continuous assignments execute in parallel
- *begin/end* groups statements

always example

- *always* triggers at the specified conditions
  - Example: A D-type register

```
module register(Q, D, clock);
  input   D, clock;
  output  Q;
  reg     Q;

  always @(posedge clock) begin
     Q = D;
  end
endmodule
```

always example

```
module and_gate(out, in1, in2);
  input   in1, in2;
  output  out;
  reg     out;

  always @(in1 or in2) begin
     out = in1 & in2;
  end
endmodule
```

Not a real register!!
Holds assignment in always block

The compiler will not use a register when it synthesizes this gate, because *out* changes whenever the inputs change. Can omit the module and write

```
wire out, in1, in2;
and (out, in1, in2);
```

specifies when block is executed
i.e. triggered by changes in in1 or in2

Incomplete trigger or incomplete assignment

- What if you omit an input trigger (e.g. *in*2)
  - Compiler will insert a register to hold the state
    - Becomes a sequential circuit — *NOT* what you want

```
module and_gate (out, in1, in2);
  input          in1, in2;
  output         out;
  reg            out;

  always @(in1) begin
     out = in1 & in2;
  end

endmodule
```

2 rules:

1) Include all inputs in the trigger list

2) Use complete assignments
   - Every path must lead to an assignment for *out*
   - Otherwise *out* needs a state element

A better way...

- Use functions for complex combinational logic
  - Functions can't have state

```
module and_gate (out, in1, in2);
  input          in1, in2;
  output         out;

  assign out = myfunction(in1, in2);
  function myfunction;
    input in1, in2;
    begin
      myfunction = in1 & in2;
    end
  endfunction

endmodule
```

Benefits:

Functions force a result

- Compiler will fail if function does not generate a result

If you build a function wrong, the circuit will not synthesize

- If you build an always block wrong, you get a register

if

- Same as C *if* statement
  - Single *if* statements synthesize to multiplexers
  - Nested *if* /*else* statements usually synthesize to logic

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;       // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;                 // target of assignment

  always @(sel or A or B or C or D)
    if (sel == 2'b00) Y = A;
    else if (sel == 2'b01) Y = B;
    else if (sel == 2'b10) Y = C;
    else if (sel == 2'b11) Y = D;

endmodule
```

if: Another way

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;       // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;                 // target of assignment

  always @(sel or A or B or C or D)
    if (sel[0] == 0)
      if (sel[1] == 0)  Y = A;
      else              Y = B;
    else
      if (sel[1] == 0)  Y = C;
      else              Y = D;
endmodule
```

case

- Sequential execution
  - Executes only first case that matches (don't need a break)
  - *case* statements synthesizes to multiplexers

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel; // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;                    // target of assignment

  always @(sel or A or B or C or D)
    case (sel)
      2'b00: Y = A;
      2'b01: Y = B;
      2'b10: Y = C;
      2'b11: Y = D;
    endcase
endmodule
```

case: A better way

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel; // 2-bit control signal
input A, B, C, D;
output Y;

 assign out = mymux(sel, A, B, C, D);
 function mymux;
    input [1:0] sel, A, B, C, D;
    begin
         case (sel)
           2'b00: mymux = A;
           2'b01: mymux = B;
           2'b10: mymux = C;
           2'b11: mymux = D;
        endcase
    end
 endfunction
endmodule
```

Note: You can define a function in a file
Then *include* it into your Verilog module

default case

```
// Simple binary encoder (input is 1-hot)
module encode (A, Y);
input  [7:0] A;           // 8-bit input vector
output [2:0] Y;           // 3-bit encoded output
reg    [2:0] Y;           // target of assignment

  always @(A)
    case (A)
      8'b00000001: Y = 0;
      8'b00000010: Y = 1;
      8'b00000100: Y = 2;
      8'b00001000: Y = 3;
      8'b00010000: Y = 4;
      8'b00100000: Y = 5;
      8'b01000000: Y = 6;
      8'b10000000: Y = 7;
      default:  Y = 3'bx; // Don't care when input isnt 1-hot
    endcase
endmodule
```

If you omit the *default*, the compiler will create a latch for Y

- Or you can list all 256 cases

Better way: Use a function

- Compiler will warn you of missing cases

case (con't)

```
// Priority encoder
module encode (A, Y);
input  [7:0] A;          // 8-bit input vector
output [2:0] Y;          // 3-bit encoded output
reg    [2:0] Y;          // target of assignment

  always @(A)
    case (1'b1)
      A[0]: Y = 0;
      A[1]: Y = 1;
      A[2]: Y = 2;
      A[3]: Y = 3;
      A[4]: Y = 4;
      A[5]: Y = 5;
      A[6]: Y = 6;
      A[7]: Y = 7;
      default:  Y = 3'bx; // Don't care when input is all 0's
    endcase
endmodule
```

Case statements execute sequentially

- Take the first alternative that matches

casez and *casex*

```
// 2-bit priority encoder
module encode (A, Y);
input  [7:0] A;           // 8-bit input vector
output [1:0] Y;           // 3-bit encoded output
reg    [1:0] Y;           // target of assignment

  always @(A)
    casez (A)
      8'bzzzz0001: Y = 0;
      8'bzzzz0010: Y = 1;
      8'bzzzz0100: Y = 2;
      8'bzzzz1000: Y = 3;
      default:  Y = 2'bx; // Don't care when input isnt 1-hot
    endcase
endmodule
```

casez: alternatives can include *z*

- *z* bits are not used in the evaluation

casex example

```
// Priority encoder
module encode (A, valid, Y);
input  [7:0] A;              // 8-bit input vector
output [2:0] Y;              // 3-bit encoded output
output valid;                // Asserted when an input is not all 0's
reg    [2:0] Y;              // target of assignment
reg    valid;

  always @(A) begin
    valid = 1;
    casex (A)
      8'bXXXXXXX1: Y = 0;
      8'bXXXXXX10: Y = 1;
      8'bXXXXX100: Y = 2;
      8'bXXXX1000: Y = 3;
      8'bXXX10000: Y = 4;
      8'bXX100000: Y = 5;
      8'bX1000000: Y = 6;
      8'b10000000: Y = 7;
      default:  begin
         valid = 0;
         Y = 3'bx;  // Don't care when input is all 0's
      end
    endcase
  end
endmodule
```

casex: alternatives can include ***x*** and ***z***

- ***x*** and ***z*** bits are not used in the evaluation

for

```
// simple encoder
module encode (A, Y);
input  [7:0] A;           // 8-bit input vector
output [2:0] Y;           // 3-bit encoded output
reg    [2:0] Y;           // target of assignment

integer i;                // Temporary variables for program only
reg [7:0] test;

  always @(A) begin
    test = 8b'00000001;
    Y = 3'bx;
    for (i = 0; i < 8; i = i + 1) begin
       if (A == test) Y = i;
       test = test << 1;  // Shift left, pad with 0s
    end
  end
endmodule
```

for statements synthesize as cascaded combinational logic

- Verilog unrolls the loop

Sequential Verilog

- Sequential circuits: Registers & combinational logic
  - We will use positive edge-triggered registers
    - Avoid latches and negative edge-triggered registers
- Register is triggered by "posedge clk"

```
module register(Q, D, clock);
  input   D, clock;
  output  Q;
  reg     Q;

  always @(posedge clock) begin
     Q = D;
  end
endmodule
```

8-bit register with synchronous reset

```
module reg8 (Q, reset, CLK, D);
 input    reset;
 input    CLK;
 input    [7:0] D;
 output   [7:0] Q;
 reg      [7:0] Q;

   always @(posedge CLK)
     if (reset)
       Q = 0;
     else
       Q = D;

endmodule    // reg8
```

N-bit register with asynchronous reset

```
module regN (Q, reset, CLK, D);
 input      reset;
 input      CLK;
 parameter N = 8;      // Allow N to be changed
 input      [N-1:0] D;
 output     [N-1:0] Q;
 reg        [N-1:0] Q;

    always @(posedge CLK or posedge reset)
      if (reset)
        Q = 0;
      else if (CLK == 1)
        Q = D;

endmodule  // regN
```

Parameters are constants

- Not variables
- Parameter values are inserted at compile time

Shift-register

```
// 8-bit register can be cleared, loaded, shifted left
// Retains value if no control signal is asserted

module shiftReg (CLK, clr, shift, ld, Din, SI, Dout);
input            CLK;
input            clr;             // clear register
input            shift;           // shift
input            ld;              // load register from Din
input    [7:0]   Din;             // Data input for load
input            SI;              // Input bit to shift in
output   [7:0]   Dout;
reg      [7:0]   Dout;

  always @(posedge CLK) begin
    if (clr)              Dout <= 0;
    else if (ld)          Dout <= Din;
    else if (shift)       Dout <= { Dout[6:0], SI };
  end

endmodule                 // shiftReg
```

Blocking and non-blocking assignments

- Blocking assignments (Q = A)
  - Variable is assigned immediately
    - New value is used by subsequent statements
- Non-blocking assignments (Q <= A)
  - Variable is assigned after all scheduled statements are executed
    - Value to be assigned is computed but saved for later
  - Usual use: Register assignment
    - Registers simultaneously take new values after the clock edge
- Example: Swap

```
always @(posedge CLK)
    begin
        temp = B;
        B = A;
        A = temp;
    end
```

```
always @(posedge CLK)
    begin
        A <= B;
        B <= A;
    end
```

Swap (cont'd)

- The following code executes incorrectly
  - One block executes first
  - Loses previous value of variable

```
always @(posedge CLK)
    begin
        A = B;
    end
```

```
always @(posedge CLK)
    begin
        B = A;
    end
```

- Non-blocking assignment fixes this
  - Both blocks are scheduled by posedge CLK

```
always @(posedge CLK)
    begin
        A <= B;
    end
```

```
always @(posedge CLK)
    begin
        B <= A;
    end
```

Non-blocking assignment

- Non-blocking assignment is also known as RTL assignment
  - If used in an *always* block triggered by a clock edge
  - Mimics register-transfer–level semantics: All flip-flops change together

```
// this implements 3 parallel flip-flops
always @(posedge clk)
   begin
      B = A;
      D = C;
      F = E;
   end
```

```
// this implements a shift register
always @(posedge clk)
   begin
      {D, C, B} = {C, B, A};
   end
```

```
// this implements a shift register
always @(posedge clk)
   begin
      B <= A;
      C <= B;
      D <= C;
   end
```

Counter

```
// 8-bit counter with clear and count enable controls
module count8 (CLK, clr, cntEn, Dout);
 input         CLK;
 input         clr;          // clear counter
 input         cntEn;        // enable count
 output [7:0]  Dout;         // counter value
 reg    [7:0]  Dout;

  always @(posedge CLK)
    if (clr)           Dout <= 0;
    else if (cntEn)    Dout <= Dout + 1;

endmodule
```