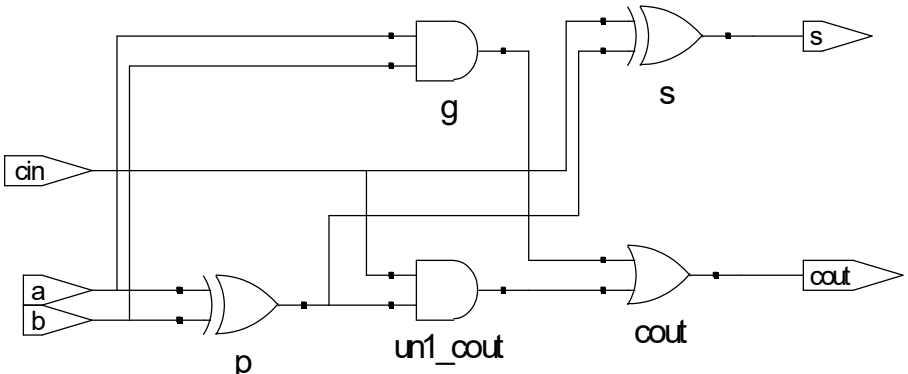
Lecture 2: Continuation of SystemVerilog

UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

Adder Examples

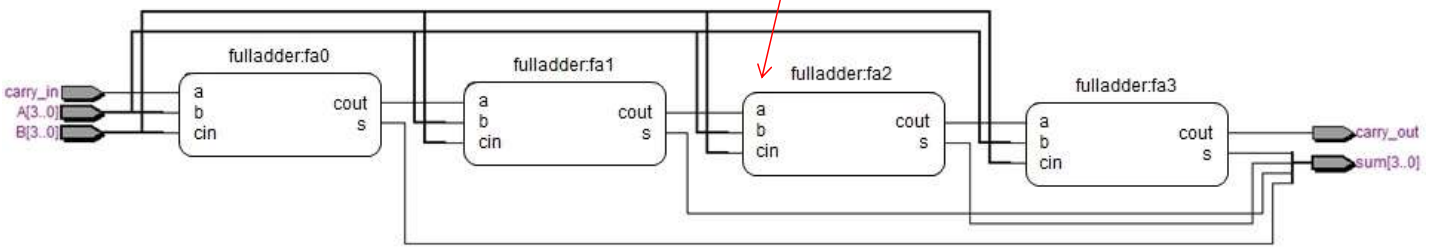
```
module fulladder(input logic a, b, cin,
                 output logic s, cout);
  logic p, g;   // internal nodes

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

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



Adder Examples

```
/* hierarchical 4-bit adder */
module h4ba(input logic [3:0] A, B,
            input logic carry_in,
           output logic [3:0] sum,
           output logic carry_out);

  logic carry_out_0, carry_out_1, carry_out_2; // internal signals

  fulladder fa0 (A[0], B[0], carry_in, sum[0], carry_out_0);
  fulladder fa1 (A[1], B[1], carry_out_0, sum[1], carry_out_1);
  fulladder fa2 (A[2], B[2], carry_out_1, sum[2], carry_out_2);
  fulladder fa3 (A[3], B[3], carry_out_2, sum[3], carry_out);

endmodule
```

each of these is an instantiation of “full_adder”

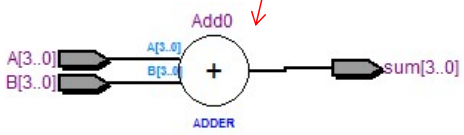


Adder Examples

```
module add4(input logic [3:0] A, B,
            output logic [3:0] sum);
  assign sum = A + B;
endmodule
```

Verilog compilers will replace arithmetic operators with default logic implementations (e.g. ripple carry adder)

this expands into logic for a ripple carry adder



Numbers

Format: N'Bvalue

N = number of bits, **B** = base

N'B is optional but recommended (default is decimal)

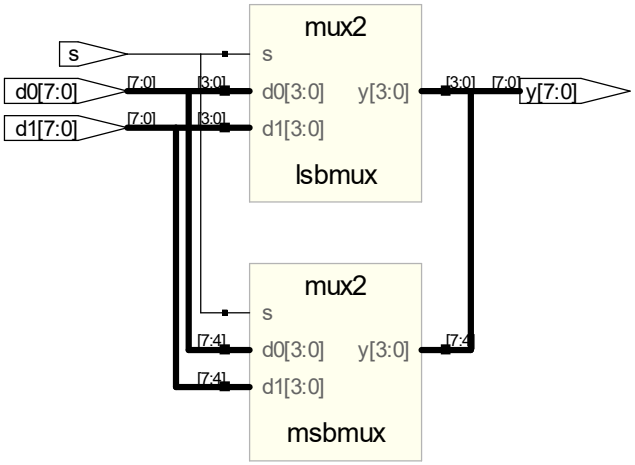
Number	# Bits	Base	Decimal Equivalent	Stored
3'b101	3	binary	5	101
'b11	unsized	binary	3	00…0011
8'b11	8	binary	3	00000011
8'b1010_1011	8	binary	171	10101011
3'd6	3	decimal	6	110
6'o42	6	octal	34	100010
8'hAB	8	hexadecimal	171	10101011
42	unsized	decimal	42	00…0101010

Bit Manipulations: Example 1

```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};

// if y is a 12-bit signal, the above statement produces:
// y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0

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

Bit Manipulations: Example 2

```
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
```



More Examples

```
module ex1(input logic [3:0] X, Y, Z,
           input logic a, cin,
           output logic [3:0] R1, R2, R3, Q1, Q2,
           output logic [7:0] P1, P2,
           output logic t, cout);

  assign R1 = X | (Y & ~Z);

  assign t = &X;

  assign R2 = (a == 1'b0) ? X : Y;

  assign P1 = 8'hff;

  assign P2 = {4{a}, X[3:2], Y[1:0]};

  assign {cout, R3} = X + Y + cin;

  assign Q1 = X << 2;

  assign Q2 = {X[1], X[0], 1'b0, 1'b0};
endmodule
```

← use of bitwise Boolean operators

← example reduction operator

← conditional operator

← example constants

← replication, same as {a, a, a, a}

← example concatenation

← bit shift operator

← equivalent bit shift

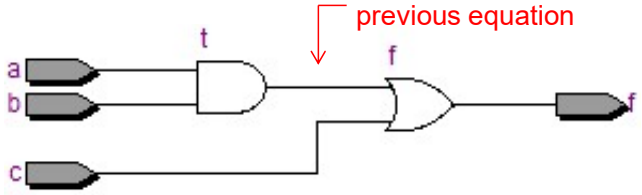
Combinational logic using always

```
module ex2(input logic a, b, c,
           output logic f);

  logic t; // internal signal

  always_comb
  begin
    t = a & b;
    f = t | c;
  end

endmodule
```

should use "**=**" (called "**blocking**" assignment) in comb. logic always statements. RHS just takes output from the previous equation.

The order of statements matters!

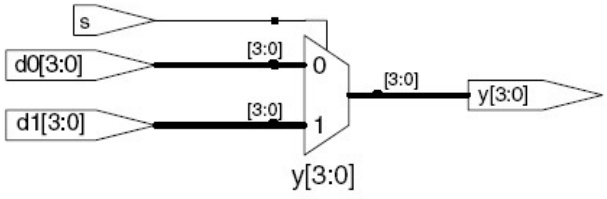
takes output of "t" from previous equation



Combinational logic using always

```
module ex3(input  logic [3:0] d0, d1,
           input  logic       s,
           output logic [3:0] y);

  always_comb
  begin
    if (s)
      y = d1;
    else
      y = d0;
  end
endmodule
```

If-then-else translates into a 2:1 multiplexor



Combinational logic using always

```
/* behaviorial description of a 4-bit adder */
module p4ba(input logic [3:0] A, B,
            input logic carry_in,
           output logic [3:0] sum,
           output logic carry_out);

  logic [4:0] carry; // internal signal

  always_comb
  begin
    carry[0] = carry_in;
    for (int i = 0; i < 4; i++) begin
      sum[i] = A[i] ^ B[i] ^ carry[i];
      carry[i+1] = A[i] & B[i] |
                   A[i] & carry[i] |
                   B[i] & carry[i];
    end
    carry_out = carry[4];
  end

endmodule
```

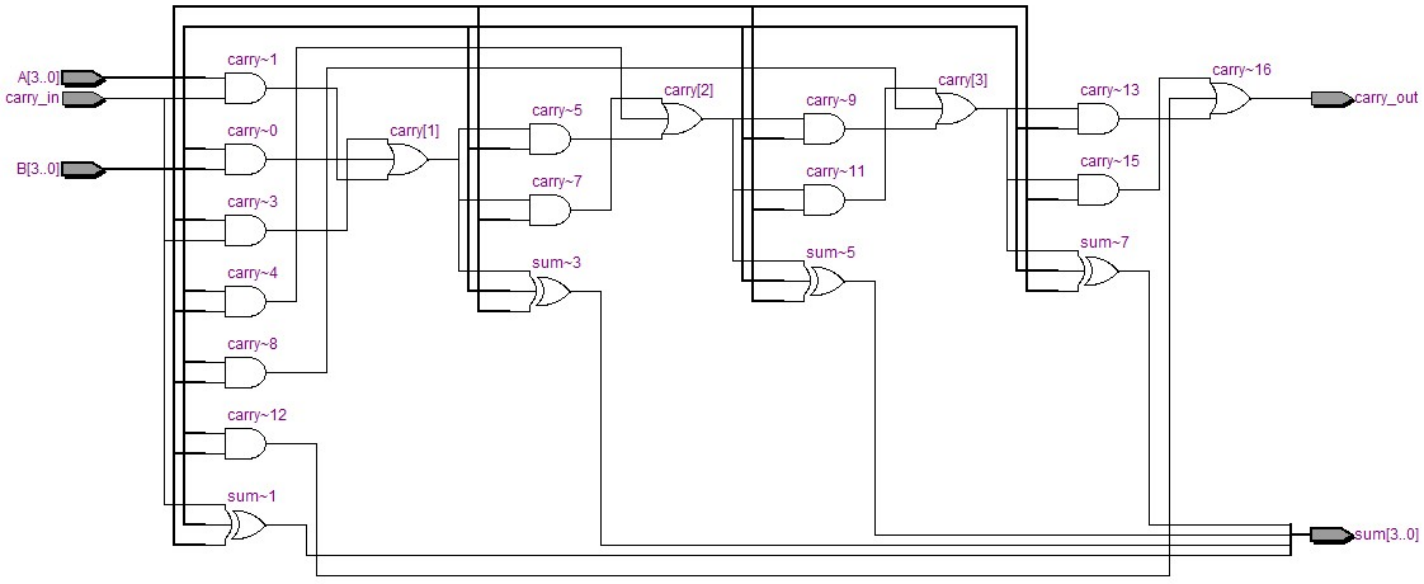
entire "**always_comb**" block is called an "always statement" for combiantional logic

for loops must have a specified range. simply interpreted as "replication".

Note we can declare the loop control variable within the for loop

Verilog calls the use of "=" inside an always statement as a "**blocking**" assignment. all it means is that the Verilog will "parse" the lines of code inside the always block in "sequential" order in the generation of logic. (will make more sense later when we discuss "non-blocking" assignments.)

Combinational logic using always



Combinational logic using case

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case (data)
      //                      abc_defg
      0: segments =       7'b111_1110;
      1: segments =       7'b011_0000;
      2: segments =       7'b110_1101;
      3: segments =       7'b111_1001;
      4: segments =       7'b011_0011;
      5: segments =       7'b101_1011;
      6: segments =       7'b101_1111;
      7: segments =       7'b111_0000;
      8: segments =       7'b111_1111;
      9: segments =       7'b111_0011;
      default: segments = 7'b000_0000; // required
    endcase
endmodule
```

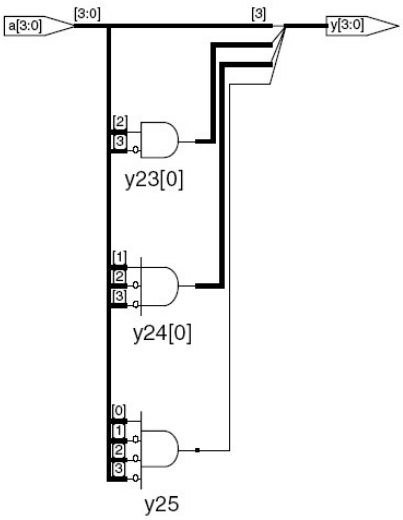
case statement translates into a more complex "multiplexor" similar to if-then-else

Combinational logic using case

- `case` statement implies combinational logic **only if** all possible input combinations described
- Remember to use **`default`** statement
- Otherwise, compiler will create an "asynchronous latch" to remember previous value: **bad** because this is not intended!

Combinational logic using casez

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



Nesting

- In general, for loop, if-then-else, and case statements can be "nested". e.g.,

```
for (…)
   if (…)
      case (…)
      …
      endcase
   else
      …
```

- Compiler will compile from the "inner-most" scope outwards: i.e., it will first produce multiplexor logic for "case" statement, then produce multiplexor logic for the "if-then-else" part, then replicate all that logic based on the number of iterations in the "for loop".

Functions

```
/* adder subtractor */
module add_sub(input logic op,
               input logic [3:0] A, B,
               input logic carry_in,
              output logic [3:0] sum,
              output logic carry_out);
   function logic [4:0] adder(input logic [3:0] x, y,
                              input logic cin);
     logic [3:0] s; // internal signals
     logic c;

     c = cin;
     for (int i = 0; i < 4; i++) begin
       s[i] = x[i] ^ y[i] ^ c;
       c = (x[i] & y[i]) | (c & x[i]) | (c & y[i]);
     end
     adder = {c, s};
   endfunction

   always_comb
     if (op) // subtraction
       {carry_out, sum} = adder(A, ~B, 1);
     else
       {carry_out, sum} = adder(A, B, 0);
endmodule
```

function is like a comb. always statement, but can be called (instantiated) later.