ET5080E
Digital Design Using Verilog HDL

Fall ‘21

Simulator Mechanics

Testbench Basics (stimulus generation)

Dataflow Verilog

Administrative Matters

- Readings
  - Text Chapter 6 *(Dataflow Verilog, vector concatenation, operators)*

- Synthesis Tutorial coming in couple of weeks

- HW2 assigned…posted

Analog Simulation (Spice Engine)

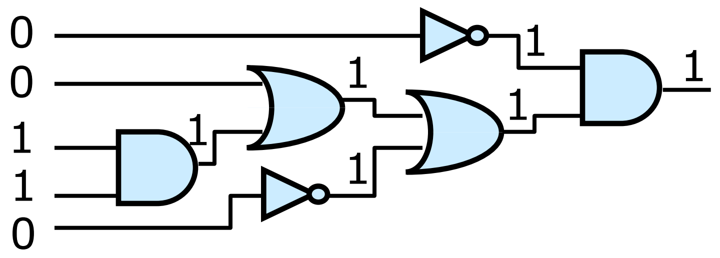
- Divide “time” into slices
- Update information in whole circuit at each slice
- Used by SPICE

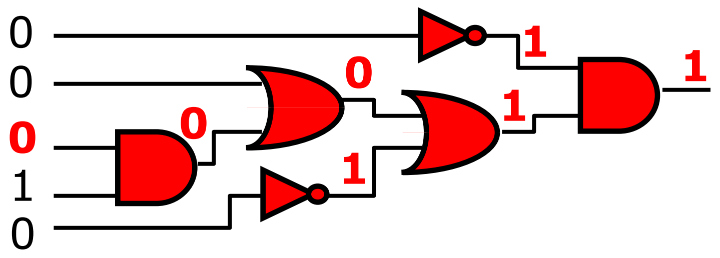
- Allows detailed modeling of current and voltage
- Computationally intensive and slow

- Don’t need this level of detail for most digital logic simulation

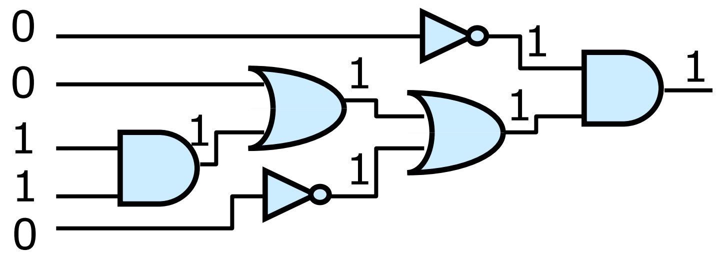
Digital Simulation

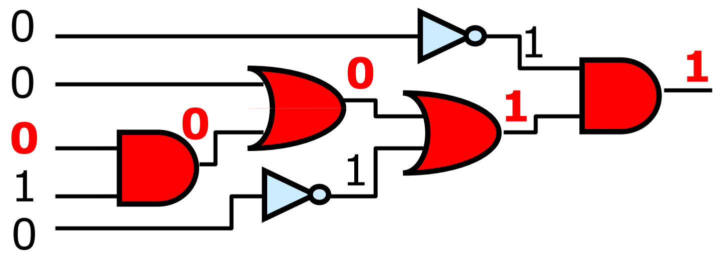
- Could update every signal on an input change





- Could update just the full path on input change





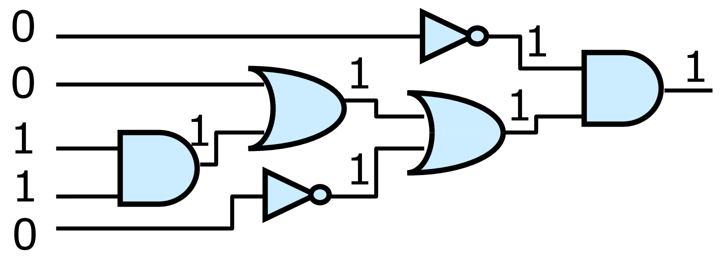
- Don’t even need to do that much work!

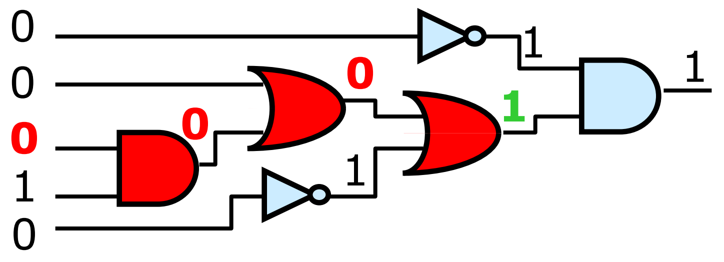
Event-Driven Simulation

- When an input to the simulating circuit changes, put it on a “changed” list
- Loop while the “changed” list isn’t empty:
  - Remove a signal from the “changed” list
  - For each sink of the signal
    - ✓ Recompute its new output(s)
    - ✓ For any output(s) that have changed value, add that signal to the “changed” list
- When the “changed” list is empty:
  - Keep simulation results
  - Advance simulation time to next stimulus (input) event

Simulation

- Update only if changed

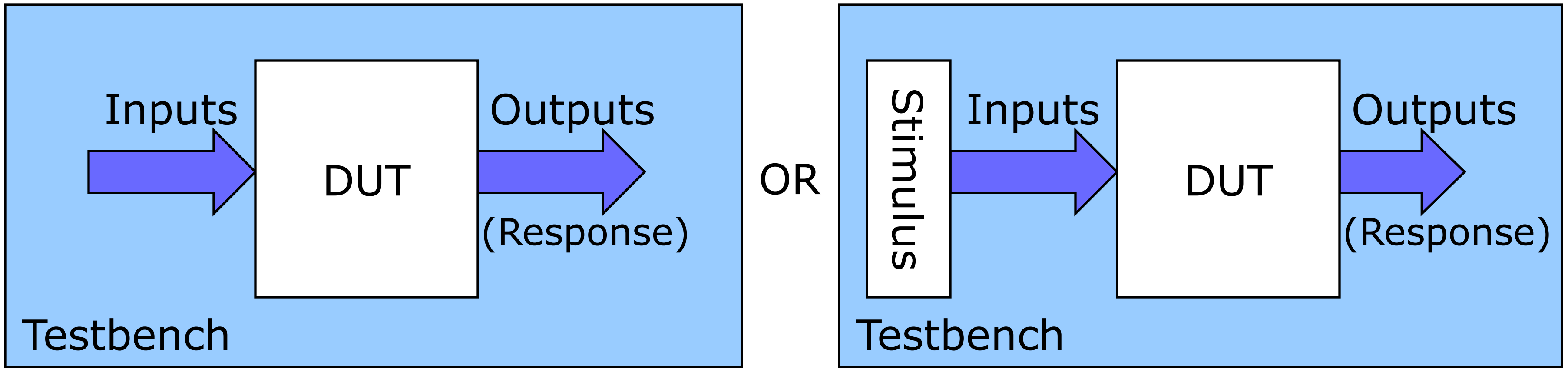




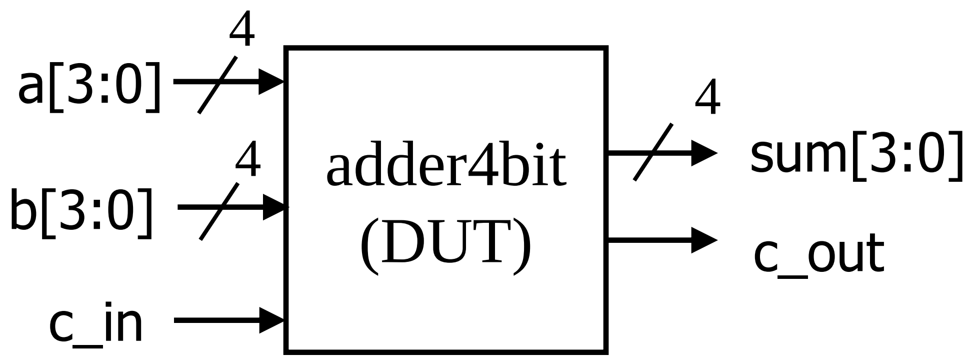
- Some circuits are very large
  - Updating every signal => very slow simulation
  - Event-driven simulation is much faster!

Testbench Basics (stimulus generation)

- Need to verify your design
  - "Design Under Test" (DUT)
- Use a "testbench"
  - Special Verilog module with no ports
  - Generates or routes inputs to the DUT
  - For now we will monitor outputs via human interface



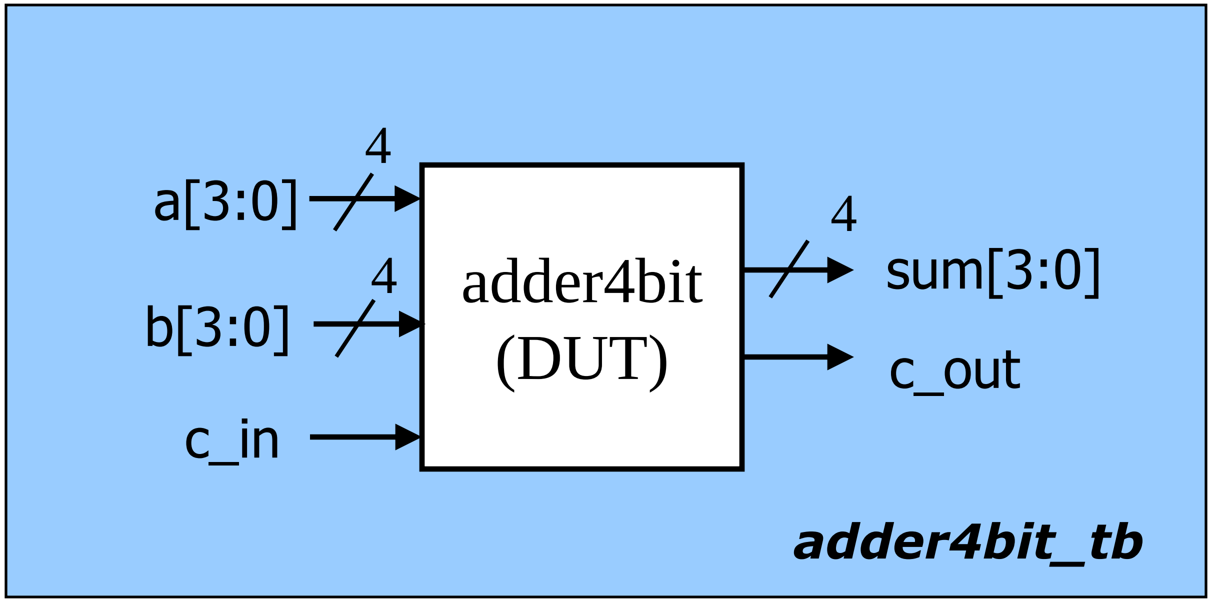
Simulation Example



Our DUT is a simple 4-bit adder, with carry in and carry out

Use a consistent naming convention for your test benches:

I usually add _tb to the end of the unit name



Testbench Requirements

- Instantiate the unit being tested (DUT)
- Provide input to that unit
  - Usually a number of different input combinations!
- Watch the “results” (outputs of DUT)
  - Can watch ModelSim Wave window…
  - Can print out information to the screen or to a file
- This way of monitoring outputs (human interface) is dangerous & incomplete.
  - Subject to human error
  - Cannot be automated into batch jobs (regression suite)
  - Self checking testbenches will be covered later

Output Test Info

- Several different system calls to output info
  - $monitor
    - ✓Output the given values whenever one changes
    - ✓Can use when simulating Structural, RTL, and/or Behavioral
  - $display, $strobe
    - ✓Output specific information like a printf in a C program
    - ✓Used in Behavioral Verilog
- Can use formatting strings with these commands
- Only means anything in simulation
- Ignored by synthesizer

Output Format Strings

- Formatting string

%h, %H	hex
%d, %D	decimal
%o, %O	octal
%b, %B	binary
%t	time

- $monitor(“%t: %b %h %h %h %b\n”,
  $time, c_out, sum, a, b, c_in);

- Can get more details from Verilog standard

Output Example

```
module adder4bit_tb;
    reg[8:0] stim;                                    // inputs to DUT are regs
    wire[3:0] S;                                      // outputs of DUT are wires
    wire C4;

    // instantiate DUT
    adder4bit(.sum(S), .c_out(C4), .a(stim[8:5]), .b(stim[4:1]), .c(stim[0]));

    initial $monitor(“%t A:%h B:%h ci:%b Sum:%h co:%b\n”,$time,
                      stim[8:5],stim[4:1],stim[0],C4,S);

// stimulus generation
    initial begin
        stim = 9'b0000_0000_0;                        // at 0 ns
        #10 stim = 9'b1111_0000_1;                    // at 10 ns
        #10 stim = 9'b0000_1111_1;                    // at 20 ns
        #10 stim = 9'b1111_0001_0;                    // at 30 ns
        #10 stim = 9'b0001_1111_0;                    // at 40 ns
        #10 $stop;                                    // at 50 ns – stops simulation
    end
endmodule
```

Exhaustive Testing

- For combinational designs w/ up to 8 or 9 inputs
  - Test ALL combinations of inputs to verify output
  - Could enumerate all test vectors, but don't…
  - Generate them using a "for" loop!

```
reg [4:0] x;
initial begin
        for (x = 0; x <  16; x = x + 1)
                #5;    // need a delay here!
end
```

- Need to use "reg" type for loop variable? Why?

Why Loop Vector Has Extra Bit

- Want to test all vectors 0000 to 1111

```
reg [3:0] x;
initial begin
        for (x = 0; x <  16; x = x + 1)
                #5;   // need a delay here!
end
```

- If x is 4 bits, it only gets up to 1111 => 15
  - 1100 => 1101 => 1110 => 1111 => 0000 => 0001
- x is never >= 16… so loop goes forever

Example: DUT

```
module Comp_4 (A_gt_B, A_lt_B, A_eq_B, A, B);
output  A_gt_B, A_lt_B, A_eq_B;
input    [3:0] A, B;
     // Code to compare A to B
     // and set A_gt_B, A_lt_B, A_eq_B accordingly
endmodule
```

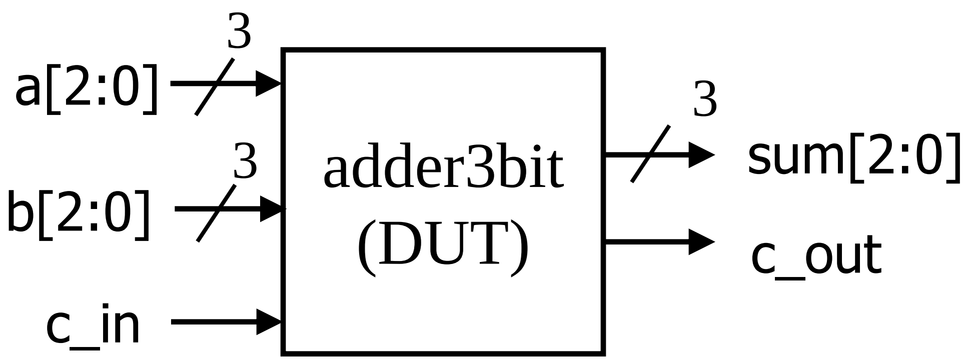
Example: Testbench

```
module Comp_4_tb();
wire     A_gt_B, A_lt_B, A_eq_B;
reg      [4:0] A, B;                  // sized to prevent loop wrap around

Comp_4 M1 (A_gt_B, A_lt_B, A_eq_B, A[3:0], B[3:0]);           // DUT

initial $monitor("%t  A: %h  B: %h  AgtB: %b  AltB: %b  AeqB: %b",
    $time, A[3:0], B[3:0], A_gt_B, A_lt_B, A_eq_B);
initial #2000 $finish;                // end simulation, quit program

initial begin
#5 for (A = 0; A < 16; A = A + 1) begin          // exhaustive test of valid
    inputs
     for (B = 0; B < 16; B = B + 1) begin #5;  // may want to test x's and z's
     end // first for
    end // second for
end // initial
endmodule
```

note multiple **initial** *blocks*

Do ModelSim Example Here

- ModelSim Example of 3-bit wide adder block



Combinational Testbench

```
module comb(output d, e, input a, b, c);
    and(d, a, b);
    nor(e, a, b, c);
endmodule
```

```
module comb_tb();
    wire        d, e;
    reg   [3:0] abc;

    comb CMD(.d(d),.e(e), .a(abc[2]), .b(abc[1]), .c(abc[0])); // DUT

    initial $monitor("%t  a: %b  b: %b  c: %b  d: %b  e: %b", $time,
            abc[2], abc[1], abc[0], d, e);
    initial #2000 $finish;           // end simulation, quit program

// exhaustive test of valid inputs
    initial begin
            for (abc = 0; abc < 8; abc = abc + 1)  #5;
    end // initial
endmodule
```

Does exhaustively testing a simple combinational block made with Verilog primitives make sense?

Generating Clocks

- Wrong way:

```
initial begin
   #5 clk = 0;
   #5 clk = 1;
   #5 clk = 0;
   … (repeat hundreds of times)
end
```

- Right way:

```
initial clk = 0;
always @(clk) clk = #5 ~clk;
```

✓

```
initial begin
      clk = 0;
      forever #5 clk = ~clk;
end
```

- LESS TYPING
- Easier to read, harder to make mistake

FSM Testing

- Response to input vector depends on state
- For each state:
  - Check all transitions
  - For Moore, check output at each state
  - For Mealy, check output for each transition
  - This includes any transitions back to same state!

- Can be time consuming to traverse FSM repeatedly…

Example : Gray Code Counter – Test1 *(the instructor is a chicken)*

- Write a testbench to test a gray code counter
  module gray_counter(out, clk, rst);
- Rst in this case is a synchrounous reset
  - It does not directly asynchronously reset the flops
  - It is an input to the combinational logic that sets the next state to all 0’s.
- Initially reset the counter and then test all states, but do not test reset in each state.

Solution : Gray Code Counter – Test1

```
module t1_gray_counter();
    wire [2:0] out;
    reg  clk, rst;
     gray_counter GC(out, clk, rst); // DUT

    initial $monitor(“%t  out: %b  rst: %b ”, $time, out, rst); // no
    clock
    initial #100 $finish;                          // end simulation, quit program

    initial begin
       clk = 0; forever #5 clk = ~clk; // What is the clock period?
    end
    initial begin
       rst = 1; #10 rst = 0;          // just reset and let it run
     end // initial
endmodule
```

Simulation: Gray Code Counter – Test1

```
#                    0  out: xxx  rst: 1                    // reset system
#                    5  out: 000  rst: 1                    // first positive edge
#                   10  out: 000  rst: 0                    // release reset
#                   15  out: 001  rst: 0                    // traverse states
#                   25  out: 011  rst: 0
#                   35  out: 010  rst: 0
#                   45  out: 110  rst: 0
#                   55  out: 111  rst: 0
#                   65  out: 101  rst: 0
#                   75  out: 100  rst: 0
#                   85  out: 000  rst: 0
#                   95  out: 001  rst: 0
```

Force/Release In Testbenches

- Allows you to "override" value FOR SIMULATION
- Doesn't do anything in "real life"
- How does this help testing?
  - Can help to pinpoint bug
  - Can use with FSMs to override state
    - ✓ Force to a state
    - ✓ Test all edges/outputs for that state
    - ✓ Force the next state to be tested, and repeat
- Can help achieve code coverage *(more on that later)*

Force/Release Example

```
assign y = a & b;
assign z = y | c;
initial begin
   a = 0; b = 0; c = 0;
   #5 a = 0; b = 1; c = 0;
   #5 force y = 1;
   #5 b = 0;
   #5 release y;
   #5 $stop;
end
```

T	a	b	c	y	z
0	0	0	0	0	0
5	0	1	0	0	0
10	0	1	0	1	1
15	0	0	0	1	1
20	0	0	0	0	0

Example : Gray Code Counter – Test2

- Write a testbench to exhaustively test the gray code counter example above

```
module gray_counter(out, clk, rst);
```

- Initially reset the counter and then test all states, then test reset in each state.

- Remember that in this example, rst is treated as an input to the combinational logic.

Example : Gray Code Counter – Test2

```
module t2_gray_counter();
    wire [2:0] out;
    reg   clk, rst;
    gray_counter GC(out, clk, rst); // DUT

    initial $monitor("%t  out: %b  rst: %b ", $time, out, rst); // no clock
    initial #300 $finish;                  // end simulation, quit program

    initial begin
      clk = 0; forever #5 clk = ~clk; // What is the clock period?
    end
    initial begin
      rst = 1; #10 rst = 0;
      #90 rst = 1; force GC.ns = 3'b001; #10 release GC.ns;
      #10 force GC.ns = 3'b011; #10 release GC.ns;
      #10 force GC.ns = 3'b010; #10 release GC.ns;


      …
    end // initial
endmodule
```

Note the Use of hierarchical Naming to get at the internal Signals. This is a very handy Thing in testbenches.

Simulation: Gray Code Counter – Test2

```
# 0  out: xxx  rst: 1
# 5  out: 000  rst: 1
# 10  out: 000  rst: 0
# 15  out: 001  rst: 0
# 25  out: 011  rst: 0
# 35  out: 010  rst: 0
# 45  out: 110  rst: 0
# 55  out: 111  rst: 0
# 65  out: 101  rst: 0
# 75  out: 100  rst: 0
# 85  out: 000  rst: 0
# 95  out: 001  rst: 0
# 100  out: 001  rst: 1
# 105  out: 000  rst: 1
// at #115 out = 000
# 125  out: 001  rst: 1
# 135  out: 000  rst: 1
# 145  out: 011  rst: 1
# 155  out: 000  rst: 1
# 165  out: 010  rst: 1
# 175  out: 000  rst: 1
# 185  out: 110  rst: 1
# 195  out: 000  rst: 1
# 205  out: 111  rst: 1
# 215  out: 000  rst: 1
# 225  out: 101  rst: 1
# 235  out: 000  rst: 1
# 245  out: 100  rst: 1
# 255  out: 000  rst: 1
```

Dataflow Verilog

- The continuous **assign** statement
  - It is the main construct of Dataflow Verilog
  - It is deceptively powerful & useful
- Generic form:

assign [drive_strength] [delay] list_of_net_assignments;

Where:
list_of_net_assignment ::= net_assignment [{,net_assignment}]
& Where:
Net_assignment ::= net_lvalue = expression

OK…that means just about nothing to me…how about some examples?

Continuous Assign Examples

- Simplest form:

```
// out is a net, a & b are also nets
assign out = a & b;  // and gate functionality
```

- Using vectors

```
wire [15:0] result, src1, src2;  // 3 16-bit wide vectors
assign result = src1 ^ src2;     // 16-bit wide XOR
```

- Can you implement a 32-bit adder in a single line?

```
wire [31:0] sum, src1, src2;  // 3 32-bit wide vectors
assign {c_out,sum} = src1 + src2 + c_in;    // wow!
```

What is this?

Vector concatenation

- Can “build” vectors using smaller vectors and/or scalar values
- Use the {} operator
- Example 1

becomes
8-bit vector:
$a_1a_0b_1b_0c_1c_0da_2$

```
module concatenate(out, a, b, c, d);
    input [2:0] a;
    input [1:0] b, c;
    input d;
    output [9:0] out;

    assign out = {a[1:0],b,c,d,a[2]};

endmodule
```

Vector concatenation

- Example 2

```
module add_concatenate(out, a, b, c, d);
    input [7:0] a;
    input [4:0] b;
    input [1:0] c;
    input d;
    output [7:0] out;

    add8bit(.sum(out), .cout(), .a(a), .b({b,c,d}), .cin());

endmodule
```

- Vector concatenation is not limited to assign statements. In this example it is done in a port connection of a module instantiation.

Replication within Concatenation

- Sometimes it is useful to replicate a bit (or vector) within the concatenation of another vector.
  - This is done with a replication constant (number) in front of the {}
  - Example: (sign extend an 8-bit value to a 16bit bus)

```
input [7:0] offset;// 8-bit offset term from EEPROM

wire [15:0] src1,src2;       // 16-bit source busses to ALU

assign src1 = {8{offset[7]},offset};              // sign extend offset term
```

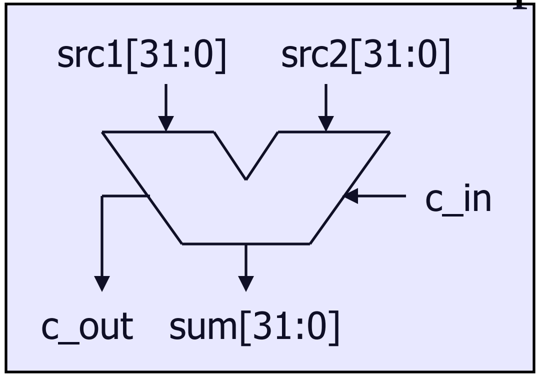
- Recall, to sign extend a 2's complement number you just replicate the MSB.
  - In this example the MSB of the 8-bit offset term has to be replicated 8 times to flush out to a full 16-bit value

Back to the Continuous Assign

- More basic generic form:

  assign <LHS> = <RHS expression>;

- If RHS result changes, LHS is updated with new value
  - Constantly operating ("continuous")
  - It's ***hardware!***
  - RHS can use operators (i.e. +,-,&,|,^,~,>>,…)



➔

```
assign {c_out,sum} = src1 + src2 + c_in;
```

Operators: Arithmetic

- Much easier than structural!

*	multiply	**	exponent
/	divide	%	modulus
+	add	-	subtract

- Some of these don't synthesis
- Also have unary operators +/- (pos/neg)
- Understand bitsize!
  - Can affect sign of result
  - Is affected by bitwidth of BOTH sides

```
Prod[7:0] = a[3:0] * b[3:0]
```

Operators

- Shift (<<, >>, <<<, >>>)
- Relational (<, >, <=, >=)
- Equality (==, !=, ===, !==)
  - ===, !== test x's, z's! ONLY USE FOR SIMULATION!
- Logical Operators (&&, ||, !)
  - Build clause for if statement or conditional expression
  - Returns single bit values
- Bitwise Operators (&, |, ^, ~)
  - Applies bit-by-bit!
- Watch ~ vs !, | vs. ||, and & vs. &&

Reduction Operators

- Reduction operators are reduce all the bits of a vector to a single bit by performing an operation across all bits.
  - Reduction AND
    - ✓ assign all_ones = &accumulator;   // are all bits set?
  - Reduction OR
    - ✓ assign not_zero = |accuumulator;   // are any bits set?
  - Reduction XOR
    - ✓ assign parity = ^data_out;   // even parity bit

Lets Kick up the Horse Power

- You thought a 32-bit adder in one line was powerful. Lets try a 32-bit MAC…

Design a multiply-accumulate (MAC) unit that computes

Z[31:0] = A[15:0]*B[15:0] + C[31:0]

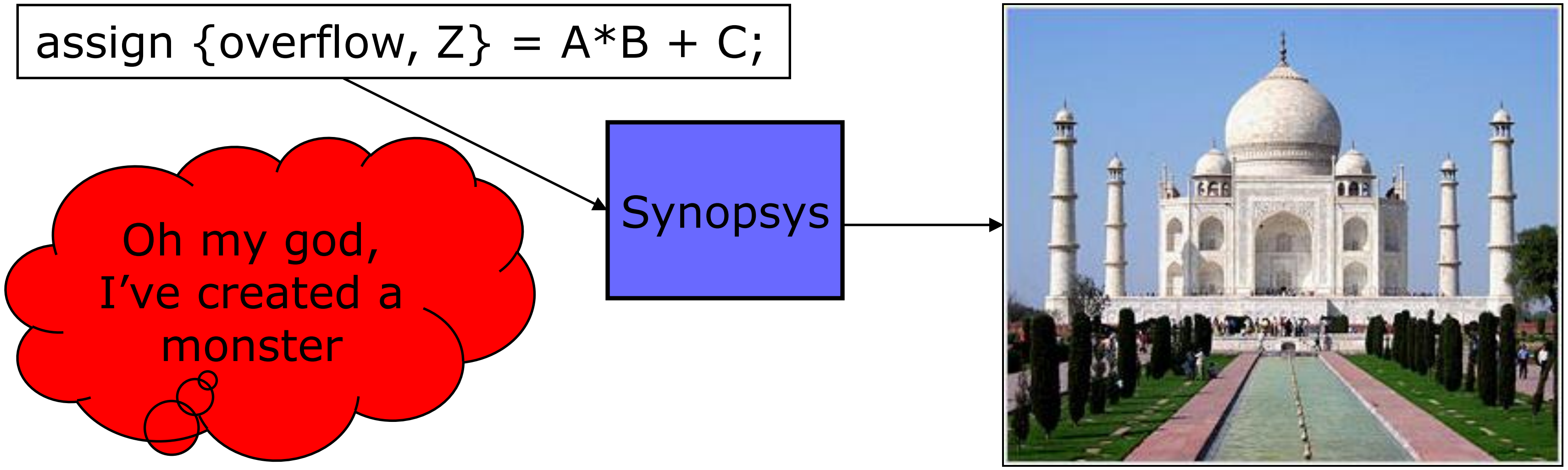
It sets overflow to one, if the result cannot be represented using 32 bits.

```
module mac(output Z [31:0], output overflow,
           input [15:0] A, B, input [15:0] C);
```

Lets Kick up the Horse Power

```
module mac(output Z [31:0], output overflow,
                input [15:0] A, B, input [31:0] C);
  assign {overflow, Z} = A*B + C;
endmodule
```

I am a brilliant genius. I am a HDL coder extraordinaire. I created a 32-bit MAC, and I did it in a single line.

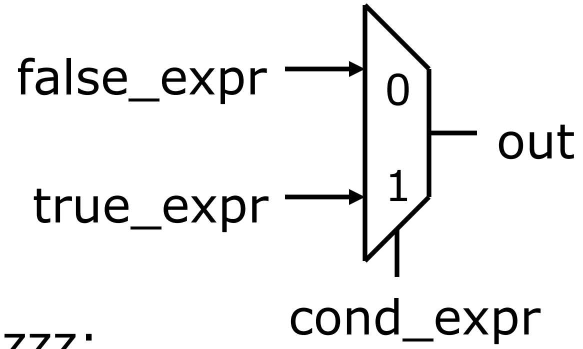


```
assign {overflow, Z} = A*B + C;
```

Conditional Operator

- This is a favorite!
  - The functionality of a 2:1 Mux
  - assign out = conditional_expr ? true_expr : false_expr;

```
Examples:

// a 2:1 mux
assign out = select ? in0 : in1;

// tri-state bus
assign src1 = rf2src1 ? Mem[addr1] : 16'hzzzz;

// Either true_expr or false_expr can also be a conditional operator
// lets use this to build a 4:1 mux
assign out = sel[1] ? (sel[0] ? in3 : in2) : (sel[0] ? in1 : in0);
```



Conditional assign (continued)

Examples: (nesting of conditionals)

```
`define add 3'b000
`define and 3'b001
`define xor 3'b010
`define shft_l 3'b011
`define shft_r 3'b100

// an ALU capable of arithmetic,logical, shift, and zero
assign {cout,dst} = (op==`add)    ?     src1+src2+cin :
                    (op==`and)    ?     {1'b0,src1 & src2} :
                    (op==`xor)    ?     {1'b0,src1 ^ src2} :
                    (op==`shft_l) ?     {src1,cin} :
                    (op==`shft_r) ?     {src1[0],src1[15],src1[15:1]} :
                                        17'h00000;
```

This can be very confusing to read if not coded with proper formating

A Few Other Possibilities

- The implicit assign
  - Goes in the wire declaration

    ```
    wire [3:0] sum = a + b;
    ```

  - Can be a useful shortcut to make code succinct, but doesn't allow fancy LHS combos

    ```
    assign {cout, sum} = a + b + cin;
    ```

  - Personal choice
    - ✓ You are welcome to use it when appropriate (I never do)

Latches with Continuous Assign

- What does the following statement imply/do?

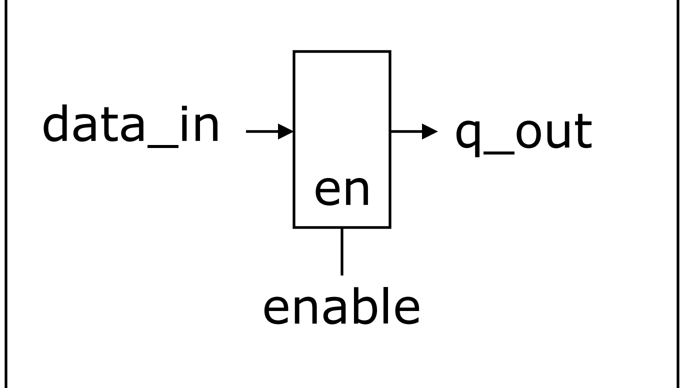
  assign q_out = enable ? data_in : q_out;

  - It acts as a latch. If enable is high new data goes to q_out. Otherwise q_out maintains its previous value.
  - Ask yourself…It that what I meant when I coded this?
  - It simulates fine…just like a latch

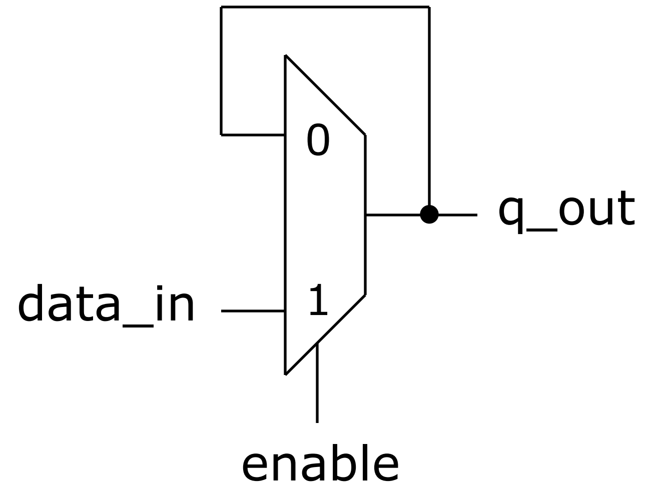
Latches with Continuous Assign

```
assign q_out = enable ? data_in : q_out;
```

- How does it synthesize??



Is the synthesizer smart enough to see this as a latch and pick a latch element from the standard cell library?



Or is it what you code is what you get? A combinational feedback loop. Still acts like a latch. Do you feel comfortable with this?