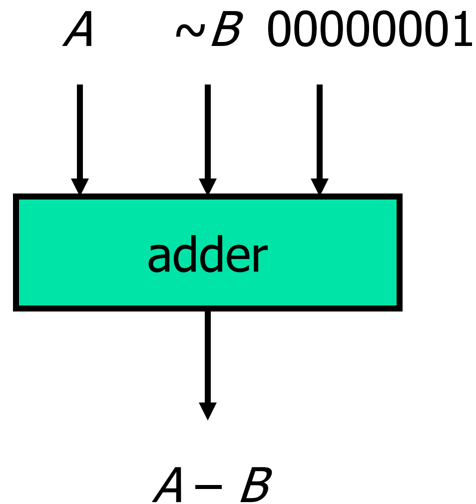
CMPE 415

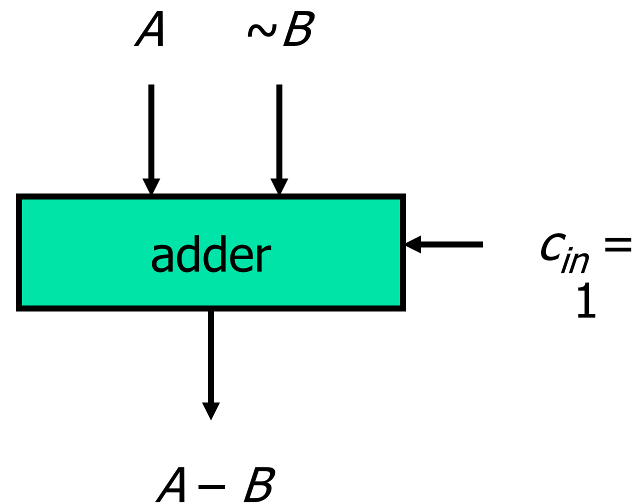
Subtraction

- Essentially the same hardware as an adder
- $A - B = A + (-B)$
- Recall that for 2's complement numbers,
  $-B = (\sim B) + 1$
- So now we have
  $A - B = A + (\sim B) + 1$

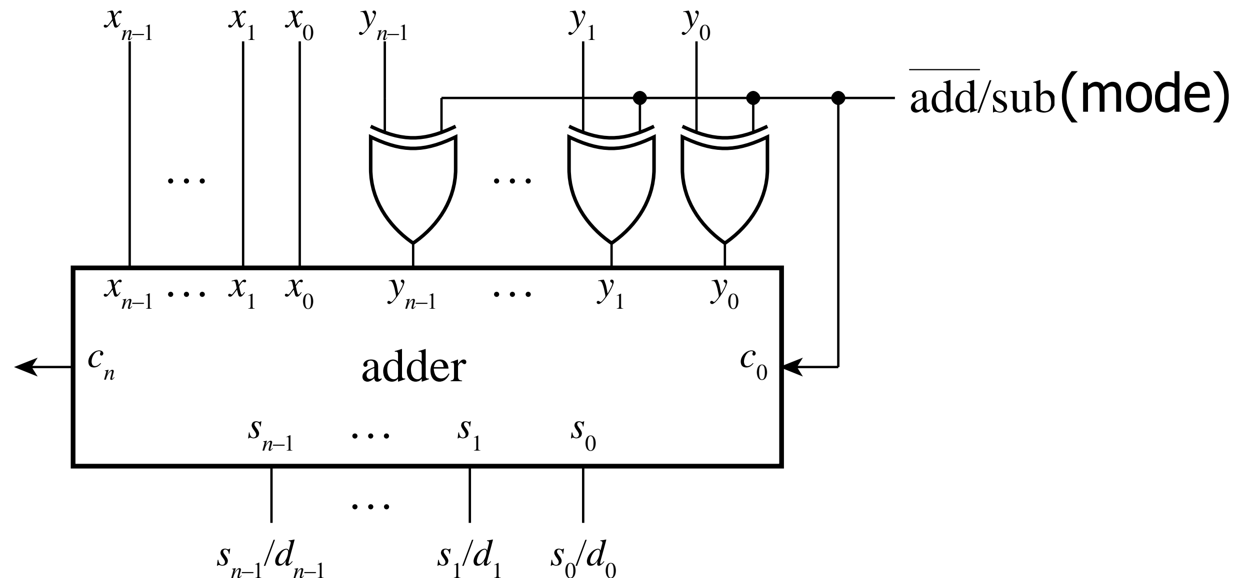
Subtraction

- Often easy to find a place to add in a "1" in the lsb position





Adder/Subtracter Circuits



- Adder can be any of those we have seen
  - depends on constraints

Subtraction in Verilog

```
module adder_subtracter ( output [12:0] s,
                          input  [11:0] x, y,
                          input         mode );
  assign {s} = !mode ? (x + y) : (x - y);
endmodule
```

- When mode=0 => adds
  - else subtracts

Unary Reduction Operator

```
module and4_rtl(y_out,x_in);
  input [3:0]  x_in;
  output y_out;

  assign y_out = & x_in;

endmodule
```

Equivalents:

```
assign y_out =
x_in[4]& x_in[2]&
x_in[1]& x_in[0];

and(y_out,x_in[4],
x_in[2], x_in[1,
x_in[0];
```

Defining Port Connections

```
module Add_half (sum, c_out, a, b);
 ...
```

Module de

```
half_adder m1(w1, w2, a, b);
half_adder m2(sum, w3, w1, c_in);
```

Ordered conn
Order define
Module defi

```
half_adder m1(.sum(w1), .c_out(w2),
              .a(a), .b(b));
half_adder m2(.sum(sum), .c_out(w3),
              .a(w1), .b(c_in));
```

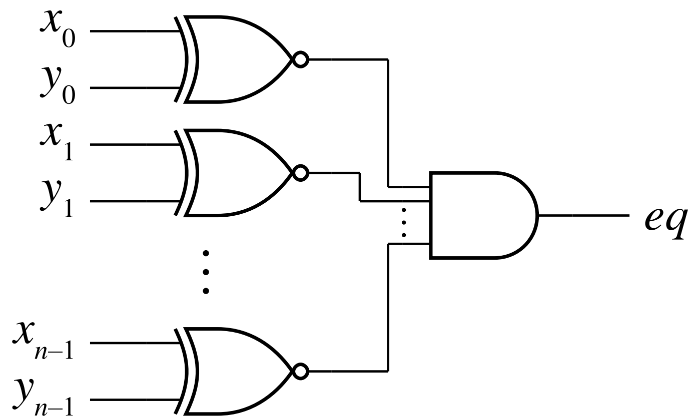
Ports
Expl
name

```
half_adder m1(.a(a), .b(b),
              .sum(w1), .c_out(w2));
```

so ord
doesn'
matter

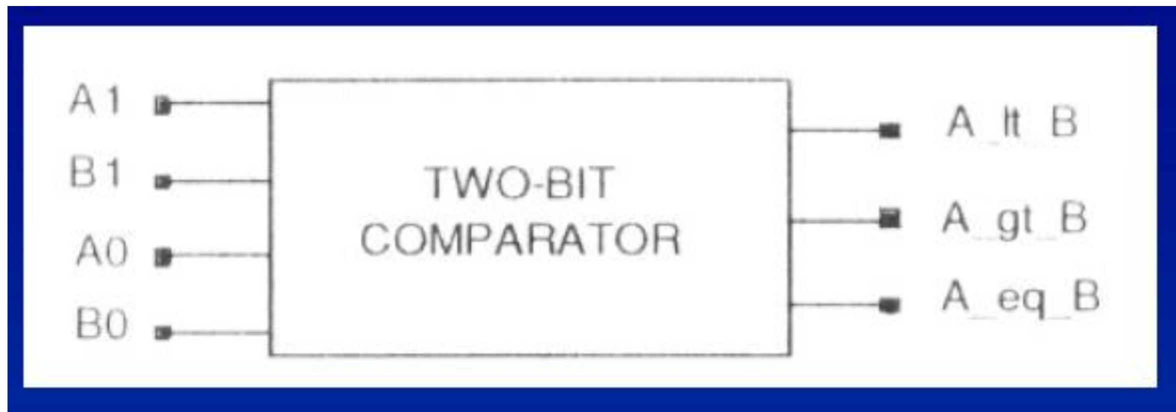
Equality Comparison

- XNOR gate: equality of two bits
  - Apply bitwise to two unsigned numbers



- In Verilog, x == y gives a bit result
  - 1'b0 for false, 1'b1 for true

```
assign eq = x == y;
```


A1
B1
A0
B0
TWO-BIT
COMPARATOR
A_lt_B
A_gt_B
A_eq_B

Long

```
module compare_2_str (A_lt_B, A_gt_B,
A_eq_B, AO, A1, BO, B1);
  input   AO, A1.B0, B1;
  output A_lt_B, A^gLB, A_eq_B;
  wire    w1, w2, w3, w4, w5, w6, w7;
  or (A_lt_B, w1, w2, w3);
  nor (A„gt_B, A_lt_B. A_eq_B);
  and (A.. eq_B, w4, w5);
  and (w1,w6, B1);
  and (w2, w6, w7, BO);
  and (w3, w7, BO, B1);
  not (w6, A1);
  not (w7, AO);
  xnor (w4, A1.B1);
  xnor (w5, AO, BO);
endmodule
```

Cleaner

```
module compare_2a (A_lt_B, A_gt_B, A_eq_B, A1, A0,
B1, B0);
  input A1,A0,B1,B0;
  output A_lt_B, A_gt._B, A..eq_B;
  assign A_lt_B = (~A1) & B1 | (~A1) & (-A0) & B0
| (~A0) & B1 & B0;
  assign A_gt_B = A1 & (-B1) | A0 & (~B1) & (-B0)
| A1 & A0 & (~B0);
  assign A_eq_B = (~A1) & (~A0) & (~B1) & (~B0) |
(-A1) & A0 & (~B1) & B0 | A1 | A0 & B1 & B0 | A1 &
(-A0) & B1 & (-B0);
endmodule
```

Shorter

```
module compare_2b (A_lt_B, A_gt_B, A_eq_B, A1,
AO, B1, BO);
input A1,A0,B1,B0;
output A_lt_B, A_gt_B, A_eq_B;
assign A_lt_B = ({A1 ,A0} < {B1 ,B0});
assign A_gt_B = ({A1 ,A0) > {B1 ,B0});
assign A_eq_B = ({A1 ,A0} == {B1 ,B0});
endmodule
```

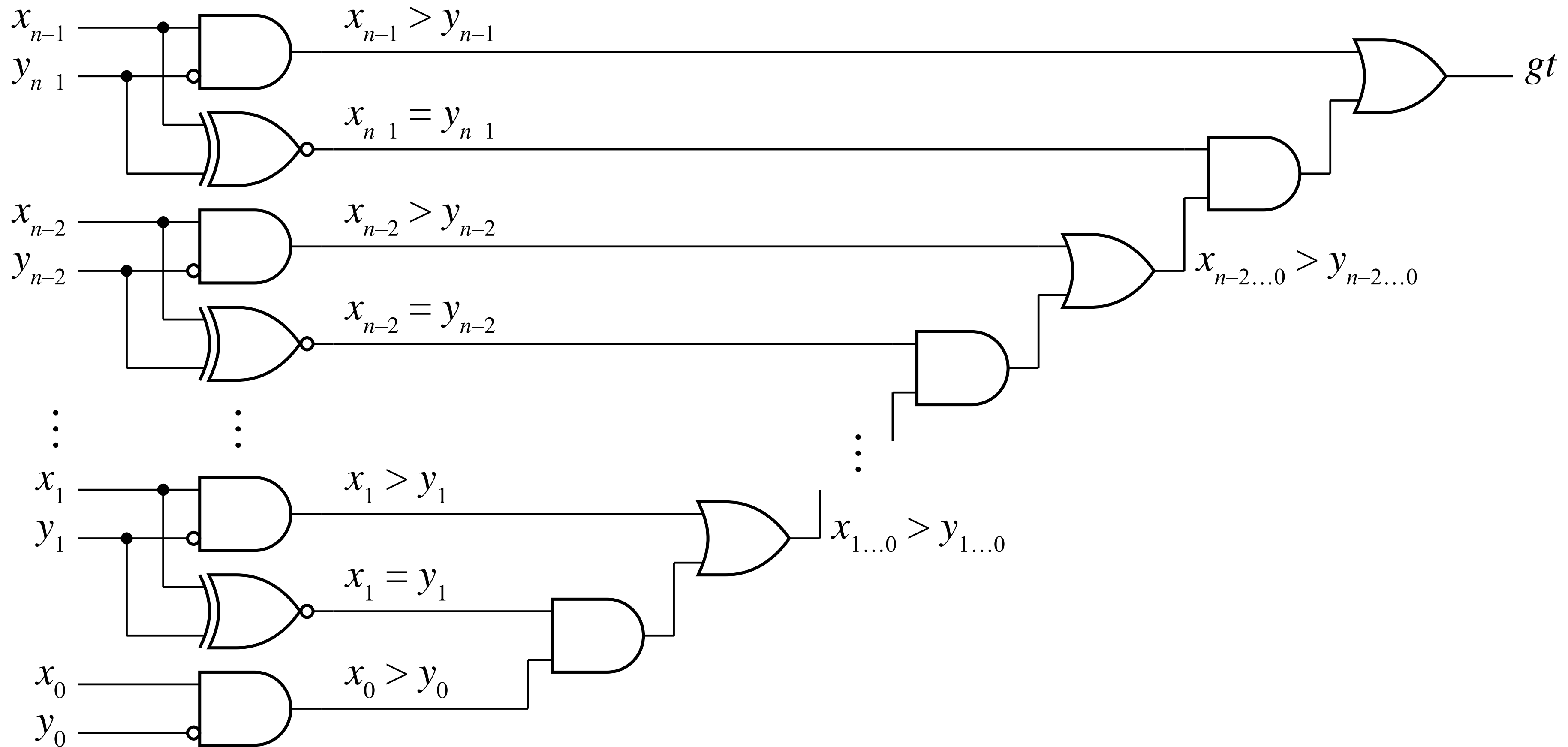
```
module compare_2_ca (A_lt_B, A_gt_B, A_eq_B, A, B);
  input [1:0] A, B;
  output A_lt_B, A_gt_B, A_eq_B;
  assign A_lt_B = (A < B);
  assign A_gt_B = (A > B);
  assign A_eq_B = (A == B);
endmodule
```

"Algorithmic"

```
module compare_2_algo (A_lt_B, A_gt_B,
                       A_eq_B, A, B);
  input   [1:0] A, B;
  output A_lt_B, A_gt_B, A_eq_B;
  reg A_lt_B, A_gt_B, A_eq_B;
  always @ (A or B) // Behavior and event expression
  begin
    A_lt_B = 0;
    A_gt_B = 0;
    A_eq_B = 0;
    if (A==B) A_eq_B = 1;
    else if (A > B) A_gt_B = 1;
    else A_lt_B = 1;
  end
endmodule
```

Inequality Comparison

- Magnitude comparator for x > y



Comparison Example in Verilog

- Thermostat with target termperature
  - Heater or cooler on when actual temperature is more than 5° from target

```
module thermostat ( output       heater_on, cooler_on,
                    input  [7:0] target, actual );
  assign heater_on = actual < target - 5;
  assign cooler_on = actual > target + 5;
endmodule
```

Scaling by Power of 2

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_0 2^0$$

$$2^k x = x_{n-1}2^{k+n-1} + x_{n-2}2^{k+n-2} + \cdots + x_0 2^k + (0)2^{k-1} + \cdots + (0)2^0$$

- This is x shifted left k places, with k bits of 0 added on the right
  - *logical shift left* by k places
  - e.g., $00010110_2 \times 2^3 = 00010110000_2$
- Truncate if result must fit in n bits
  - overflow if any truncated bit is not 0

Scaling by Power of 2

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_0 2^0$$

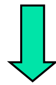
$$x/2^k = x_{n-1}2^{n-1-k} + x_{n-2}2^{n-2-k} + \cdots + x_k 2^0 + x_{k-1}2^{-1} + \cdots + x_0 2^{-k}$$

- This is x shifted right k places, with k bits truncated on the right
  - *logical shift right* by k places
  - e.g., $01110110_2 / 2^3 = 01110_2$
- Fill on the left with k bits of 0 if result must fit in n bits

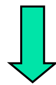
Scaling in Verilog

- Shift-left (<<) and shift-right (>>) operations
  - result is same size as operand

$s = 00010011_2 = 19_{10}$



```
assign y = s << 2;
```



$y = 01001100_2 = 76_{10}$

$s = 00010011_2 = 19_{10}$

```
assign y = s >> 2;
```

$y = 000100_2 = 4_{10}$

Multipliers

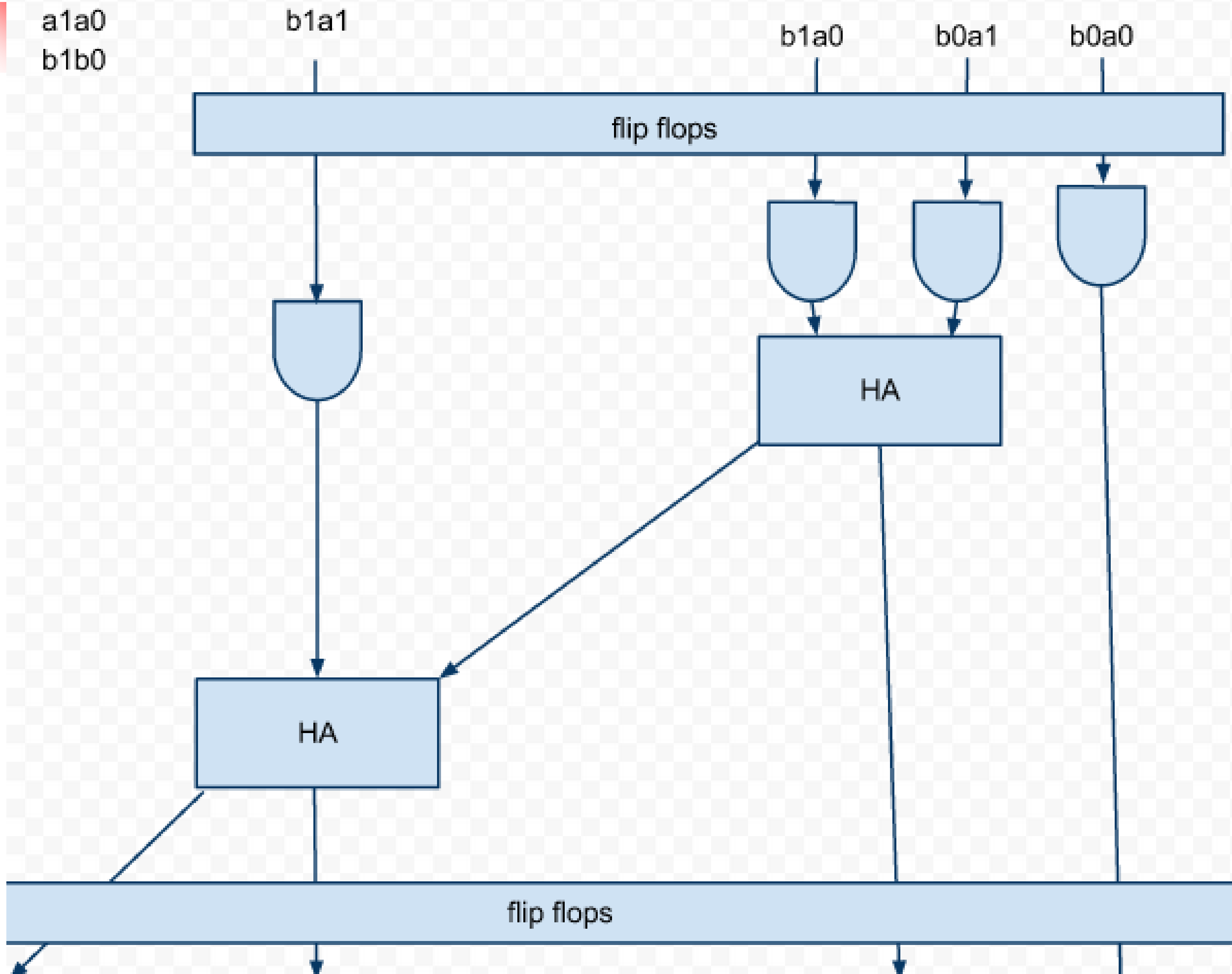
- Widely used in DSP processors, less so in general-purpose processors
- Hardware is typically done the same as you would do it with paper and pencil
- Partial products are copies of the *multiplicand* AND'd by bits of the *multiplier*

<draw picture in notes>

Multipliers 3 Main Steps

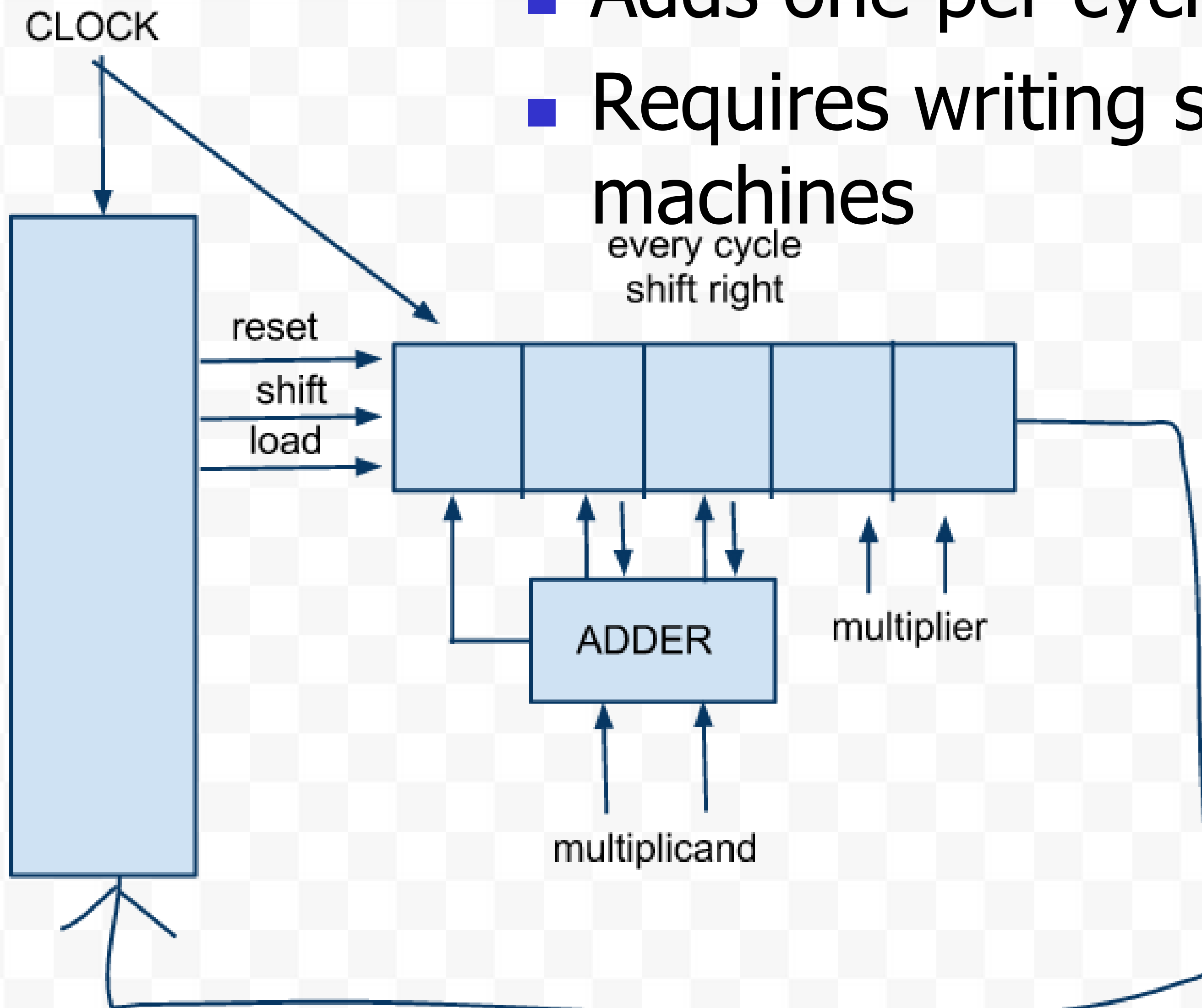
- Generate partial products
- Reduce the partial product array (normally using carry-save addition)
  - Linear array addition
  - Tree addition (Wallace tree)
- Final adder
  - Convert carry-save form to single word form
  - Whichever one you like, probably a faster one
  - Carry-propagate adder (CPA)

Array Multiplier



Serial Multiplier

- Adds one per cycle
- Requires writing state machines



Unsigned Multiplication

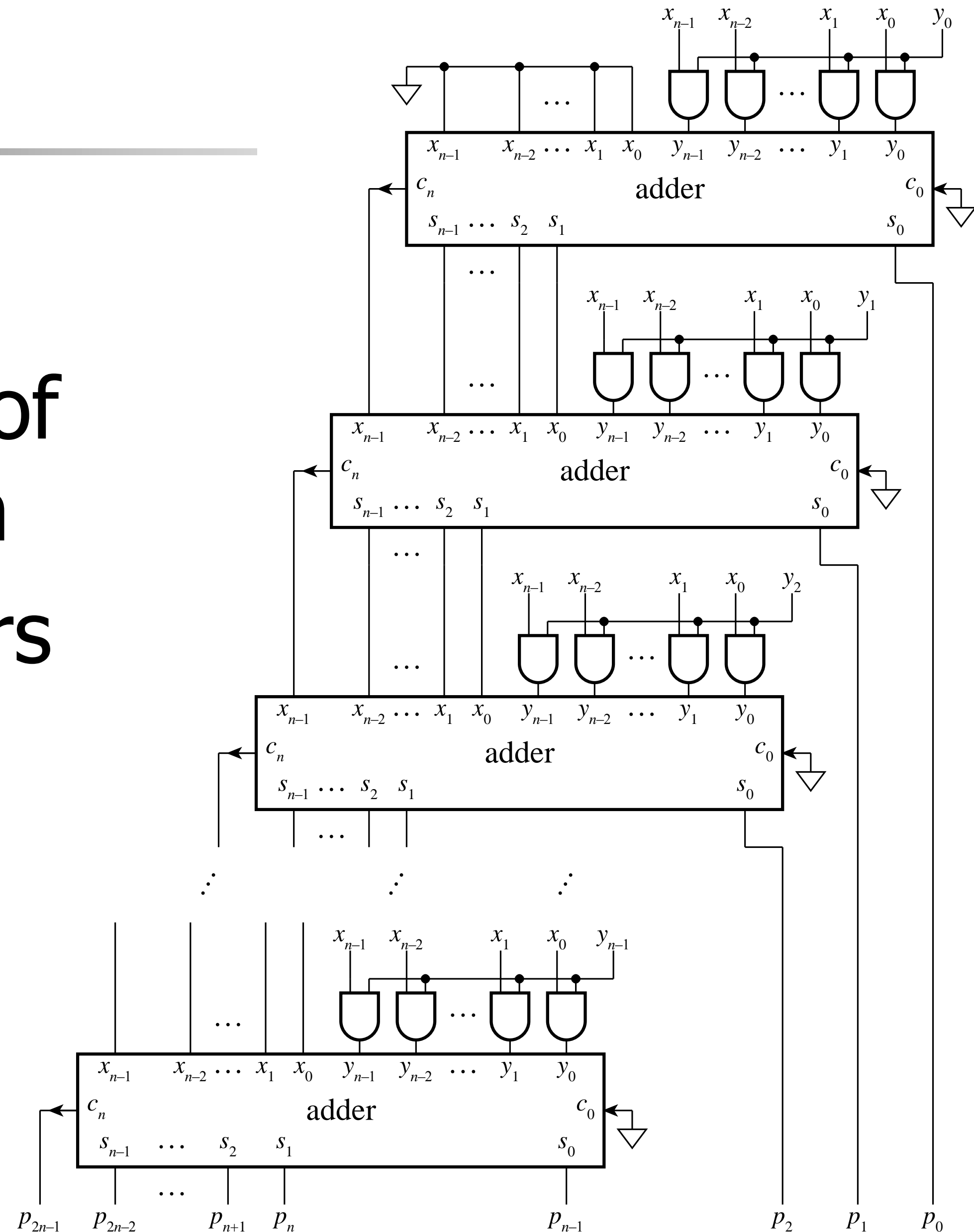
$$xy = x\left(y_{n-1}2^{n-1} + y_{n-2}2^{n-2} + \cdots + y_0 2^0\right)$$

$$= y_{n-1}x2^{n-1} + y_{n-2}x2^{n-2} + \cdots + y_0 x2^0$$

- $y_i x\, 2^i$ is called a partial product
  - if $y_i = 0$, then $y_i x\, 2^i = 0$
  - if $y_i = 1$, then $y_i x\, 2^i$ is x shifted left by i
- Combinational array multiplier
  - AND gates form partial products
  - adders form full product

Unsigned Multiplication

- Adders can be any of those we have seen
- Optimized multipliers combine parts of adjacent adders



Product Size

- Greatest result for n-bit operands:

$$(2^n - 1)(2^n - 1) = 2^{2n} - 2^n - 2^n + 1 = 2^{2n} - \left(2^{n+1} - 1\right)$$

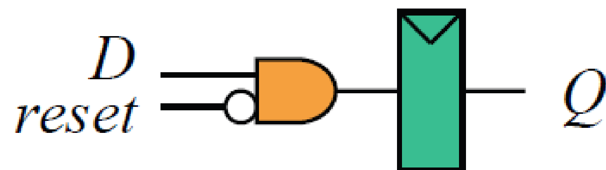
- Requires 2n bits to avoid overflow
- Multiplying n-bit and m-bit operands
  - requires $n + m$ bits

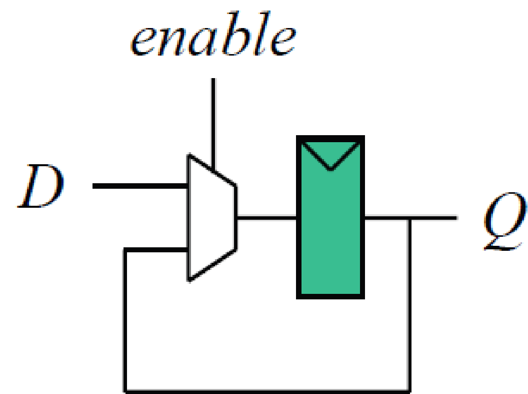
```
wire [ 7:0] x; wire [13:0] y; wire [21:0] p;
...

assign p = {14'b0, x} * {8'b0, y};
```

```
assign p = x * y;  // implicit resizing
```

Reset-able and Enable-able Registers

- Sometimes it is convenient or necessary to have flip-flops with special inputs like *reset* and *enable*
- When designing flip-flops/registers, it is ok (possibly required) for there to be cases where the `always` block is entered, but the `reg` is not assigned
- No fancy code, just make it work
- Normally use synchronous reset instead of asynchronous reset (easier to test)





Reset-able and Enable-able Registers

- Example FF with reset and enable (reset has priority)

```
always @(posedge clk) begin
   if (reset)              // highest priority
      out <= #1 1'b0;
   else if (enable)
      out <= #1 c_out;
   // ok if no assignment (out holds value)
end
```

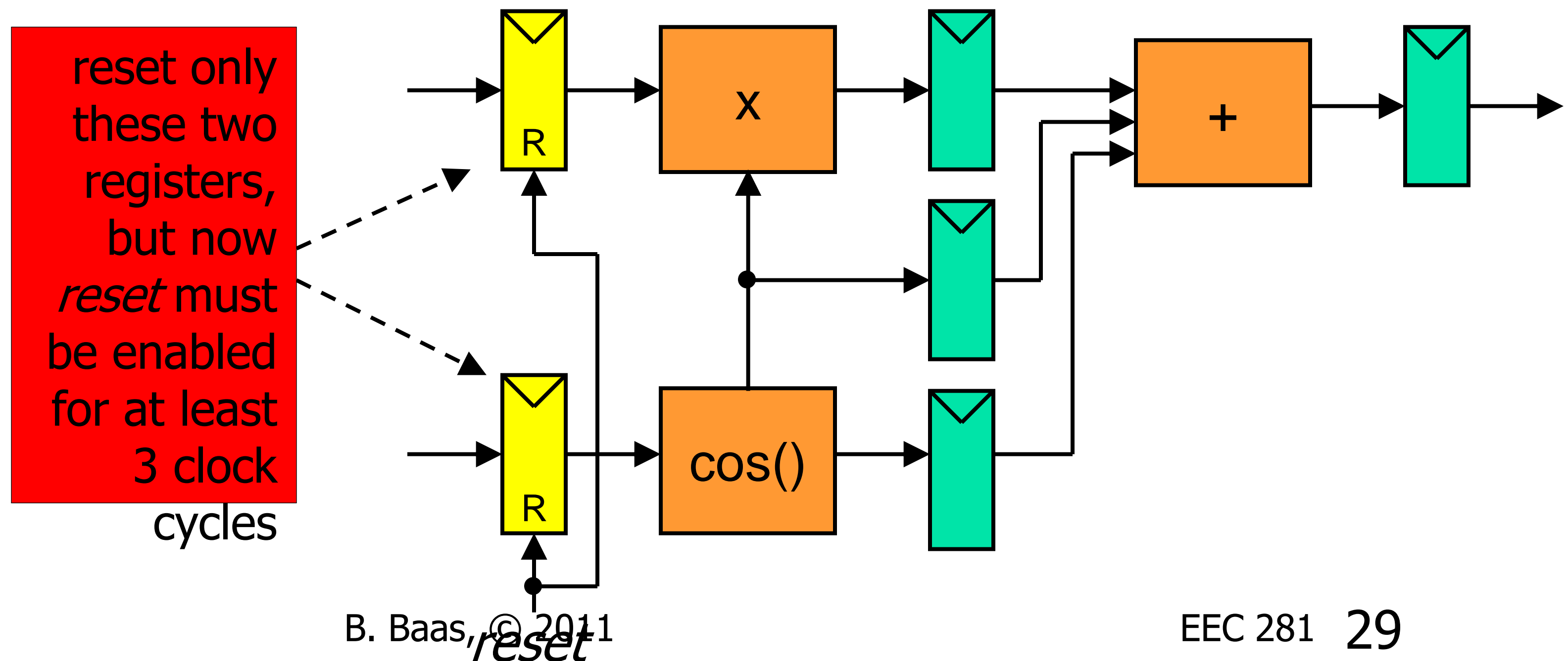
Reset-able and Enable-able Registers

- Example FF with reset and enable (enable has priority)

```
always @(posedge clk) begin
   if (enable) begin   // highest priority
      if (reset)
         out <= #1 1'b0;
      else
         out <= #1 c_out;
   end
   // ok if no assignment (out holds value)
end
```

Reset-able and Enable-able Registers

- Use reset-able FFs only where needed
  - FFs are a little larger and higher power
  - Requires the global routing of the high-fanout *reset* signal



Three types of "case" statements in verilog

1) `case`
   - Normal case statement
2) `casez`
   - Allows use of wildcard "?" character for don't cares.

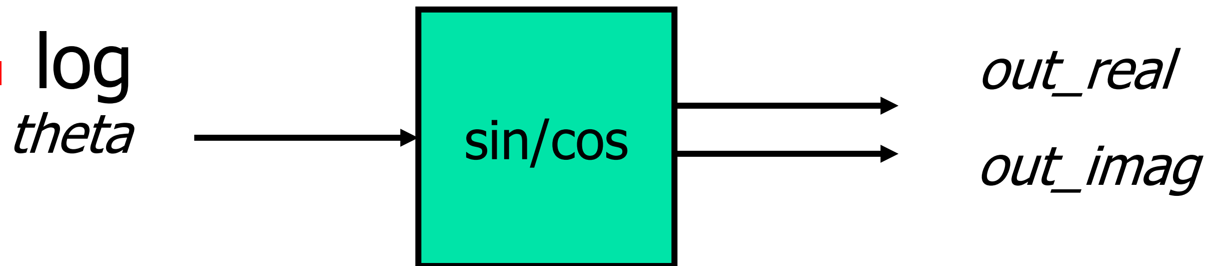
```
casez(in)
    4'b1???: out = a;
    4'b01??: out = b;
    4'b00??: out = c;
    default: out = d;
endcase
```

3) `casex`
   - Don't use it. Could use "z" or "x" logic.

- `default`
  - Normally set output to an easily-recognizable value (such as x's) in a `default` statement to make mistakes easier to spot

Hardwired Complex Functions

- Complex or “arbitrary” functions are not uncommon
- Examples
  - sin/cos
  - tangent^{-1}
  - log



Hardwired Function in Verilog using a Lookup Table

```
always @(input) begin
    case (input)
    4’b0000: begin  real=3’b100; imag=3’b001; end
    4’b0001: begin  real=3’b000; imag=3’b101; end
    4’b0010: begin  real=3’b110; imag=3’b011; end
        ...
    default: begin  real=3’bxxx; imag=3’bxxx; end
    endcase
end
```

- Often best to write a matlab program to write the verilog table as plain text
  - You will need several versions to get it right
  - Easy to adapt to other specifications
- Not efficient for very large tables
- Tables with data that is less random will have smaller synthesized area