/* Half adder module */
module Add_half (sum,c_out,a,b);

input a,b;
output sum,c_out;

assign {c_out,sum} = a+b;
endmodule

/* Full adder module – a structural Verilog module */
module Add_full (sum,c_out,a,b,c_in);

input a,b,c_in;
output c_out,sum;
wire w1,w2,w3;

Add_half M1(w1,w2,a,b);
Add_half M2(sum,w3,w1,c_in);
or (c_out,w2,w3);
endmodule

/* Testbench for verifying full adder module */

module test_add; // declare module

reg a,b,c_in;    // declare test input variables
wire sum,c_out; // declare outputs of the module under test

reg correct_sum, correct_c_out;  // testbench variables which hold expected values
wire error;  // declare an "error" variable that will detect errors in module under test

Add_full A1(sum,c_out,a,b,c_in); // instantiate full adder module

// define test inputs and expected outputs
initial
begin

a = 0; b = 0; c_in = 0; correct_sum = 0; correct_c_out = 0;
#10 a = 0; b = 0; c_in = 1; correct_sum = 1; correct_c_out = 0;
#10 a = 0; b = 1; c_in = 0; correct_sum = 1; correct_c_out = 0;
#10 a = 0; b = 1; c_in = 1; correct_sum = 0; correct_c_out = 1;
#10 a = 1; b = 0; c_in = 0; correct_sum = 1; correct_c_out = 0;
#10 a = 1; b = 0; c_in = 1; correct_sum = 0; correct_c_out = 1;
#10 a = 1; b = 1; c_in = 0; correct_sum = 0; correct_c_out = 1;
#10 a = 1; b = 1; c_in = 1; correct_sum = 1; correct_c_out = 1;

end

// define an expression for "error" that will detect deviations between
// expected and actual outputs
assign error = (sum ^ correct_sum) || (c_out ^ correct_c_out);

endmodule
/* Flip flop module */
module flop (data, clock, clear, q, qb);

output q, qb;
noutput data, clock, clear;

wire q, qb;
wire data, clock, clear;

wire a, b, c, d, e, f, g, h; // interconnecting wires as described in the problem
// wires 'g' and 'h' have been added to signify the output of the iv1 and iv2 inverters respectively

not #10 iv1(g, data);
not #10 iv2(h, clock);
nand #10 nd1(a, clear, data, clock);
nand #10 nd2(b, clock, g);
nand #9 nd3(c,a,d);
nand #10 nd4(d,c,b,clear);
nand #10 nd5(e,c,h);
nand #10 nd6(f,d,h);
nand #9 nd7(q,e,qb);
nand #10 nd8(qb,q,clear,f);

module

/*Test bench for flip flop module*/
/* It should be noted that the test inputs are aimed at verifying the functional correctness of the flip flop implementation – mainly that the output should go to 1 when the ‘D’ input is 1 and the output should go to 0 when the ‘D’ input goes to 0. In this process, it can also be seen that all possible combination of inputs are implicitly chosen. Also note that the clock cycle time has been chosen in such a way as to make sure that the output will appear before the next cycle. Nevertheless, the ‘D’ input is held constant for atleast 2 clock cycles before being changed. The rising and falling delays from clock to q and again from clock to qbar can also be determined by the application of these test inputs*/

module test_flop;

reg clear,clock,data;
wire q,qb;

flop MSff(data,clock,clear,q,qb); // instantiate module under test

initial
begin
    data = 0; clear = 0; clock = 0;
    #225 data = 1;
    #200 clear = 1;
    #200 data = 0;
    #200 data = 1;
    #200 data = 0;
end

initial
begin
    #1300 $finish;
end

always
    #50 clock = ~clock;
endmodule
From the simulation, it is observed that

- The rising delay from clock to q is 79 ns.
- The falling delay from clock to q is 89 ns.
- The rising delay from clock to qbar is 80 ns.
- The falling delay from clock to qbar is 89 ns.

Now consider the case when the simulation has started but the inputs are not assigned values until sometime after the simulation begins. When all 3 inputs are ‘x’, the output is also ‘x’. When the clear input alone gets a value of 0 initially but the rest of the inputs are unknown, the q output goes to 0 and qbar goes to 1. This is due to the nature of the structural connections between the different gates. When the clear input is ‘1’, the flip flop is sensitive to both the clock and data inputs and when either one of them is ‘x’, the output is ‘x’. But when both of them have known values (either 1 or 0) the flip flop exhibits the expected behavior. Note that this is rendered possible because the output of the D flip flop is dependent only on the current input and not on any previous state. If that were not the case, then the output would still be at an unknown value even after the clock and data inputs have been assigned valid values.
module latch_hw2(q, qbar, data, G);

// inputs and outputs
input data, G;
output q, qbar;
wire data_bar, nand1, nand2;
wire q, qbar;

//module description
not not1(data_bar, data);
nand n1(nand1, data, G);
nand n2(nand2, data_bar, G);
nand n3(q, nand1, qbar); nand n4(qbar, nand2, q);
endmodule

module test_latch_hw2;

// signals and variables
wire q, qbar;
reg data, G;

latch_hw2 latch(q, qbar, data, G);

// module description
initial
begin
  #10 G = 0; // stay at s0
  #10 G = 1; data = 1'b1; // go to s1
  #10 data =0; //go to s1
  #10 G = 0; // stay at s1
  #10 G = 1; data =1'b1; // stay at s0
  #10 data =1; //go to s2
  #10 G = 0; // go back to s0
  #10 G=1; data =1'b1; // go back to s0
  #10 data =0; //go to s1
end
endmodule
* 10 means G=1, Data=0
** z state is ignored in the graph

State Transition Diagram

PAGE 113 PROBLEM 10
The wand signal is equal to the output of an AND gate.
Say

wire a, b, c;
wand d;

Then the logic after the following assignments

assign d = a;
assign d = b;
assign c = a & b;

The logic values for c and d are equal. However, if d is not a wand signal, then d will possibly get x if a and b have different values.

**PAGE 116 PROBLEM 17**

module add_sub_hw2(Y, Ov, A, B, S);

    // inputs and outputs

    input [4:0] A, B;
    input S; // mode selection. '1' if the operation is subtraction

    output [4:0] Y;
    output Ov; // overflow flag. set to 1 if overflow happens

    // module description

    assign Y = (S ==='b0)? (A+B) : (S ==='b1)? (A-B) : 5'bxxxxx;


endmodule

module test_add_sub_hw2;

    reg [4:0] A, B;
    reg S;

    wire [4:0] Y;
    wire Ov;

    reg [4:0] d1, d2, d3, d4;
add_sub_hw2 U1(Y, Ov, A, B, S);

initial
begin
    #10 S=1'b0; A=5'd10; B=5'd3; // normal addition
    #10 B=5'd11; // overflow
    #10 A=5'd0-5'd10; B=5'd0-5'd11; // underflow
    #10 S=1'b1; // normal subtraction
    #10 B=5'd11; // underflow
    #10 A=5'd10; B=5'd0-5'd11; // overflow
end
endmodule

Simulation result

PAGE 116 PROBLEM 20 – 22

The rules to follow when assign a value to a register variable:

1. Write down the exact value for LHS (Left Hand Side) value into the full expression. In Problem 5, the value is 8 bxxxxxx01. In the other two problems, the values have 32 bits and x (in problem 6) and z (in problem 7) will be filled for all the unspecified bits.

2. Assign the value. If RHS (Right Hand Side) variable has more bits than LHS value, then 0 is used to extend the unspecified bits. If RHS has less bits than LHS, than LHS is truncated to the desired width and assigned to RHS.
PAGE 130 PROBLEM 5

/*UDP declaration for a D flip flop with active low reset*/

primitive dflip (q_out, reset, clock, data);
output q_out;
input reset, clock, data;

reg q_out;

table

  \[
  \begin{array}{cccccc}
  \text{reset} & \text{clock} & \text{data} & \text{state} & \text{q\_out/next\_state} \\
  0 & ? & ? & ? & 0; & \text{define reset behavior} \\
  1 & (01) & 0 & ? & 0; & \text{Rising clock edge} \\
  1 & (01) & 1 & ? & 1; \\
  1 & (0?) & 1 & 1 & 1; \\
  1 & (?0) & ? & ? & -; & \text{Falling clock edge} \\
  1 & ? & (??) & ? & -; & \text{Steady clock, ignore data} \\
  \end{array}
  \]

endtable
endprimitive

PAGE 130 PROBLEM 7

primitive comparator (q_out, a1,a0,b1,b0);

output q_out;
input a1,a0,b1,b0;

table

  \[
  \begin{array}{cccccc}
  \text{a[1]} & \text{a[0]} & \text{b[1]} & \text{b[0]} & \text{q\_out} \\
  0 & 0 & 0 & 0 & 1; \\
  0 & 1 & 0 & 1 & 1; \\
  1 & 0 & 1 & 0 & 1; \\
  1 & 1 & 1 & 1 & 1; \\
  \end{array}
  \]

  //cases where both inputs match exactly-output is 1

  \[
  \begin{array}{cccccc}
  \text{a[1]} & \text{a[0]} & \text{b[1]} & \text{b[0]} & \text{q\_out} \\
  0 & ? & 1 & ? & 0; \\
  \end{array}
  \]

  //cases where inputs don't match exactly - output is 0
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>?</td>
<td>0</td>
<td>?</td>
<td>: 0;</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>: 0;</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>: 0;</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>: 0;</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>: 0;</td>
</tr>
</tbody>
</table>

//all other cases match to 'x' which need not be specified

detable
dendprimitive