1. Port assigning through Instantiated design by name is as below:

module mydesign(input x,y, output z);

assign z = x ^ y;

endmodule

module my\_tb;

reg a, b;

wire c;

**mydesign d0 (.x(a),**

**.y(b),**

**.z(c)); // .x means x is from the module mydesign that is being instantiated here**

// putting them in separate line so that any compilation error points to that port on that line number reducing debug time

endmodule

1. Bit select from part select means : w[x+:y] == w[(x+y-1):x] // this is little endian so its w[MSB:LSB]

1. Simulation wise -- testbench is the top level module because the top level design is instantiated inside it.
2. Signals inside a module can be accessed using hierarchical operator "." ….. So a signal inside a top level design module can be accessed from a testbench using the "dot" operator such as:

testbench.d0.\_net // \_net is the signal inside the design module

/\*

module testbench;

design d0(<port list>);

endmodule

module design;

wire \_net;

mod1 mod1\_inst;

mod2 mod2\_inst;

endmodule

\*/

1. 2 drivers for the same bit during assign statement creates conflict for the bit and is therefore assigned "X"

Eg.

module design(input [3:0]x,

y,

output [4:0]z);

x = 4'b1100;

y = 1'b1;

assign z[3:1] = {x,y}; // z = Z001Z; here z[3] = 0

assign z[3] = 1; // here z[3] = 1 , so creates conflict of 2 drivers for the same signal bit3 making it a X

endmodule

//if x = 4'b1100, y=1

assign z = {x[1:0],y}; // z = 5'b00\_001 , it will not be z the initial 2 values

assign z[2:0] = {x, y}; // z= 5'bZZ\_001 , because z[4:3] is explicitly left open so they will be Z

1. Little endian --> lower index has LSB means { reg [3:0]a } where a[0] is LSB so little means lower
2. Implicit nets are always one-bit wires and causes bugs if you had intended to use a vector. Disabling creation of implicit nets can be done using the `default\_nettype none directive.

eg.

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.

1. **Important question: reversal of bits of input and display them in output.**

**For loops** do NOT get iterated when the circuit hardware gets designed, meaning the circuit does not perform iteration after the schematic is ready….

//for-loops describe circuit \*behaviour\*, not \*structure\*, so they can only be used

// inside procedural blocks (e.g., always block).

// The circuit created (wires and gates) does NOT do any iteration: It only produces the same result

// AS IF the iteration occurred. In reality, a **logic synthesizer** will **do the iteration at compile time** to

// figure out what circuit to produce. (In contrast, a Verilog simulator will execute the loop sequentially

// during simulation.)

always @(\*) begin

for (int i=0; i<$bits(out); i++)        // int is a SystemVerilog type. Use integer for pure Verilog.

**out[i] = in[$bits(out)-i-1]; //BIT REVERSAL**

end

// It is also possible to do this with a generate-for loop. Generate loops look like procedural for loops,

// but are quite different in concept, and not easy to understand. Generate loops are used to make instantiations

// of "things" (Unlike procedural loops, it doesn't describe actions). These "things" are assign statements,

// module instantiations, net/variable declarations, and procedural blocks (things you can create when NOT inside

// a procedure). Generate loops (and genvars) are evaluated entirely at compile time. You can think of generate

// blocks as a form of preprocessing to generate more code, which is then run though the logic synthesizer.

// In the example below, the generate-for loop first creates 8 assign statements at compile time, which is then

// synthesized.

// Note that because of its intended usage (generating code at compile time), there are some restrictions

// on how you use them. Examples: 1. Quartus requires a generate-for loop to have a named begin-end block

// attached (in this example, named "my\_block\_name"). 2. Inside the loop body, genvars are read only.

generate

genvar i;

for (i=0; i<8; i = i+1) begin: my\_block\_name

assign out[i] = in[8-i-1];

end

endgenerate

1. signed negative numbers can be represented as -6'sd3

A screenshot of a computer

Description automatically generated

1. here the sum[15:0] cannot be given to 2 different instances of modules like add16 and add1 , 2 different wires will be required to represent the last cout of add1 and add16 and sum as well .. its logical because output of sum[15] cannot come from 2 different drivers of add16 and add1 hence separate wires needed for both cout last and sum[15];

wire c0,c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,**c15**;

wire s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,**s15**;

wire **carry1**;

add16 a1(a[15:0],b[15:0],0,**sum[15:0]**,carry1);

add1 b15(a[15],b[15],c14,**s15**,**c15**);

1. **xor gate using mux** for one of the input is used for adder/subtractor circuit (eg. 32 bit adder/sub done using 2-16 bit adders with a a,b being 32 bit inputs and sub being the cin… xor implementation is "sub? ~b: b" within a 32 bit wire [31:0]w1,

assign w1 = sub ? ~b : b; so this wire is used in the instantiation of b in the 2-16 bit adder modules )

1. (\**Procedural continuous assignments* do exist, but are somewhat different from *continuous assignments*, and are not synthesizable.)
2. We can **infer latches in case even with a default statement is present , situation is when we have more than 1 outputs**… So eg.

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

module top\_module (

input [15:0] scancode,

output reg left,

output reg down,

output reg right,

output reg up );

always @(\*)

case(scancode)

16'he06b:        begin **left = 1'b1**; right=0;up=0;down=0; end

16'he072:        begin left=0;right=0;up=0;**down = 1'b1**; end

16'he074:        begin left=0;**right = 1'b1**;up=0;down=0; end

16'he075:        begin left=0;right=0;**up = 1'b1**;down=0; end

**default**:         begin left=0;right=0;up=0;down=0; end

endcase

endmodule

**Note: In the 1st case 16'he06b if we don’t mention the values of right,up,down we will infer a latch because maybe previously the 4th case 16'he075 was true and made up=1 , so now both up and left will be 1 during this 1st case showing a latch.**

1. COUNTING THE NUMBER OF 1's IN A 255 BIT vector in an efficient way ::: Note: A 255 bit adder will have max 255 ->1s , so 8 bit output is sufficient.

module top\_module(

input [254:0] in,

output [7:0] out );

integer i;

always @(\*)begin

out = {8{1'b0}};

for(i=0;i<255;i=i+1)

begin

//Less efficient way --> if(in[i]) begin out = out + 1; end

//Better way is --> adding each bit of the input in[i] to the previous value of out[8:0]

//because even if in[i] =0 it doesnt affect the out[8:0] so it is a 256 bit adder

out = out + in[i];

        end

end

endmodule

1. 100 bit adder using instantiations CANNOT be done using always block … ONLY USING GENERATE LOOP

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;

fa inst0(a[0],b[0],cin,sum[0],cout[0]); //any of these 2 can be used to instantiate the 0th adder

genvar i;

generate

for(i=1;i<$bits(a);i=i+1)

begin:**gen**

fa inst1(a[i],b[i],cout[**i-1**],sum[i],cout[i]);

end:**gen**

endgenerate

endmodule

module fa(input a\_1,b\_1,cin\_1,output sum\_1,cout\_1);

assign {cout\_1,sum\_1} = a\_1 + b\_1 + cin\_1;

endmodule

1. 400 bit BCD adder , in -short 4 bit BCD adder instantiated 100 times using GENERATE LOOP:

module top\_module(

input [399:0] a, b,

input cin,

output cout, // 99 bits of cout is not given so I have used wire of 100 bits to help cin of the 100 digits

output [399:0] sum );

wire [99:0]wout;

bcd\_fad inst0(a[3:0],b[3:0],cin,wout[0],sum[3:0]);

genvar i;

generate

for (i=4;i<400;i=i+4) begin: gen

bcd\_fad inst(a[i+3:i],b[i+3:i],wout[i/4-1],wout[i/4],sum[i+3:i]);

/\*BCD adder instantiation from 5th bit onwards

// wout[i/4-1] is cin of the previous bit just like full adder

// and wout[i/4] is cout starting from cout[0] to cout[99] all the 100 -> 4bit added results\*/

end

endgenerate

assign cout = wout[99];

endmodule

module bcd\_fad(

input [3:0]a,

input [3:0]b,

input cin,

output cout,

output [3:0]sum

);

wire [3:0]w\_sum;

wire w\_cout;

assign {w\_cout,w\_sum} = a + b + cin;

assign cout = (w\_sum[3] & w\_sum[2]) | (w\_sum[3] & w\_sum[1]) | w\_cout; // Look at below diagram and kmap will be an output for 4 bit sum>9 so it gives (s3s2 | s3s1)

assign sum = w\_sum + {1'b0,{2{cout}},1'b0};

endmodule

A diagram of a block diagram

Description automatically generated

1. assign out = b[99:0] & **{100{sel}}** | a[99:0] & ~**{100{sel}}** ; --> **100 bit MUX of 2x1** , without the 100 times replicated select line it won't work, gets stuck at one input.

or assign out = sel? b:a; // correct for any no. of bits wide vector

1. assign out\_both[98:0] = in[98:0] & in[99:1]; // in[98] & in[99] is--> out\_both[98] … Similarly other bits to the left of LSB

assign out\_any[99:1] = in[99:1] | in[98:0]; // in[99] | in[98] is--> out\_any[99] … Similarly other bits to the right of MSB

assign out\_different[99:0] = {(in[99] ^ in[0]), in[98:0] ^ in[99:1]} ; // in[99] ^ in[0] is--> out\_different[99] … in[98] ^ in[99] -->out\_diff[98] Similarly other bits in circular manner to the left of LSB

1. assign out = **in[sel]**; // 256X1MUX for a 256 BIT wide input [255:0]in , [7:0]sel, --> **in 1 line** assigning 256x1 mux
2. assign out = {in[sel\*4+3],in[sel\*4+2],in[sel\*4+1],in[sel\*4+0]};

assign out = in[sel\*4 +:4]; // example: final value sel=8'd255, so in[1020 +:4] --> 1020 to 1023

                 //for a 4 bit bus of input

1. // assign sum = {x+y};        // Concatenation operator: This discards the carry-out , hence just (x+y) is the adder operation required
2. assign **overflow** = (~(a[7] ^ b[7])) && (s[7] != a[7]); // Overflow condition is last digit of the bit are equal (means xnor) and result sum's MSB is not same as those any one of //those a[7] or b[7].. then it is overflow because it means addition of 2 negative nos. gave a positive no. because this is addition of 2's complemented 2 vectors a and b
3. **XOR** OF MORE THAN 2 INPUTS MEANS: If input has **ODD** number of 1's its output is 1
4. always@(posedge clk)

begin

out<=in^out; // means at every posedge :result out:: give the xor of "in" and its "previous out"

end

A black and white drawing of a person's face

Description automatically generated

1. SINGLE EDGE DETECTION OF A 8 BIT INPUT VECTOR IS DONE WITH PREVIOUS INPUT CONSIDERATION like below:

reg [7:0] d\_last;

always @(posedge clk) begin

d\_last <= in;                        // Remember the state of the previous cycle

pedge <= in & ~d\_last;        // A positive edge occurred if input was 0 and is now 1..pedge is the output

end

1. BOTH EDGE DETECTION will be xor of d\_last and in::

reg [7:0] d\_last;

always @(posedge clk) begin

d\_last <= in;                        // Remember the state of the previous cycle

anyedge <= in ^ d\_last;        // A positive edge occurred if input was 0 and is now 1..anyedge is the output

end

A diagram of a circuit

Description automatically generated

1. Dual edge triggered flip flop which cant be simply written with a always @(posedge clk or negedge clk) it requires the above circuit:

always@(posedge clk)

p<= d ^ n;

always@(negedge clk)

n<= d ^ p;

assign q = p ^ n;

reg [31:0] d\_last;

1. For each bit in a 32-bit vector, capture when the input signal changes from 1 in one clock cycle to 0 the next. "Capture" means that the output will remain 1 until the register is reset (synchronous reset).

Each output bit behaves like a SR flip-flop: The output bit should be set (to 1) the cycle after a 1 to 0 transition occurs. The output bit should be reset (to 0) at the positive clock edge when reset is high. If both of the above events occur at the same time, reset has precedence. In the last 4 cycles of the example waveform below, the 'reset' event occurs one cycle earlier than the 'set' event, so there is no conflict here. <<https://hdlbits.01xz.net/wiki/Edgecapture>> ::

1. always @(posedge clk) begin

d\_last <= in;

if(reset)

out <= 0;

else begin

out <= out | (d\_last & ~in);

end

end

1. Decade counter with enable signal:

module bcdcount (

input clk,

input reset,

input slowena,

output [3:0] q);

always @(posedge clk) begin

if (reset) q <= 4'h0;

else if (slowena) begin

if (q == 4'h9) q<= 4'h0;

        else q <= q + 4'h1;

end

else q<=q;

end

endmodule

1. WALL CLOCK for:::

wire [3:0] q0, q1, q2;

bcdcount counter0 (clk, reset, c\_enable[0], q0);

bcdcount counter1 (clk, reset, c\_enable[1], q1);

bcdcount counter2 (clk, reset, c\_enable[2], q2);

assign c\_enable = {(q1 == 4'd9) & (q0 == 4'd9), q0 == 4'd9, 1'b1};

assign OneHertz = (q2 == 4'd9) & (q1 == 4'd9) & (q0 == 4'd9);

1. 3 BCD counter ::

always @(posedge clk) begin

if(reset) begin

q = 0;

ena = 0;

end

else begin

q[3:0] = q[3:0] + 1;

ena[1] = (q[3:0] == 9) ? 1 : 0;

if(q[3:0] == 10) begin

q[3:0] = 0;

q[7:4] = q[7:4] + 1;

end

ena[2] = ((q[7:4] == 9) && (q[3:0] == 9)) ? 1 : 0;

if(q[7:4] == 10) begin

q[7:4] = 0;

q[11:8] = q[11:8] + 1;

end

ena[3] = ((q[11:8] == 9) && (q[7:4] == 9) && (q[3:0] == 9)) ? 1 : 0;

if(q[11:8] == 10) begin

q[11:8] = 0;

q[15:12] = q[15:12] + 1;

end

if(q[15:12] == 10) begin

q = 0;

end

end

end