GTU

DEPARTMENT OF

COMPUTER ENGINEERING

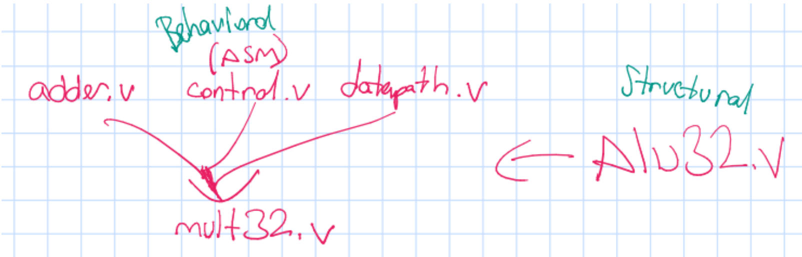
CSE 331 – Autumn 2022

HOMEWORK 2
REPORT

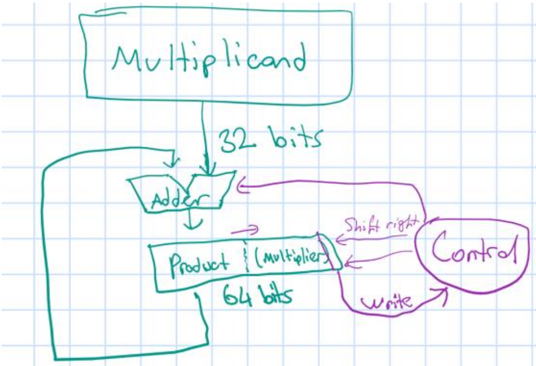
SÜLEYMAN GÖLBOL

1801042656

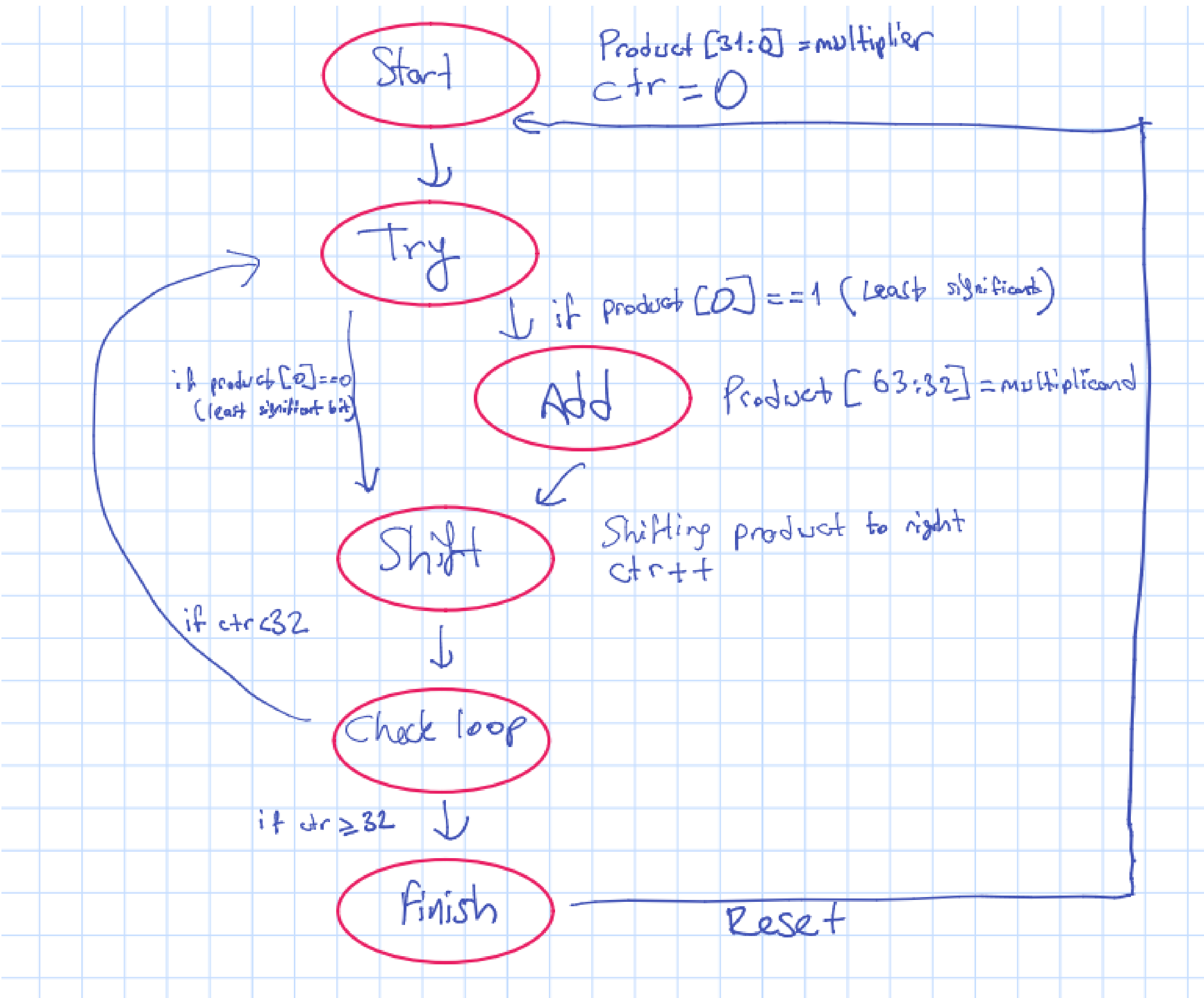
Main Components



Multiplication Component



HL STATE MACHINE



FINITE STATE MACHINE OF VARIABLES

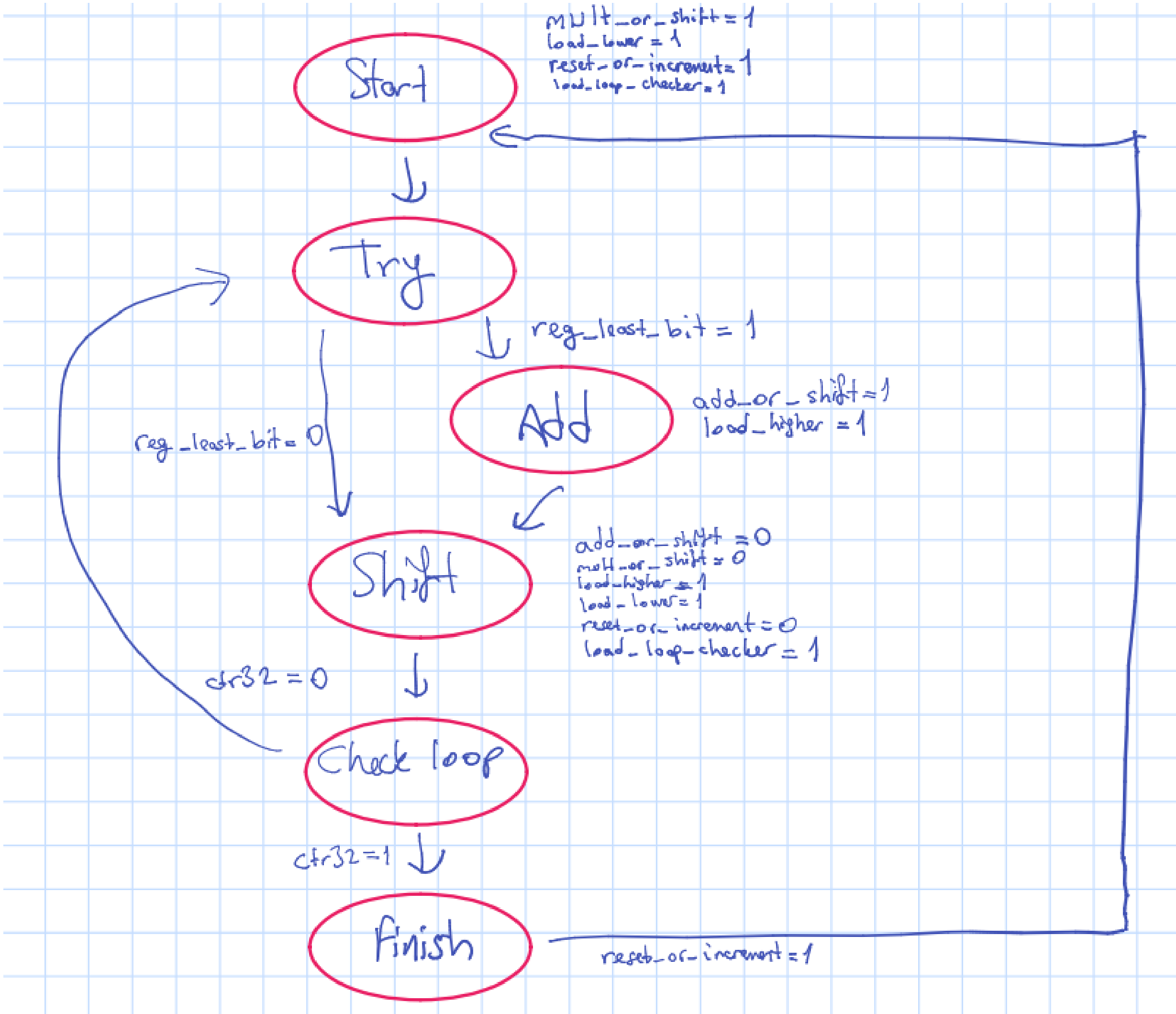


TABLE FOR EVERY STATE

States	R[2]	R[1]	R[0]	input	Next	N[2]	N[1]	N[0]
Start	0	0	0	1	Try	0	0	1
Try	0	0	1	reg_least_bit	Add	0	1	0
Try	0	0	1	¬reg_least_bit	Shift	0	1	1
Add	0	1	0	1	Shift	0	1	1
Shift	0	1	1	1	Check loop	1	0	0
Check loop	1	0	0	¬ctr32	try	0	0	1
Check loop	1	0	0	ctr32	finish	1	0	1
finish	1	0	1	reset	Start	0	0	0
finish	1	0	1	¬reset	finish	1	0	1

DERIVING LOGIC

$$N2 = R2' . R1 . R0 + R2 . R1' . R0' . ctr32 + R2 . R1' . R0 . reset'$$

$$N1 = R2' . R1' . R0 + R2' . R1 . R0'$$

$$N0 = R2' . R1' . R0' + R2' . R1' . R0 . reg\_least\_bit + R2' . R1 . R0' + R2 . R1' . R0' + R2 . R1' . R0 . reset$$

$$mult\_or\_shift = R2' . R1' . R0'$$

$$load\_lower = R2' . R1' . R0' + R2' . R1 . R0$$

$$load\_loop\_checker = R2' . R1' . R0' + R2' . R1 . R0$$

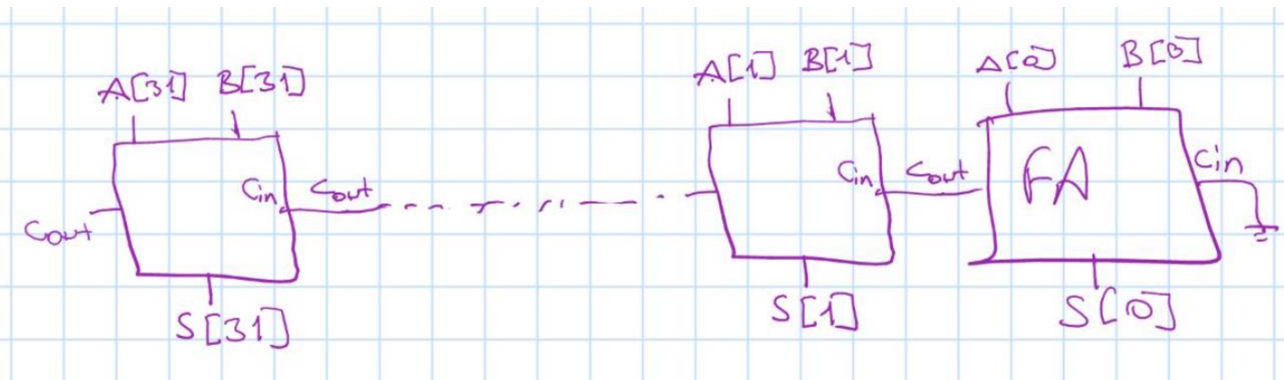
$$reset\_or\_increment = R2' . R1' . R0'$$

$$add\_or\_shift = R2' . R1 . R0'$$

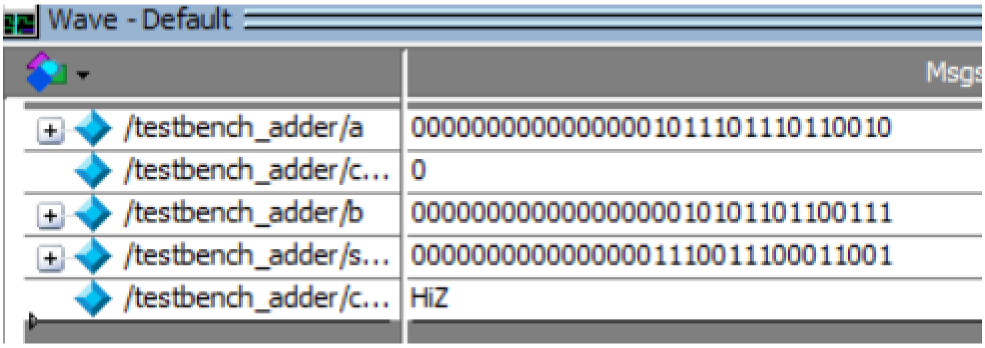
$$load\_higher = R2' . R1 . R0' + R2' . R1 . R0$$

1. ADDER COMPONENT

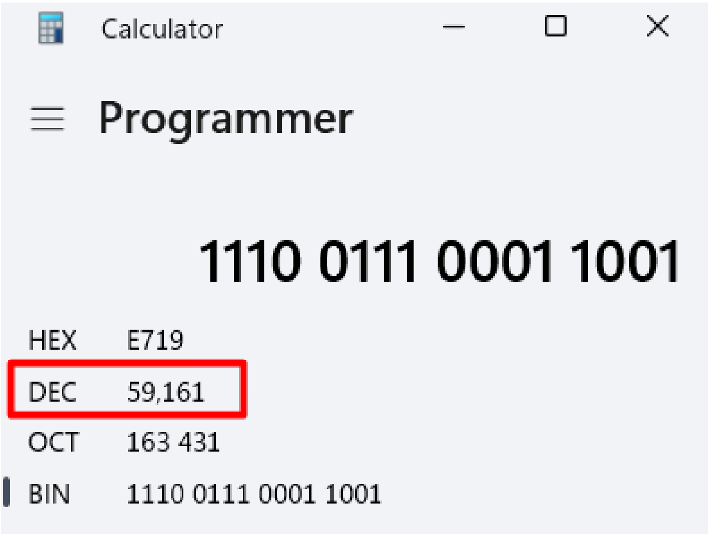
For adder component, I created a 1-bit full adder. To create 32 bits adder, I used every full adder’s “carry out” value to connect. Then I put the output into 32 bits sum variable.



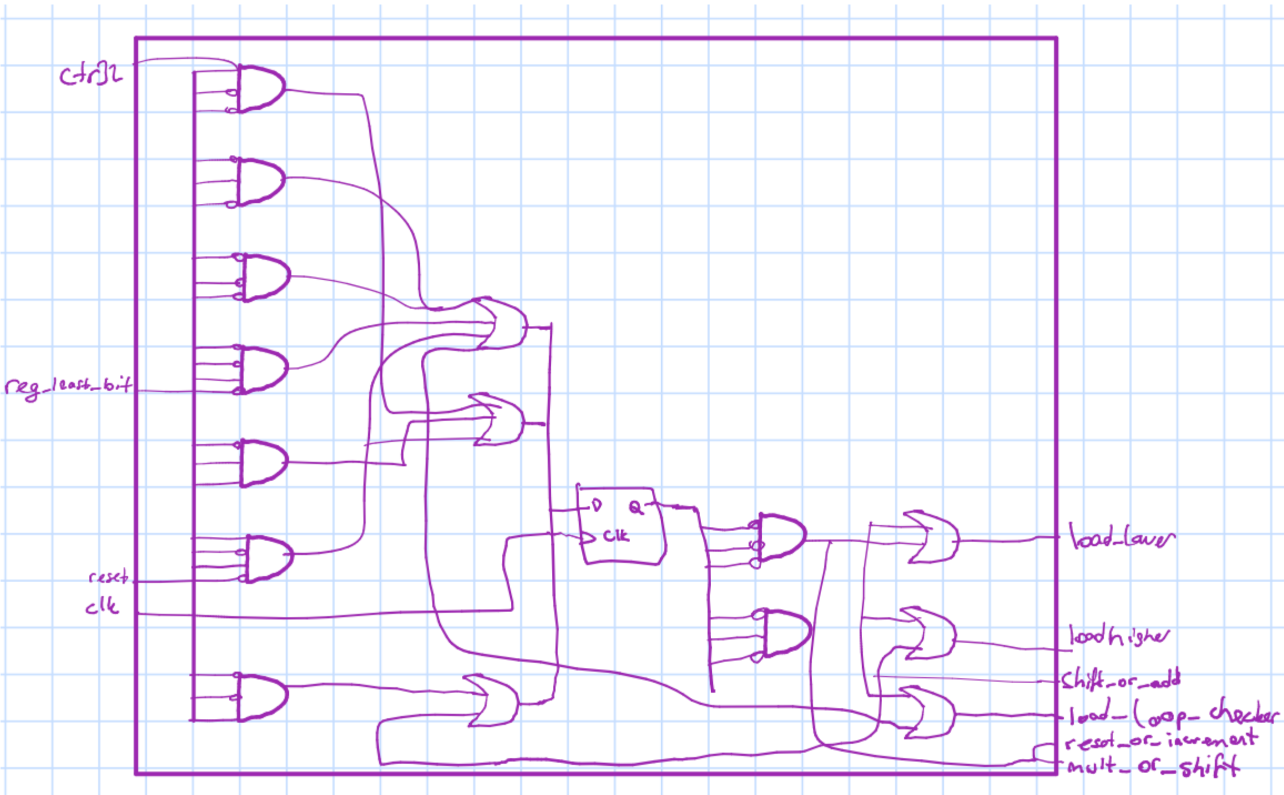
Testing



I tried to add 48050 and 11111. The sum should be 59161.

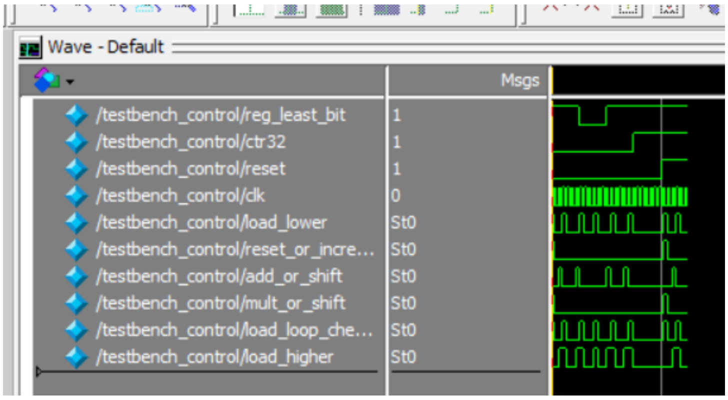


2. CONTROL COMPONENT

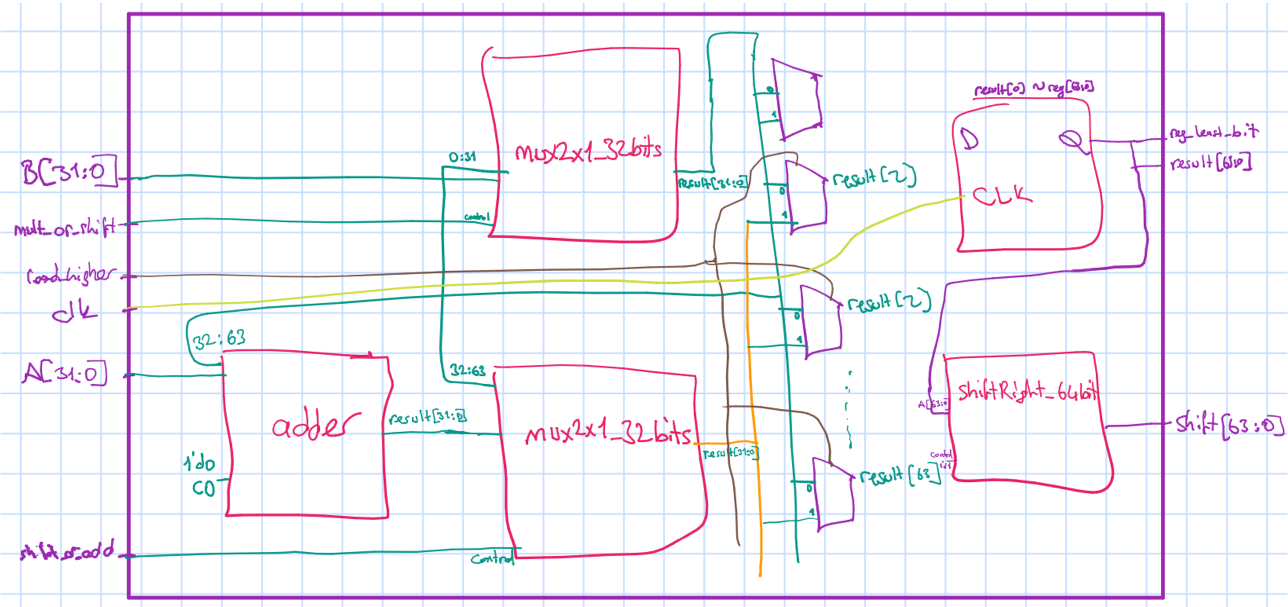


For the control part, I needed to use state table variables values to achieve mult_or_shift, reset_or_increment, load_loop_checker, load_higher, load_lower values using inputs reset, clk and reg_least_bit.

At the end values are like that.



3. DATAPATH COMPONENT



For the datapath component I used 2 multiplexer to select current state or selecting 64 bit number to produce. Multiplexer above takes first 32 bits and below takes last 32 bits. Mult_or_Shift is connected to first multiplexer to control selection. Below, add_or_shift is connected to multiplexer to control selection. Shiftright_64bit has control signal which is 1, so it will shift right the 64 bit number.

Since it will continue until 32 there is a loop checker and ctr32 connected to incrementer. Incrementer uses mux2x1_8bits, adder_8bits and setonlessthan_32bits to increment the number.

Also, to prevent repetition for array-like variables, I used generate keyword with “genvar” variable.

```
// generation variable to prevent repetition
genvar i;
// NOTE: FOR LOOP IN GENERATE IS NOT BEHVARIORAL. IT JUST TO PREVENT REPETITION.
// for loop for and1
generate
   for (i=0; i<=62; i = i+1) begin: mymux
      mux2x1_1bit mux(A[i], A[i+1], control, result[i]);
   end
endgenerate
```

It’s just a helper not to write same thing tens of times.

4. MULT32.V

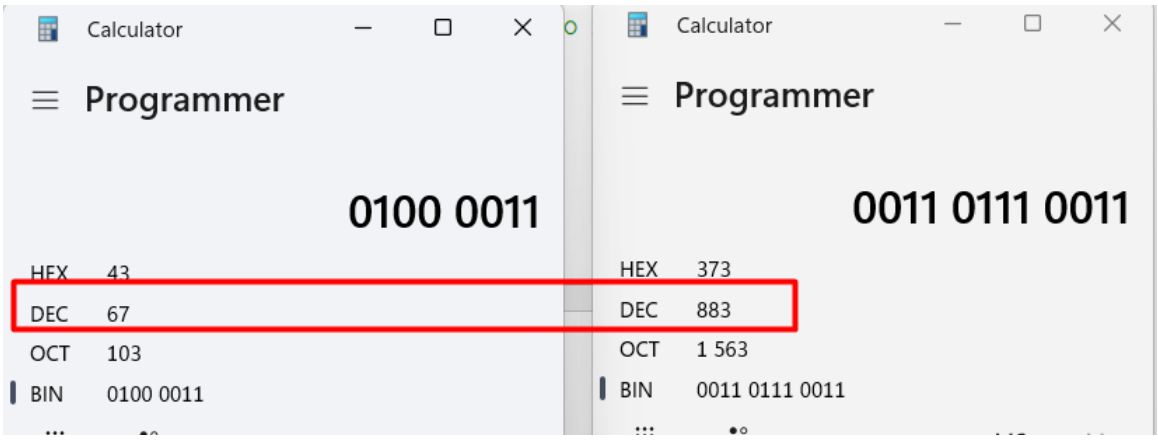
Multiplier module connects them all to multiply number.

```
control mycontroller (reg_least_bit, ctr32, reset, clk, add_or_shift, load_higher, mult_or_shift,
                      load_lower, reset_or_increment, load_loop_checker);
datapath mydatapath (add_or_shift, load_higher, mult_or_shift, load_lower, reset_or_increment, load_loop_checker,
                      a, b, clk, shift_wire, result, reg_least_bit, ctr32);
```

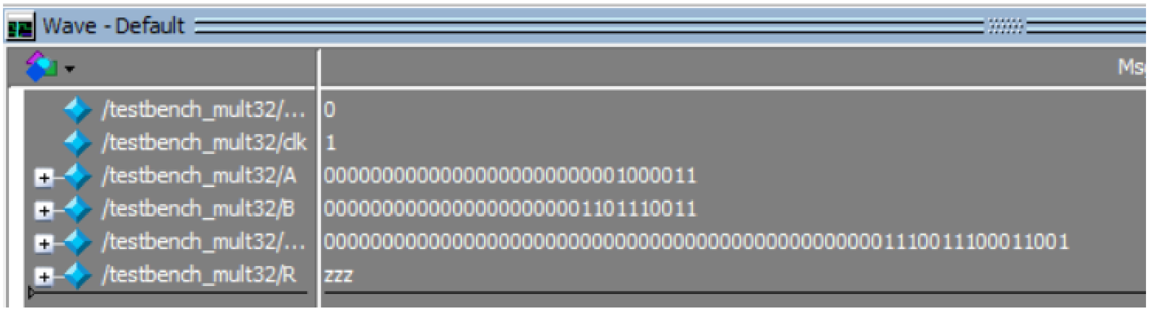
As it seen first uses control then uses datapath module with all needed parameters.

Testing

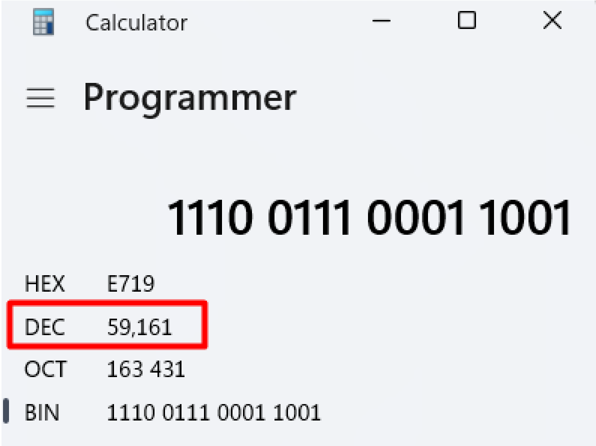
Multiplying 67 * 883



After adding waves in ModelSim app to testbench, this is the result.



This is the result as expected because 67 * 883 = 59161



5. ALU32.V

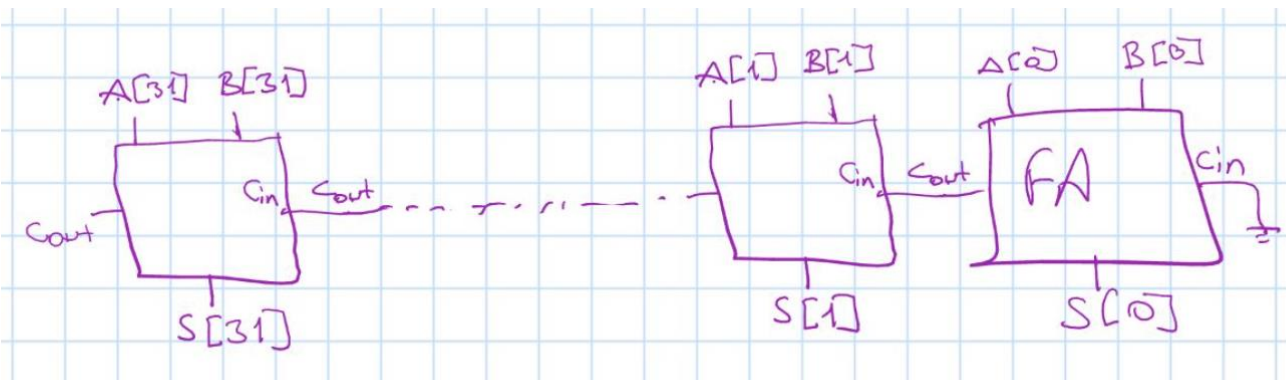
Alu32 module have a multiplexer to select one of the 8 options.

000->ADD	100->AND
001->SUB	101->OR
010->MULT	110->SLT
011->XOR	111->NOR

```
adder gatea (a,b, 1'b0, add_wire);

xor_32bits gateb (a, b, xor_wire);

substractor_32bits gatec (a, b, subs_wire);

mult32 gated(a, b, reset, clk, mult_wire);

slt_32bits gatee (a, b, slt_wire);

nor_32bits gatef (a,b, nor_wire);

and_32bits gateg (a,b, and_wire);

or_32bits gateh (a,b, or_wire);

mux8x1_64bits gatei({32'd0,add_wire}, {32'd0,subs_wire}, mult_wire, {32'd0,xor_wire},
                  {32'd0,and_wire}, {32'd0,or_wire}, {32'd0,slt_wire}, {32'd0,nor_wire}, ALUop, result);
```

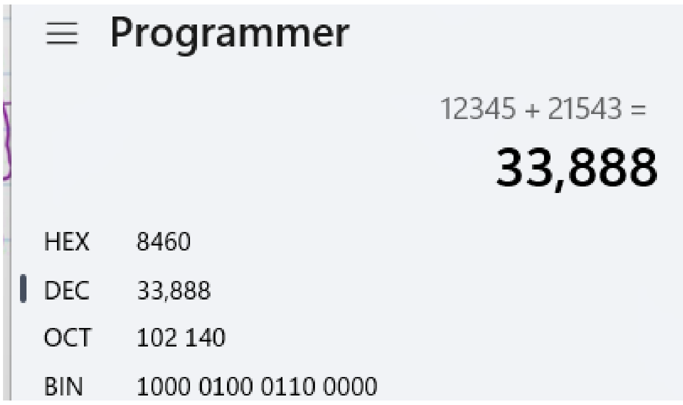
Adder

Like I wrote above, it’s made by connecting full adders.



A and B Inputs are: 12345 and 21543. A+B = 33888

```
# ALUop: 000
# A and B:
# 00000000000000000011000000111001
# 00000000000000000101010000100111
# result:
# 00000000000000001000010001100000
```



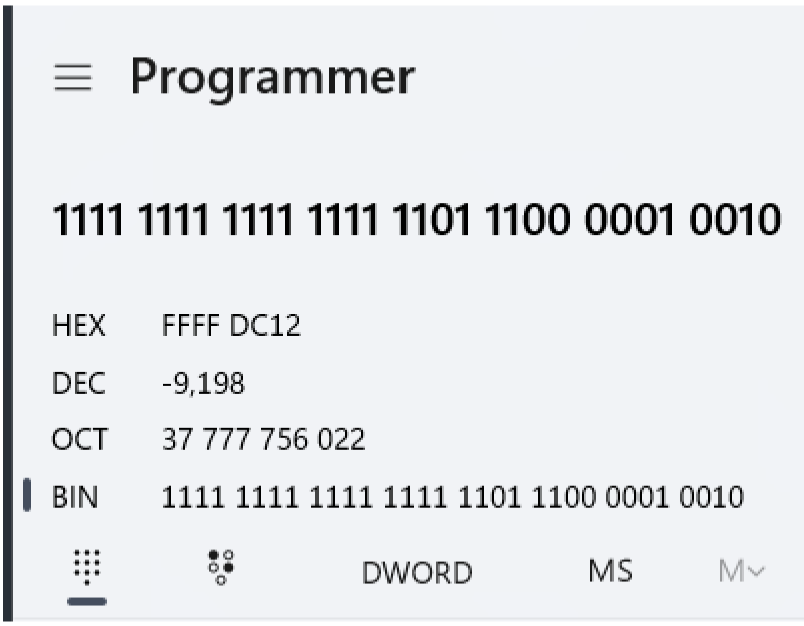
Sub

```
module substractor_32bits(input [31:0] a, input [31:0] b, output [31:0] result);
   wire [31:0] wirevar;

   xor_32bits gate1(b, 32'b11111111111111111111111111111111, wirevar);
   adder gate2(a, wirevar /*b*/, 1'b1, result);

endmodule
```

It only takes 2's complement for sub

```
ALUop: 001
A and B:
00000000000000000011000000111001
00000000000000000101010000100111
result:
11111111111111111101110000010010
```

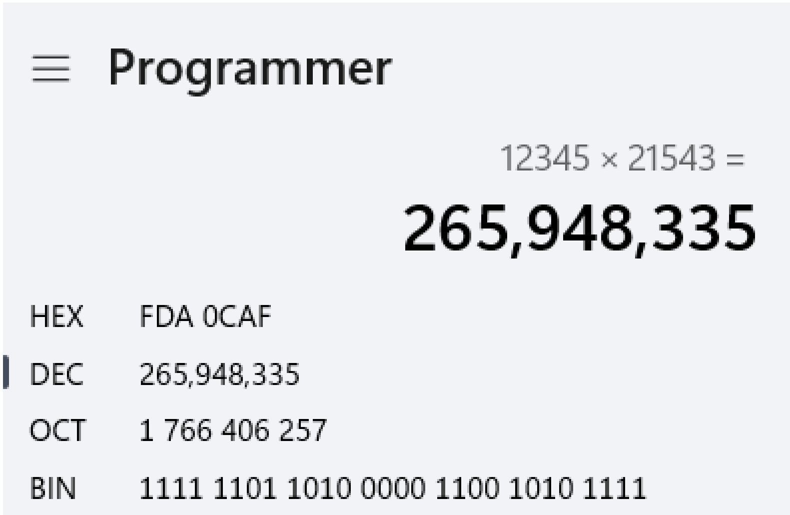
A and B Inputs are: 12345 and 21543. A-B = -9198



Mult

```
ALUop: 010
A and B:
00000000000000000011000000111001
00000000000000000101010000100111
result:
00001111110110100000110010101111
```

A and B Inputs are: 12345 and 21543. A*B= 265948335



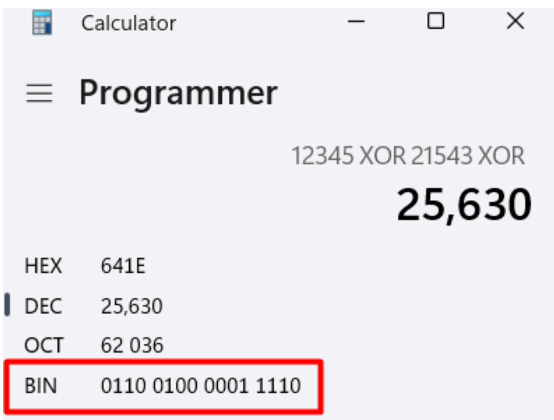
Ignore 0’s in the beginning in the mult result.

Important Note:

ModelSim is printing all intermediate stages for multiplication. Please ignore Other AluOp:010 results and look only to the last ‘ALUop:010’ result.

Xor

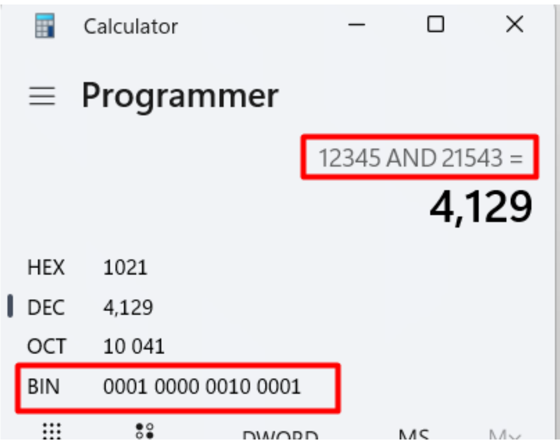
```
# ALUop: 011
# A and B:
# 00000000000000000011000000111001
# 00000000000000000101010000100111
# result:
# 00000000000000000110010000011110
#
```



A and B Inputs are: 12345 and 21543. Result is same.

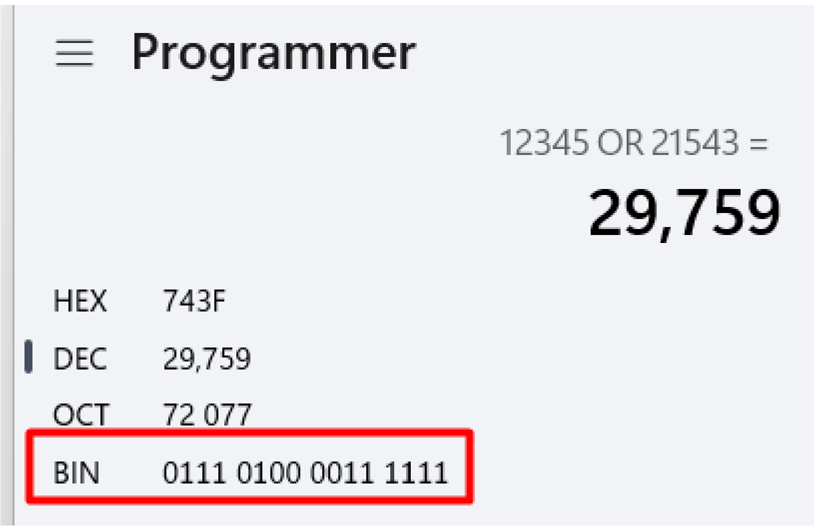
And

```
# ALUop: 100
# A and B:
# 00000000000000000011000000111001
# 00000000000000000101010000100111
# result:
# 00000000000000000001000000100001
```



Or

```
# ALUop: 101
# A and B:
# 00000000000000000011000000111001
# 00000000000000000101010000100111
# result:
# 00000000000000000111010000111111
```



They are same.

Slt

```
module slt_32bits (input [31:0] A, input [31:0] B, output [31:0] result);

wire [31:0] wirevar;

    // generation variable to prevent repetition
    genvar i;
    // for loop for and1
    generate
        for (i=1; i<32; i = i+1) begin: mynotgate
            not notgate(result[i], 1'b1);
        end
    endgenerate

    substractor_32bits gate1(A, B, wirevar);
    or (result[0], wirevar[31], 1'b0);

endmodule
```

First 32 bits are full of 0's. And the last bit is found out by using subtractions most significant bit.

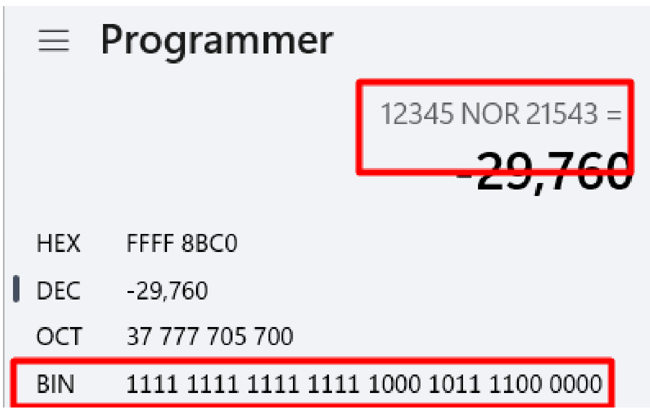
```
# ALUop: 110
# A and B:
# 00000000000000000011000000111001
# 00000000000000000101010000100111
# result:
# 00000000000000000000000000000001
#
```

Since A(12345) is smaller than B(21543), result will be 1.

Nor

Just nors every bit.

```
# A and B:
# 00000000000000000011000000111001
# 00000000000000000101010000100111
# result:
# 11111111111111111000101111000000
#
```



Results are same.