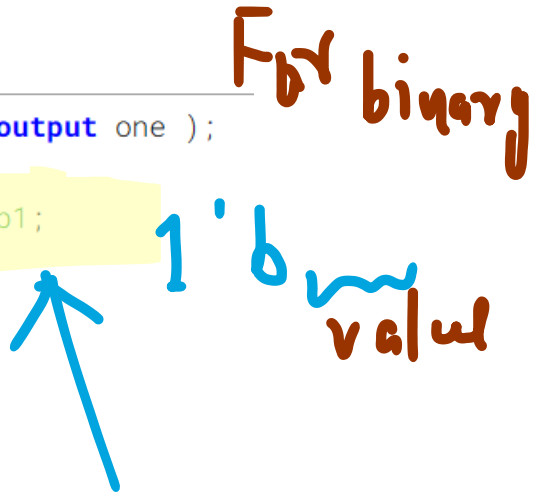
BASICS

06 December 2023 16:23

Solution

```
1  module top_module( output one );
2
3      assign one = 1'b1;
4
5  endmodule
6
```



Screen clipping taken: 06-12-2023 16:23

```
assign zero = 1'b0
```

Screen clipping taken: 06-12-2023 16:25

Unlike physical wires, wires (and other signals) in Verilog are *directional*. This means information flows in only one direction, from (usually one) *source* to the *sinks* (The source is also often called a *driver* that *drives* a value onto a wire). In a Verilog "continuous assignment" (`assign left_side = right_side;`), the value of the signal on the right side is driven onto the wire on the left side. The assignment is "continuous" because the assignment continues all the time even if the right side's value changes. A continuous assignment is not a one-time event.

The ports on a module also have a direction (usually input or output). An input port is *driven by* something from outside the module, while an output port *drives* something outside. When viewed from inside the module, an input port is a driver or source, while an output port is a sink.

Screen clipping taken: 06-12-2023 16:28

```
1  module top_module( input in, output out );
2  assign out=in;
3  endmodule
4
```

Screen clipping taken: 06-12-2023 16:30

Hint...

Verilog has separate bitwise-NOT (~) and logical-NOT (!) operators, like C. Since we're working with a one-bit here, it doesn't matter which we choose.

```
module top_module( input in, output out );
assign out = ~in;
endmodule

```

Screen clipping taken: 06-12-2023 16:42

Hint...

Verilog has separate bitwise-AND (&) and logical-AND (&&) operators, like C. Since we're working with a one-bit here, it doesn't matter which we choose.

Screen clipping taken: 06-12-2023 16:44

```
module top_module(
    input a,
    input b,
    output out );
assign out = a&b;
endmodule

```

Screen clipping taken: 06-12-2023 16:44

Nor gate

```
module top_module(
    input a,
    input b,
    output out );
assign out = ~a&~b;
endmodule

```

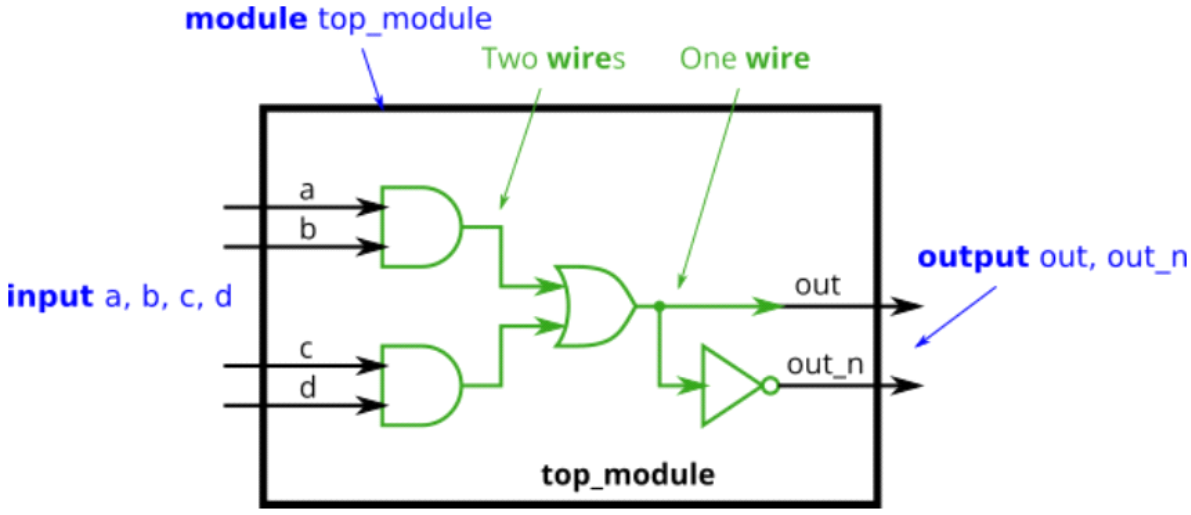
Screen clipping taken: 07-12-2023 13:09

```
module top_module(
    input a,
    input b,
    output out );
    assign out =(a&b)|(~a&~b);
endmodule
```

Submit | Submit (new window)

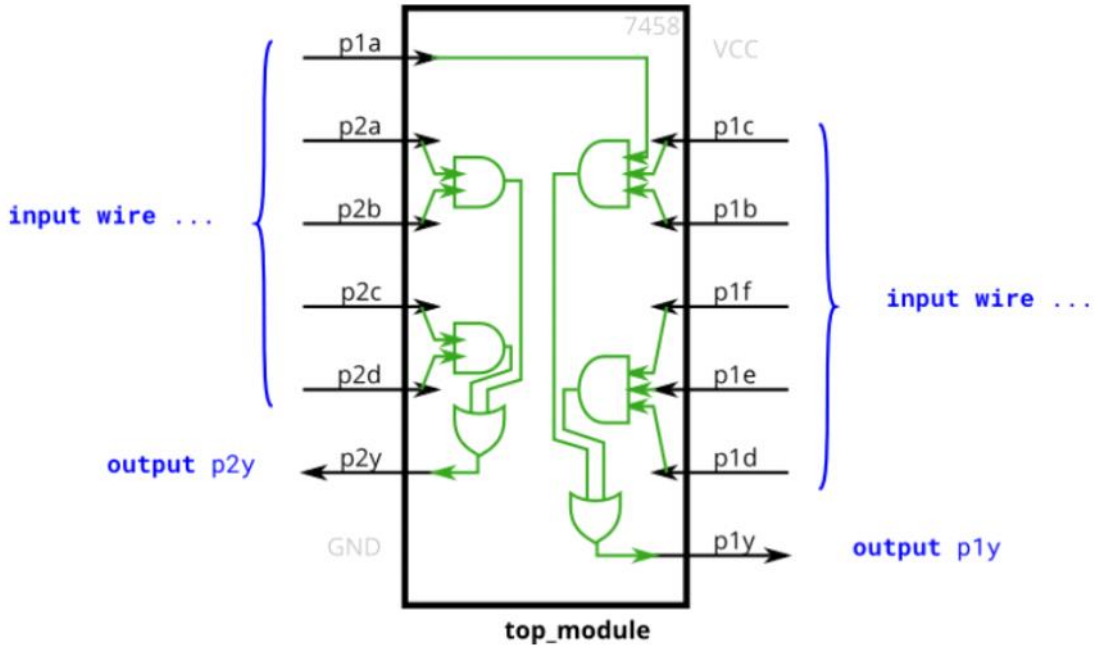
Upload a source file...

xnorgate — Compile and simulate



Screen clipping taken: 07-12-2023 13:21

```
`default_nettype none
module top_module(
    input a,
    input b,
    input c,
    input d,
    output out,
    output out_n   );
    wire x,y,z;
    assign x=a&b;
    assign y=c&d;
    assign z= x|y;
    assign out =z;
    assign out_n =~z;
endmodule

```



```
module top_module (
    input p1a, p1b, p1c, p1d, p1e, p1f,
    output p1y,
    input p2a, p2b, p2c, p2d,
    output p2y );
    wire x,y,a,b;
    assign x= p2a & p2b;
    assign y= p2c & p2d;
    assign p2y=x|y;
    assign a= p1c & p1b & p1a;
    assign b= p1e & p1d & p1f;
    assign p1y=a|b;
endmodule

```

Vectors

Vectors are used to group related signals using one name to make it more convenient to manipulate. For example, `wire [7:0] w;` declares an 8-bit vector named `w` that is functionally equivalent to having 8 separate wires.

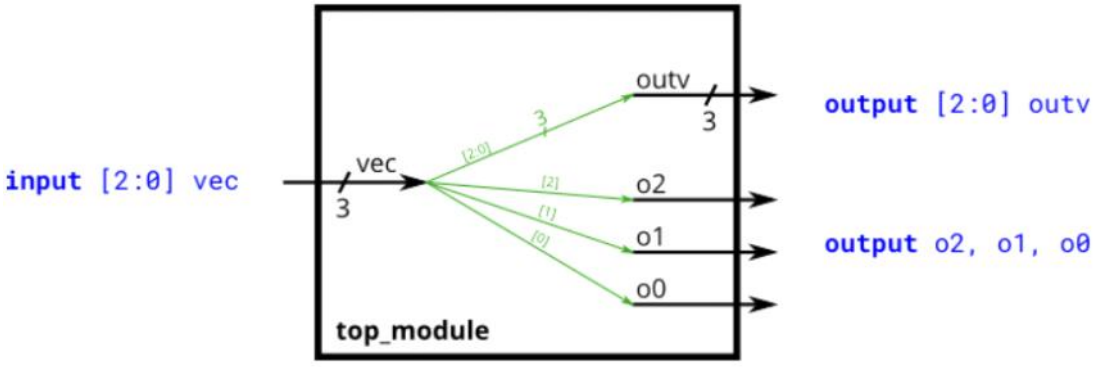
Notice that the *declaration* of a vector places the dimensions *before* the name of the vector, which is unusual compared to C syntax. However, the *part select* has the dimensions *after* the vector name as you would expect.

```
wire [99:0] my_vector;      // Declare a 100-element vector
assign out = my_vector[10]; // Part-select one bit out of the vector
```

Build a circuit that has one 3-bit input, then outputs the same vector, and also splits it into three separate 1-bit outputs. Connect output `o0` to the input vector's position 0, `o1` to position 1, etc.

In a diagram, a tick mark with a number next to it indicates the width of the vector (or "bus"), rather than drawing a separate line for each bit in the vector.

Screen clipping taken: 07-12-2023 15:12



Screen clipping taken: 07-12-2023 15:12

```
module top_module (
    input wire [2:0] vec,
    output wire [2:0] outv,
    output wire o2,
    output wire o1,
    output wire o0  ); // Module body starts after module declaration
    assign o0=vec[0];
    assign o1=vec[1];
    assign o2=vec[2];
    assign outv=vec;
endmodule

```

Screen clipping taken: 07-12-2023 15:14

Vectors must be declared:

`type [upper:lower] vector_name;`

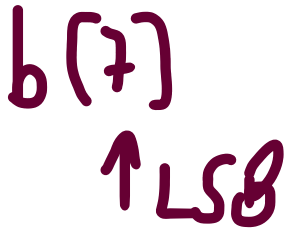
Vectors are used to group related signals using one name to make it more convenient to manipulate. For example, `wire [7:0] w;` declares an 8-bit vector named `w` that is equivalent to having 8 separate wires.

Screen clipping taken: 07-12-2023 15:16

```
wire [7:0] w;         // 8-bit wire
reg  [4:1] x;         // 4-bit reg
output reg [0:0] y;   // 1-bit reg that is also an output port (this
is still a vector)
input wire [3:-2] z;  // 6-bit wire input (negative ranges are
allowed)
output [3:0] a;       // 4-bit output wire. Type is 'wire' unless
specified otherwise.
```

```
output [3:0] a;          // 4-bit output wire. Type is "wire" unless
specified otherwise.
wire [0:7] b;            // 8-bit wire where b[0] is the most-
significant bit.
```

Screen clipping taken: 07-12-2023 15:17




```
wire [2:0] a, c;    // Two vectors
assign a = 3'b101;  // a = 101
assign b = a;       // b =   1  implicitly-created wire
assign c = b;       // c = 001  <-- bug
my_module i1 (d,e); // d and e are implicitly one-bit wide if
not declared.
                    // This could be a bug if the port was
intended to be a vector.
```

Screen clipping taken: 07-12-2023 15:25

Unpacked vs. Packed Arrays

You may have noticed that in *declarations*, the vector indices are written *before* the vector name. This declares the "packed" dimensions of the array, where the bits are "packed" together into a blob (this is relevant in a simulator, but not in hardware). The *unpacked* dimensions are declared *after* the name. They are generally used to declare memory arrays. Since ECE253 didn't cover memory arrays, we have not used packed arrays in this course. See http://www.asic-world.com/systemverilog/data_types10.html for more details.

```
reg [7:0] mem [255:0];   // 256 unpacked elements, each of which is a
8-bit packed vector of reg.
reg mem2 [28:0];         // 29 unpacked elements, each of which is a
1-bit reg.
```

The part-select operator can be used to access a portion of a vector:

```
w[3:0]      // Only the lower 4 bits of w
x[1]        // The lowest bit of x
x[1:1]      // ...also the lowest bit of x
z[-1:-2]    // Two lowest bits of z
b[3:0]      // Illegal. Vector part-select must match the direction
of the declaration.
b[0:3]      // The *upper* 4 bits of b.
assign w[3:0] = b[0:3];    // Assign upper 4 bits of b to lower 4
bits of w. w[3]=b[0], w[2]=b[1], etc.
```

Build a combinational circuit that splits an input half-word (16 bits, [15:0]) into lower [7:0] and upper [15:8] bytes.

Screen clipping taken: 07-12-2023 15:36

```
`default_nettype none     // Disable implicit nets. Reduces some types
module top_module(
    input wire [15:0] in,
    output wire [7:0] out_hi,
    output wire [7:0] out_lo );
    assign out_hi = in[15:8];
    assign out_lo = in[7:0];
endmodule

```

Screen clipping taken: 07-12-2023 15:38

A 32-bit vector can be viewed as containing 4 bytes (bits [31:24], [23:16], etc.). Build a circuit that will reverse the *byte* ordering of the 4-byte word.

AaaaaaaaBbbbbbbbCccccccDddddddd => DddddddCccccccBbbbbbbbAaaaaaaa

Screen clipping taken: 07-12-2023 15:46

```
module top_module(
    input [31:0] in,
    output [31:0] out );//
    assign out[31:24]=in[7:0];
    assign out[23:16]=in[15:8];
    assign out[15:8]=in[23:16];
    assign out[7:0]=in[31:24];
endmodule

```

Screen clipping taken: 07-12-2023 15:50

VECTORS

07 December 2023 15:56

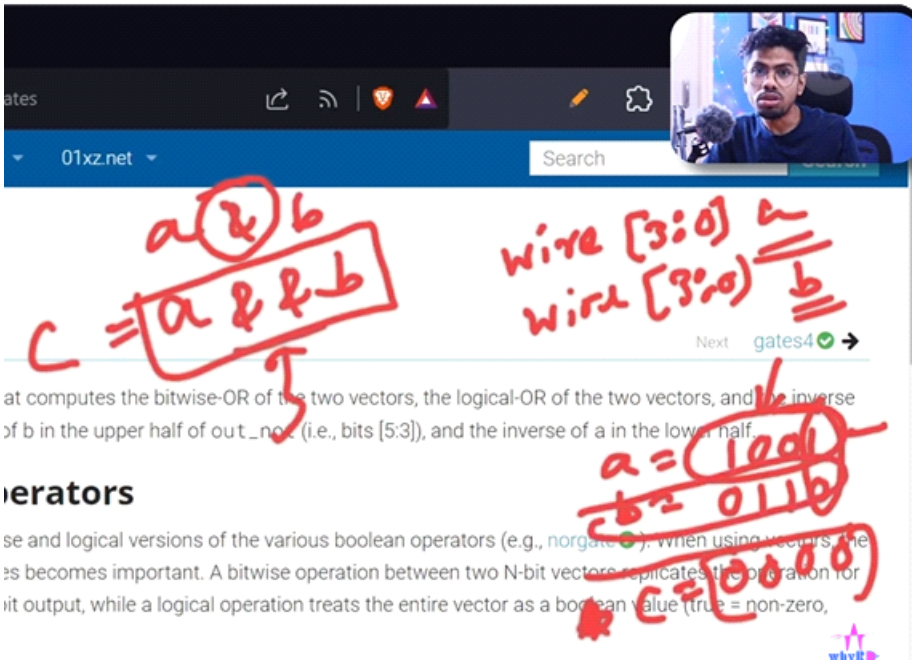
Vectorgates

Screen clipping taken: 07-12-2023 15:57

Bitwise vs. Logical Operators

Screen clipping taken: 07-12-2023 16:03

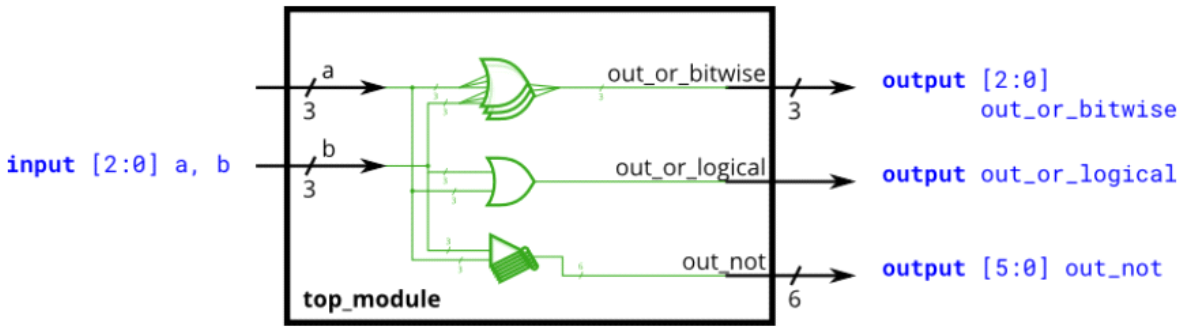
types becomes important. A bitwise operation between two N-bit vectors replicates the operation for each bit of the vector and produces a N-bit output, while a logical operation treats the entire vector as a boolean value (true = non-zero, false = zero) and produces a 1-bit output.



Screen clipping taken: 07-12-2023 16:03

Build a circuit that has two 3-bit inputs that computes the bitwise-OR of the two vectors, the logical-OR of the two vectors, and the inverse (NOT) of both vectors. Place the inverse of b in the upper half of `out_not` (i.e., bits [5:3]), and the inverse of a in the lower half.

Screen clipping taken: 07-12-2023 16:05




```
module top_module(
    input [2:0] a,
    input [2:0] b,
    output [2:0] out_or_bitwise,
    output out_or_logical,
    output [5:0] out_not
);

    assign out_or_bitwise[0] = a[0] | b[0];
    assign out_or_bitwise[1] = a[1] | b[1];
    assign out_or_bitwise[2] = a[2] | b[2];

    assign out_or_logical =a || b;

    assign out_not[0]=~a[0];
    assign out_not[1]=~a[1];
    assign out_not[2]=~a[2];
    assign out_not[3]=~b[0];
    assign out_not[4]=~b[1];
    assign out_not[5]=~b[2];
endmodule

```

Screen clipping taken: 07-12-2023 16:25

Build a combinational circuit with four inputs, `in[3:0]`

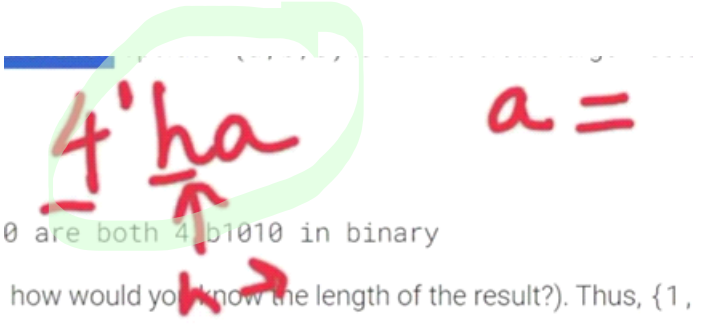
There are 3 outputs:

- out_and: output of a 4-input AND gate.
- out_or: output of a 4-input OR gate.
- out_xor: output of a 4-input XOR gate.

Screen clipping taken: 07-12-2023 16:28

```verilog
module top_module(
    input [3:0] in,
    output out_and,
    output out_or,
    output out_xor
);
    assign out_and =in[0] & in[1] & in[2] & in[3];
    assign out_or =in[0] | in[1] | in[2] | in[3];
    assign out_xor =in[0] ^ in[1] ^ in[2] ^ in[3];
endmodule

```

Screen clipping taken: 07-12-2023 16:31



Part selection was used to select portions of a vector. The concatenation operator `{a,b,c}` is used to create larger vectors by concatenating smaller portions of a vector together.

```
{3'b111, 3'b000} => 6'b111000
{1'b1, 1'b0, 3'b101} => 5'b10101
{4'ha, 4'd10} => 8'b10101010     // 4'ha and 4'd10
are both 4'b1010 in binary
```

Screen clipping taken: 08-12-2023 11:05

Concatenation needs to know the width of every component (or how would you know the length of the result?). Thus, `{1, 2, 3}` is illegal and results in the error message: `unsized constants are not allowed in concatenations`.

Screen clipping taken: 08-12-2023 11:08

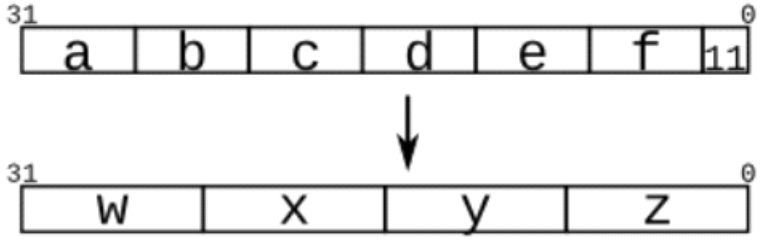
The concatenation operator can be used on both the left and right sides of assignments.

```
input [15:0] in;
output [23:0] out;
assign {out[7:0], out[15:8]} = in;          // Swap
two bytes. Right side and left side are both 16-bit
vectors.
assign out[15:0] = {in[7:0], in[15:8]};     // This is
the same thing.
assign out = {in[7:0], in[15:8]};       // This is
different. The 16-bit vector on the right is extended
to
                                            // match the
24-bit vector on the left, so out[23:16] are zero.
                                            // In the
first two examples, out[23:16] are not assigned.
```

Screen clipping taken: 08-12-2023 11:08

A Bit of Practice

Given several input vectors, concatenate them together then split them up into several output vectors. There are six 5-bit input vectors: a, b, c, d, e, and f, for a total of 30 bits of input. There are four 8-bit output vectors: w, x, y, and z, for 32 bits of output. The output should be a concatenation of the input vectors followed by two 1 bits:



Screen clipping taken: 08-12-2023 11:10

```
module top_module (
    input [4:0] a, b, c, d, e, f,
    output [7:0] w, x, y, z );//

    // assign { ... } = { ... };
    assign { w, x, y, z} = {a, b, c, d, e, f,2'b11};
endmodule

```

Screen clipping taken: 08-12-2023 11:15

Given an 8-bit input vector [7:0], reverse its bit ordering.

Screen clipping taken: 08-12-2023 11:22

```
module top_module(
    input [7:0] in,
    output [7:0] out
);
    assign out= {in[0],in[1],in[2],in[3],in[4],in[5],
endmodule

```

This replicates *vector* by *num* times. *num* must be a constant. Both sets of braces are required.

Examples:

```
{5{1'b1}}               // 5'b11111 (or 5'd31 or 5'h1f)
{2{a,b,c}}              // The same as {a,b,c,a,b,c}
{3'd5, {2{3'd6}}}   // 9'b101_110_110. It's a
concatenation of 101 with
                        // the second vector, which is
two copies of 3'b110.
```

One common place to see a replication operator is when sign-extending a smaller number to a larger one, while preserving its signed value. This is done by replicating the sign bit (the most significant bit) of the smaller number to the left. For example, sign-

significant bit) of the smaller number to the left. For example, sign-extending 4'b0101 (5) to 8 bits results in 8'b00000101 (5), while sign-extending 4'b1101 (-3) to 8 bits results in 8'b11111101 (-3).

Screen clipping taken: 08-12-2023 11:34

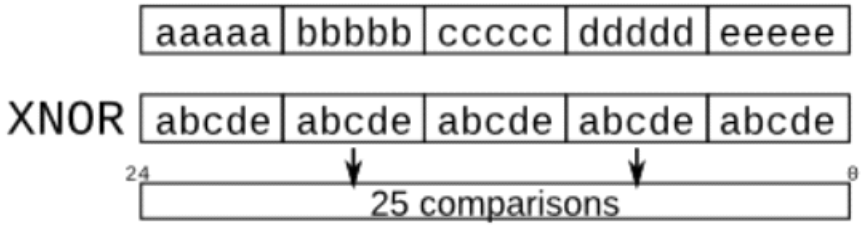
Build a circuit that sign-extends an 8-bit number to 32 bits. This requires a concatenation of 24 copies of the sign bit (i.e., replicate bit[7] 24 times) followed by the 8-bit number itself.

```
1 module top_module (
2     input [7:0] in,
3     output [31:0] out );//
4     assign out = {{24{in[7]}}, in[7:0]};
5 
6 endmodule
7 
```

Screen clipping taken: 08-12-2023 11:42

Given five 1-bit signals (a, b, c, d, and e), compute all 25 pairwise one-bit comparisons in the 25-bit output vector. The output should be 1 if the two bits being compared are equal.

```
out[24] = ~a ^ a;    // a == a, so out[24] is always 1.
out[23] = ~a ^ b;
out[22] = ~a ^ c;
...
out[ 1] = ~e ^ d;
out[ 0] = ~e ^ e;
```



Screen clipping taken: 08-12-2023 12:06

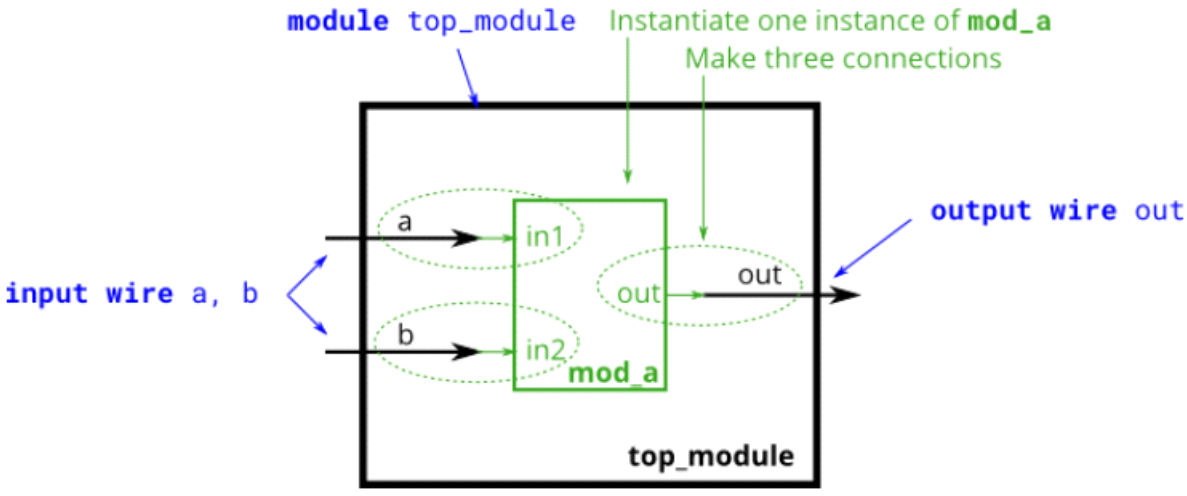
```systemverilog
module top_module (
    input a, b, c, d, e,
    output [24:0] out );//

    // The output is XNOR of two vectors created by
    // concatenating and replicating the five inputs.
    assign out = {~{5{a}} ^ {a,b,c,d,e},
                  ~{5{b}} ^ {a,b,c,d,e},
                  ~{5{c}} ^ {a,b,c,d,e},
                  ~{5{d}} ^ {a,b,c,d,e},
                  ~{5{e}} ^ {a,b,c,d,e}};
endmodule

```

Screen clipping taken: 08-12-2023 12:12

MODULES

20 December 2023 10:11

You may connect signals to the module by port name or port position. For extra practice, try both methods.

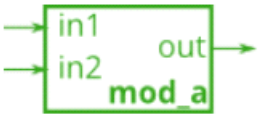


Screen clipping taken: 20-12-2023 10:20

Child Module

The code for module mod_a looks like this:

```
module mod_a ( input in1, input in2,
output out );
    // Module body
endmodule
```



Screen clipping taken: 20-12-2023 10:23

Method-1

Write your solution here

[Load a previous submission] Load

```
1  module top_module (
2      input a,
3      input b,
4      output out );
5      mod_a in1(a,b,out);
6
7  endmodule
8
```

Screen clipping taken: 20-12-2023 10:27

Method-2

By name

Connecting signals to a module's ports *by name* allows wires to remain correctly connected even if the port list changes. This syntax is more verbose, however.

Screen clipping taken: 20-12-2023 10:28

The above line instantiates a module of type `mod_a` named "instance2", then connects signal wa (outside the module) to the port **named** `in1`, wb to the port **named** `in2`, and wc to the port **named** `out`. Notice how the ordering of ports is irrelevant here because the connection will be made to the correct name, regardless of its position in the sub-module's port list. Also notice

position in the sub-module's port list. Also notice the period immediately preceding the port name in this syntax.

Screen clipping taken: 20-12-2023 11:57

```
module top_module (
    input a,
    input b,
    output out );
    mod_a in1(.out(out),.in1(a),.in2(b))

endmodule

```

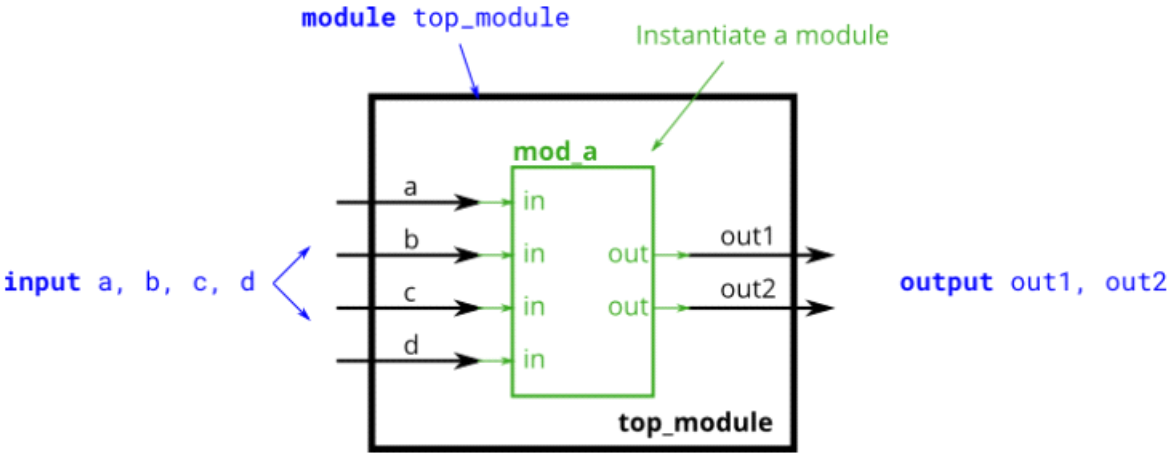
Screen clipping taken: 20-12-2023 11:57

This problem is similar to the previous one (module ✅). You are given a module named `mod_a` that has 2 outputs and 4 inputs, in that order. You must connect the 6 ports *by position* to your top-level module's ports `out1`, `out2`, a, b, c, and d, in that order.

You are given the following module:

```
module mod_a ( output, output, input, input, input, input );
```

Screen clipping taken: 20-12-2023 12:01



Screen clipping taken: 20-12-2023 12:01

```
module top_module (
    input a,
    input b,
    input c,
    input d,
    output out1,
    output out2
);
    mod_a in1(out1,out2,a,b,c,d);
endmodule

```

Screen clipping taken: 20-12-2023 12:12

This problem is similar to module. You are given a module named mod_a that has 2 outputs and 4 inputs, in some order. You must connect the 6 ports *by name* to your top-level module's ports:

Port in mod_a	Port in top_module
output out1	out1
output out2	out2
input in1	a
input in2	b
input in3	c
input in4	d

Screen clipping taken: 20-12-2023 12:13

```
module top_module (
    input a,
    input b,
    input c,
    input d,
    output out1,
    output out2
);
mod_a in1(.out1(out1),.out2(out2),.in1(a),.in2(b),.in3(c),.ir
endmodule

```

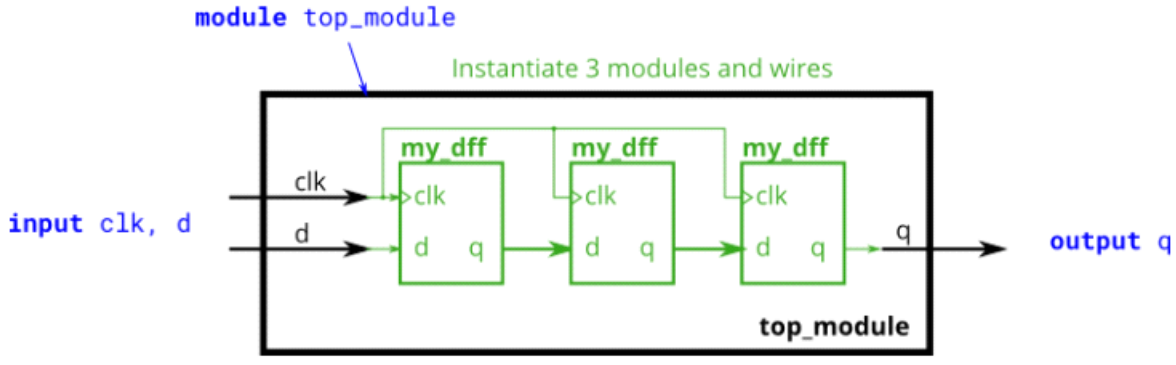
Screen clipping taken: 20-12-2023 12:17

You are given a module `my_dff` with two inputs and one output (that implements a D flip-flop). Instantiate three of them, then chain them together to make a shift register of length 3. The `clk` port needs to be connected to all instances.

Screen clipping taken: 20-12-2023 12:26

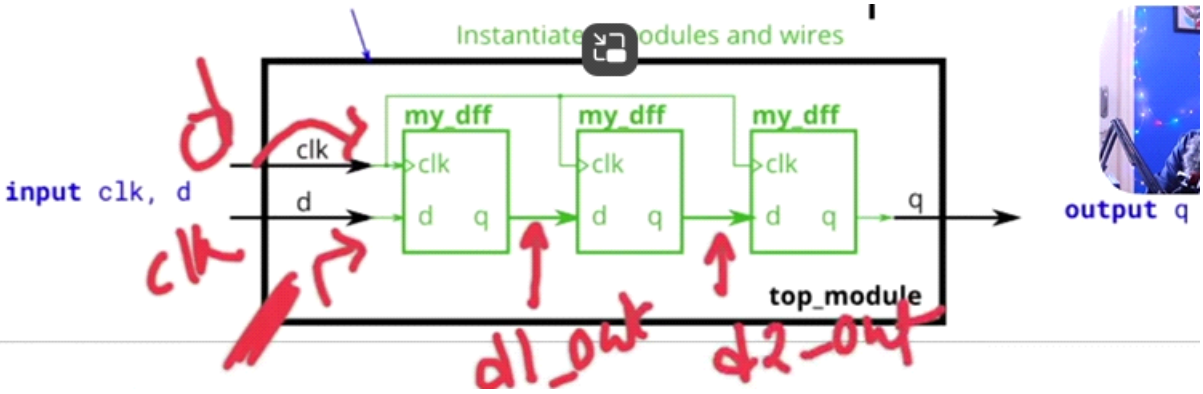
The module provided to you is: `module my_dff ( input clk, input d, output q );`

Screen clipping taken: 20-12-2023 12:26



Screen clipping taken: 20-12-2023 12:27

Name these wires for simplicity



Screen clipping taken: 20-12-2023 12:28

```
module top_module (input clk,
                   input d,
                   output q );
    wire d1_out,d2_out;
    my_dff d1(.clk(clk),.d(d),.q(d1_out));
    my_dff d2(.clk(clk),.d(d1_out),.q(d2_out));
    my_dff d3(.clk(clk),.d(d2_out),.q(q));
endmodule
```

```
module top_module ( input a, input b, output out1, out2, );
    assign a=3'b101;
    assign b=3'b100;
    assign out1=a | b ;
    assign out2=a || b;
endmodule
```

Answer option :

A) out1=100 out2=1

B) out1=101 out2=0

C) out1=100 out2=0

D) out1=101 out2=1

E) None is true

Screen clipping taken: 20-12-2023 14:39

This exercise is an extension of module_shift ✅ . Instead of module ports being only single pins, we now have modules with vectors as ports, to which you will attach wire vectors instead of plain wires. Like everywhere else in Verilog, the vector length of the port does not have to match the wire connecting to it, but this will cause zero-padding or trucation of the vector. This exercise does not use connections with mismatched vector lengths.

Screen clipping taken: 20-12-2023 14:39

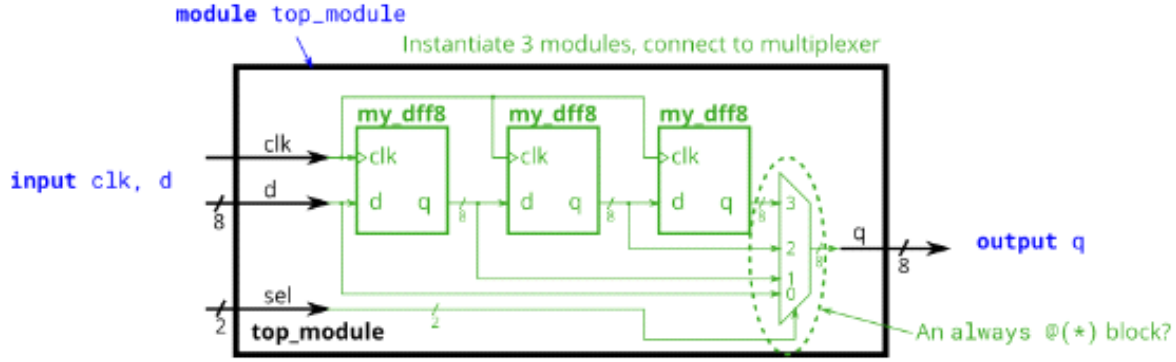
You are given a module `my_dff8` with two inputs and one output (that implements a set of 8 D flip-flops). Instantiate three of them, then chain them together to make a 8-bit wide shift register of length 3. In addition, create a 4-to-1 multiplexer (not provided) that chooses what to output depending on `sel[1:0]`: The value at the input d, after the first, after the second, or after the third D flip-flop. (Essentially, `sel` selects how many cycles to delay the input, from zero to three clock cycles.)

Screen clipping taken: 20-12-2023 14:42

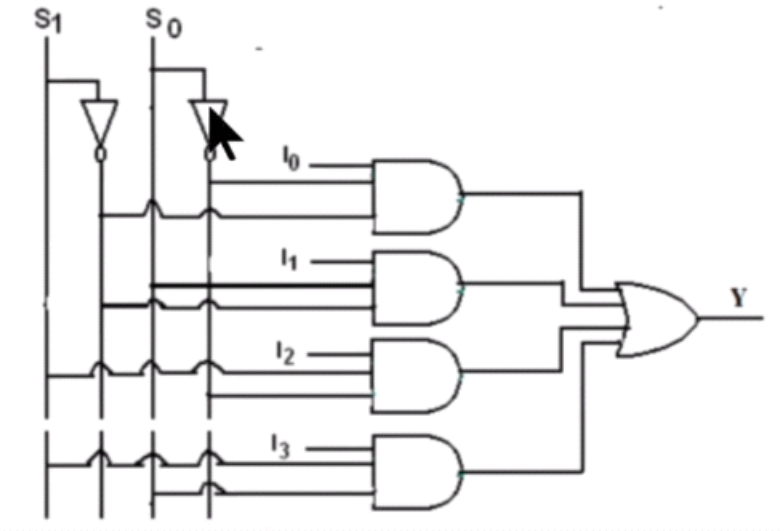
The module provided to you is: `module my_dff8 ( input clk, input [7:0] d, output [7:0] q );`

The multiplexer is not provided. One possible way to write one is inside an `always` block with a `case` statement inside. (See also: mux9to1v ⚪)

Screen clipping taken: 20-12-2023 14:42



Screen clipping taken: 20-12-2023 14:42




```
module top_module (
    input clk,
    input [7:0] d,
    input [1:0] sel,
    output [7:0] q
);
    wire [7:0] d1_out,d2_out,d3_out;
    my_dff8 d1(.clk(clk),.d(d),.q(d1_out));
    my_dff8 d2(.clk(clk),.d(d1_out),.q(d2_out));
    my_dff8 d3(.clk(clk),.d(d2_out),.q(d3_out));
    always @(*)
        begin
        case(sel)
        2'b00: q=d;
        2'b01: q=d1_out;
        2'b10: q=d2_out;
        2'b11: q=d3_out;
         endcase
        end
endmodule

```

In1=4'b001x;

In2=4'b1010;

sum=in1+in2;

Answer option are:

A)sum=4'b100x

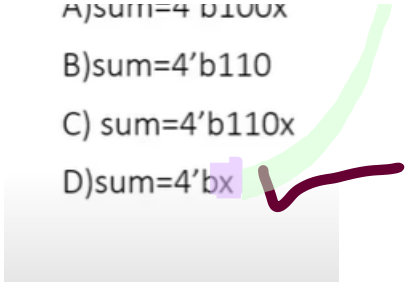
B)sum=4'b110

A)sum=4'b100x

B)sum=4'b110

C) sum=4'b110x

D)sum=4'bx



Adders

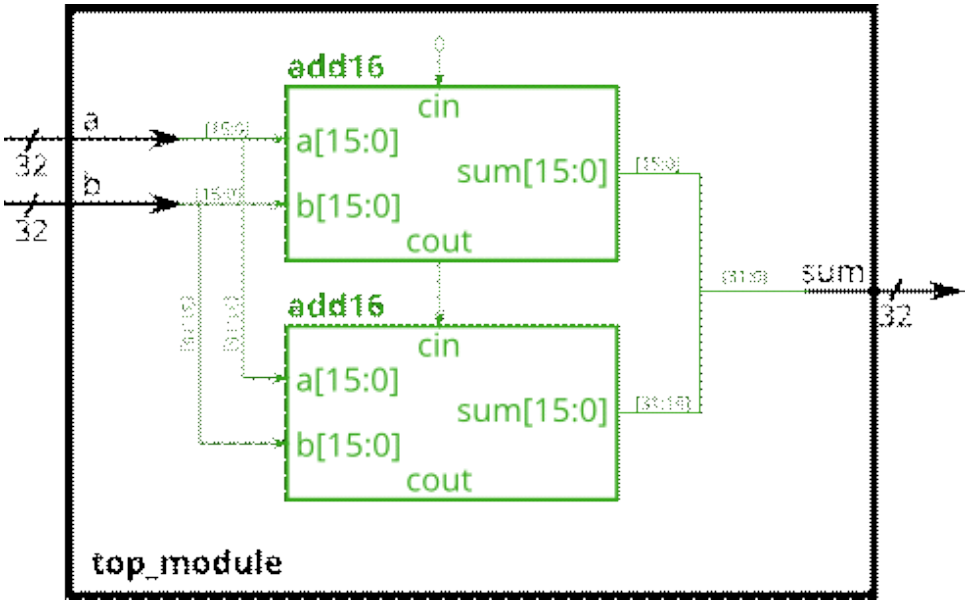
20 December 2023 15:24

Full stack VLSI mini project :
Write full adder code: synthesize using Verilog simulator to gate level and then make layout and compare both results.

You are given a module add16 that performs a 16-bit addition. Instantiate two of them to create a 32-bit adder. One add16 module computes the lower 16 bits of the addition result, while the second add16 module computes the upper 16 bits of the result, after receiving the carry-out from the first adder. Your 32-bit adder does not need to handle carry-in (assume 0) or carry-out (ignored), but the internal modules need to in order to function correctly. (In other words, the add16 module performs 16-bit a + b + cin, while your module performs 32-bit a + b).

Connect the modules together as shown in the diagram below. The provided module add16 has the following declaration:

```
module add16 ( input[15:0] a, input[15:0] b, input cin,
output[15:0] sum, output cout );
```



```
module top_module(
    input [31:0] a,
    input [31:0] b,
    output [31:0] sum
);
    wire add1_carry;
    add16 add1(.a(a[15:0]),.b(b[15:0])
    ,.sum(sum[15:0]),.cout(add1_carry));
    add16 add2(.a(a[31:16]),.b(b[31:16])
    ,.sum(sum[31:16]),.cin(add1_carry));
endmodule
```

In this exercise, you will create a circuit with two levels of hierarchy. Your `top_module` will instantiate two copies of `add16` (provided), each of which will instantiate 16 copies of `add1` (which you must write). Thus, you must write *two* modules: `top_module` and `add1`.

Like module_add ✅, you are given a module `add16` that performs a 16-bit addition. You must instantiate two of them to create a 32-bit adder. One `add16` module computes the lower 16 bits of the addition result, while the second `add16` module computes the upper 16 bits of the result. Your 32-bit adder does not need to handle carry-in (assume 0) or carry-out (ignored).

Screen clipping taken: 09-01-2024 15:26

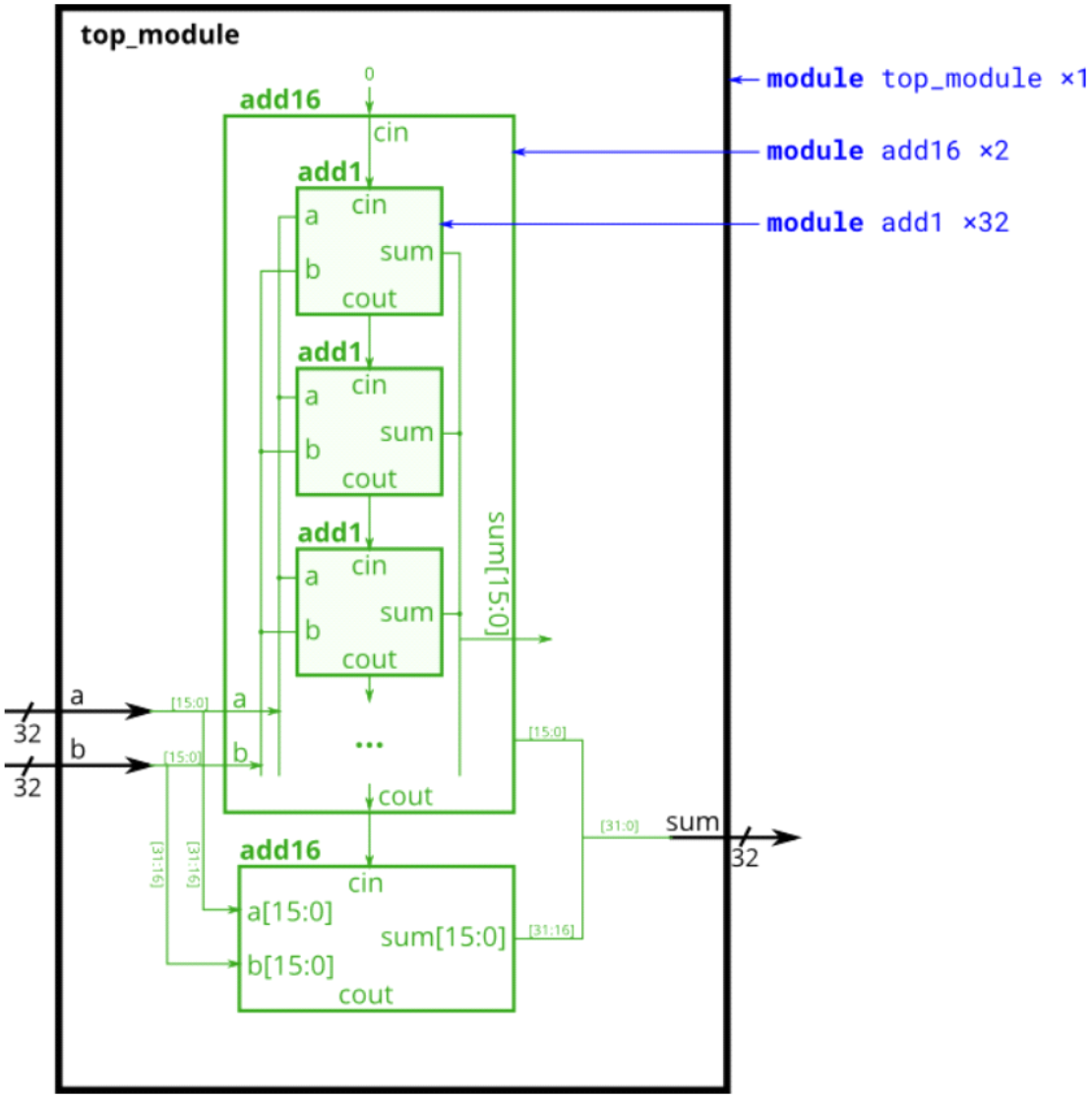
Connect the `add16` modules together as shown in the diagram below. The provided module `add16` has the following declaration:

```
module add16 ( input[15:0] a, input[15:0] b, input cin,
output[15:0] sum, output cout );
```

Within each `add16`, 16 full adders (module `add1`, not provided) are instantiated to actually perform the addition. You must write the full adder module that has the following declaration:

```
module add1 ( input a, input b, input cin, output sum,
output cout );
```

Screen clipping taken: 09-01-2024 15:27



Screen clipping taken: 09-01-2024 15:28

Recall that a full adder computes the sum and carry-out of a+b+cin.

In summary, there are three modules in this design:

- `top_module` — Your top-level module that contains two of...
- add16, provided — A 16-bit adder module that is composed of 16 of...
- `add1` — A 1-bit full adder module.

If your submission is missing a `module add1`, you will get an error message that says `Error (12006): Node instance "user_fadd[0].a1" instantiates undefined entity "add1"`.

Screen clipping taken: 09-01-2024 15:28

```
module top_module (
    input [31:0] a,
    input [31:0] b,
    output [31:0] sum
);//

endmodule

module add1 ( input a, input b, input cin,   output sum, output cout );

// Full adder module here

endmodule

```

Screen clipping taken: 09-01-2024 15:30

Code in this….

```
module top_module (
    input [31:0] a,
    input [31:0] b,
    output [31:0] sum
);//
 wire add1_carry;
    add16 add1(.a(a[15:0]),.b(b[15:0]),.sum(sum[15:0]),.cout(add1_carry));
    add16 add2(.a(a[31:16]),.b(b[31:16]),.sum(sum[31:16]),.cin(add1_carry))

endmodule

module add1 ( input a, input b, input cin,   output sum, output cout );

assign sum=a^b^cin;
    assign cout=a&b |cin &(a^b);

endmodule

```

Screen clipping taken: 12-01-2024 16:58

```
module fulladd4(output [3:0] sum,output c_out,input [3:0]a , input [3:0] b,  input c_in);
//fulladd4 definition is not required for this question
module Top;
        reg [3:0]A,B;
        reg C_in;
        reg [3:0] sum;
        wire C_out;
        fulladd4 fa0(sum, C_out, A, B, C_in )
endmodule
```

Whats the issue with this Code ?
Please Explain

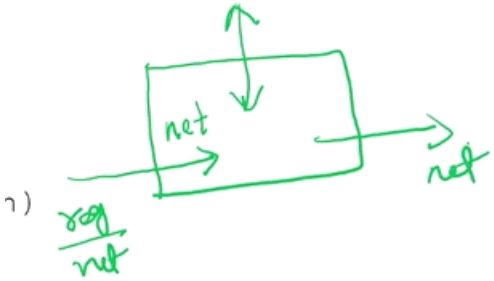
Screen clipping taken: 12-01-2024 16:59

Module without ports are also allowed.
A and B are reg type.
Correct till wire C_out it is correct.

```
module fulladd4(output [3:0] sum,output c_out,input [3:0]a , input [3:0] b,  input c_in);
//fulladd4 definition is not required for this question
module Top;
        reg [3:0]A,B;
        reg C_in;
        reg [3:0] sum;
        wire C_out;
        fulladd4 fa0(sum, C_out, A, B, C_in )
endmodule
```

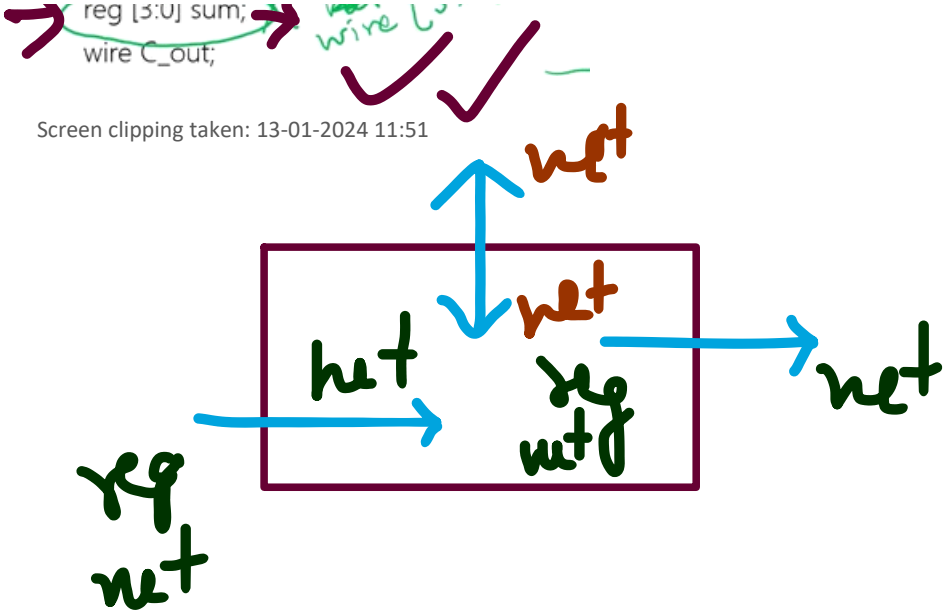
Screen clipping taken: 13-01-2024 11:46

Sum should be of net type and not register type.

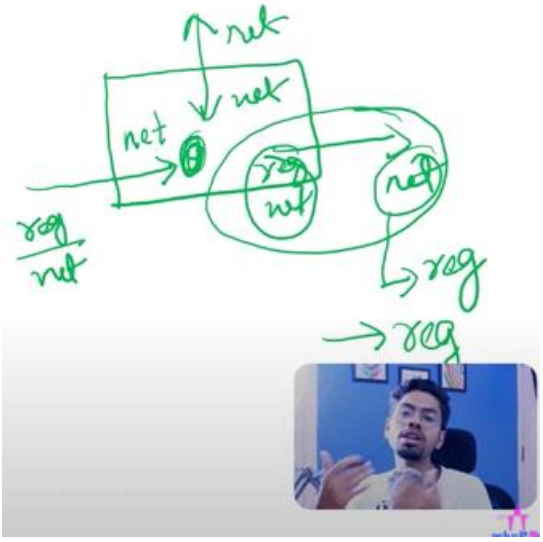


Inside should be a net type.

```
reg [3:0]A,B;
reg C_in;
reg [3:0] sum;
wire C_out;
```



Screen clipping taken: 13-01-2024 11:51



Screen clipping taken: 13-01-2024 11:57

End of arrow should be net only
Start can be reg or net
Double side arrow can be net only.

Module cseladd

module_fadd module_addsub

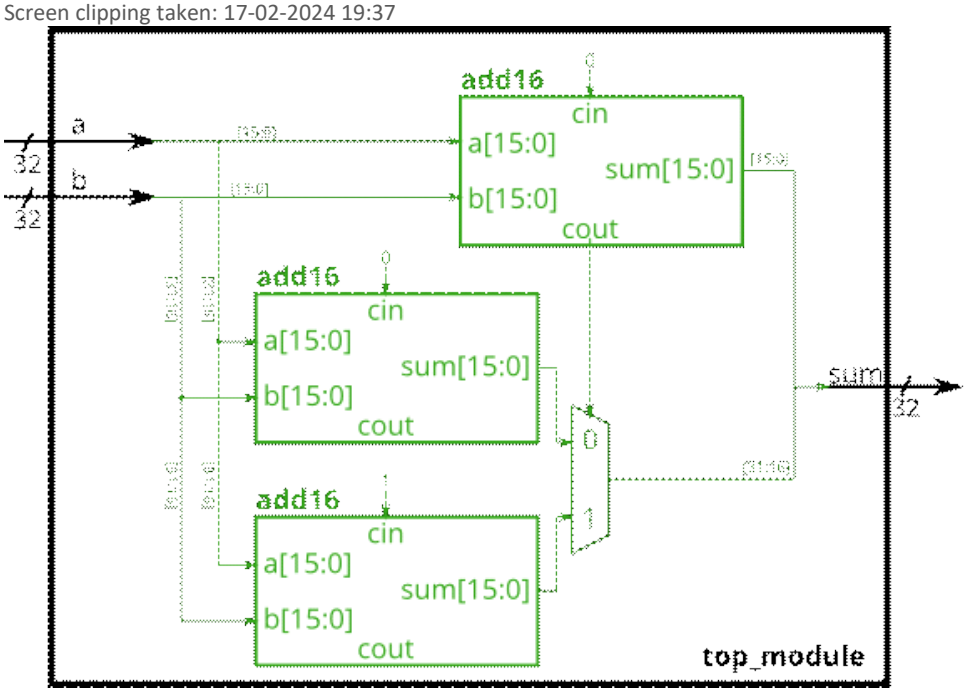
One drawback of the ripple carry adder (See previous exercise) is that the delay for an adder to compute the carry out (from the carry-in, in the worst case) is fairly slow, and the second-stage adder cannot begin computing *its* carry-out until the first-stage adder has finished. This makes the adder slow. One improvement is a carry-select adder, shown below. The first-stage adder is the same as before, but we duplicate the second-stage adder, one assuming carry-in=0 and one assuming carry-in=1, then using a fast 2-to-1 multiplexer to select which result happened to be correct.

In this exercise, you are provided with the same module add16 as the previous exercise, which adds two 16-bit numbers with carry-in and produces a carry-out and 16-bit sum. You must instantiate *three* of these to build the carry-select adder, using your own 16-bit 2-to-1 multiplexer.

Connect the modules together as shown in the diagram below. The provided module add16 has the following declaration:

```
module add16 ( input[15:0] a, input[15:0] b, input
cin, output[15:0] sum, output cout );
```

Screen clipping taken: 17-02-2024 19:37



Procedures

13 January 2024 11:53

Use of Always block

```
always @(*)
    begin
//multiplexer code
        case(sel)
            2'b00: q=d;
            2'b01: q=d1_out;
            2'b10: q=d2_out;
            2'b11: q=d3_out;
        endcase
    end
```

Screen clipping taken: 13-01-2024 12:07

Remove always and write assign

```
//multiplexer code
        case(sel)
            2'b00: assign q=d;
            2'b01: assign q=d1_out;
            2'b10: q=d2_out;
            2'b11: q=d3_out;
        endcase
```

Screen clipping taken: 13-01-2024 12:07

Case and if else are being used only in procedural block.

Since digital circuits are composed of logic gates connected with wires, any circuit can be expressed as some combination of modules and assign statements. However, sometimes this is not the most convenient way to describe the circuit. *Procedures* (of which `always` blocks are one example) provide an alternative syntax for describing circuits.

For synthesizing hardware, two types of always blocks are relevant:

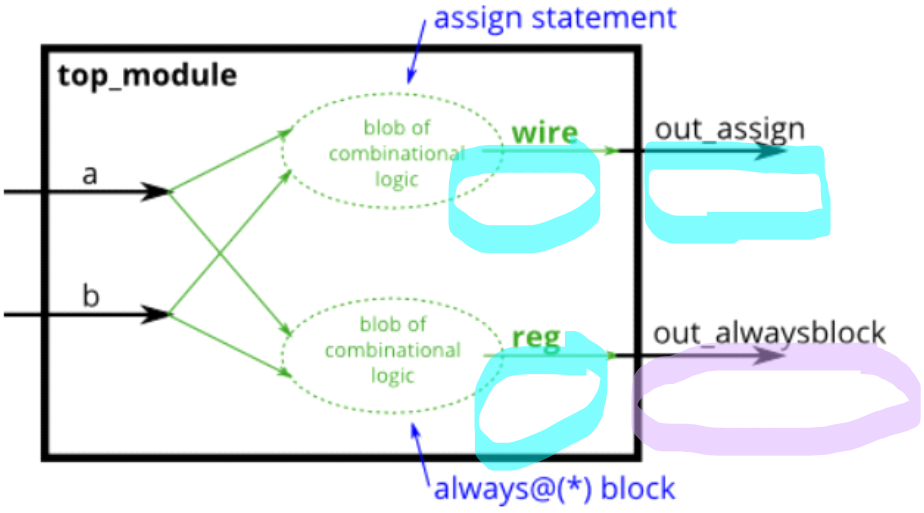
- Combinational: `always @(*)`
- Clocked: `always @(posedge clk)`

Combinational always blocks are equivalent to assign statements, thus there is always a way to express a combinational circuit both ways. The choice between which to use is mainly an issue of which syntax is more convenient. **The syntax for code inside a procedural block is different from code that is outside.** Procedural blocks have a richer set of statements (e.g., if-then, case), cannot contain continuous assignments*, but also introduces many new non-intuitive ways of making errors. (**Procedural continuous assignments* do exist, but are somewhat different from *continuous assignments*, and are not synthesizable.)

Screen clipping taken: 13-01-2024 12:08

For example, the assign and combinational always block describe the same circuit. Both create the same blob of combinational logic. Both will recompute the output whenever any of the inputs (right side) changes value.

```
assign out1 = a & b | c ^ d;
always @(*) out2 = a & b | c ^ d;
```



Screen clipping taken: 13-01-2024 12:09

For combinational always blocks, always use a sensitivity list of `(*)`. Explicitly listing out the signals is error-prone (if you miss one), and is ignored for hardware synthesis. If you explicitly specify the sensitivity list and miss a signal, the synthesized hardware will still behave as though `(*)` was specified, but the simulation will not and not match the hardware's behaviour. (In SystemVerilog, use `always_comb`.)

A note on wire vs. reg: The left-hand-side of an assign statement must be a *net* type (e.g., `wire`), while the left-hand-side of a procedural assignment (in an always block) must be a *variable* type (e.g., `reg`). These types (wire vs. reg) have nothing to do with what hardware is synthesized, and is just syntax left over from Verilog's use as a hardware *simulation* language.

Screen clipping taken: 13-01-2024 12:09

```
1 // synthesis verilog_input_version verilog_2001
2 module top_module(
3     input a,
4     input b,
5     output wire out_assign,
6     output reg out_alwaysblock
7 );
8 assign out_alwaysblock =a & b ;
9 endmodule
```

Screen clipping taken: 13-01-2024 12:20

What will be the error?

```
Error (10219): Verilog HDL Continuous Assignment error at top_module.v(8): object "out_alwaysblock" on left-hand
side of assignment must have a net type File: /home/h/work/hdlbits.12683886/top_module.v Line: 8
Error: Quartus Prime Analysis & Synthesis was unsuccessful. 1 error, 1 warning
```

Screen clipping taken: 13-01-2024 12:20

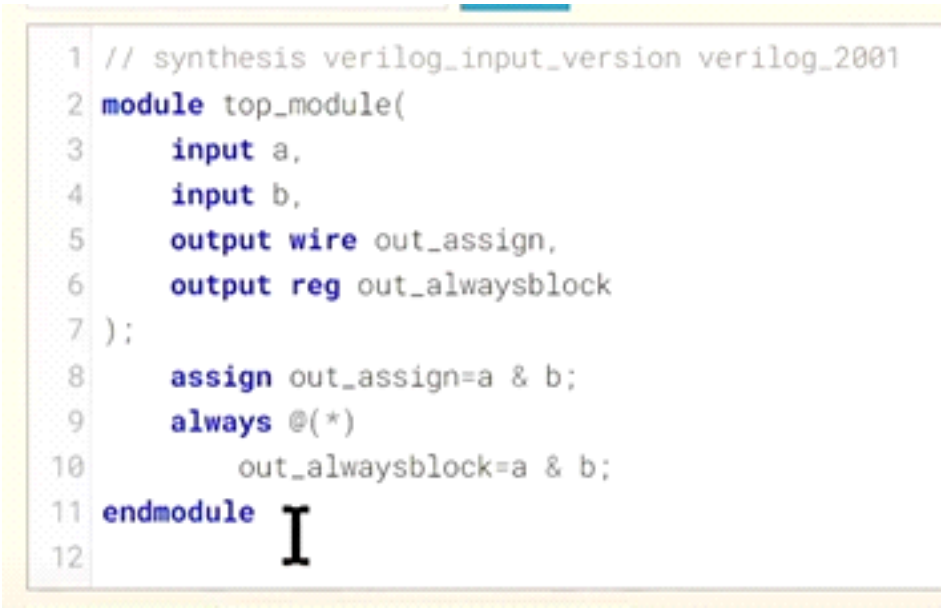
This is not a net type as it’s a reg type.

```
1 // synthesis verilog_input_version verilog_2001
2 module top_module(
3     input a,
4     input b,
5     output wire out_assign,
6     output reg out_alwaysblock
7 );
8     always @(*)
9         out_assign=a & b;
10 endmodule
```

Screen clipping taken: 13-01-2024 12:24

Use out_alwaysblock and not out_assign

as reg type is available so use that …
that's why it is giving error.



```
1  // synthesis verilog_input_version verilog_2001
2  module top_module(
3      input a,
4      input b,
5      output wire out_assign,
6      output reg out_alwaysblock
7  );
8      assign out_assign=a & b;
9      always @(*)
10         out_alwaysblock=a & b;
11 endmodule
12
```

Screen clipping taken: 13-01-2024 12:25

This is totally correct as both out_assign and out_alwaysblock are used.

A bit of practice

Build an AND gate using both an assign statement and a combinational always block. (Since assign statements and combinational always blocks function identically, there is no way to enforce that you're using both methods. But you're here for practice, right?...)

Module Declaration

```
// synthesis verilog_input_version verilog_2001
module top_module(
    input a,
    input b,
    output wire out_assign,
    output reg out_alwaysblock
);
```

Screen clipping taken: 13-01-2024 12:27

```verilog
// synthesis verilog_input_version verilog_2001
module top_module(
    input a,
    input b,
    output wire out_assign,
    output reg out_alwaysblock
);
assign out_assign = a&b;
    always @(*)
        out_alwaysblock= a&b;
endmodule

```

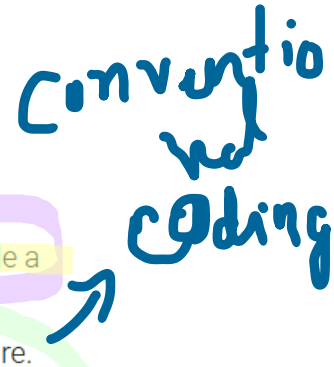
Screen clipping taken: 13-01-2024 12:27

For hardware synthesis, there are two types of `always` blocks that are relevant:

- Combinational: `always @(*)`
- Clocked: `always @(posedge clk)`

Clocked always blocks create a blob of combinational logic just like combinational always blocks, but also creates a set of flip-flops (or "registers") at the output of the blob of combinational logic. Instead of the outputs of the blob of logic being visible immediately, the outputs are visible only immediately after the next (posedge clk).

Blocking vs. Non-Blocking Assignment

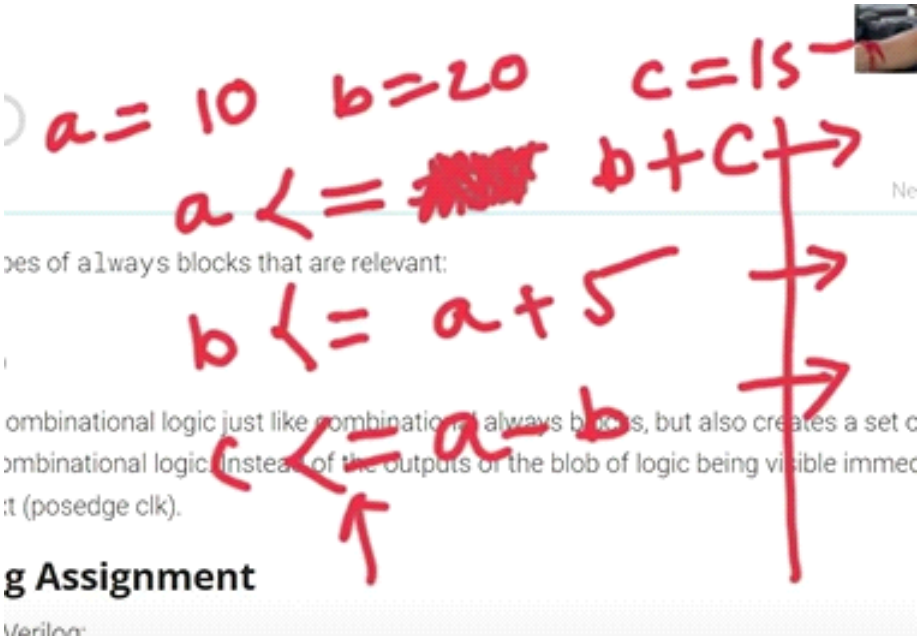


There are three types of assignments in Verilog:

- **Continuous** assignments (`assign x = y;`). Can only be used when **not** inside a procedure ("always block").
- Procedural **blocking** assignment: (`x = y;`). Can only be used inside a procedure.
- Procedural **non-blocking** assignment: (`x <= y;`). Can only be used inside a procedure.

In a **combinational** always block, use **blocking** assignments. In a **clocked** always block, use **non-blocking** assignments. A full understanding of why is not particularly useful for hardware design and requires a good understanding of how Verilog simulators keep track of events. Not following this rule results in extremely hard to find errors that are both non-deterministic and differ between simulation and synthesized hardware.

Example: for non-blocking assignment.



Screen clipping taken: 13-01-2024 15:25

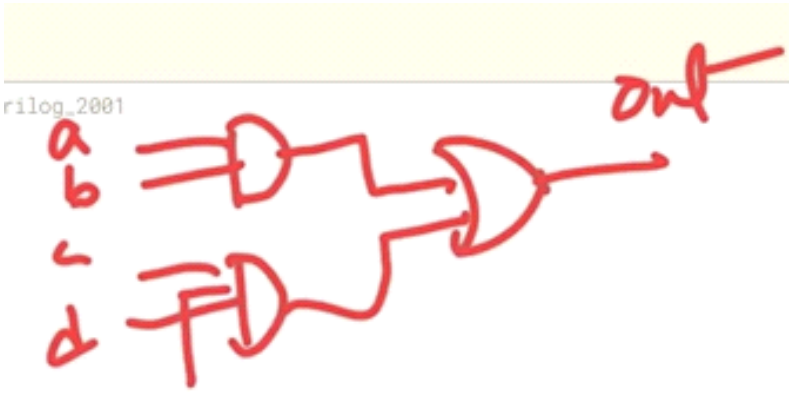
What are the answers?

All work at the same time.

a= 35

b= 15

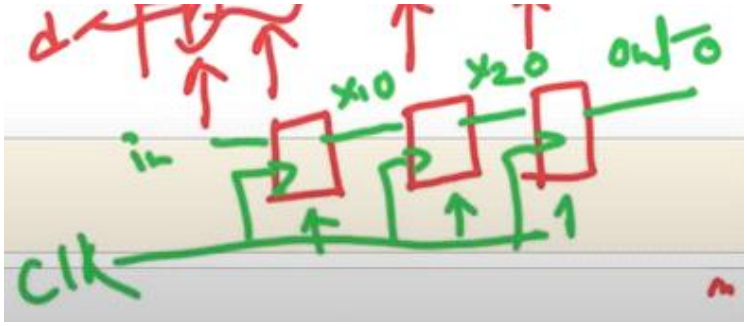
c= -10



Screen clipping taken: 13-01-2024 16:19

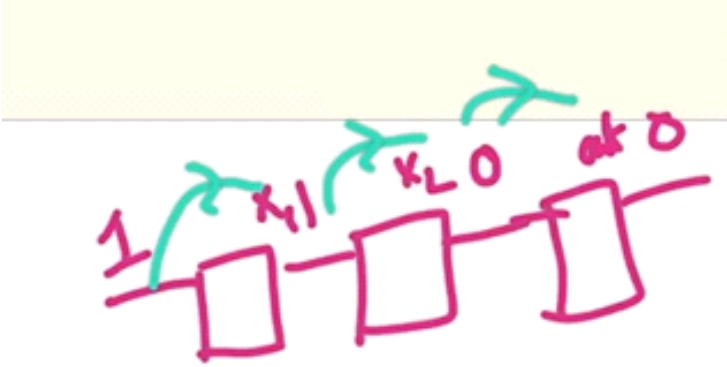
Not holding any data in between

But If you consider a shift register



Screen clipping taken: 13-01-2024 16:21

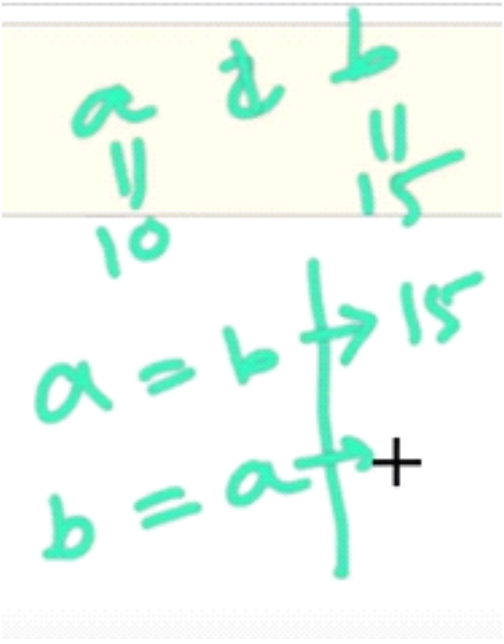
Till you get the next edge clock hold the values



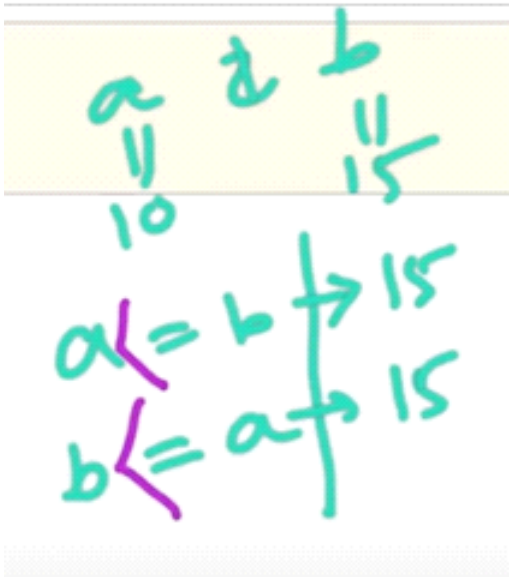
Screen clipping taken: 13-01-2024 16:22

3 Transfers are happening

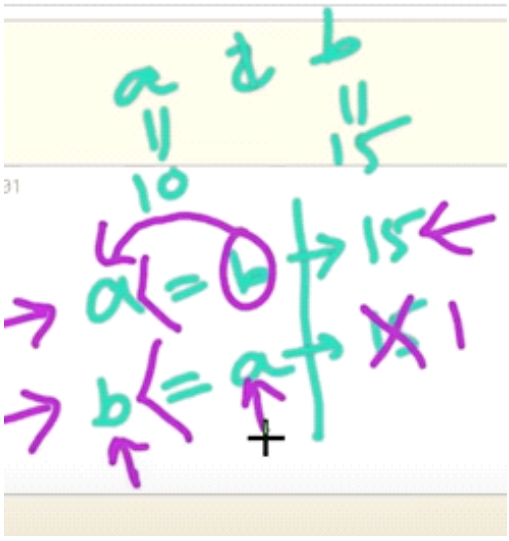
For any sequential ckt use non-blocking statement.



Screen clipping taken: 13-01-2024 16:27



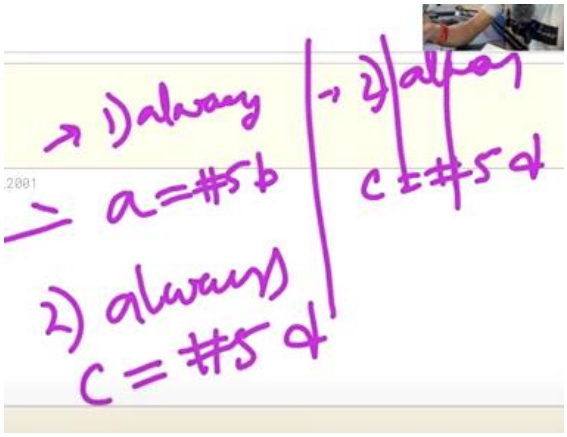
Screen clipping taken: 13-01-2024 16:27



Screen clipping taken: 13-01-2024 16:28

Swapping done perfectly with non-blocking assignment.

New topic



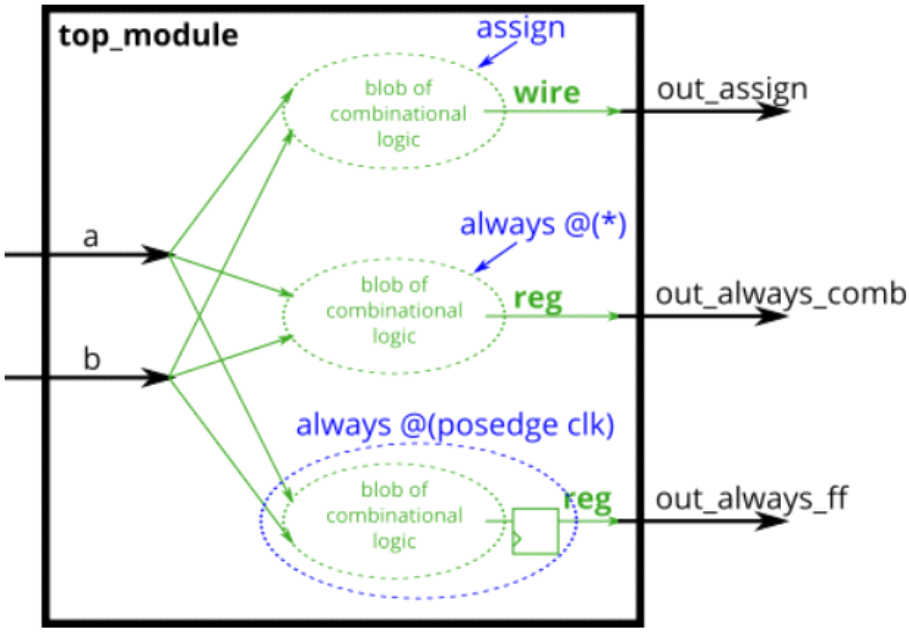
Screen clipping taken: 13-01-2024 16:31

In verilog world all the procedural block start executing at the same time.

Blocking statement will block the next statement only inside that always block.

A bit of practice

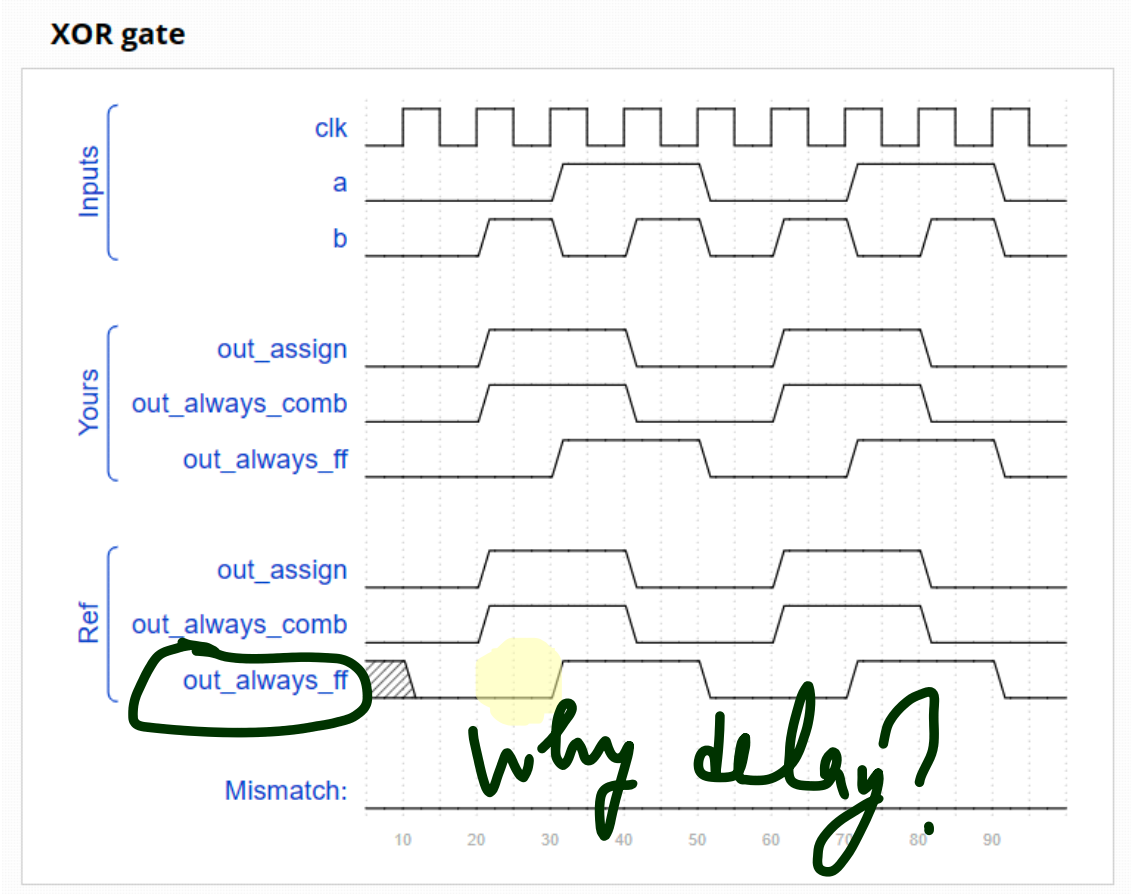
Build an XOR gate three ways, using an assign statement, a combinational always block, and a clocked always block. Note that the clocked always block produces a different circuit from the other two: There is a flip-flop so the output is delayed.



```
// synthesis verilog_input_version verilog_2001
module top_module(
    input clk,
    input a,
    input b,
    output wire out_assign,
    output reg out_always_comb,
    output reg out_always_ff   );

endmodule
```

```
// synthesis verilog_input_version verilog_2001
module top_module(
    input clk,
    input a,
    input b,
    output wire out_assign,
    output reg out_always_comb,
    output reg out_always_ff   );

   assign out_assign =a^b;
    always @(*)
        out_always_comb = a^b;
    always @(posedge clk)
        out_always_ff = a^b;
endmodule

```

Screen clipping taken: 13-01-2024 16:38

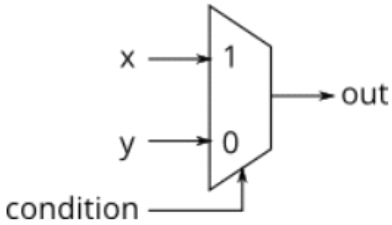


Screen clipping taken: 13-01-2024 16:39

Always if

An `if` statement usually creates a 2-to-1 multiplexer, selecting one input if the condition is true, and the other input if the condition is false.

```
always @(*) begin
    if (condition) begin
        out = x;
    end
    else begin
        out = y;
    end
end
```



This is equivalent to using a continuous `assign`ment with a conditional operator:

```
assign out = (condition) ? x : y;
```

However, the procedural `if` statement provides a new way to make mistakes. The circuit is combinational only if `out` is always assigned a value.

A bit of practice

Build a 2-to-1 mux that chooses between a and b. Choose b if *both* `sel_b1` and `sel_b2` are true. Otherwise, choose a. Do the same twice, once using `assign` statements and once using a procedural if statement.

sel_b1	sel_b2	out_assign out_always
0	0	a
0	1	a
1	0	a
1	1	b

```
// synthesis verilog_input_version verilog_2001
module top_module(
    input a,
    input b,
    input sel_b1,
    input sel_b2,
    output wire out_assign,
    output reg out_always   );

endmodule

```

```verilog
// synthesis verilog_input_version verilog_2001
module top_module(
    input a,
    input b,
    input sel_b1,
    input sel_b2,
    output wire out_assign,
    output reg out_always );

    assign out_assign = (sel_b1 && sel_b2) ? b : a;
    always @(*)
        begin
            if (sel_b1 && sel_b2)
                out_always =b;
            else
               out_always =a;
        end
endmodule

```

Screen clipping taken: 13-01-2024 17:09

A common source of errors: How to avoid making latches

When designing circuits, you *must* think first in terms of circuits:

- I want this logic gate
- I want a *combinational* blob of logic that has these inputs and produces these outputs
- I want a combinational blob of logic followed by a set of flip-flops

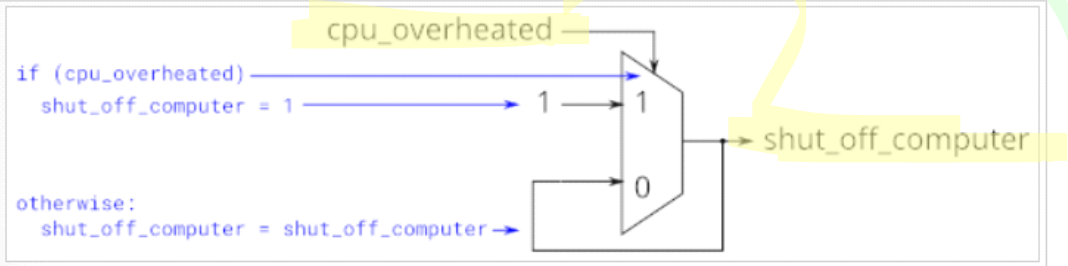
What you *must not* do is write the code first, then hope it generates a proper circuit.

- If (cpu_overheated) then shut_off_computer = 1;
- If (~arrived) then keep_driving = ~gas_tank_empty;

Screen clipping taken: 08-02-2024 16:07

Demonstration

The following code contains incorrect behaviour that creates a latch. Fix the bugs so that you will shut off the computer only if it's really overheated, and stop driving if you've arrived at your destination or you need to refuel.



This is the circuit described by the code, not the circuit you want to build.

Screen clipping taken: 08-02-2024 16:14

```
always @(*) begin
    if (cpu_overheated)
        shut_off_computer = 1;
end
```

if no else used then?

```
always @(*) begin
    if (~arrived)
        keep_driving = ~gas_tank_empty;
end
```

Screen clipping taken: 08-02-2024 16:14

```verilog
// synthesis verilog_input_version verilog_2001
module top_module (
    input      cpu_overheated,
    output reg shut_off_computer,
    input      arrived,
    input      gas_tank_empty,
    output reg keep_driving  ); //

    always @(*) begin
        if (cpu_overheated)
            shut_off_computer = 1;
       else
           shut_off_computer = 0;
    end
    always @(*) begin
        if (~arrived && ~gas_tank_empty)
            keep_driving = 1;
    else
            keep_driving =0;
    end
endmodule

```

Screen clipping taken: 08-02-2024 16:35

Always case

always_if2 always_case2

Case statements in Verilog are nearly equivalent to a sequence of if-elseif-else that compares one expression to a list of others. Its syntax and functionality differs from the `switch` statement in C.

```
always @(*) begin     // This is a combinational circuit
    case (in)
      1'b1: begin
                out = 1'b1;  // begin-end if >1 statement
            end
      1'b0: out = 1'b0;
      default: out = 1'bx;
    endcase
end
```

Screen clipping taken: 08-02-2024 16:37

- The case statement begins with `case` and each "case item" ends with a colon. There is no "switch".
- Each case item can execute *exactly one* statement. This makes the "break" used in C unnecessary. But this means that if you need more than one statement, you must use `begin ... end`.
- Duplicate (and partially overlapping) case items are permitted. The first one that matches is used. C does not allow duplicate case items.

Screen clipping taken: 08-02-2024 16:39

A bit of practice

Case statements are more convenient than if statements if there are a large number of cases. So, in this exercise, create a 6-to-1 multiplexer. When `sel` is between 0 and 5, choose the corresponding data input. Otherwise, output 0. The data inputs and outputs are all 4 bits wide.

Be careful of inferring latches (See.always_if2)

Screen clipping taken: 08-02-2024 16:42

Write your solution here

[Load a previous submission] Load

```
// synthesis verilog_input_version verilog_2001
module top_module (
    input [2:0] sel,
    input [3:0] data0,
    input [3:0] data1,
    input [3:0] data2,
    input [3:0] data3,
    input [3:0] data4,
    input [3:0] data5,
    output reg [3:0] out   );//

    always@(*) begin  // This is a combinational circuit
        case(sel)
            3'b000: out =data0;
            3'b001: out =data1;
            3'b010: out =data2;
            3'b011: out =data3;
            3'b100: out =data4;
            3'b101: out =data5;
            default: out=1'b0;
        endcase
    end

endmodule

```

Screen clipping taken: 08-02-2024 16:43

Always case2

always_case ✓ always_casez

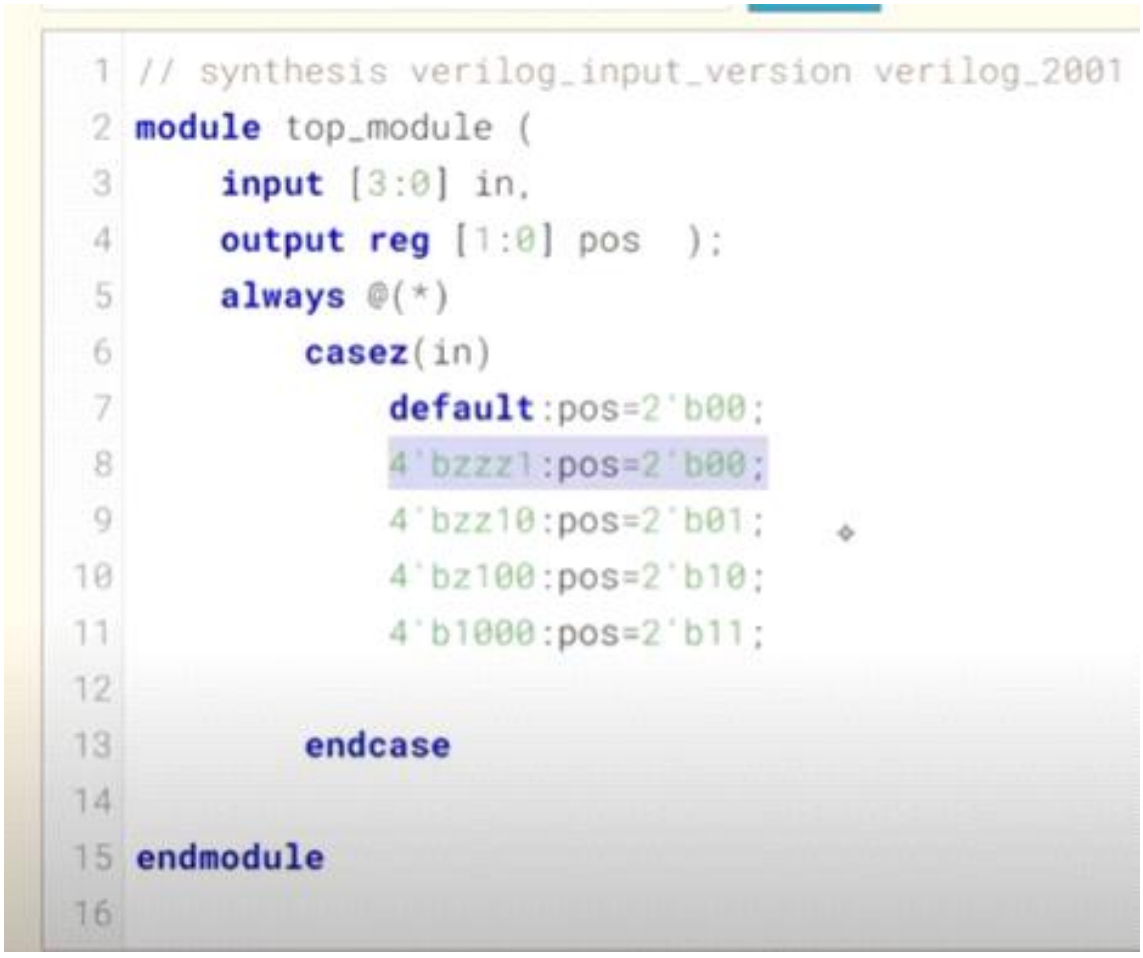
A *priority encoder* is a combinational circuit that, when given an input bit vector, outputs the position of the first 1 bit in the vector. For example, a 8-bit priority encoder given the input 8'b10010000 would output 3'd4, because bit[4] is first bit that is high.

Build a 4-bit priority encoder. For this problem, if none of the input bits are high (i.e., input is zero), output zero. Note that a 4-bit number has 16 possible combinations.

Screen clipping taken: 08-02-2024 16:45

```verilog
// synthesis verilog_input_version verilog_2001
module top_module (
    input [3:0] in,
    output reg [1:0] pos  );
    always@(*)
        case(in)
            4'b0000:pos=2'd0;
            4'b0001:pos=2'd0;
            4'b0010:pos=2'd1;
            4'b0011:pos=2'd0;
            4'b0100:pos=2'd2;
            4'b0101:pos=2'd0;
            4'b0110:pos=2'd1;
            4'b0111:pos=2'd0;
            4'b1000:pos=2'd3;
            4'b1001:pos=2'd0;
            4'b1010:pos=2'd1;
            4'b1011:pos=2'd0;
            4'b1100:pos=2'd2;
            4'b1101:pos=2'd0;
            4'b1110:pos=2'd1;
            4'b1111:pos=2'd0;
        endcase
endmodule

```

Using casez



```
1  // synthesis verilog_input_version verilog_2001
2  module top_module (
3      input [3:0] in,
4      output reg [1:0] pos  );
5      always @(*)
6          casez(in)
7              default:pos=2'b00;
8              4'bzzz1:pos=2'b00;
9              4'bzz10:pos=2'b01;
10             4'bz100:pos=2'b10;
11             4'b1000:pos=2'b11;
12
13         endcase
14
15 endmodule
16
```

Screen clipping taken: 08-02-2024 18:18

Always casez

always_case2 always_nolatches

Build a priority encoder for 8-bit inputs. Given an 8-bit vector, the output should report the first (least significant) bit in the vector that is 1. Report zero if the input vector has no bits that are high. For example, the input 8'b10010000 should output 3'd4, because bit[4] is first bit that is high.

From the previous exercise (always_case2), there would be 256 cases in the case statement. We can reduce this (down to 9 cases) if the case items in the case statement supported don't-care bits. This is what **casez** is for: It treats bits that have the value z as don't-care in the comparison.

For example, this would implement the 4-input priority encoder from the previous exercise:

```
always @(*) begin
    casez (in[3:0])
        4'bzzz1: out = 0;    // in[3:1] can be anything
        4'bzz1z: out = 1;
        4'bz1zz: out = 2;
        4'b1zzz: out = 3;
        default: out = 0;
    endcase
end
```

A case statement behaves as though each item is checked sequentially (in reality, a big combinational logic function). Notice how there are certain inputs (e.g., `4'b1111`) that will match more than one case item. The first match is chosen (so `4'b1111` matches the first item, `out = 0`, but not any of the later ones).

- There is also a similar `casex` that treats both `x` and `z` as don't-care. I don't see much purpose to using it over `casez`.
- The digit `?` is a synonym for `z`. so `2'bz0` is the same as `2'b?0`

It may be less error-prone to explicitly specify the priority behaviour rather than rely on the ordering of the case items. For example, the following will still behave the same way if some of the case items were reordered, because any bit pattern can only match at most one case item:

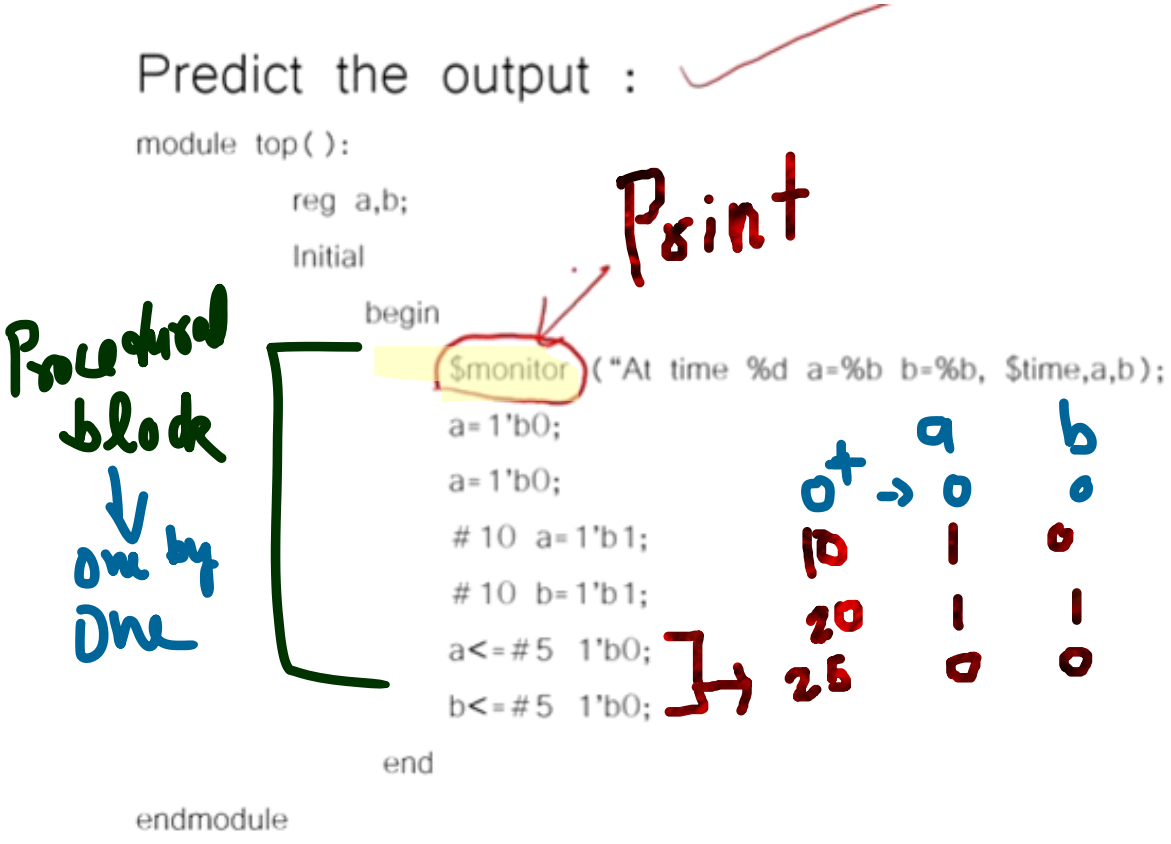
```
casez (in[3:0])
    4'bzzz1: ...
    4'bzz10: ...
    4'bz100: ...
    4'b1000: ...
    default: ...
endcase
```

Screen clipping taken: 08-02-2024 18:06

```
// synthesis verilog_input_version verilog_2001
module top_module (
    input [7:0] in,
    output reg [2:0] pos );
    always@(*)
        casez(in)
            default:pos=3'b000;
            8'bzzzzzzz1:pos=3'b000;
            8'bzzzzzz10:pos=3'b001;
            8'bzzzzz100:pos=3'b010;
            8'bzzzz1000:pos=3'b011;
            8'bzzz10000:pos=3'b100;
            8'bzz100000:pos=3'b101;
            8'bz1000000:pos=3'b110;
            8'b10000000:pos=3'b111;
        endcase
endmodule

```

Screen clipping taken: 08-02-2024 18:20

Predict the output :



Always nolatches

always_casez Previous Next conditional

Suppose you're building a circuit to process scancodes from a PS/2 keyboard for a game. Given the last two bytes of scancodes received, you need to indicate whether one of the arrow keys on the keyboard have been pressed. This involves a fairly simple mapping, which can be implemented as a case statement (or if-elseif) with four cases.

Scancode [15:0]	Arrow key
16'he06b	left arrow
16'he072	down arrow
16'he074	right arrow
16'he075	up arrow
Anything else	none

Your circuit has one 16-bit input, and four outputs. Build this circuit that recognizes these four scancodes and asserts the correct output.

Screen clipping taken: 08-02-2024 18:24

To avoid creating latches, all outputs must be assigned a value in all possible conditions (See also always_if2). Simply having a `default` case is not enough. You must assign a value to all four outputs in all four cases and the default case. This can involve a lot of unnecessary typing. One easy way around this is to assign a "default value" to the outputs *before* the case statement:

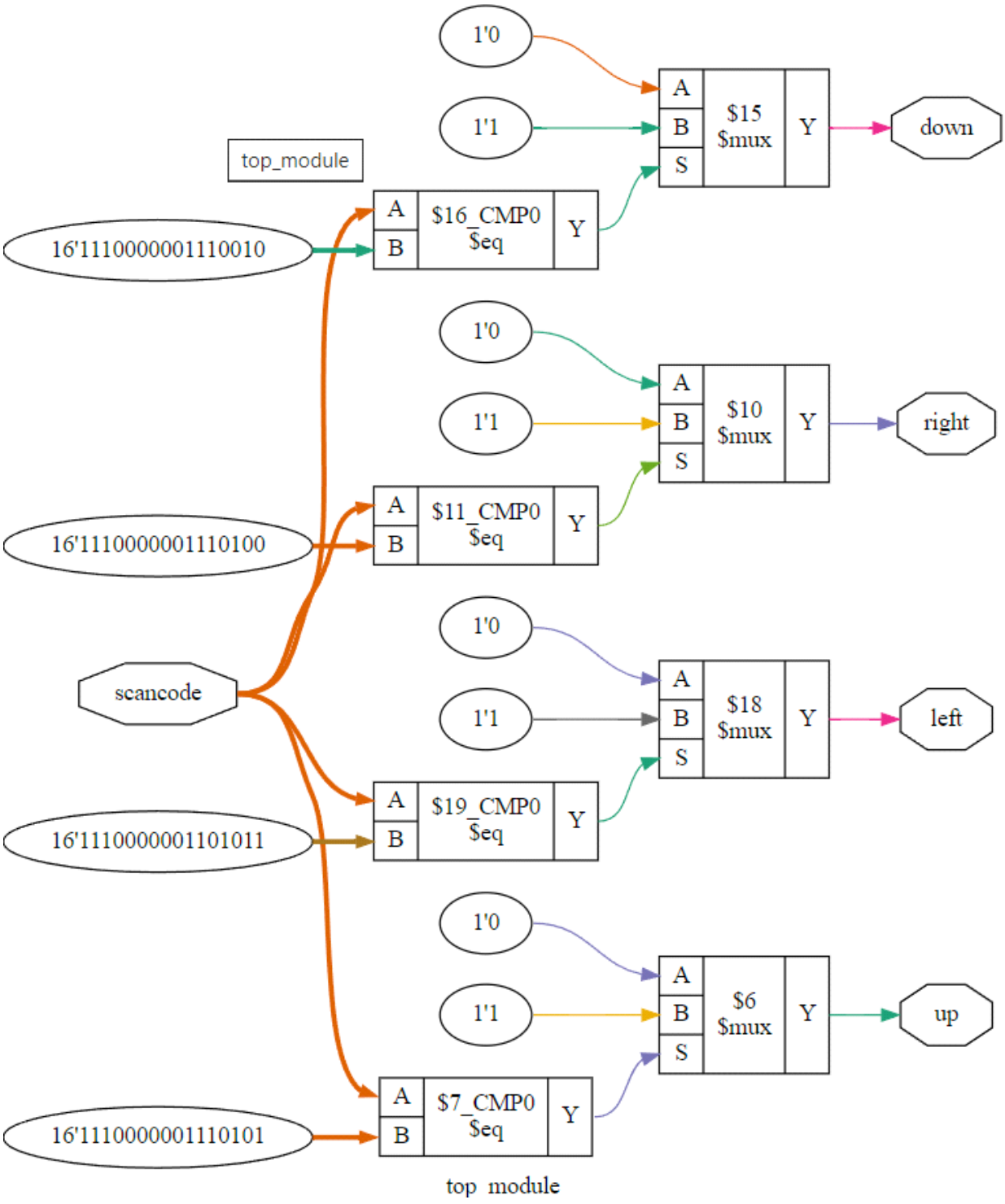
```
always @(*) begin
    up = 1'b0; down = 1'b0; left = 1'b0; right = 1'b0;
    case (scancode)
        ... // Set to 1 as necessary.
    endcase
end
```

This style of code ensures the outputs are assigned a value (of 0) in all possible cases unless the case statement overrides the assignment. This also means that a `default :` case item becomes unnecessary.

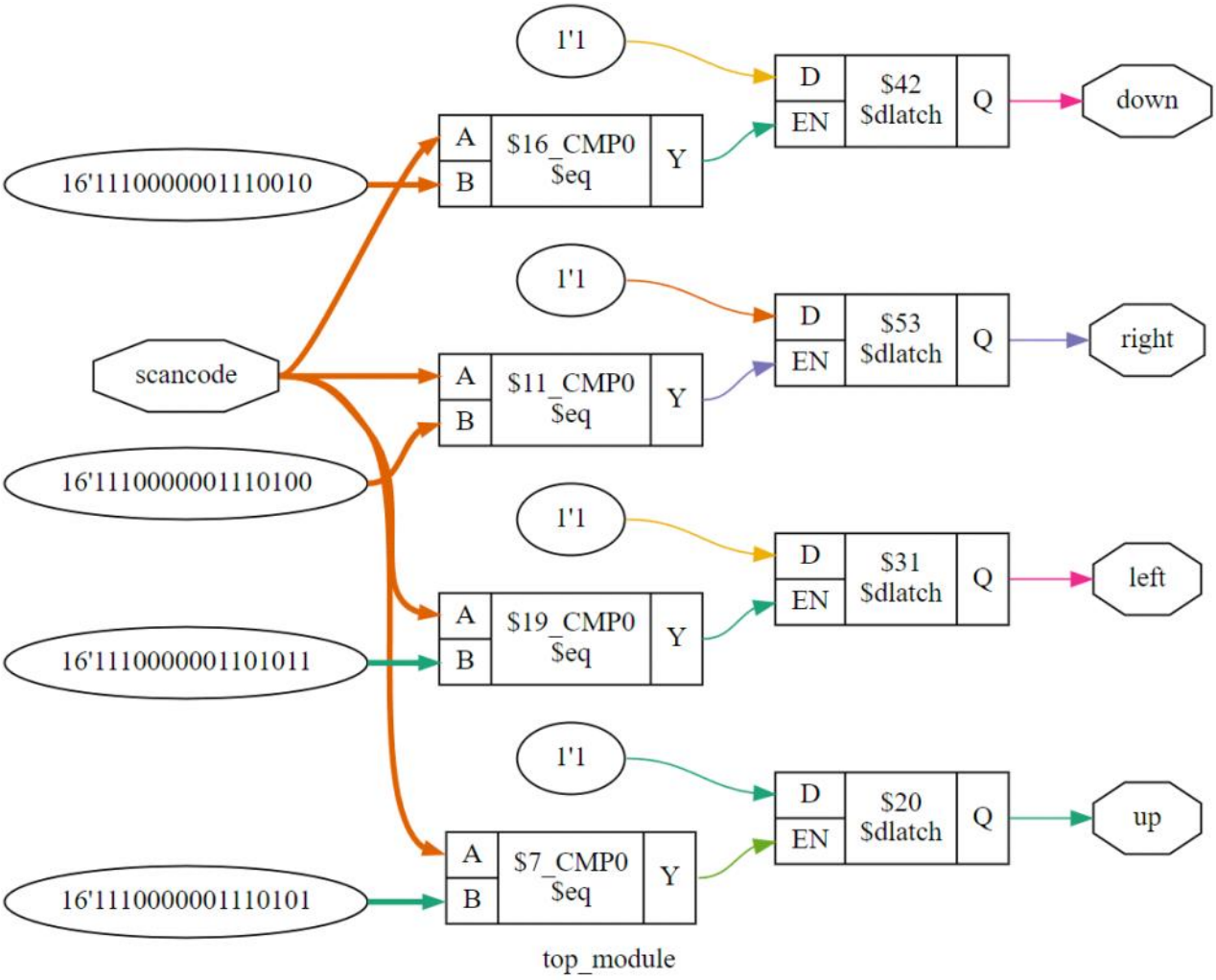
Screen clipping taken: 08-02-2024 18:27

```
// synthesis verilog_input_version verilog_2001
module top_module (
    input [15:0] scancode,
    output reg left,
    output reg down,
    output reg right,
    output reg up  );
    always @(*) begin
        up =16'h0;
        down =16'h0;
        right =16'h0;
        left =16'h0;
        case(scancode)
            16'he075:up=1'b1;
            16'he072:down=1'b1;
            16'he074:right=1'b1;
            16'he06b:left=1'b1;
        endcase
    end
endmodule

```

Screen clipping taken: 08-02-2024 18:38



Screen clipping taken: 08-02-2024 21:37


1'1
D
EN
$42
$dlatch
Q
down
A
B
$16_CMP0
$eq
Y
16'1110000001110010
1'1
D
EN
$53
$dlatch
Q
right
scancode
A
B
$11_CMP0
$eq
Y
16'1110000001110100
1'1
D
EN
$31
$dlatch
Q
left
A
B
$19_CMP0
$eq
Y
16'1110000001101011
1'1
D
EN
$20
$dlatch
Q
up
A
B
$7_CMP0
$eq
Y
16'1110000001110101
top_module

More Verilog Features

08 February 2024 18:21

Conditional

always_nolatches reduction

Verilog has a ternary conditional operator (? :) much like C:

```
(condition ? if_true : if_false)
```

This can be used to choose one of two values based on *condition* (a mux!) on one line, without using an if-then inside a combinational always block.

Examples:

```
(0 ? 3 : 5)     // This is 5 because the condition is
false.
(sel ? b : a)   // A 2-to-1 multiplexer between a and b
selected by sel.

always @(posedge clk)          // A T-flip-flop.
  q <= toggle ? ~q : q;

always @(*)                    // State transition logic for
a one-input FSM
  case (state)
    A: next = w ? B : A;
    B: next = w ? A : B;
  endcase
    assign out = ena ? q : 1'bz;  // A tri-state buffer

    ((sel[1:0] == 2'h0) ? a :     // A 3-to-1 mux
     (sel[1:0] == 2'h1) ? b :
                          c )
```

A Bit of Practice

Given four unsigned numbers, find the minimum. Unsigned numbers can be compared with standard comparison operators (a < b). Use the conditional operator to make two-way *min* circuits, then compose a few of them to create a 4-way *min* circuit. You'll probably want some wire vectors for the intermediate results.

Screen clipping taken: 08-02-2024 21:54

```
module top_module (
    input [7:0] a, b, c, d,
    output [7:0] min);//
    wire [7:0]t1,t2;
    assign t1=a<b?a:b;
    assign t2=c<d?c:d;
    assign min =t1<t2?t1:t2;
endmodule

```

Screen clipping taken: 08-02-2024 21:59

Reduction

conditional gates100

You're already familiar with bitwise operations between two values, e.g., `a & b` or `a ^ b`. Sometimes, you want to create a wide gate that operates on all of the bits of *one* vector, like `(a[0] & a[1] & a[2] & a[3] ...)`, which gets tedious if the vector is long.

The *reduction* operators can do AND, OR, and XOR of the bits of a vector, producing one bit of output:

```
& a[3:0]      // AND: a[3]&a[2]&a[1]&a[0].
Equivalent to (a[3:0] == 4'hf)
| b[3:0]      // OR:  b[3]|b[2]|b[1]|b[0].
Equivalent to (b[3:0] != 4'h0)
^ c[2:0]      // XOR: c[2]^c[1]^c[0]
```

These are *unary* operators that have only one operand (similar to the NOT operators ! and ~). You can also invert the outputs of these to create NAND, NOR, and XNOR gates, e.g., `(~& d[7:0])`.

A Bit of Practice

Parity checking is often used as a simple method of detecting errors when transmitting data through an imperfect channel. Create a circuit that will compute a parity bit for a 8-bit byte (which will add a 9th bit to the byte). We will use "even" parity, where the parity bit is just the XOR of all 8 data bits.

```
module top_module (
    input [7:0] in,
    output parity);
    assign parity=^in[7:0];
endmodule

```

Gates100

reduction vector100r

Build a combinational circuit with 100 inputs, `in[99:0]`.

There are 3 outputs:

- out_and: output of a 100-input AND gate.
- out_or: output of a 100-input OR gate.
- out_xor: output of a 100-input XOR gate.

```verilog
module top_module(
    input [99:0] in,
    output out_and,
    output out_or,
    output out_xor
);
assign out_and=&in;
assign out_or=|in;
assign out_xor=^in;
endmodule

```

Screen clipping taken: 08-02-2024 22:34

Given a 100-bit input vector [99:0], reverse its bit ordering.

Module Declaration

```verilog
module top_module(
    input [99:0] in,
    output [99:0] out
);
```

Screen clipping taken: 08-02-2024 22:36

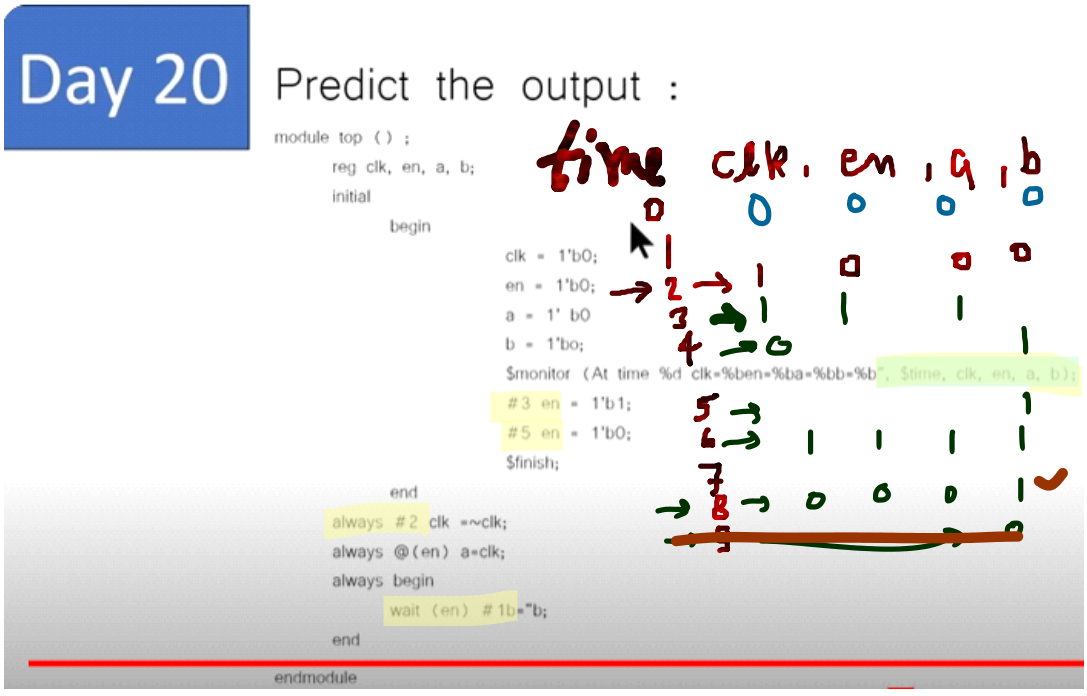
```verilog
module top_module(
    input [99:0] in,
    output [99:0] out
);
    always @(*) begin
        for(int i=0 ; i<100 ; i++)
            out[99-i]=in[i];
    end
endmodule

```

Screen clipping taken: 08-02-2024 23:08

```
1  module top_module(
2      input [99:0] in,
3      output [99:0] out
4  );
5      genvar i;
6      generate
7      always @(*) begin
8          for(int i=0 ; i<100 ; i++)
9              out[99-i]=in[i];
10     end
11     endgenerate
12 endmodule
13
```

Screen clipping taken: 08-02-2024 23:14

All always block run in parallel.



Screen clipping taken: 08-02-2024 23:18

Popcount255

vector100r adder100i

A "population count" circuit counts the number of '1's in an input vector. Build a population count circuit for a 255-bit input vector.

Module Declaration

```
module top_module(
    input [254:0] in,
    output [7:0] out );
```

```
module top_module(
    input [254:0] in,
    output [7:0] out );
genvar i;
    generate
        always @(*)
            begin
         out =8'b00000000;
         for(int i=0; i<255; i++)
             begin
            if (in[i]==1'b1)
            out=out+1;
            else
            out=out;
            end
        end
    endgenerate
endmodule

```

Screen clipping taken: 09-02-2024 15:19

Adder100i

popcount255 bcdadd100

Create a 100-bit binary ripple-carry adder by instantiating 100 full adders. The adder adds two 100-bit numbers and a carry-in to produce a 100-bit sum and carry out. To encourage you to actually instantiate full adders, also output the carry-out from *each* full adder in the ripple-carry adder. cout[99] is the final carry-out from the last full adder, and is the carry-out you usually see.

Module Declaration

```
module top_module(
    input [99:0] a, b,
    input cin,
    output [99:0] cout,
    output [99:0] sum );
```

Screen clipping taken: 09-02-2024 15:34

Screen clipping taken: 09-02-2024 15:34

```
module top_module(
    input [99:0] a, b,
    input cin,
    output [99:0] cout,
    output [99:0] sum );
    assign {cout[0],sum[0]} = a[0]+b[0]+cin;
    genvar i;
    generate
        begin
            for(i=1; i<100 ; i=i+1)
            begin :gen
                FA fa1(.fa_sum(sum[i]),
                       .fa_cout(cout[i]),
                       .in1(a[i]),
                       .in2(b[i]),
                       .cin(cout[i-1]));
            end :gen
        end
   endgenerate
endmodule

module FA(output fa_sum,fa_cout,
          input in1,in2,cin);
    assign {fa_cout,fa_sum}=in1+in2+cin;
endmodule
```

Bcdadd100

adder100i exams/m2014_q4h

You are provided with a BCD one-digit adder named bcd_fadd that adds two BCD digits and carry-in, and produces a sum and carry-out.

```
module bcd_fadd (
    input [3:0] a,
    input [3:0] b,
    input     cin,
    output   cout,
    output [3:0] sum );
```

Instantiate 100 copies of bcd_fadd to create a 100-digit BCD ripple-carry adder. Your adder should add two 100-digit BCD numbers (packed into 400-bit vectors) and a carry-in to produce a 100-digit sum and carry out.

Screen clipping taken: 09-02-2024 18:26

```
module top_module(
    input [399:0] a, b,
    input cin,
    output cout,
    output [399:0] sum );
    wire[99:0] cout_wires;
    genvar i;
    generate
    bcd_fadd(
    a[3:0],
    b[3:0],
    cin,
    cout_wires[0],
    sum[3:0]);
        begin
            for(i=4;i<400;i=i+4)
                begin :bcd
                bcd_fadd bcd(.sum(sum[i+3:i]),
                      .cout(cout_wires[i/4]),
                      .a(a[i+3:i]),
                      .b(b[i+3:i]),
                      .cin(cout_wires[i/4-1]));
                end:bcd
            end
      endgenerate
    assign cout=cout_wires[99];
endmodule
```

Screen clipping taken: 09-02-2024 22:02

Basic gates

08 February 2024 22:20

Implement the following circuit:

Module Declaration

```
module top_module (
    input in,
    output out);
```

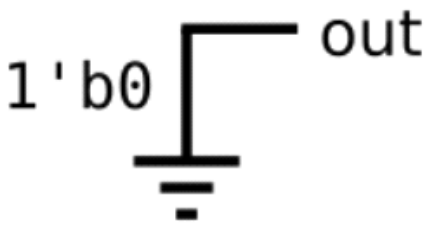
Write your solution here

[Load a previous submission] Load

```
1 module top_module (
2     input in,
3     output out);
4 assign out=in;
5 endmodule
6
```

Screen clipping taken: 09-02-2024 22:06

Implement the following circuit:



Module Declaration

```
module top_module (
    output out);
```

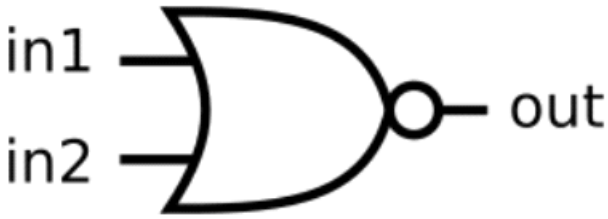
Write your solution here

[Load a previous submission] Load

```
1  module top_module (
2      output out);
3  assign out=1'b0;
4  endmodule
5
```

Screen clipping taken: 09-02-2024 22:07

Implement the following circuit:



Module Declaration

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

Write your solution here

[Load a previous submission] Load

```
1 module top_module (
2     input in1,
3     input in2,
4     output out);
5     assign out=~(in1||in2);
6 endmodule
7
```

Screen clipping taken: 09-02-2024 22:11

Implement the following circuit:

Module Declaration

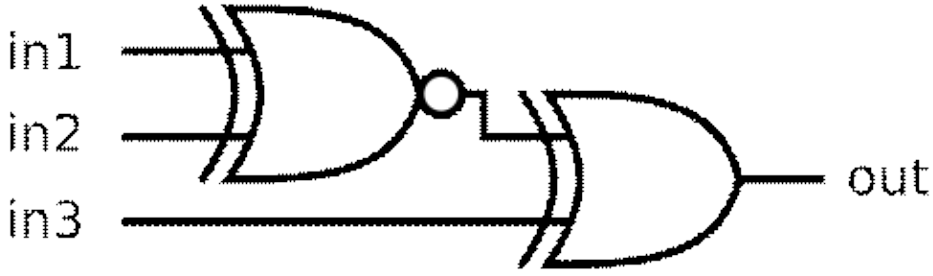
```
module top_module (
    input in1,
    input in2,
    output out);
```

Write your solution here

[Load a previous submission] Load

```
module top_module (
    input in1,
    input in2,
    output out);
    assign out=in1&(~in2);
endmodule

```

Screen clipping taken: 09-02-2024 22:14




```
module top_module (
    input in1,
    input in2,
    input in3,
    output out);
    wire t;
    assign t=~(in1^in2);
    assign out=t^in3;
endmodule

```

Screen clipping taken: 09-02-2024 22:16

Ok, let's try building several logic gates at the same time. Build a combinational circuit with two inputs, a and b.

There are 7 outputs, each with a logic gate driving it:

- out_and: a and b
- out_or: a or b
- out_xor: a xor b
- out_nand: a nand b
- out_nor: a nor b
- out_xnor: a xnor b
- out_anotb: a and-not b

Expected solution length: *Around 7 lines.*

Screen clipping taken: 09-02-2024 22:19

```
module top_module(
    input a, b,
    output out_and,
    output out_or,
    output out_xor,
    output out_nand,
    output out_nor,
    output out_xnor,
    output out_anotb
);
    assign out_and =a&b;
    assign out_or =a|b;
    assign out_xor =a^b;
    assign out_nand =~(a&b);
    assign out_nor =~(a|b);
    assign out_xnor =~(a^b);
    assign out_anotb =a&(~b);
endmodule

```

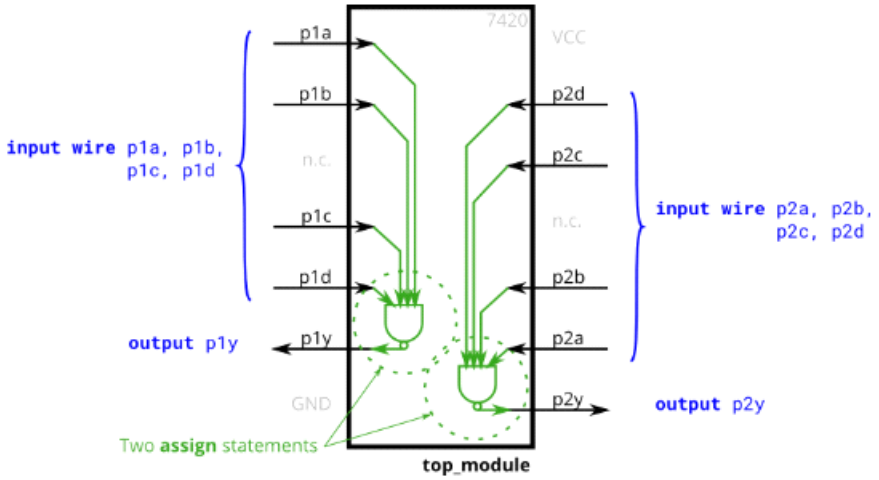
Screen clipping taken: 09-02-2024 22:19

7420

gates truthtable1

The 7400-series integrated circuits are a series of digital chips with a few gates each. The 7420 is a chip with two 4-input NAND gates.

Create a module with the same functionality as the 7420 chip. It has 8 inputs and 2 outputs.



Screen clipping taken: 09-02-2024 22:31

```
module top_module (
    input p1a, p1b, p1c, p1d,
    output p1y,
    input p2a, p2b, p2c, p2d,
    output p2y );
    assign p1y =~(p1a&p1b&p1c&p1d);
    assign p2y =~(p2a&p2b&p2c&p2d);
endmodule

```

Screen clipping taken: 09-02-2024 22:32

Truthtable1

7420 mt2015_eq2

In the previous exercises, we used simple logic gates and combinations of several logic gates. These circuits are examples of *combinational* circuits. Combinational means the outputs of the circuit is a function (in the mathematics sense) of only its inputs. This means that for any given input value, there is only one possible output value. Thus, one way to describe the behaviour of a combinational function is to explicitly list what the output should be for every possible value of the inputs. This is a truth table.

For a boolean function of N inputs, there are 2^N possible input combinations. Each row of the truth table lists one input combination, so there are always 2^N rows. The output column shows what the output should be for each input value.

Screen clipping taken: 09-02-2024 22:37

Row	*Inputs*			*Outputs*
number	**x3**	**x2**	**x1**	**f**
0	0	0	0	0
1	0	0	1	0
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

The above truth table is for a three-input, one-output function. It has 8 rows for each of the 8 possible input combinations, and one output column. There are four inputs combinations where the output is 1, and four where the output is 0.

```verilog
module top_module(
    input x3,
    input x2,
    input x1,   // three inputs
    output f    // one output
);
    wire t1,t2,t3,t4;
    and(t1,~x3,x2,x1);
    and(t2,~x3,x2,~x1);
    and(t3,x3,~x2,x1);
    and(t4,x3,x2,x1);
    or(f,t1,t2,t3,t4);
endmodule

```

Also f= $\overline{x_3}\ .x_2 + x_3x_1$

Why generate block is used in verilog?

Mt2015 eq2

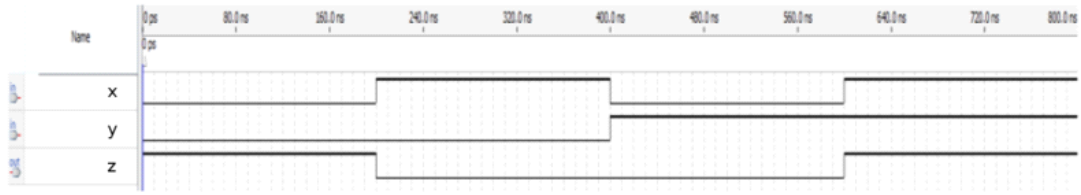
truthtable1 ✓ mt2015_q4a

Taken from 2015 midterm question 1k

Create a circuit that has two 2-bit inputs A[1:0] and B[1:0], and produces an output z. The value of z should be 1 if A = B, otherwise z should be 0.

```verilog
module top_module (input [1:0] A,
                   input [1:0] B,
                   output z );
wire t1,t2,t3,t4;
    and(t1,~A[0],~A[1],~B[0],~B[1]);
    and(t2,~A[0],A[1],~B[0],B[1]);
    and(t3,A[0],~A[1],B[0],~B[1]);
    and(t4,A[0],A[1],B[0],B[1]);
    or(z,t1,t2,t3,t4);
endmodule

```

Screen clipping taken: 09-02-2024 23:19




```
module top_module ( input x, input y, output z );
    assign z=~(x^y);
endmodule
```

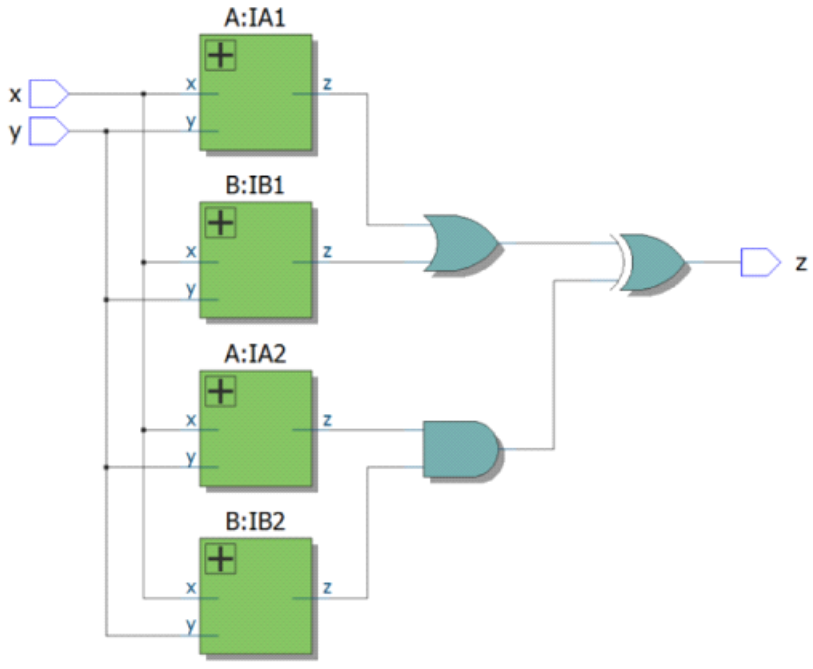
Screen clipping taken: 09-02-2024 23:25

Mt2015 q4

mt2015_q4b ringer

Taken from 2015 midterm question 4

See mt2015_q4a and mt2015_q4b for the submodules used here. The top-level design consists of two instantiations each of subcircuits A and B, as shown below.



Implement this circuit.

Screen clipping taken: 09-02-2024 23:30

```
module top_module (input x, input y, output z);
wire t1,t2,t3,t4,t5,t6;
    a a1(.x(x),.y(y),.z(t1));
    b b1(.x(x),.y(y),.z(t2));
    a a2(.x(x),.y(y),.z(t3));
    b b2(.x(x),.y(y),.z(t4));
    assign t5= t1|t2;
    assign t6= t3&t4;
    assign z=t5^t6;
endmodule

module a (input x, input y, output z);
    assign z=(x^y)&x;
endmodule

module b ( input x, input y, output z );
    assign z=~(x^y);
endmodule
```

Ringer

← mt2015_q4 thermostat →

Suppose you are designing a circuit to control a cellphone's ringer and vibration motor. Whenever the phone needs to ring from an incoming call (`input` **`ring`**), your circuit must either turn on the ringer (`output` **`ringer`** = 1) or the motor (`output` **`motor`** = 1), but not both. If the phone is in vibrate mode (`input` **`vibrate_mode`** = 1), turn on the motor. Otherwise, turn on the ringer.

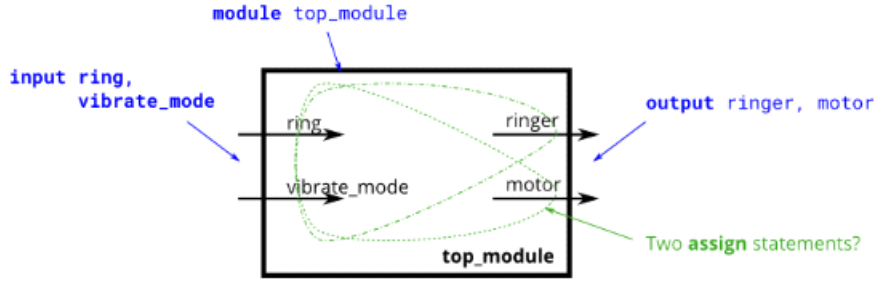
Try to use only `assign` statements, to see whether you can translate a problem description into a collection of logic gates.

Screen clipping taken: 12-02-2024 16:48

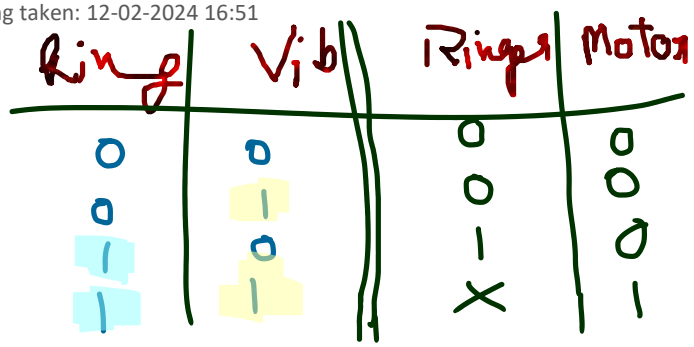
Design hint: When designing circuits, one often has to think of the problem "backwards", starting from the outputs then working backwards towards the inputs. This is often the opposite of how one would think about a (sequential, imperative) programming problem, where one would look at the inputs first then decide on an action (or output). For sequential programs, one would often think "If (inputs are ___) then (output should be ___)". On the other hand, hardware designers often think "The (output should be ___) when (inputs are ___)".

The above problem description is written in an imperative form suitable for software programming (*if ring then do this*), so you must convert it to a more declarative form suitable for hardware implementation (`assign ringer = ___`). Being able to think in, and translate between, both styles is one of the most important skills needed for hardware design.

Screen clipping taken: 12-02-2024 16:49



Screen clipping taken: 12-02-2024 16:51



```
module top_module (
    input ring,
    input vibrate_mode,
    output ringer,        // Make sound
    output motor          // Vibrate
);
assign ringer =ring && ~vibrate_mode ;
assign motor =ring && vibrate_mode ;
endmodule

```

Screen clipping taken: 12-02-2024 17:02

Thermostat

ringer popcount3

A heating/cooling thermostat controls both a heater (during winter) and an air conditioner (during summer). Implement a circuit that will turn on and off the heater, air conditioning, and blower fan as appropriate.

The thermostat can be in one of two modes: heating (`mode = 1`) and cooling (`mode = 0`). In heating mode, turn the heater on when it is too cold (`too_cold = 1`) but do not use the air conditioner. In cooling mode, turn the air conditioner on when it is too hot (`too_hot = 1`), but do not turn on the heater. When the heater or air conditioner are on, also turn on the fan to circulate the air. In addition, the user can also request the fan to turn on (`fan_on = 1`), even if the heater and air conditioner are off.

Try to use only `assign` statements, to see whether you can translate a problem description into a collection of logic gates.

Screen clipping taken: 12-02-

FAN ON	MODE	TOO_cold	TOO_hot	Heater	Aircon	Fan
0	0	0	0	0	0	0
0	0	0	1	0	1	1
0	0	1	0	0	0	0
0	0	1	1	x	x	x
0	1	0	0	0	0	0
0	1	0	1	0	0	0
0	1	1	0	1	0	1
0	1	1	1	x	x	x
1	0	0	0	0	0	1
1	0	0	1	0	1	1
1	0	1	0	0	0	1
1	0	1	1	x	x	1

1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	1	0	1
1	1	1	1	x	x	1

```
module top_module (
    input too_cold,
    input too_hot,
    input mode,
    input fan_on,
    output heater,
    output aircon,
    output fan
);
assign heater =mode &too_cold;
    assign aircon =too_hot&~(mode) ;
    assign fan =fan_on |(mode&too_cold)|(too_hot&~(mode));
endmodule

```

Screen clipping taken: 12-02-2024 19:43

Popcount3

thermostat gatesv

A "population count" circuit counts the number of '1's in an input vector. Build a population count circuit for a 3-bit input vector.

Module Declaration

```
module top_module(
    input [2:0] in,
    output [1:0] out );
```

Screen clipping taken: 12-02-2024 21:21

```
module top_module(
    input [2:0] in,
    output [1:0] out );
//genvar i;
    generate
        always @(*)
            begin
                out =2'b00;
                for(int i=0; i<3; i++)
                   begin
                   if(in[i] ==1'b1)
```

```
module top_module(
    input [2:0] in,
    output [1:0] out );
//genvar i;
    generate
        always @(*)
            begin
                out =2'b00;
                for(int i=0; i<3; i++)
                   begin
                   if(in[i] ==1'b1)
                        out=out+1;
                   else
                        out=out;
                   end
               end
    endgenerate
endmodule

```

Screen clipping taken: 12-02-2024 21:21

Gatesv

popcount3 gatesv100

You are given a four-bit input vector in[3:0]. We want to know some relationships between each bit and its neighbour:

- **out_both**: Each bit of this output vector should indicate whether *both* the corresponding input bit and its neighbour to the **left** (higher index) are '1'. For example, `out_both[2]` should indicate if `in[2]` and `in[3]` are both 1. Since `in[3]` has no neighbour to the left, the answer is obvious so we don't need to know `out_both[3]`.
- **out_any**: Each bit of this output vector should indicate whether *any* of the corresponding input bit and its neighbour to the **right** are '1'. For example, `out_any[2]` should indicate if either `in[2]` or `in[1]` are 1. Since `in[0]` has no neighbour to the right, the answer is obvious so we don't need to know `out_any[0]`.
- **out_different**: Each bit of this output vector should indicate whether the corresponding input bit is different from its neighbour to the **left**. For example, `out_different[2]` should indicate if `in[2]` is different from `in[3]`. For this part, treat the vector as wrapping

- **out_different**: Each bit of this output vector should indicate whether the corresponding input bit is different from its neighbour to the **left**. For example, `out_different[2]` should indicate if `in[2]` is different from `in[3]`. For this part, treat the vector as wrapping around, so `in[3]`'s neighbour to the left is `in[0]`.

EXOR


```
module top_module(
    input [3:0] in,
    output [2:0] out_both,
    output [3:1] out_any,
    output [3:0] out_different );
assign out_both ={in[3]&in[2],
                  in[2]&in[1],
                  in[1]&in[0]};
    assign out_any ={in[3]|in[2],
                     in[2]|in[1],
                     in[1]|in[0]};
assign out_different ={in[0]^in[3],
                       in[3]^in[2],
                       in[2]^in[1],
                       in[1]^in[0]};
endmodule

```

Gatesv100

gatesv mux2to1

See also the shorter version: Gates and vectors.

You are given a 100-bit input vector in[99:0]. We want to know some relationships between each bit and its neighbour:

- **out_both**: Each bit of this output vector should indicate whether *both* the corresponding input bit and its neighbour to the **left** are '1'. For example, `out_both[98]` should indicate if `in[98]` and `in[99]` are both 1. Since `in[99]` has no neighbour to the left, the answer is obvious so we don't need to know `out_both[99]`.
- **out_any**: Each bit of this output vector should indicate whether *any* of the corresponding input bit and its neighbour to the **right** are '1'. For example, `out_any[2]` should indicate if either `in[2]` or `in[1]` are 1. Since `in[0]` has no neighbour to the right, the answer is obvious so we don't need to know `out_any[0]`.

- **out_different**: Each bit of this output vector should indicate whether the corresponding input bit is different from its neighbour to the **left**. For example, `out_different[98]` should indicate if `in[98]` is different from `in[99]`. For this part, treat the vector as wrapping around, so `in[99]`'s neighbour to the left is `in[0]`.

Screen clipping taken: 12-02-2024 21:57

Last non-success: 12/2/2024, 10:08:36 pm Load

```
module top_module(
    input [99:0] in,
    output [98:0] out_both,
    output [99:1] out_any,
    output [99:0] out_different );

assign out_both=in[98:0]&in[99:1];
  assign out_any=in[99:1]|in[98:0];
    assign out_different[99]=in[0]^in[99];
    assign out_different[98:0]=in[99:1]^in[98:0];

endmodule

```

Screen clipping taken: 12-02-2024 22:28

Multiplexers

12 February 2024 22:28

Mux2to1

gatesv100 mux2to1v

Create a one-bit wide, 2-to-1 multiplexer. When sel=0, choose a. When sel=1, choose b.

Screen clipping taken: 12-02-2024 22:52

```
module top_module(
    input a, b, sel,
    output out );
    always @(*)
        begin
            if(sel==1'b0)
            out=a;
            else
            out=b;
        end
endmodule

```

Screen clipping taken: 12-02-2024 22:57

```
module top_module(
    input a, b, sel,
    output out );
assign out = sel?b:a;
endmodule

```

Screen clipping taken: 12-02-2024 22:58

Mux2to1v

mux2to1 mux9to1v

Create a 100-bit wide, 2-to-1 multiplexer. When sel=0, choose a. When sel=1, choose b.

Screen clipping taken: 12-02-2024 22:59

```
module top_module(
    input [99:0] a, b,
    input sel,
    output [99:0] out );
assign out =sel ? b:a;
endmodule

```

Screen clipping taken: 12-02-2024 22:59

Mux9to1v

mux2to1v mux256to1

Create a 16-bit wide, 9-to-1 multiplexer. sel=0 chooses a, sel=1 chooses b, etc. For the unused cases (sel=9 to 15), set all output bits to '1'.

Screen clipping taken: 12-02-2024 23:06

```
module top_module(
    input [15:0] a, b, c, d, e, f, g, h, i,
    input [3:0] sel,
    output [15:0] out );
    always @(*)
        begin
            case(sel)
             default: out=16'hffff;
                4'b0000: out=a;
                4'b0001: out=b;
                4'b0010: out=c;
                4'b0011: out=d;
                4'b0100: out=e;
                4'b0101: out=f;
                4'b0110: out=g;
                4'b0111: out=h;
                4'b1000: out=i;
            endcase
        end
endmodule

```

Screen clipping taken: 12-02-2024 23:13

Mux256to1

mux9to1v ✓ mux256to1v

Create a 1-bit wide, 256-to-1 multiplexer. The 256 inputs are all packed into a single 256-bit input vector. sel=0 should select `in[0]`, sel=1 selects bits `in[1]`, sel=2 selects bits `in[2]`, etc.

```
module top_module(
    input [255:0] in,
    input [7:0] sel,
    output out );
    assign out=in[sel];
endmodule

```

Mux256to1v

mux256to1 ✓ hadd

Create a 4-bit wide, 256-to-1 multiplexer. The 256 4-bit inputs are all packed into a single 1024-bit input vector. sel=0 should select bits `in[3:0]`, sel=1 selects bits `in[7:4]`, sel=2 selects bits `in[11:8]`, etc.

Screen clipping taken: 12-02-2024 23:19

```
module top_module(
    input [1023:0] in,
    input [7:0] sel,
    output [3:0] out );
    assign out=in[sel*4+3:sel*4];
endmodule

```

Hint...

- With this many options, a case statement isn't so useful.
- Vector indices can be variable, as long as the synthesizer can figure out that the width of the bits being selected is constant. It's not always good at this. An error saying "... is not a constant" means it couldn't prove that the select width is constant. In particular, `in[ sel*4+3 : sel*4 ]` does not work.
- Bit slicing ("Indexed vector part select", since Verilog-2001) has an even more compact syntax.

```
module top_module(
    input [1023:0] in,
    input [7:0] sel,
    output [3:0] out );
    assign out={in[sel*4+3],
                in[sel*4+2],
                in[sel*4+1],
                in[sel*4]};
endmodule

```

Screen clipping taken: 12-02-2024 23:31

a. Arithmetic Circuits

12 February 2024 23:31

Hadd

mux256to1v fadd

Create a half adder. A half adder adds two bits (with no carry-in) and produces a sum and carry-out.

Expected solution length: *Around 2 lines.*

Module Declaration

```
module top_module(
    input a, b,
    output cout, sum );
```

Write your solution here

[Load a previous submission] Load

```
1 module top_module(
2     input a, b,
3     output cout, sum );
4 assign cout=a&b;
5 assign sum=a^b;
6 endmodule
```

Screen clipping taken: 17-02-2024 16:42

Fadd

hadd adder3

Create a full adder. A full adder adds three bits (including carry-in) and produces a sum and carry-out.

Screen clipping taken: 17-02-2024 16:44

```
module top_module(
    input a, b, cin,
    output cout, sum );
assign sum=a^b^cin;
    assign cout=a&b | b&cin | a&cin;
endmodule

```

Screen clipping taken: 17-02-2024 16:44

Adder3

fadd exams/m2014_q4j

Now that you know how to build a full adder, make 3 instances of it to create a 3-bit binary ripple-carry adder. The adder adds two 3-bit numbers and a carry-in to produce a 3-bit sum and carry out. To encourage you to actually instantiate full adders, also output the carry-out from *each* full adder in the ripple-carry adder. cout[2] is the final carry-out from the last full adder, and is the carry-out you usually see.

Screen clipping taken: 17-02-2024 16:44

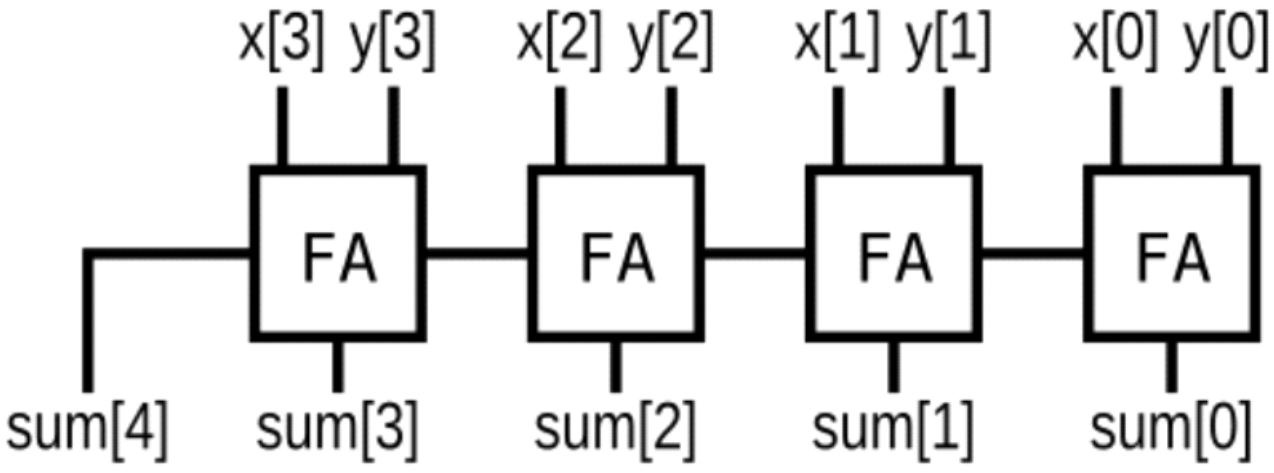
```
module top_module(
    input [2:0] a, b,
    input cin,
    output [2:0] cout,
    output [2:0] sum );
    FA fa1(.a(a[0]),.b(b[0]),.cin(cin),
           .sum(sum[0]),.cout(cout[0]));
    FA fa2(.a(a[1]),.b(b[1]),.cin(cout[0]),
           .sum(sum[1]),.cout(cout[1]));
    FA fa3(.a(a[2]),.b(b[2]),.cin(cout[1]),
           .sum(sum[2]),.cout(cout[2]));
endmodule

module FA(input a,b,cin,
          output sum,cout);
    assign sum=a^b^cin;
    assign cout= a&b | b&cin | a&cin;
endmodule
```

Screen clipping taken: 17-02-2024 17:07

Exams/m2014 q4j

adder3 exams/ece241_2014_q1c

Implement the following circuit:



("FA" is a full adder)

Screen clipping taken: 17-02-2024 17:08

```
module top_module (
    input [3:0] x,
    input [3:0] y,
    output [4:0] sum);
    wire [2:0]cout;
    FA fa1(.a(x[0]),.b(y[0]),.cin(0)
           ,.sum(sum[0]),.cout(cout[0]));
    FA fa2(.a(x[1]),.b(y[1]),.cin(cout[0])
           ,.sum(sum[1]),.cout(cout[1]));
    FA fa3(.a(x[2]),.b(y[2]),.cin(cout[1])
           ,.sum(sum[2]),.cout(cout[2]));
    FA fa45 (.a(x[3]),.b(y[3]),.cin(cout[2])
             ,.sum(sum[3]),.cout(sum[4]));
endmodule

module FA(input a,b,cin,
          output sum,cout);
    assign sum=a^b^cin;
    assign cout= a&b | b&cin | a&cin;
endmodule

```

Screen clipping taken: 17-02-2024 17:15

b. Arithmetic Circuits

12 February 2024 23:31

Exams/ece241 2014 q1c

exams/m2014_q4j adder100

Assume that you have two 8-bit 2's complement numbers, a[7:0] and b[7:0]. These numbers are added to produce s[7:0]. Also compute whether a (signed) overflow has occurred.

Module Declaration

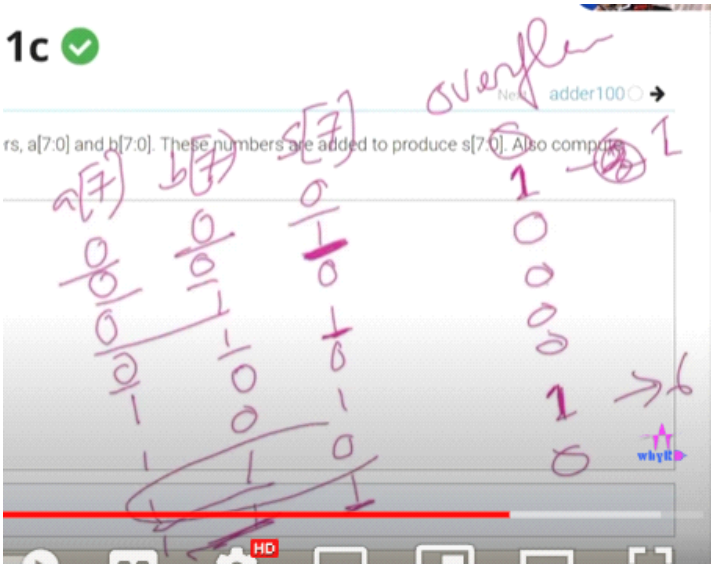
```
module top_module (
    input [7:0] a,
    input [7:0] b,
    output [7:0] s,
    output overflow
);
```

Screen clipping taken: 13-02-2024 00:12

Hint...

A *signed* overflow occurs when adding two positive numbers produces a negative result, or adding two negative numbers produces a positive result. There are several methods to detect overflow: It could be computed by comparing the signs of the input and output numbers, or derived from the carry-out of bit n and n-1.

Screen clipping taken: 13-02-2024 00:12



Screen clipping taken: 13-02-2024 00:43

```
module top_module (
    input [7:0] a,
    input [7:0] b,
    output [7:0] s,
    output overflow
);

assign s =a+b ;
    assign overflow =~a[7]&~b[7]&s[7]
        | a[7]&b[7]&~s[7] ;
// Check overflow from truth table
endmodule

```

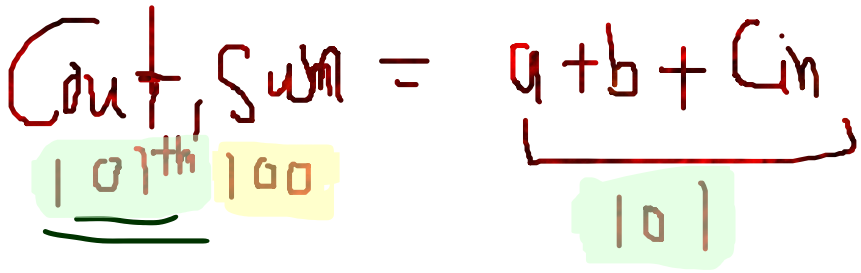
//2's complement for a binary number is found by inverting the numbers and adding 1
// i.e: 2'scomplement of 0011 is : 1100+0001 = 1101
//
//Also remember that MSB of the 2's complement number represents the sign. i.e if MSB is one then the number is -ve
// i.e: 1101 = -8+4+0+1=-3
//
//If 2= 0010 then -2= 1101+1 = 1110
//Also 2+(-2) = 1111 (This property holds for all the numbers)
//
//While adding two 2's complement numbers, overflow can be detected two ways:
// 1. If carryout and carry-on to MSB are different then overflow occurred
// 2. If both numbers are +ve and result is-ve or vise versa, overflow occurs

Adder100

exams/ece241_2014_q1c bcdadd4

Create a 100-bit binary adder. The adder adds two 100-bit numbers and a carry-in to produce a 100-bit sum and carry out.

Screen clipping taken: 17-02-2024 17:20




```
module top_module(
    input [99:0] a, b,
    input cin,
    output cout,
    output [99:0] sum);
    assign{cout,sum}=a+b+cin;
endmodule

```

Screen clipping taken: 17-02-2024 17:56

Bcdadd4

adder100 kmap1

You are provided with a BCD (binary-coded decimal) one-digit adder named bcd_fadd that adds two BCD digits and carry-in, and produces a sum and carry-out.

```
module bcd_fadd (
    input [3:0] a,
    input [3:0] b,
    input     cin,
    output   cout,
    output [3:0] sum );
```

Instantiate 4 copies of bcd_fadd to create a 4-digit BCD ripple-carry adder. Your adder should add two 4-digit BCD numbers (packed into 16-bit vectors) and a carry-in to produce a 4-digit sum and carry out.

Screen clipping taken: 17-02-2024 17:56

```
module top_module (
    input [15:0] a, b,
    input cin,
    output cout,
    output [15:0] sum );
    wire co[2:0];
    bcd_fadd bcd1(a[3:0],b[3:0],cin,co[0],sum[3:0]);
    bcd_fadd bcd2(a[7:4],b[7:4],co[0],co[1],sum[7:4]);
    bcd_fadd bcd3(a[11:8],b[11:8],co[1],co[2],sum[11:8]);
    bcd_fadd bcd4(a[15:12],b[15:12],co[2],(cout),sum[15:12])
// Use by name instantiation beacuse you don't
//    know the name of ports of bcd_fadd
endmodule

/*module bcd_fadd(input a,b,cin,
                output s,co);
    assign s=a^b^cin;
    assign co = a&b | b&cin | a&cin;
endmodule*/
```

Screen clipping taken: 17-02-2024 18:25

BUT

```
module top_module (
    input [15:0] a, b,
    input cin,
    output cout,
    output [15:0] sum );
    wire co[2:0];
    bcd_fad bcd1(a[3:0],b[3:0],cin,co[0],sum[3:0]);
    bcd_fad bcd2(a[7:4],b[7:4],co[0],co[1],sum[7:4]);
    bcd_fad bcd3(a[11:8],b[11:8],co[1],co[2],sum[11:8]);
    bcd_fad bcd4(a[15:12],b[15:12],co[2],(cout),sum[15:12]);
// Use by name instantiation beacuse you don't
//    know the name of ports of bcd_fadd
endmodule

module bcd_fad(input a,
               input b,
               input cin,
               output co,
               output s);
    assign s=a^b^cin;
    assign co = a&b | b&cin | a&cin;
endmodule
```

Screen clipping taken: 17-02-2024 18:35

It is not Working.

Status: Incorrect

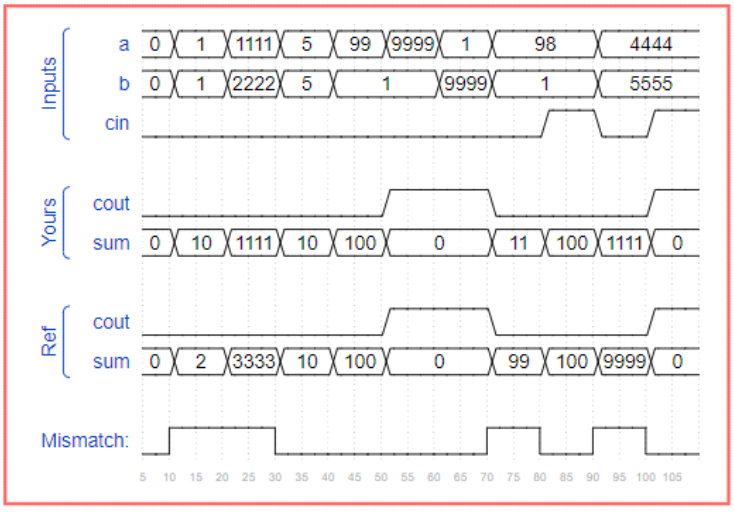
Compile and simulation succeeded, but the circuit's output wasn't entirely correct. The hints below may help.

```
# Hint: Output 'cout' has 87 mismatches. First mismatch occurred at time 115.
# Hint: Output 'sum' has 215 mismatches. First mismatch occurred at time 10.
# Hint: Total mismatched samples is 215 out of 233 samples
```

Timing diagrams for selected test cases

These are timing diagrams from some of the test cases we used. They may help you debug your circuit. The diagrams show inputs to the circuit, outputs from your circuit, and the expected reference outputs. The "Mismatch" trace shows which cycles your outputs don't match the reference outputs (0 = correct, 1 = incorrect).

BCD addition



Screen clipping taken: 17-02-2024 18:35

Because I think the module
they have given works differently.

Module cseladd

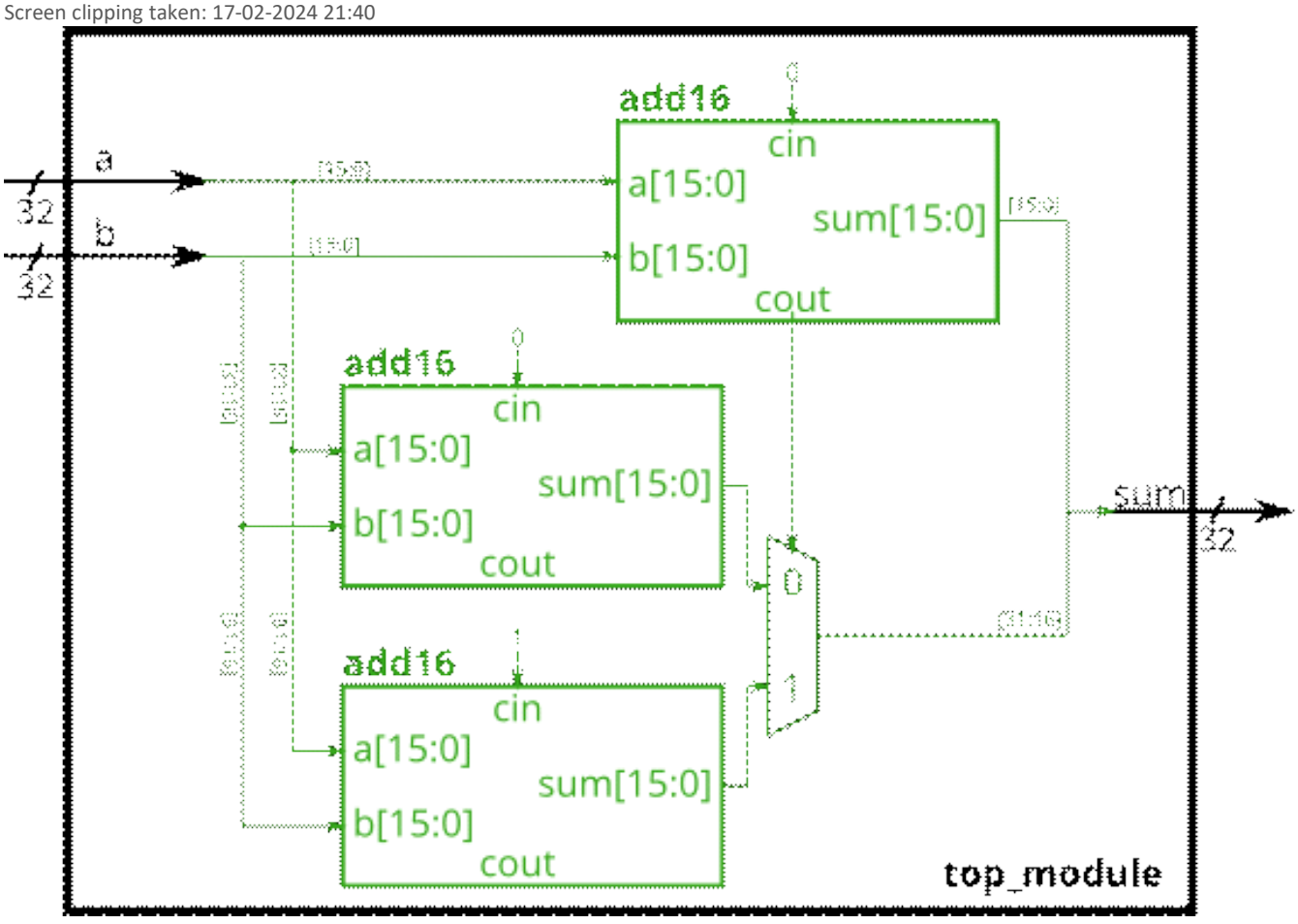
module_fadd module_addsub

One drawback of the ripple carry adder (See previous exercise) is that the delay for an adder to compute the carry out (from the carry-in, in the worst case) is fairly slow, and the second-stage adder cannot begin computing *its* carry-out until the first-stage adder has finished. This makes the adder slow. One improvement is a carry-select adder, shown below. The first-stage adder is the same as before, but we duplicate the second-stage adder, one assuming carry-in=0 and one assuming carry-in=1, then using a fast 2-to-1 multiplexer to select which result happened to be correct.

In this exercise, you are provided with the same module add16 as the previous exercise, which adds two 16-bit numbers with carry-in and produces a carry-out and 16-bit sum. You must instantiate *three* of these to build the carry-select adder, using your own 16-bit 2-to-1 multiplexer.

Connect the modules together as shown in the diagram below. The provided module add16 has the following declaration:

Screen clipping taken: 17-02-2024 21:40



```
module top_module(
    input [31:0] a,
    input [31:0] b,
    output [31:0] sum
);
    wire z;
    wire [15:0]sum1,sum2;
    add16 a1(.a(a[15:0]),.b(b[15:0]),.sum(sum[15:0]),.cout(z));
    add16 a2(.a(a[31:16]),.b(b[31:16]),.sum(sum1),.cin(1'b0));
    add16 a3(.a(a[31:16]),.b(b[31:16]),.sum(sum2),.cin(1'b1));
always  @(*)
 begin
     case (z)
         1'b0:sum[31:16]=sum1;
         1'b1:sum[31:16]=sum2;
     endcase
 end
endmodule
```

Screen clipping taken: 17-02-2024 21:40

Module addsub

← module_cseladd ✓ alwaysblock1 ✓ →

An adder-subtractor can be built from an adder by optionally negating one of the inputs, which is equivalent to inverting the input then adding 1. The net result is a circuit that can do two operations: (a + b + 0) and (a + ~b + 1). See Wikipedia if you want a more detailed explanation of how this circuit works.

Build the adder-subtractor below.

You are provided with a 16-bit adder module, which you need to instantiate twice:

```
module add16 ( input[15:0] a, input[15:0] b, input cin,
output[15:0] sum, output cout );
```

Use a 32-bit wide XOR gate to invert the b input whenever sub is 1. (This can also be viewed as b[31:0] XORed with sub replicated 32 times. See replication operator ✓.). Also connect the sub input to the carry-in of the adder.

Screen clipping taken: 17-02-2024 21:42

The concatenation operator allowed concatenating together vectors to form a larger vector. But sometimes you want the same thing concatenated together many times, ar it is still tedious to do something like `assign a = {b,b,b,b,b,b};`. The replicatio operator allows repeating a vector and concatenating them together:

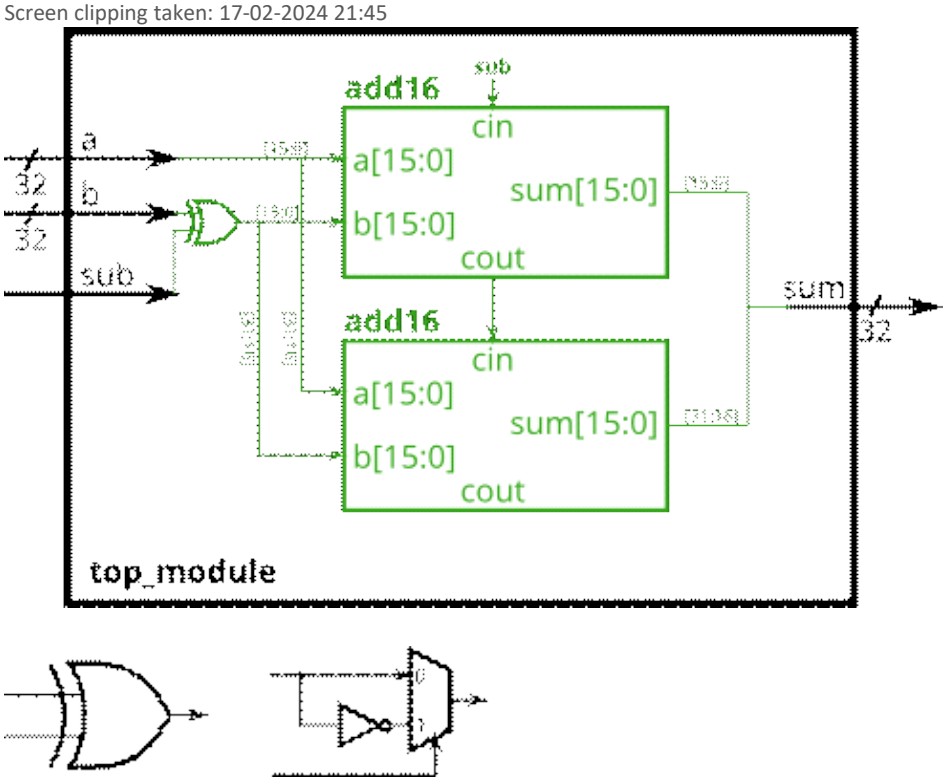
```
{num{vector}}
```

This replicates *vector* by *num* times. *num* must be a constant. Both sets of braces are required.

Examples:

```
{5{1'b1}}               // 5'b11111 (or 5'd31 or 5'h1f)
{2{a,b,c}}              // The same as {a,b,c,a,b,c}
{3'd5, {2{3'd6}}}       // 9'b101_110_110. It's a concatenation of 101
with
                        // the second vector, which is two copies of
3'b110.
```

Screen clipping taken: 17-02-2024 21:45




```
module top_module(
    input [31:0] a,
    input [31:0] b,
    input sub,
    output [31:0] sum
);
    wire [31:0]b1;
    wire cout;
```

```
module top_module(
    input [31:0] a,
    input [31:0] b,
    input sub,
    output [31:0] sum
);
    wire [31:0]b1;
    wire cout;
always @(*) begin
    case (sub)
            1'b0:begin
                b1 <=b[31:0];
            end
            1'b1:begin
                b1 <=~b[31:0];
            end
        endcase
    end
    add16 a1(.a(a[15:0]),.b(b1[15:0]),.cin(sub),
             .cout(cout),.sum(sum[15:0]));
    add16 a2(.a(a[31:16]),.b(b1[31:16]),.cin(cout),
             .sum(sum[31:16]));
endmodule

```

Screen clipping taken: 17-02-2024 22:34

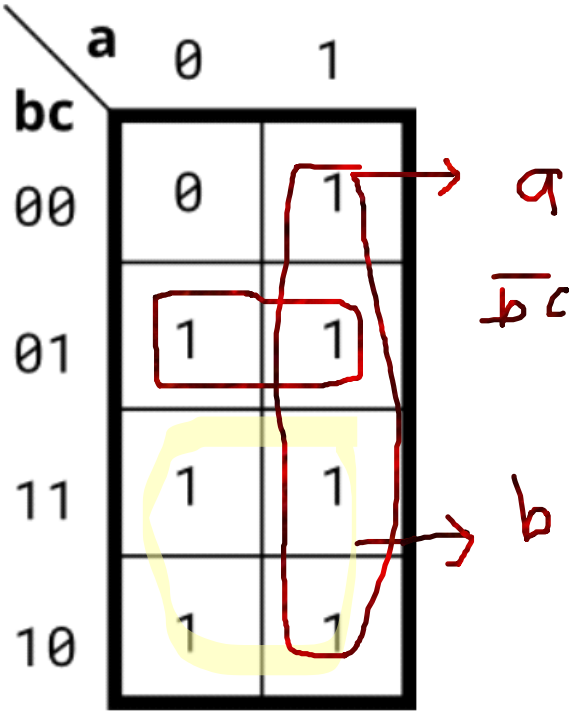
K-Map

12 February 2024 23:32

Kmap1

bcdadd4 kmap2

Implement the circuit described by the Karnaugh map below.



Screen clipping taken: 12-02-2024 23:32

Try to simplify the k-map before coding it. Try both product-of-sums and sum-of-products forms. We can't check whether you have the optimal simplification of the k-map. But we can check if your reduction is equivalent, and we can check whether you can translate a k-map into a circuit.

Screen clipping taken: 12-02-2024 23:32

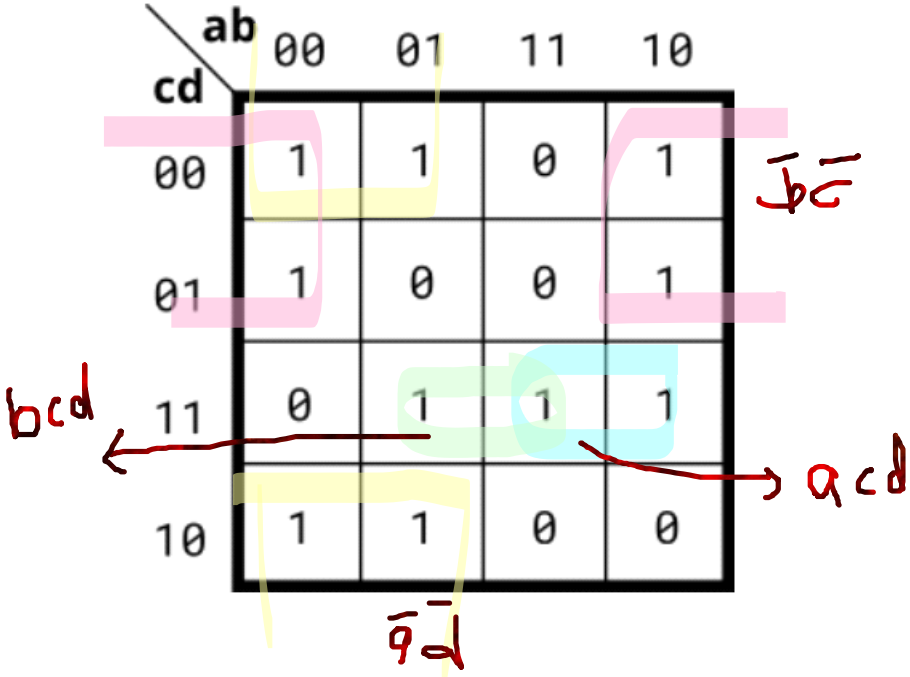
```
module top_module(
    input a,
    input b,
    input c,
    output out  );
    assign out = a|b|(c&~(b));
endmodule

```

Screen clipping taken: 12-02-2024 23:34

Kmap2

kmap1 ✔ kmap3 ○

Implement the circuit described by the Karnaugh map below.



Screen clipping taken: 12-02-2024 23:34

Previous Ques:

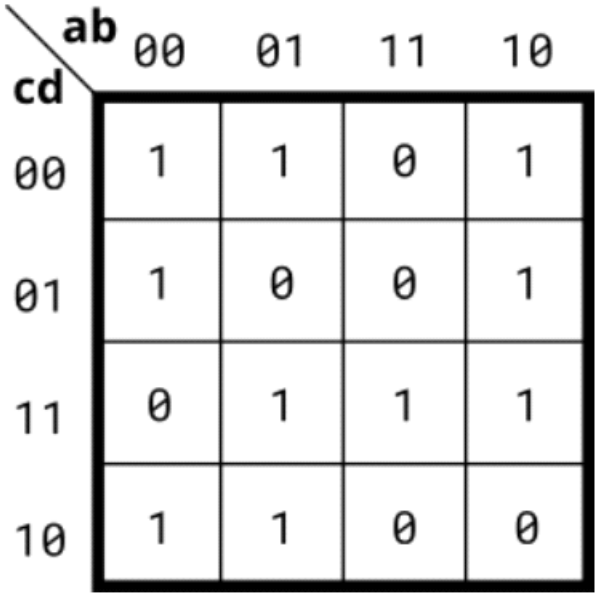
```
module top_module(
    input a,
    input b,
    input c,
    output out  );
assign out=a | b | c ;
endmodule

```

Screen clipping taken: 12-02-2024 23:40

Kmap2

← kmap1 ✓ kmap3 ○ →

Implement the circuit described by the Karnaugh map below.



cd \ ab	00	01	11	10
00	1	1	0	1
01	1	0	0	1
11	0	1	1	1
10	1	1	0	0

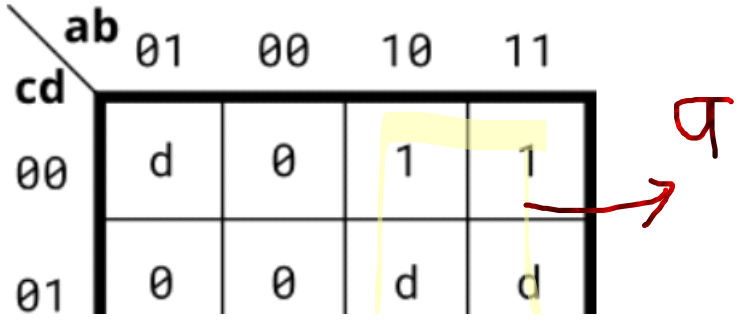
```
module top_module(
    input a,
    input b,
    input c,
    input d,
    output out  );
    assign out=(a&d&c)|(b&c&d)|(~b&~c)|(~a&~d);
endmodule

```

Screen clipping taken: 12-02-2024 23:48

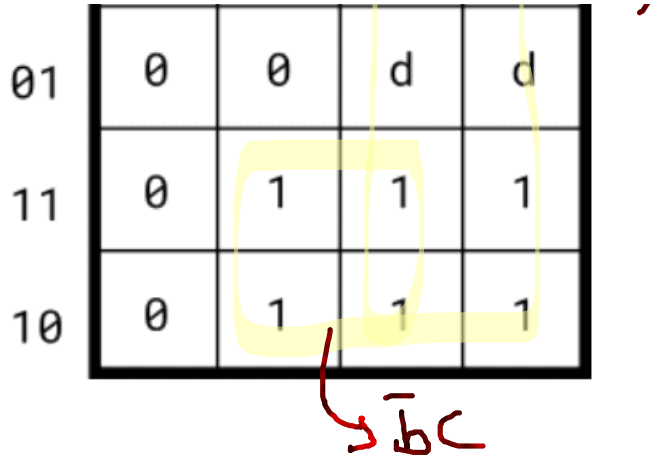
Kmap3

← kmap2 ✓ kmap4 ○ →

Implement the circuit described by the Karnaugh map below.



cd \ ab	01	00	10	11
00	d	0	1	1
01	0	0	d	d



Screen clipping taken: 12-02-2024 23:48

```
module top_module(
    input a,
    input b,
    input c,
    input d,
    output out  );
    assign out=a|(~b&c);
endmodule

```

Kmap4

kmap3 exams/ece241_2013_q2

Implement the circuit described by the Karnaugh map below.

cd \ ab	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

Screen clipping taken: 16-02-2024 10:24

```
module top_module(
    input a,
    input b,
    input c,
    input d,
    output out  );
assign out = (~a&b&~c&~d) | (a&~b&~c&~d)
    | (~a&~b&~c&d) |  (a&b&~c&d)
    | (~a&b&c&d)   |  (a&~b&c&d)
    | (~a&~b&c&~d) |  (a&b&c&~d);
endmodule

```

Screen clipping taken: 16-02-2024 10:25

▼ **Karnaugh Map to Circuit**

- ✓ 3-variable
- ✓ 4-variable
- ✓ 4-variable
- ✓ 4-variable
- ○ **Minimum SOP and POS**

Exams/ece241 2013 q2 ○

← kmap4 ✓ exams/m2014_q3 ○ →

A single-output digital system with four inputs (a,b,c,d) generates a logic-1 when 2, 7, or 15 appears on the inputs, and a logic-0 when 0, 1, 4, 5, 6, 9, 10, 13, or 14 appears. The input conditions for the numbers 3, 8, 11, and 12 never occur in this system. For example, 7 corresponds to a,b,c,d being set to 0,1,1,1, respectively.

Determine the output out_sop in minimum SOP form, and the output out_pos in minimum POS form.

```
module top_module (
    input a,
    input b,
    input c,
    input d,
    output out_sop,
    output out_pos
);
    assign out_sop=c&d | (~a&~b&c);
    assign out_pos=(~a|d)&c&(~b|d);
endmodule

```

Screen clipping taken: 16-02-2024 11:01

Exams/m2014 q3

exams/ece241_2013_q2 exams/2012_q1g

Consider the function *f* shown in the Karnaugh map below.

x_3x_4 \ x_1x_2	00	01	11	10
00	d	0	d	d
01	0	d	1	0
11	1	1	d	d
10	1	1	0	d

Implement this function. **d** is don't-care, which means you may choose to output whatever value is convenient.

Screen clipping taken: 16-02-2024 11:03

```
module top_module (
    input [4:1] x,
    output f );
    assign f=(~x[1]&x[3]) | (~x[3]&x[2]&x[4]);
endmodule

```

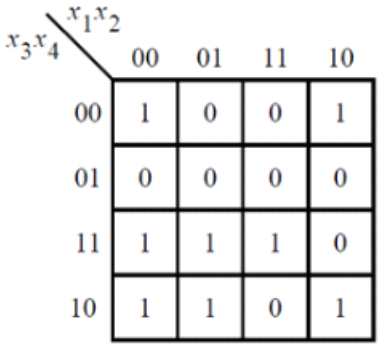
Screen clipping taken: 16-02-2024 11:06

Exams/2012 q1g

exams/m2014_q3 exams/ece241_2014_q3

Consider the function *f* shown in the Karnaugh map below. Implement this function.

(The original exam question asked for simplified SOP and POS forms of the function.)



x_3x_4 \ x_1x_2	00	01	11	10
00	1	0	0	1
01	0	0	0	0
11	1	1	1	0
10	1	1	0	1

```
module top_module (
    input [4:1] x,
    output f
);
    assign f=(~x[1]&x[3])
        | (~x[2]&~x[4])
        | (x[2]&x[3]&x[4]);
endmodule

```

Screen clipping taken: 16-02-2024 11:13

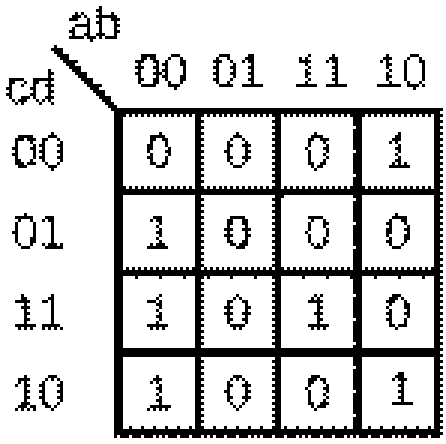
Exams/ece241 2014 q3

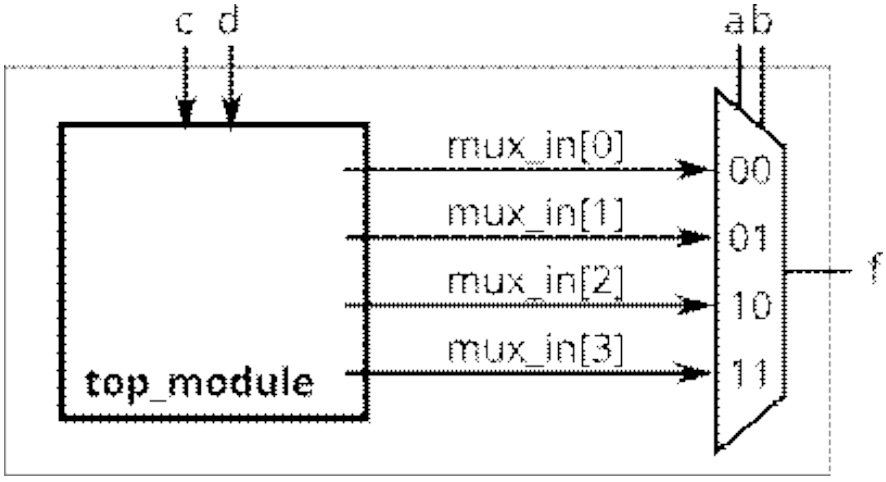
exams/2012_q1g dff

For the following Karnaugh map, give the circuit implementation using one 4-to-1 multiplexer and as many 2-to-1 multiplexers as required, but using as few as possible. You are not allowed to use any other logic gate and you must use *a* and *b* as the multiplexer selector inputs, as shown on the 4-to-1 multiplexer below.

You are implementing just the portion labelled **top_module**, such that the entire circuit (including the 4-to-1 mux) implements the K-map.

Screen clipping taken: 16-02-2024 11:15





(The requirement to use only 2-to-1 multiplexers exists because the original exam question also wanted to test logic function simplification using K-maps and how to synthesize logic functions using only multiplexers. If you wish to treat this as purely a Verilog exercise, you may ignore this constraint and write the module any way you wish.)

Screen clipping taken: 16-02-2024 11:16

```
module top_module (
    input c,
    input d,
    output [3:0] mux_in
);
    assign mux_in[3]=c&d;
    assign mux_in[2]=~d;
    assign mux_in[1]=1'b0;
    assign mux_in[0]=c|d;
endmodule

```

Screen clipping taken: 16-02-2024 11:29

```
module top_module (
    input c,
    input d,
    output [3:0] mux_in
);
    //assign mux_in[3]=c&d;
    mux2x1 mod_1(c,1,d,mux_in[0]);
    assign mux_in[2]=~d;
    assign mux_in[1]=1'b0;
    assign mux_in[0]=c|d;
endmodule

module mux2x1(
    input x,
    input y,
    input sel,
    output z );
    assign z =sel? y:x;
endmodule
```

Screen clipping taken: 16-02-2024 16:57

```
Warning (10034): Output port "mux_in[3]" at top_module.v(5) has no driver File:
/home/h/work/hdlbits.14773587/top_module.v Line: 5
Info (12128): Elaborating entity "mux2x1" for hierarchy "mux2x1:mod_1" File:
/home/h/work/hdlbits.14773587/top_module.v Line: 7
Error (12014): Net "mux_in[0]", which fans out to "mux_in[0]", cannot be
assigned more than one value File: /home/h/work/hdlbits.14773587/top_module.v
Line: 5
    Error (12015): Net is fed by "mux2x1:mod_1|z" File:
/home/h/work/hdlbits.14773587/top_module.v Line: 17
    Error (12015): Net is fed by "mux_in" File:
/home/h/work/hdlbits.14773587/top_module.v Line: 5
Warning (12241): 1 hierarchies have connectivity warnings - see the
Connectivity Checks report folder
Error: Quartus Prime Analysis & Synthesis was unsuccessful. 3 errors, 3
warnings
    Error: Peak virtual memory: 386 megabytes
    Error: Processing ended: Fri Feb 16 11:27:13 2024
    Error: Elapsed time: 00:00:00
    Error: Total CPU time (on all processors): 00:00:00
Error (23031): Evaluation of Tcl script /home/h/hdlbits/compile.tcl
unsuccessful
Error: Quartus Prime Shell was unsuccessful. 9 errors, 3 warnings
    Error: Peak virtual memory: 481 megabytes
    Error: Processing ended: Fri Feb 16 11:27:13 2024
    Error: Elapsed time: 00:00:01
    Error: Total CPU time (on all processors): 00:00:01
```

Screen clipping taken: 16-02-2024 16:57

Latches and Flip-flop

08 February 2024 22:20

Dff

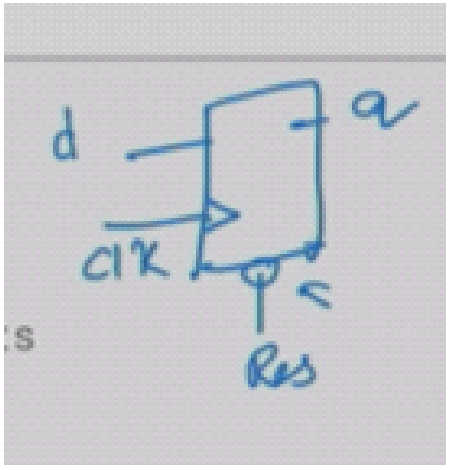
exams/ece241_2014_q3 dff8

A D flip-flop is a circuit that stores a bit and is updated periodically, at the (usually) positive edge of a clock signal.

D flip-flops are created by the logic synthesizer when a clocked always block is used (See alwaysblock2). A D flip-flop is the simplest form of "blob of combinational logic followed by a flip-flop" where the combinational logic portion is just a wire.

Create a single D flip-flop.

Screen clipping taken: 16-02-2024 16:58



Screen clipping taken: 16-02-2024 17:00

```
1  module top_module (
2      input clk,
3      input d,
4      output reg q );
5      always @(posedge clk)
6    begin
```

```
    always @(posedge clk)
  begin
// Use a clocked always block
    q <=d;
//   copy d to q at every positive edge of clk
//Clocked always blocks should use non-blocking assignments
end
endmodule

```

Screen clipping taken: 16-02-2024 17:06

Dff8

← dff ✓ dff8r ○ →

Create 8 D flip-flops. All DFFs should be triggered by the positive edge of `clk`.

Screen clipping taken: 16-02-2024 17:13

```
module top_module (
    input clk,
    input [7:0] d,
    output [7:0] q
);
    always @(posedge clk)
        begin
            q=d;
        end
endmodule

```

Screen clipping taken: 16-02-2024 17:13

Dff8r

dff8 dff8p

Create 8 D flip-flops with active high synchronous reset. All DFFs should be triggered by the positive edge of `clk`.

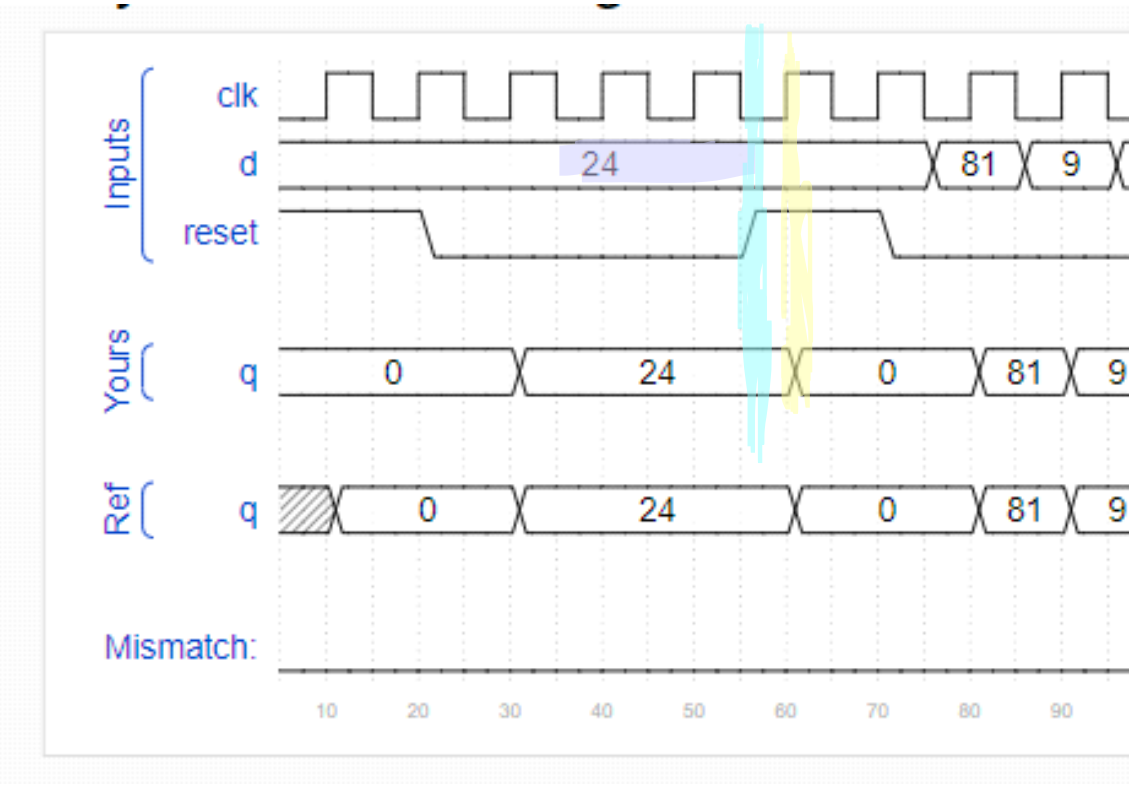
Module Declaration

```
module top_module (
    input clk,
    input reset,            // Synchronous reset
    input [7:0] d,
    output [7:0] q
);
```

Screen clipping taken: 16-02-2024 17:14

```
module top_module (
    input clk,
    input reset,            // Synchronous reset
    input [7:0] d,
    output [7:0] q
);
    always @(posedge clk)
        begin
            if (reset)
             q=0;
            else
                q=d;
        end
endmodule

```

Screen clipping taken: 16-02-2024 17:16



Screen clipping taken: 16-02-2024 17:19

Dff8p

dff8r dff8ar

Create 8 D flip-flops with active high synchronous reset. The flip-flops must be reset to 0x34 rather than zero. All DFFs should be triggered by the **negative** edge of `clk`.

Screen clipping taken: 16-02-2024 17:25

```
module top_module (
    input clk,
    input reset,
    input [7:0] d,
    output [7:0] q
);
    always @(negedge clk)
        begin
            if(reset)
              q=8'h0x34;
            else
             q=d;
        end
endmodule

```

Screen clipping taken: 16-02-2024 17:30

Dff8ar

dff8p dff16e

Create 8 D flip-flops with active high asynchronous reset. All DFFs should be triggered by the positive edge of `clk`.

Module Declaration

```
module top_module (
    input clk,
    input areset,   // active high asynchronous
reset
    input [7:0] d,
    output [7:0] q
);
```

Screen clipping taken: 16-02-2024 17:44

```
module top_module (
    input clk,
    input areset,   // active high asynchronous reset
    input [7:0] d,
    output [7:0] q
);
    always@(posedge clk or posedge areset)
        begin
            if (areset)
                q=0;
            else
                q=d;
        end
endmodule

```

Screen clipping taken: 16-02-2024 17:45

Dff16e

dff8ar exams/m2014_q4a

Create 16 D flip-flops. It's sometimes useful to only modify parts of a group of flip-flops. The byte-enable inputs control whether each byte of the 16 registers should be written to on that cycle. `byteena[1]` controls the upper byte `d[15:8]`, while `byteena[0]` controls the lower byte `d[7:0]`.

`resetn` is a synchronous, active-low reset.

All DFFs should be triggered by the positive edge of `clk`.

Screen clipping taken: 16-02-2024 17:49

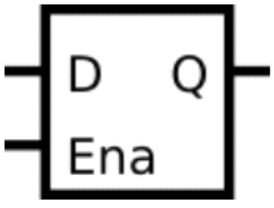
```
module top_module (
    input clk,
    input resetn,
    input [1:0] byteena,
    input [15:0] d,
    output [15:0] q
);
    always @(posedge clk)
        begin
            if(resetn==1'b0)
                q<=1'b0;
            else if (byteena==2'b00)
                q <=q;
            else if (byteena==2'b01)
                q[7:0]<=d[7:0];
            else if (byteena==2'b10)
                q[15:8]<=d[15:8];
            else if(byteena==2'b11)
                q<=d;
        end
endmodule

```

Screen clipping taken: 16-02-2024 18:02

Exams/m2014 q4a

dff16e exams/m2014_q4b

Implement the following circuit:



Note that this is a latch, so a Quartus warning about having inferred a latch is expected.

Hint...

- Latches are level-sensitive (not edge-sensitive) circuits, so in an always block, they use level-sensitive sensitivity lists.
- However, they are still sequential elements, so should use non-blocking assignments.
- A D-latch acts like a wire (or non-inverting buffer) when enabled, and preserves the current value when disabled.

Screen clipping taken: 16-02-2024 18:06

```
module top_module (
    input d,
    input ena,
    output q);
    always @(*)
        begin
            if(ena)
                q<=d;
        end
endmodule

```

Screen clipping taken: 16-02-2024 18:09

Status: Success!

You have solved 79 problems. See my progress...

Warning messages that may be important

Quartus messages (Show all)

Info (10041): Inferred latch for "q" at top_module.v(7) File: /home/h/work/hdlbits.14774149/top_module.v Line: 7

Unless you intentionally wanted to create a latch, this warning usually indicates a bug in a combinational always block. Make sure every variable is assigned a value in all cases so the previous value does not need to be remembered. Possible ways to achieve this include assigning a default value to variables at the top of the always block, using a default case, or having an else clause.

Warning (10240): Verilog HDL Always Construct warning at top_module.v(7): inferring latch(es) for variable "q", which holds its previous value in one or more paths through the always construct File: /home/h/work/hdlbits.14774149/top_module.v Line: 7

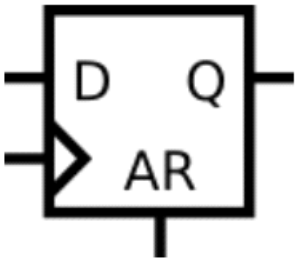
Unless you intentionally wanted to create a latch, this warning usually indicates a bug in a combinational always block. Make sure every variable is assigned a value in all cases so the previous value does not need to be remembered. Possible ways to achieve this include assigning a default value to variables at the top of the always block, using a default case, or having an else clause.

Screen clipping taken: 16-02-2024 18:09

Exams/m2014 q4b

← exams/m2014_q4a

Implement the following circuit:



Screen clipping taken: 16-02-2024 18:09

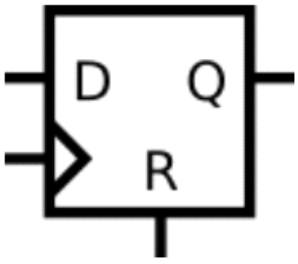
```
module top_module (
    input clk,
    input d,
    input ar,   // asynchronous reset
    output q);
    always @(posedge clk or posedge ar)
        begin
            if(ar==1'b1)
                q<=1'b0;
            else
                q<=d;
        end
endmodule

```

Screen clipping taken: 16-02-2024 18:13

Exams/m2014 q4c

exams/m2014_q4b

Implement the following circuit:



Screen clipping taken: 16-02-2024 18:14

```
module top_module (
    input clk,
    input d,
    input r,   // synchronous reset
    output q);
    always @(posedge clk)
        begin
            if(r)
                q<=0;
            else
                q<=d;
        end
endmodule

```

Screen clipping taken: 16-02-2024 18:14

```
module top_module (
    input clk,
    input in
    output out);
    always @(posedge clk)
        begin
        out <= out^in;
        end
endmodule

```

Screen clipping taken: 16-02-2024 18:22

```
Error (10170): Verilog HDL syntax error at top_module.v(4)
near text: "output";  expecting ")". Check for and fix any
syntax errors that appear immediately before or at the
specified keyword. The Intel FPGA Knowledge Database contains
many articles with specific details on how to resolve this
error. Visit the Knowledge Database at
https://www.altera.com/support/support-resources/knowledge-
base/search.html and search for this specific error message
number. File: /home/h/work/hdlbits.14774228/top_module.v
Line: 4
Error (10112): Ignored design unit "top_module" at
top_module.v(1) due to previous errors File:
/home/h/work/hdlbits.14774228/top_module.v Line: 1
```

Screen clipping taken: 16-02-2024 18:22

```
module top_module (
    input clk,
    input in,
    output out);

    always @ (posedge clk) begin
        out <= out ^ in;
        end
endmodule

```

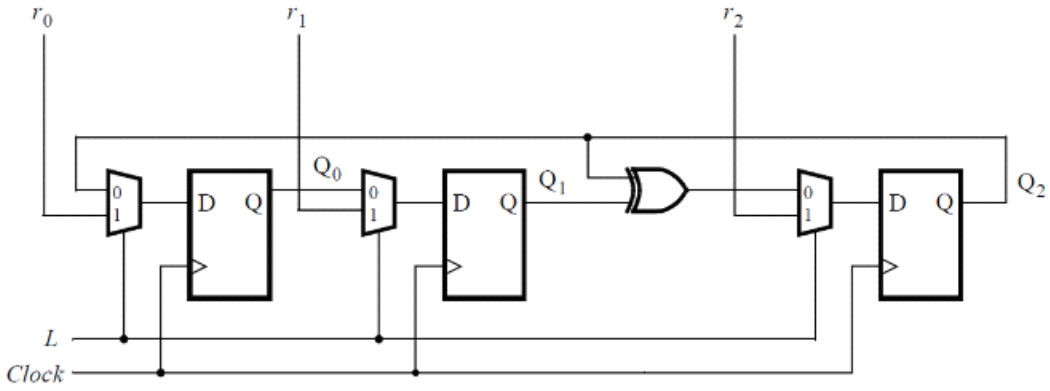
Screen clipping taken: 16-02-2024 18:26

Mt2015 muxdff

exams/m2014_q4d exams/2014_q4a

Taken from ECE253 2015 midterm question 5

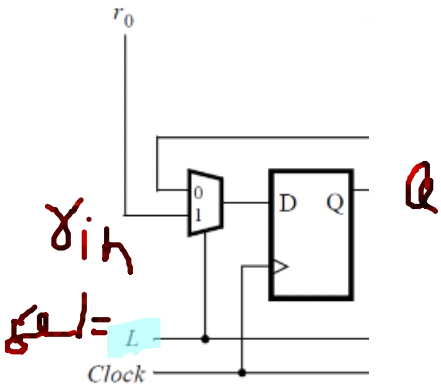
Consider the sequential circuit below:



Screen clipping taken: 16-02-2024 18:27

Assume that you want to implement hierarchical Verilog code for this circuit, using three instantiations of a submodule that has a flip-flop and multiplexer in it. Write a Verilog module (containing one flip-flop and multiplexer) named `top_module` for this submodule.

Screen clipping taken: 16-02-2024 18:27



Screen clipping taken: 16-02-2024 19:06

Module Declaration

```
module top_module (
        input clk,
        input L,
        input r_in,
        input q_in,
        output reg Q);
```

Screen clipping taken: 16-02-2024 19:07

```
module top_module (
    input clk,
    input L,
    input r_in,
    input q_in,
    output reg Q);
    always @(posedge clk)
        begin
            if(L==1'b0)
                Q<=q_in;
                else
                Q<=r_in;
        end
endmodule

```

Screen clipping taken: 16-02-2024 19:13

Status: Success!

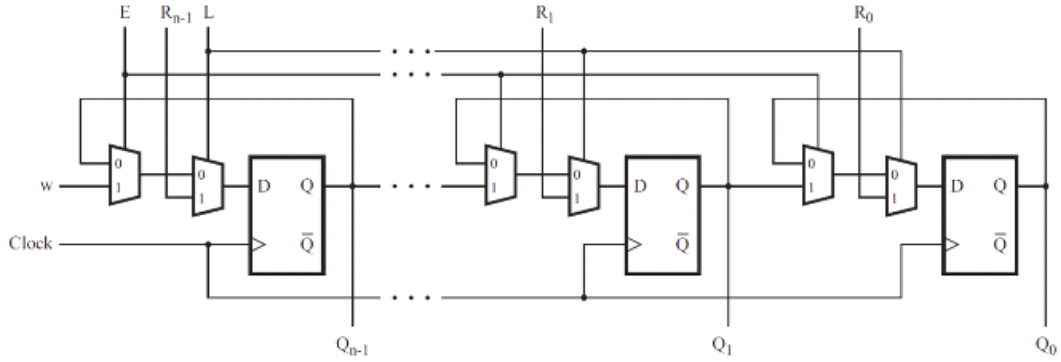
You have solved 91 problems. See my progress...

Screen clipping taken: 17-02-2024 22:35

Exams/2014 q4a

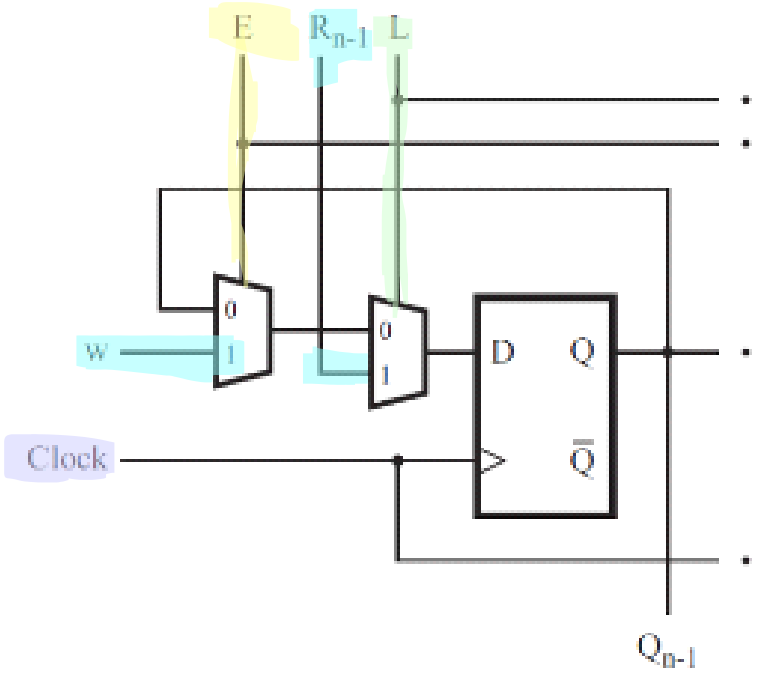
mt2015_muxdff exams/ece241_2014_q4

Consider the *n*-bit shift register circuit shown below:



Write a Verilog module named top_module for one stage of this circuit, including both the flip-flop and multiplexers.

Screen clipping taken: 17-02-2024 22:38



Screen clipping taken: 17-02-2024 22:38

```
module top_module (
    input clk,
    input w, R, E, L,
    output Q
);
    wire [1:0]t;
    assign t={E,L};
    always @(posedge clk)
        begin :ayush
            case(t)
            2'b00:  Q <= Q;
            2'b01:  Q <= R;
            2'b10:  Q <= w;
            2'b11:  Q <= R;
             endcase
        end :ayush
endmodule

```

Screen clipping taken: 17-02-2024 23:05

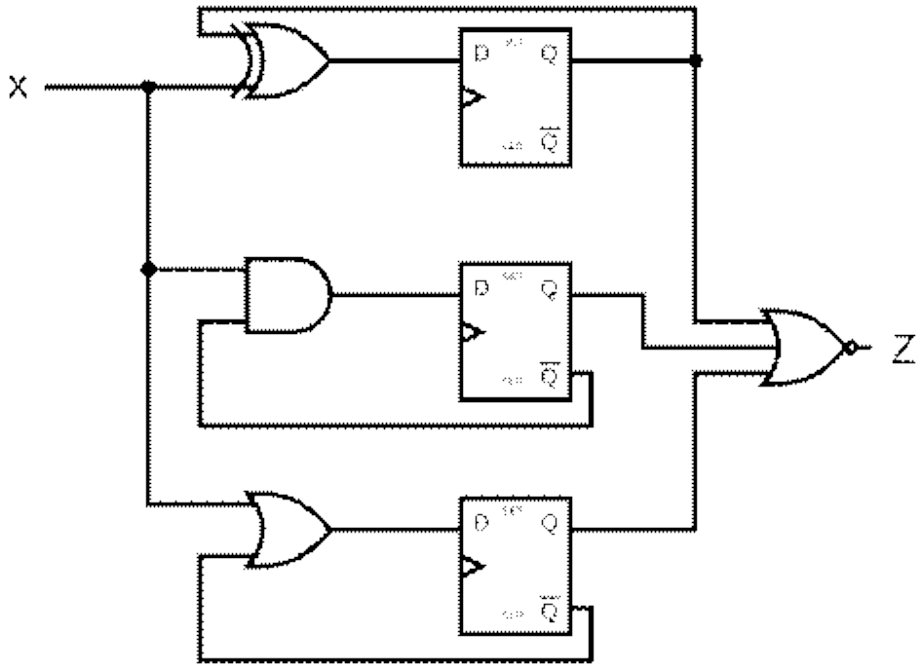
Exams/ece241 2014 q4

← exams/2014_q4a ✓ exams/ece241_2013_q7 →

Given the finite state machine circuit as shown, assume that the D flip-flops are initially reset to zero before the machine begins.

Build this circuit.

Screen clipping taken: 17-02-2024 23:10




```
module top_module (
    input clk,
    input x,
    output z
);
    wire q1,q2,q3,q2n,q3n;
            assign q2n =~q2 ;
            assign q3n =~q3 ;
            assign q1 =x^q1 ;
            assign q2 =x&q2n ;
            assign q3 =x|q3n ;
    always @(posedge clk)
        begin
            z <=~(q1|q2|q3);
        end
endmodule

```

Screen clipping taken: 17-02-2024 23:19

```
################## END OF LOOP #################
# ** Error (suppressible): (vsim-3601) Iteration limit 5000 reached at
time 45 ps.
# End time: 17:49:07 on Feb 17,2024, Elapsed time: 0:00:00
# Errors: 1, Warnings: 0
```

Status: Simulation Error

The code did not simulate. Check the error messages from Modelsim.

Warning messages that may be important

Modelsim messages (Show all)

```
# ** Error (suppressible): (vsim-3601) Iteration limit 5000 reached at time
45 ps.
```

This usually means your code is broken and the simulator can't simulate it. It's often caused by having a combinational loop or a latch with race conditions that oscillates, causing the simulator to never settle to a fixed value at this particular point in the simulation. Combinational loops can occur if some combinational logic (including `always@(*)` blocks) consumes a signal that it modifies. It can sometimes be caused by thinking in C while writing Verilog. Look for Quartus Warning (10240 or 13012) or Info (10041) regarding inferred latches or unsafe latch behaviour.

Screen clipping taken: 17-02-2024 23:19

```
module top_module (
    input clk,
    input x,
    output z
);
    reg q1,q2,q3;
    initial z=1;
    always @(posedge clk)
        begin
            q1 = x ^ q1 ;
            q2 = x & ~q2 ;
            q3 = x | ~q3 ;
           z <=~(q1|q2|q3);
        end
endmodule

```

Screen clipping taken: 17-02-2024 23:27

Exams/ece241 2013 q7

exams/ece241_2014_q4 edgedetect

A JK flip-flop has the below truth table. Implement a JK flip-flop with only a D-type flip-flop and gates. Note: Qold is the output of the D flip-flop before the positive clock edge.

J	K	Q
0	0	Qold
0	1	0
1	0	1
1	1	~Qold

Screen clipping taken: 17-02-2024 23:32

```
module top_module (
    input clk,
    input j,
    input k,
    output Q);
    wire [1:0] t;
    assign t={j,k};
    always @(posedge clk)
        begin
            case(t)
                2'b00: Q <=Q;
                2'b01: Q <=1'b0;
                2'b10: Q <=1'b1;
                2'b11: Q <=~Q;
            endcase
        end
endmodule

```

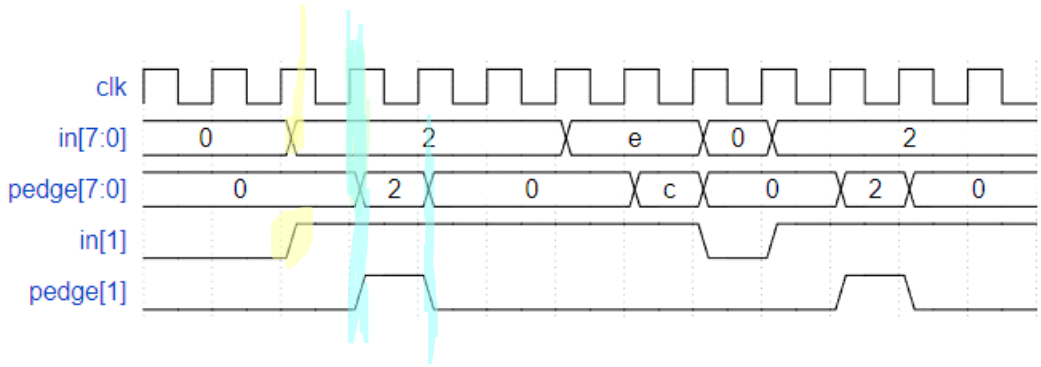
Screen clipping taken: 17-02-2024 23:33

Edgedetect

exams/ece241_2013_q7 edgedetect2

For each bit in an 8-bit vector, detect when the input signal changes from 0 in one clock cycle to 1 the next (similar to positive edge detection). The output bit should be set the cycle after a 0 to 1 transition occurs.

Here are some examples. For clarity, in[1] and pedge[1] are shown separately.



Screen clipping taken: 17-02-2024 23:34

Counters

16 February 2024 10:17

Shift Registers

16 February 2024 10:17

More Circuits

16 February 2024 10:17

FSM

16 February 2024 10:18

ALU

08 February 2024 22:20

Finding Bugs in a code

16 February 2024 10:18

Build a circuit from Simple Waveform

16 February 2024 10:19

Test-Benches

16 February 2024 10:19