Davinci System – Gate Level Model

Derek Ortega

CS147

San Jose State University

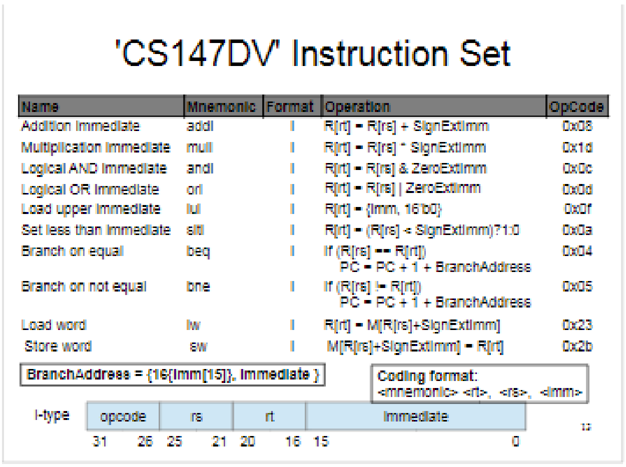
Derekortega0@gmail.com

Abstract **– This report will describe the design and implementation of our davinci system using gate level modeling. This system will be able to execute the CS147 Instruction set. It contains many low level components and logic gates that come together in the data path, registers, and control unit.**

Index:

I. Requirement for System

The objective of this project is to implement a mixed model of a computer system with a 32-bit processor and 256MB of memory. The processor is also required to support the instruction set CS147DV.



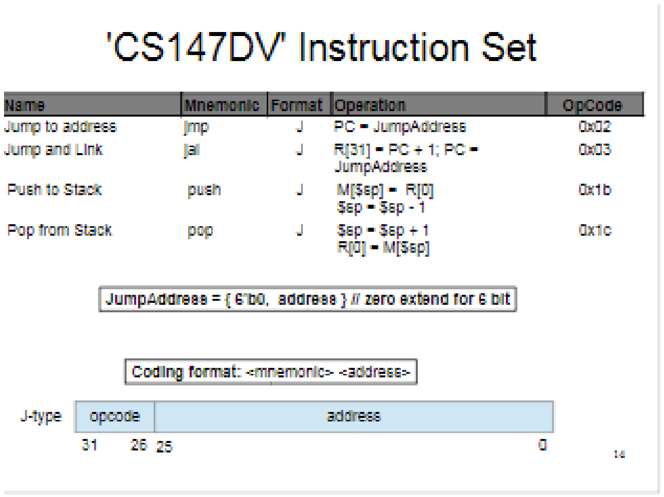
'CS147DV' Instruction Set

Name	Mnemonic	Format	Operation	OpCode
Addition Immediate	addi	I	R[rt] = R[rs] + SignExtImm	0x08
Multiplication Immediate	muli	I	R[rt] = R[rs] * SignExtImm	0x1d
Logical AND Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	0x0c
Logical OR Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	0x0d
Load upper Immediate	lui	I	R[rt] = {Imm, 16'b0}	0x0f
Set less than Immediate	slti	I	R[rt] = (R[rs] < SignExtImm)?1:0	0x0a
Branch on equal	beq	I	If (R[rs] == R[rt]) PC = PC + 1 + BranchAddress	0x04
Branch on not equal	bne	I	If (R[rs] != R[rt]) PC = PC + 1 + BranchAddress	0x05
Load word	lw	I	R[rt] = M[R[rs]+SignExtImm]	0x23
Store word	sw	I	M[R[rs]+SignExtImm] = R[rt]	0x2b

BranchAddress = {16{Imm[15]}, Immediate }

Coding format:
<mnemonic> <rt>, <rs>, <imm>

I-type	opcode	rs	rt	Immediate
	31 26	25 21	20 16	15 0



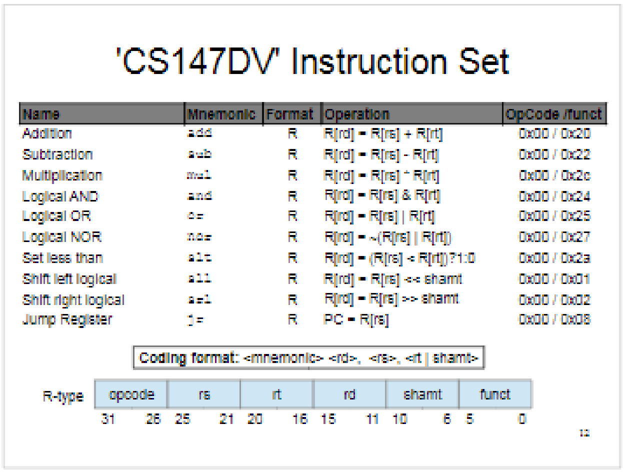
'CS147DV' Instruction Set

Name	Mnemonic	Format	Operation	OpCode
Jump to address	jmp	J	PC = JumpAddress	0x02
Jump and Link	jal	J	R[31] = PC + 1; PC = JumpAddress	0x03
Push to Stack	push	J	M[$sp] = R[0] $sp = $sp - 1	0x1b
Pop from Stack	pop	J	$sp = $sp + 1 R[0] = M[$sp]	0x1c

JumpAddress = { 6'b0, address } // zero extend for 6 bit

Coding format: <mnemonic> <address>

J-type	opcode	address
	31 26	25 0



'CS147DV' Instruction Set

Name	Mnemonic	Format	Operation	OpCode /funct
Addition	add	R	R[rd] = R[rs] + R[rt]	0x00 / 0x20
Subtraction	sub	R	R[rd] = R[rs] - R[rt]	0x00 / 0x22
Multiplication	mul	R	R[rd] = R[rs] * R[rt]	0x00 / 0x2c
Logical AND	and	R	R[rd] = R[rs] & R[rt]	0x00 / 0x24
Logical OR	or	R	R[rd] = R[rs] | R[rt]	0x00 / 0x25
Logical NOR	nor	R	R[rd] = ~(R[rs] | R[rt])	0x00 / 0x27
Set less than	slt	R	R[rd] = (R[rs] < R[rt])?1:0	0x00 / 0x2a
Shift left logical	sll	R	R[rd] = R[rs] << shamt	0x00 / 0x01
Shift right logical	srl	R	R[rd] = R[rs] >> shamt	0x00 / 0x02
Jump Register	jr	R	PC = R[rs]	0x00 / 0x08

Coding format: <mnemonic> <rd>, <rs>, <rt | shamt>

	opcode	rs	rt	rd	shamt	funct
R-type	31 26	25 21	20 16	15 11	10 6	5 0

12

1. ALU

The ALU needs to be implemented at the gate level and needs to be able to support the following functions

- Addition
- Subtraction
- Multiplication
- Shift Logical Right
- Shift Logical Left
- AND
- OR
- NOR
- SLT

2. Register File

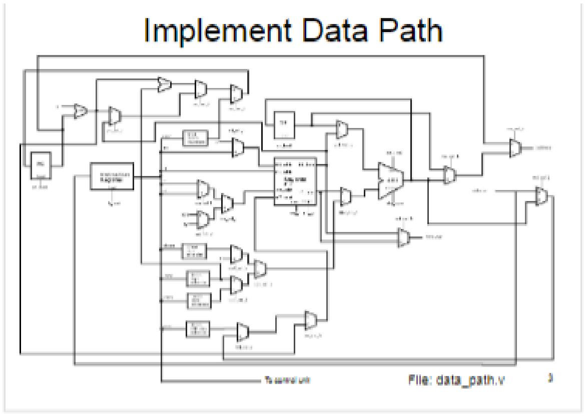
Register file is dual read with 32 registers of 32 bits. Reset is done on the –ve edge of the RST signal, while the rest of the operation is done at the +ve edge of the CLK signal. Read Operation is done if READ =1, WRITE =0. Write is done if WRITE =1, READ =0. DATA_R will equal X if both READ and WRITE are 0 or 1.

3. Memory

Memory is to be implemented at the behavioral level. Therefor we can use our old memory modules from project 2. It is 32 bit word accessible, Reset is done on –ve edge. Rest of the operation is done at the +ve edge of the clock signal.

4. Data path

Data path will also be assembled at the gate level and the PC and SP registers must also be implemented at the gate level. Data path is created to replicate the following diagram.



5. Control Unit

The control unit is made up of five different stages, FETCH, DECODE, EXECUTE, MEMORY, WB. Each step will change the control register that will be sent to the ALU and other components of the data path which will determine what operations and actions to take inside the system.

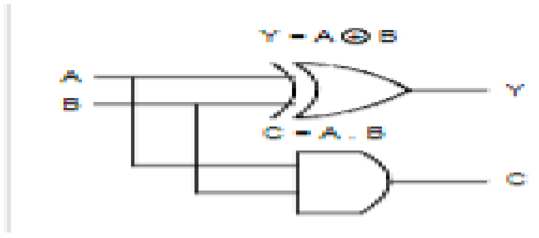
II. Design, Implementation, and testing

This section will describe how we designed our system and the implementation and testing of the various components/modules of our system

1. Half Adder

a. Design

The half adder is the basis for all of our addition and subtraction operations. It is composed of a single AND gate and a single XOR gate and follows the logic circuit below.



b. Implementation

To implement in Verilog we need a module with two inputs(A,B) and two outputs (Y,C) which is sum and the carry out.

```
`include "prj_definition.v"

module HALF_ADDER(Y,C,A,B);
output Y,C;
input A,B;

// TBD
xor inst1(Y,A,B);
and inst2(C,A,B);

endmodule;
```

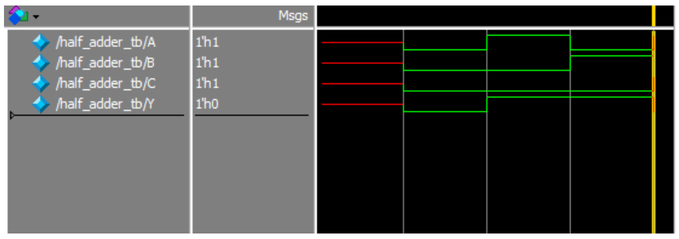
To implement we simply XOR A and B for the Y, and we AND A and B for the carry out.

c. Testing

In order to test our half adder we need to test each possible value for A and B, 00, 01, 11, 10. Below is the test bench for our half adder.

```
`timescale 1ns/1ps
module half_adder_tb;
reg A, B;
wire Y,C;

HALF_ADDER hs_inst_1(.Y(Y), .C(C),.A(A), .B(B));

initial
begin
#5 A=0; B=0;
#5 A=1; B=0;
#5 A=0; B=1;
#5 A=1; B=1;
end

endmodule
```

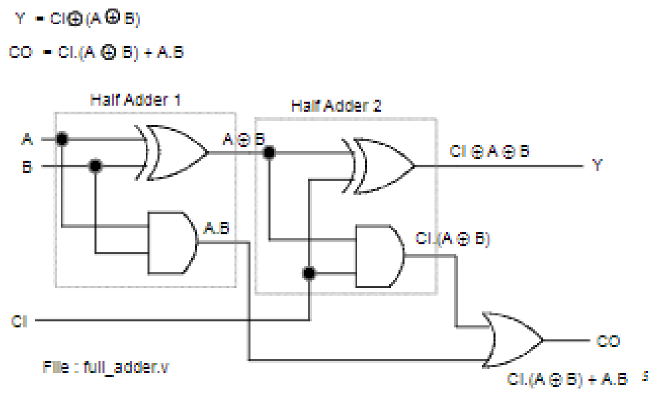
The wave for the test is below. We can see that when A=1 and B=1, Y is equal to 0 while our C value is equal to 1. And Y is equal to 1 only when A or B is 1 and 0 otherwise. Therefore proving to us that the half adder is working correctly.



2. Full Adder

a. Design

The full adder is simply two half adders combined with a Carry in value to create the full adder. It takes inputs A,B, and CI which is the carry in value. And outputs Y and CO which is the carry out bit. Its logic circuit is as follows.



b. Implementation

In order to implement our full adder we can re-use our half adder module. Then combine the Carry out values for both half adders to get our new CO value.

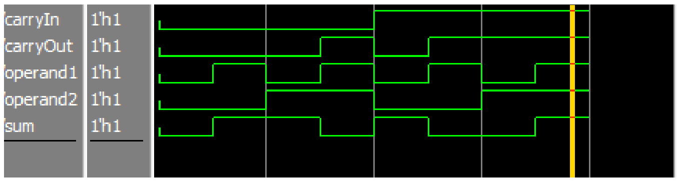
```
`include "prj_definition.v"

module FULL_ADDER(S,CO,A,B, CI);
output S,CO;
input A,B, CI;
wire aXorB, aAndB, CIandAXorB;
//TBD

HALF_ADDER h1(aXorB,aAndB, A,B);
HALF_ADDER h2(S,CIandAXorB,aXorB,CI);

or orI(CO, aAndB, CIandAXorB);
endmodule;
```

c. Testing

To test the full adder we need to try every input value for A, B, and CI. We can then observe the wave to check if our sum and CO values are correct.

```
`timescale 1ns/1ps
module a_full_adder_tb;
        reg A, B, CI;
        wire Y, CO;
        FULL_ADDER f(Y,CO, A,B,CI);
        initial begin
                A=0; B=0; CI=0;
                #5 A=1; B=0; CI=0;
                #5 A=0; B=1; CI=0;
                #5 A=1; B=1; CI=0;
                #5 A=0; B=0; CI=1;
                #5 A=1; B=0; CI=1;
                #5 A=0; B=1; CI=1;
                #5 A=1; B=1; CI=1;
                #5;
        end
endmodule
```

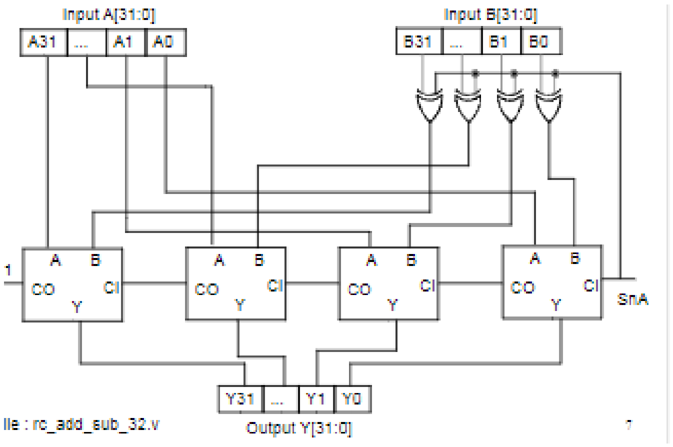
In our wave we can see that the full adder is working correctly for all possible input values. When all inputs are 1 we can see that not only is Y =1 but so is the CO value.



3. *32-bit binary Ripple Carry Adder/Subtractor*

a. Design

The 32 bit binary ripple carry Adder/Subtractor is designed to support 32 bit subtraction and addition. It has an SnA input that controls when addition or subtraction is to take place. It has 32 different full adders for each output bit. Each full adder takes in its respective bit from A and B and CI from the previous full adder.



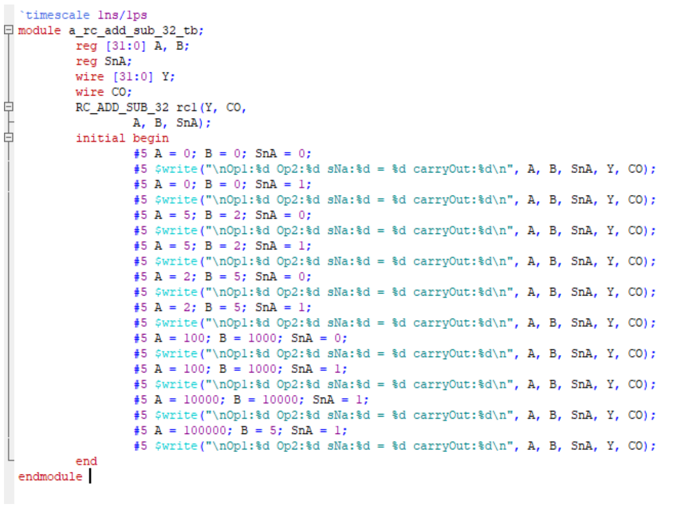
b. Implementation

In order to implement this component we need to loop 32 times to create 32 full adders that will compute each output bit. However the first and last adder we need to change certain outputs and inputs. The first adder will take SnA as its CI while the rest will take the previous adders CO as its CI. The last adder will output its CO as the final CO bit in our solution. We also need to xor each B bit with SnA for our subtraction operations.

```
module RC_ADD_SUB_32(Y, CO, A, B, SnA);
// output list
output [`DATA_INDEX_LIMIT:0] Y;
output CO;
// input list
input [`DATA_INDEX_LIMIT:0] A;
input [`DATA_INDEX_LIMIT:0] B;
input SnA;

wire [31:0] xorResult,CI;
// TBD
genvar i;
generate
        for(i = 0; i<32; i=i+1) begin: rc_add_sub_32_loop
        xor xorInst(xorResult[i], SnA, B[i]);
        if(i != 0 && i !=31)
                begin
                        FULL_ADDER f(Y[i], CI[i], A[i], xorResult[i], CI[i-1]);
                end
        else if(i==0)
                begin
                        FULL_ADDER f(Y[i], CI[i], A[i], xorResult[i], SnA);
                end
        else if(i==31)
                begin
                        FULL_ADDER f(Y[i], CO, A[i], xorResult[i], CI[i-1]);
                end
end
endgenerate

endmodule
```

c. Testing

Testing for this component is relatively the same for the previous two. We simply input some numbers to be added and subtracted and check to make sure the outputs are correct.

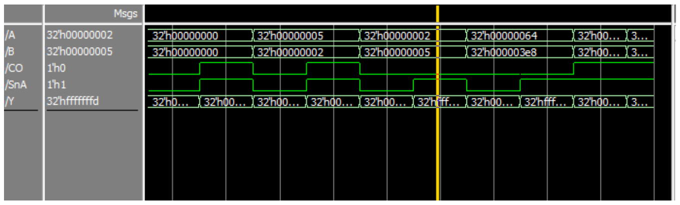


```
`timescale 1ns/1ps
module a_rc_add_sub_32_tb;
        reg [31:0] A, B;
        reg SnA;
        wire [31:0] Y;
        wire CO;
        RC_ADD_SUB_32 rc1(Y, CO,
                A, B, SnA);
        initial begin
                #5 A = 0; B = 0; SnA = 0;
                #5 $write("\nOp1:%d Op2:%d sNa:%d = %d carryOut:%d\n", A, B, SnA, Y, CO);
                #5 A = 0; B = 0; SnA = 1;
                #5 $write("\nOp1:%d Op2:%d sNa:%d = %d carryOut:%d\n", A, B, SnA, Y, CO);
                #5 A = 5; B = 2; SnA = 0;
                #5 $write("\nOp1:%d Op2:%d sNa:%d = %d carryOut:%d\n", A, B, SnA, Y, CO);
                #5 A = 5; B = 2; SnA = 1;
                #5 $write("\nOp1:%d Op2:%d sNa:%d = %d carryOut:%d\n", A, B, SnA, Y, CO);
                #5 A = 2; B = 5; SnA = 0;
                #5 $write("\nOp1:%d Op2:%d sNa:%d = %d carryOut:%d\n", A, B, SnA, Y, CO);
                #5 A = 2; B = 5; SnA = 1;
                #5 $write("\nOp1:%d Op2:%d sNa:%d = %d carryOut:%d\n", A, B, SnA, Y, CO);
                #5 A = 100; B = 1000; SnA = 0;
                #5 $write("\nOp1:%d Op2:%d sNa:%d = %d carryOut:%d\n", A, B, SnA, Y, CO);
                #5 A = 100; B = 1000; SnA = 1;
                #5 $write("\nOp1:%d Op2:%d sNa:%d = %d carryOut:%d\n", A, B, SnA, Y, CO);
                #5 A = 10000; B = 10000; SnA = 1;
                #5 $write("\nOp1:%d Op2:%d sNa:%d = %d carryOut:%d\n", A, B, SnA, Y, CO);
                #5 A = 100000; B = 5; SnA = 1;
                #5 $write("\nOp1:%d Op2:%d sNa:%d = %d carryOut:%d\n", A, B, SnA, Y, CO);
        end
endmodule
```

OutPut:

```
#
# Op1:          0 Op2:          0 sNa:0 =          0 carryOut:0
#
# Op1:          0 Op2:          0 sNa:1 =          0 carryOut:1
#
# Op1:          5 Op2:          2 sNa:0 =          7 carryOut:0
#
# Op1:          5 Op2:          2 sNa:1 =          3 carryOut:1
#
# Op1:          2 Op2:          5 sNa:0 =          7 carryOut:0
#
# Op1:          2 Op2:          5 sNa:1 = 4294967293 carryOut:0
#
# Op1:        100 Op2:       1000 sNa:0 =       1100 carryOut:0
#
# Op1:        100 Op2:       1000 sNa:1 = 4294966396 carryOut:0
#
# Op1:      10000 Op2:      10000 sNa:1 =          0 carryOut:1
#
# Op1:     100000 Op2:          5 sNa:1 =      99995 carryOut:1
```

Wave:



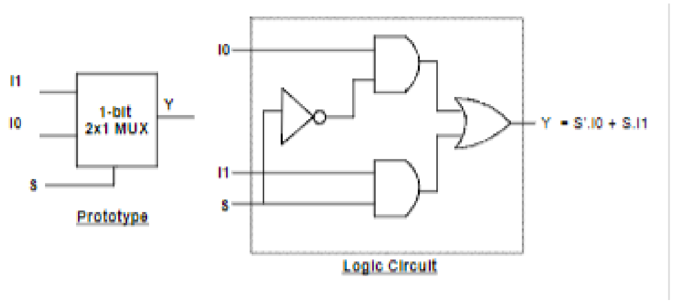
When we look at the OutPut we see that we are getting a really large number when we subtract a larger number from our first operand. This is because we are not accounting for twos compliment form. In our wave we can see that our Y value needs to be twos complimented to get the negative number. But other than that everything is working perfectly.

4. Multiplexer

a. Design

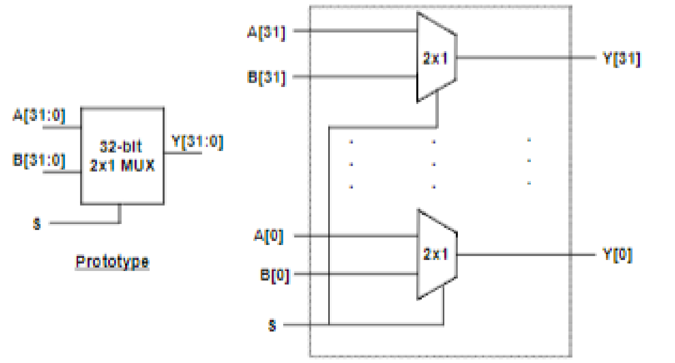
Our multiplexer component is supposed to output a specific input based on our control input. If the control is 0 we want our I0 input and vice versa in order to implement this we create our base 1-bit 2x1 MUX and then we can implement higher bit MUX with our base 2x1 MUX who's logic design is below.

1-bit 2x1 MUX:

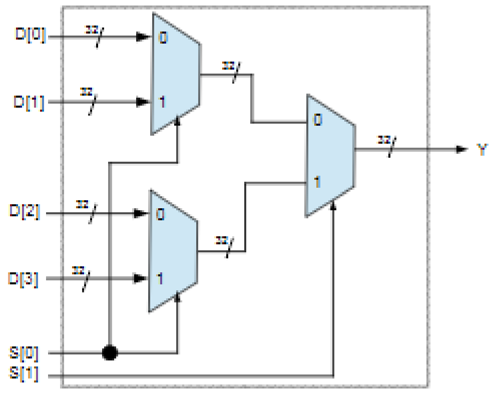


With that we can implement higher order MUX.

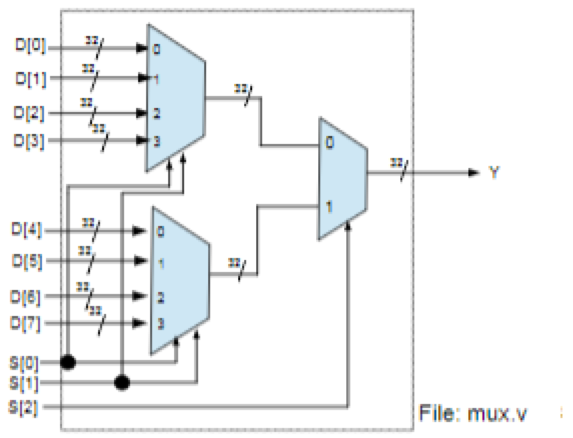
32-bit 2x1 MUX:



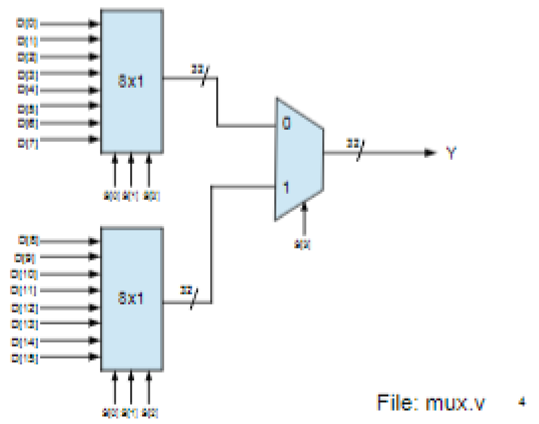
32-bit 4x1 MUX:



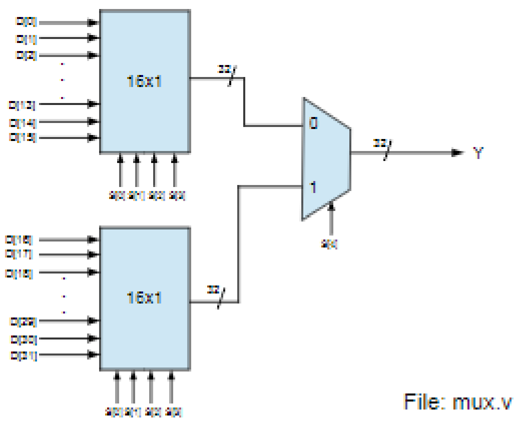
32-bit 8x1 MUX:



32-bit 16x1 MUX:



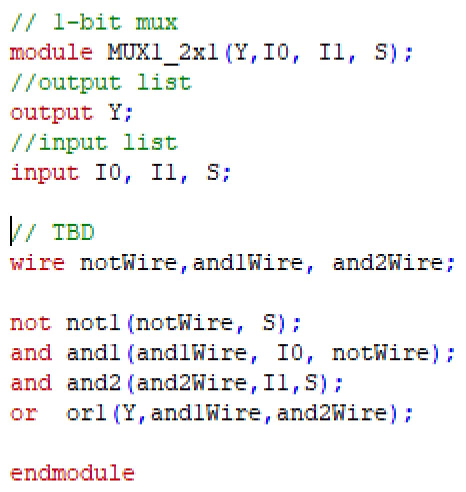
32-bit 32x1 MUX:



b. Implementation

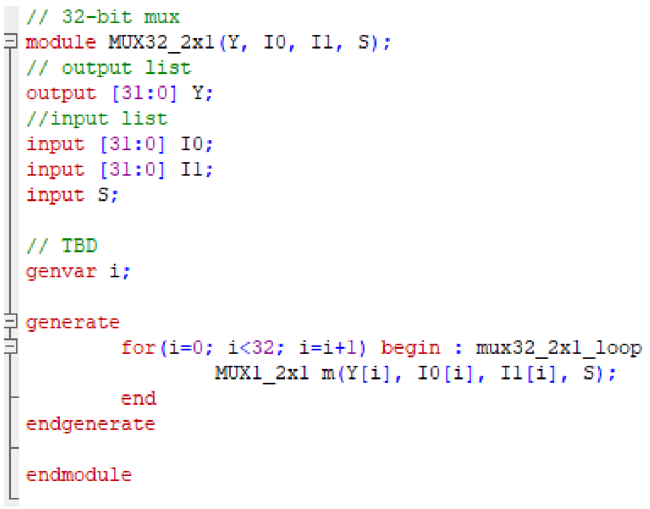
Each level of MUX simply uses the lower input MUX twice and then a 2x1 MUX to get the final result. So in each module we will have two MUX and then a MUX32_2x1 to select between the two outputs. The MUX32_2x1 will be implemented following the logic circuit in our design.

1-bit 2x1 MUX:



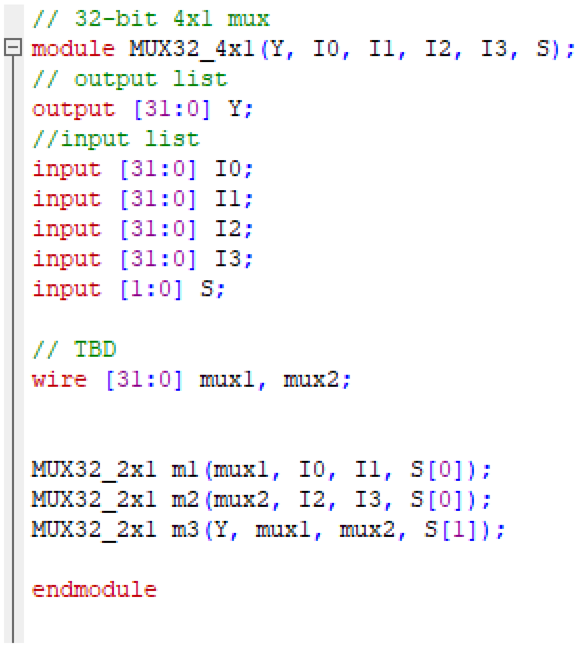
```
// 1-bit mux
module MUX1_2x1(Y,I0, I1, S);
//output list
output Y;
//input list
input I0, I1, S;

// TBD
wire notWire,and1Wire, and2Wire;

not not1(notWire, S);
and and1(and1Wire, I0, notWire);
and and2(and2Wire,I1,S);
or  or1(Y,and1Wire,and2Wire);

endmodule
```

32-bit 2x1 MUX:



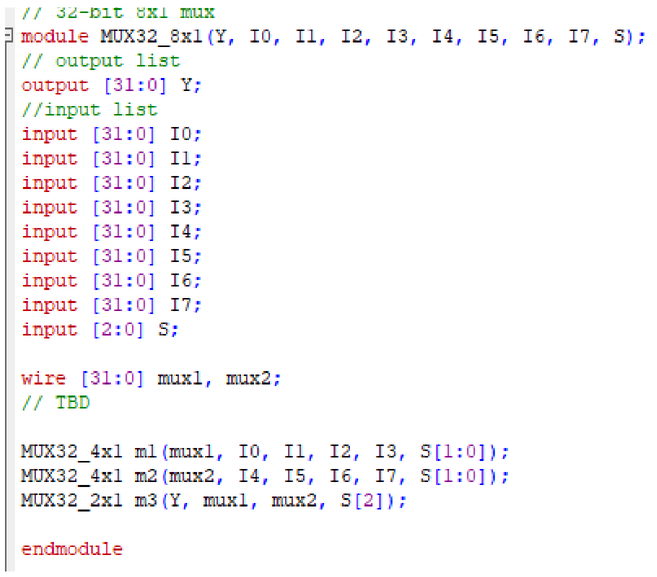
```
// 32-bit mux
module MUX32_2x1(Y, I0, I1, S);
// output list
output [31:0] Y;
//input list
input [31:0] I0;
input [31:0] I1;
input S;

// TBD
genvar i;

generate
        for(i=0; i<32; i=i+1) begin : mux32_2x1_loop
                MUX1_2x1 m(Y[i], I0[i], I1[i], S);
        end
endgenerate

endmodule
```

32-bit 4x1 MUX:



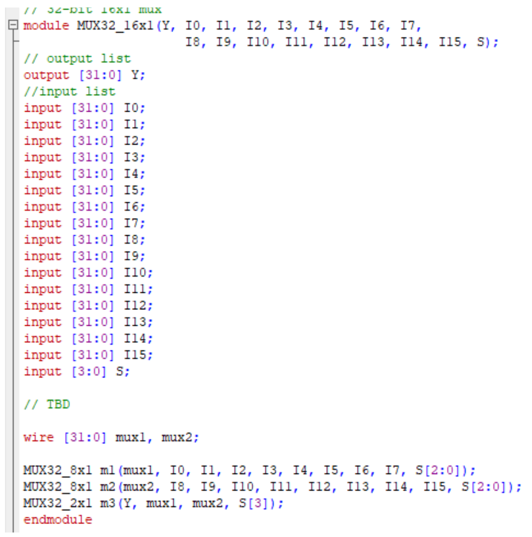
```
// 32-bit 4x1 mux
module MUX32_4x1(Y, I0, I1, I2, I3, S);
// output list
output [31:0] Y;
//input list
input [31:0] I0;
input [31:0] I1;
input [31:0] I2;
input [31:0] I3;
input [1:0] S;

// TBD
wire [31:0] mux1, mux2;


MUX32_2x1 m1(mux1, I0, I1, S[0]);
MUX32_2x1 m2(mux2, I2, I3, S[0]);
MUX32_2x1 m3(Y, mux1, mux2, S[1]);

endmodule
```

32-bit 8x1 MUX:



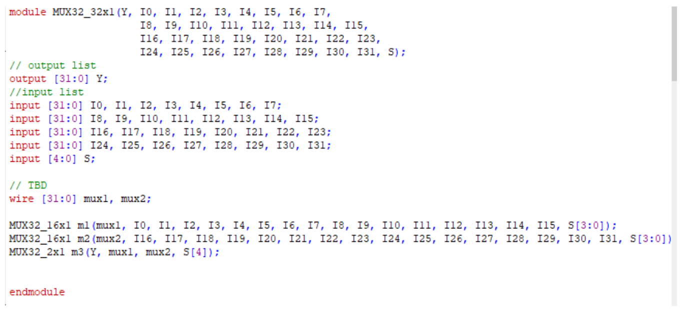
```
// 32-bit 8x1 mux
module MUX32_8x1(Y, I0, I1, I2, I3, I4, I5, I6, I7, S);
// output list
output [31:0] Y;
//input list
input [31:0] I0;
input [31:0] I1;
input [31:0] I2;
input [31:0] I3;
input [31:0] I4;
input [31:0] I5;
input [31:0] I6;
input [31:0] I7;
input [2:0] S;

wire [31:0] mux1, mux2;
// TBD

MUX32_4x1 m1(mux1, I0, I1, I2, I3, S[1:0]);
MUX32_4x1 m2(mux2, I4, I5, I6, I7, S[1:0]);
MUX32_2x1 m3(Y, mux1, mux2, S[2]);

endmodule
```

32-bit 16x1 MUX:



```
// 32-bit 16x1 mux
module MUX32_16x1(Y, I0, I1, I2, I3, I4, I5, I6, I7,
                     I8, I9, I10, I11, I12, I13, I14, I15, S);
// output list
output [31:0] Y;
//input list
input [31:0] I0;
input [31:0] I1;
input [31:0] I2;
input [31:0] I3;
input [31:0] I4;
input [31:0] I5;
input [31:0] I6;
input [31:0] I7;
input [31:0] I8;
input [31:0] I9;
input [31:0] I10;
input [31:0] I11;
input [31:0] I12;
input [31:0] I13;
input [31:0] I14;
input [31:0] I15;
input [3:0] S;

// TBD

wire [31:0] mux1, mux2;

MUX32_8x1 m1(mux1, I0, I1, I2, I3, I4, I5, I6, I7, S[2:0]);
MUX32_8x1 m2(mux2, I8, I9, I10, I11, I12, I13, I14, I15, S[2:0]);
MUX32_2x1 m3(Y, mux1, mux2, S[3]);
endmodule
```

32-bit 32x1 MUX:



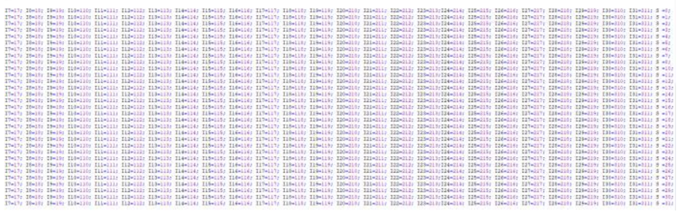
```
module MUX32_32x1(Y, I0, I1, I2, I3, I4, I5, I6, I7,
                     I8, I9, I10, I11, I12, I13, I14, I15,
                     I16, I17, I18, I19, I20, I21, I22, I23,
                     I24, I25, I26, I27, I28, I29, I30, I31, S);
// output list
output [31:0] Y;
//input list
input [31:0] I0, I1, I2, I3, I4, I5, I6, I7;
input [31:0] I8, I9, I10, I11, I12, I13, I14, I15;
input [31:0] I16, I17, I18, I19, I20, I21, I22, I23;
input [31:0] I24, I25, I26, I27, I28, I29, I30, I31;
input [4:0] S;

// TBD
wire [31:0] mux1, mux2;

MUX32_16x1 m1(mux1, I0, I1, I2, I3, I4, I5, I6, I7, I8, I9, I10, I11, I12, I13, I14, I15, S[3:0]);
MUX32_16x1 m2(mux2, I16, I17, I18, I19, I20, I21, I22, I23, I24, I25, I26, I27, I28, I29, I30, I31, S[3:0])
MUX32_2x1 m3(Y, mux1, mux2, S[4]);


endmodule
```

3. Testing

In order to test if the multiplexer works we only need to test the highest bit MUX which is our 32-bit 32x1 MUX. We only need to test this MUX because it essentially contains all other multiplexers in its code. If we test this and find it to be correct we can assume that all multiplexers are working correctly. In order to fully test this component we need to test each possible control signal. So we create a random data set for our inputs and then we try all 31 S values and check if we get the correct output in our Y value.

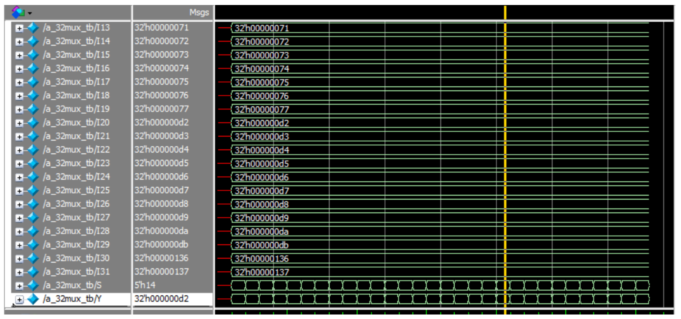
TestBench:

(Wasn't able to get a picture of the whole module because there are so many inputs, but you can see in this picture that S is incrementing by 1 in each iteration.)



[illegible]

Wave:



```
Msgs
/a_32mux_tb/I13   32'h00000071
/a_32mux_tb/I14   32'h00000072
/a_32mux_tb/I15   32'h00000073
/a_32mux_tb/I16   32'h00000074
/a_32mux_tb/I17   32'h00000075
/a_32mux_tb/I18   32'h00000076
/a_32mux_tb/I19   32'h00000077
/a_32mux_tb/I20   32'h000000d2
/a_32mux_tb/I21   32'h000000d3
/a_32mux_tb/I22   32'h000000d4
/a_32mux_tb/I23   32'h000000d5
/a_32mux_tb/I24   32'h000000d6
/a_32mux_tb/I25   32'h000000d7
/a_32mux_tb/I26   32'h000000d8
/a_32mux_tb/I27   32'h000000d9
/a_32mux_tb/I28   32'h000000da
/a_32mux_tb/I29   32'h000000db
/a_32mux_tb/I30   32'h00000136
/a_32mux_tb/I31   32'h00000137
/a_32mux_tb/S     5'h14
/a_32mux_tb/Y     32'h000000d2
```

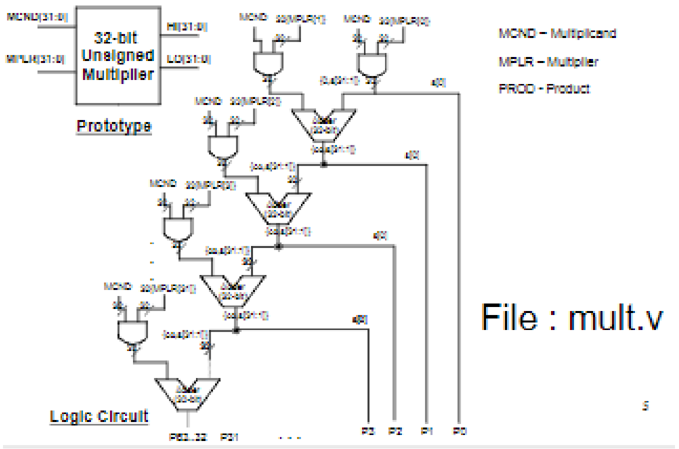
In our wave we can see that S = 5'h14 which is equivalent to 20 in decimal. So if we check our I20 input it has value 32'h000000d2. Now if we check our Y value we can see that it matches I20. For the rest of our wave the S control and Input values all correlate correctly to our output Y meaning that our multiplexers are working correctly.

5. Multiplier

a. Design

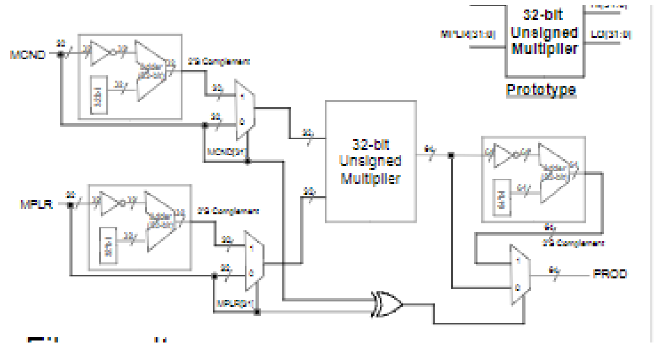
The multiplier is responsible for all multiplication operations. It takes in two operands multiplier and multiplicand and outputs HI and LO. An unsigned multiplier uses multiple ripple carry adders to compute the LO value bit by bit. The inputs for the adder come from an AND between our multiplicand and 32 bits of the Multiplier of a single repeating bit. And the second input is the result of the previous adder which is also the previous LO bit value.

32-bit Unsigned Multiplier:

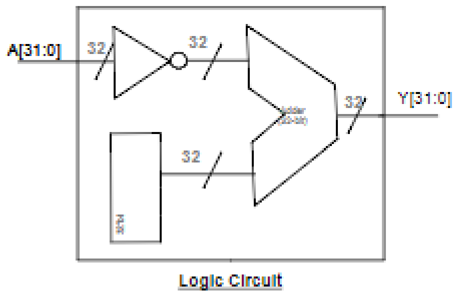


Our signed Multiplier uses an unsigned multiplier and three stages of twos compliment combined into a MUX64_2x1 multiplexer to get our answer. Our multiplier knows whether or not to use the twos compliment because of a multiplexer that will differentiate between a negative and positive number.

32-bit Signed Multiplier:



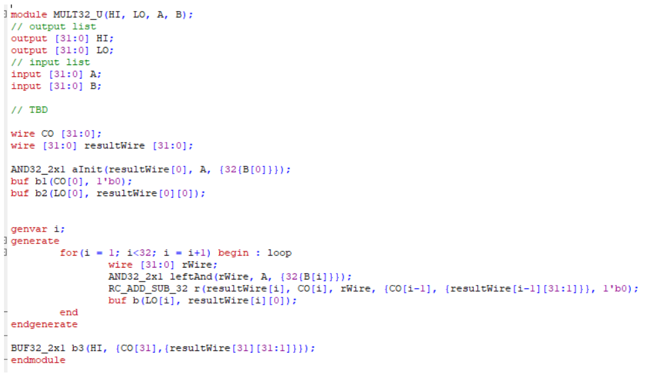
Twos Compliment:



b. Implementation

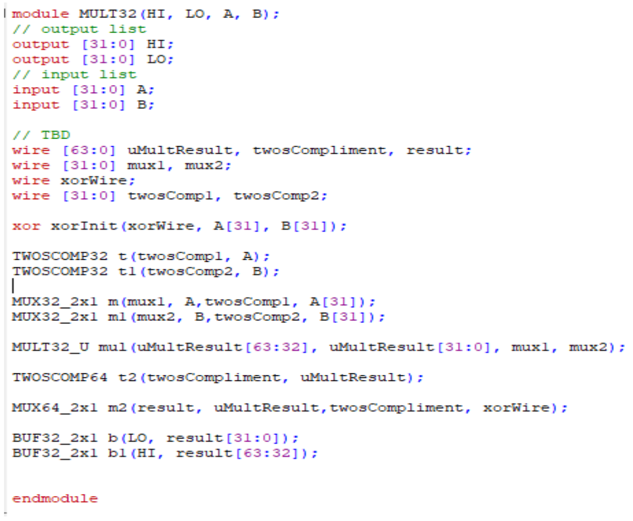
To implement our multiplier we simply follow our logic circuit for each module.

32-bit Unsigned Multiplier:



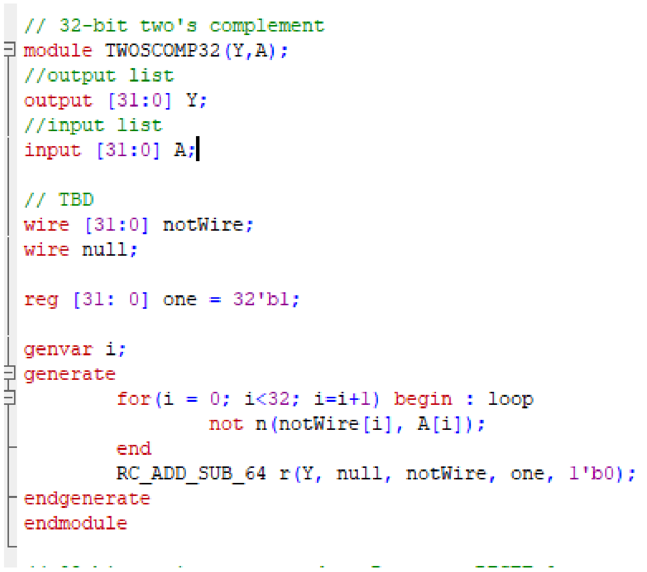
```
module MULT32_U(HI, LO, A, B);
// output list
output [31:0] HI;
output [31:0] LO;
// input list
input [31:0] A;
input [31:0] B;

// TBD

wire CO [31:0];
wire [31:0] resultWire [31:0];

AND32_2x1 aInit(resultWire[0], A, {32{B[0]}});
buf b1(CO[0], 1'b0);
buf b2(LO[0], resultWire[0][0]);


genvar i;
generate
        for(i = 1; i<32; i = i+1) begin : loop
                wire [31:0] rWire;
                AND32_2x1 leftAnd(rWire, A, {32{B[i]}});
                RC_ADD_SUB_32 r(resultWire[i], CO[i], rWire, {CO[i-1], {resultWire[i-1][31:1]}}, 1'b0);
                buf b(LO[i], resultWire[i][0]);
        end
endgenerate

BUF32_2x1 b3(HI, {CO[31],{resultWire[31][31:1]}});
endmodule
```

32-bit Signed Multiplier:



```
module MULT32(HI, LO, A, B);
// output list
output [31:0] HI;
output [31:0] LO;
// input list
input [31:0] A;
input [31:0] B;

// TBD
wire [63:0] uMultResult, twosCompliment, result;
wire [31:0] mux1, mux2;
wire xorWire;
wire [31:0] twosComp1, twosComp2;

xor xorInit(xorWire, A[31], B[31]);

TWOSCOMP32 t(twosComp1, A);
TWOSCOMP32 t1(twosComp2, B);

MUX32_2x1 m(mux1, A,twosComp1, A[31]);
MUX32_2x1 m1(mux2, B,twosComp2, B[31]);

MULT32_U mul(uMultResult[63:32], uMultResult[31:0], mux1, mux2);

TWOSCOMP64 t2(twosCompliment, uMultResult);

MUX64_2x1 m2(result, uMultResult,twosCompliment, xorWire);

BUF32_2x1 b(LO, result[31:0]);
BUF32_2x1 b1(HI, result[63:32]);


endmodule
```

Twos Compliment:



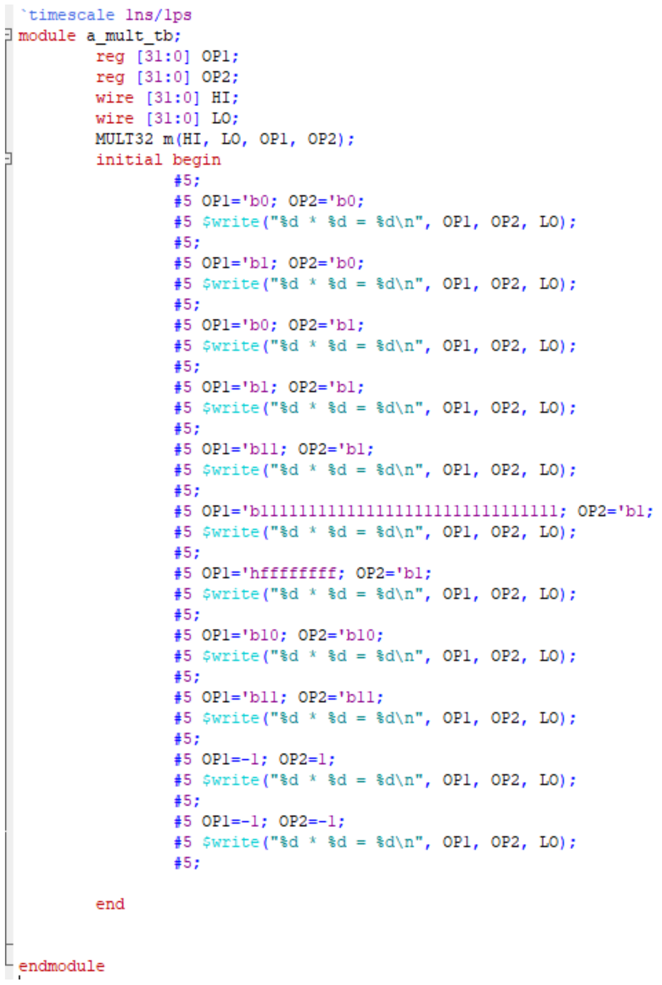
```
// 32-bit two's complement
module TWOSCOMP32(Y,A);
//output list
output [31:0] Y;
//input list
input [31:0] A;

// TBD
wire [31:0] notWire;
wire null;

reg [31: 0] one = 32'b1;

genvar i;
generate
        for(i = 0; i<32; i=i+1) begin : loop
                not n(notWire[i], A[i]);
        end
        RC_ADD_SUB_64 r(Y, null, notWire, one, 1'b0);
endgenerate
endmodule
```

c. Testing

To test our multiplier we use different inputs of both positive and negative to test whether or not we get the correct sum as well as the correct sign. We don't need to test each module as we can test just our signed multiplier because it contains all modules.

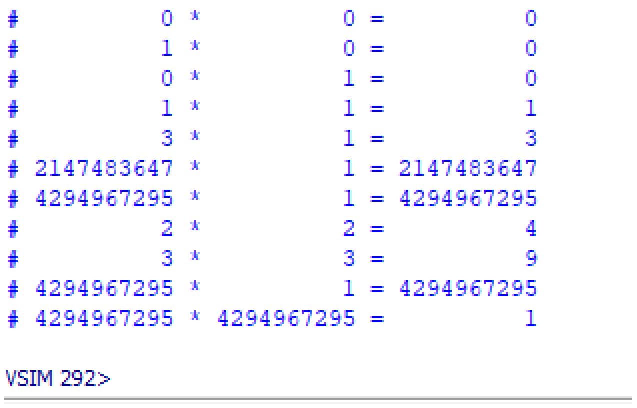
Test Bench:

Here we test multiple different inputs and write the out put.



```
`timescale 1ns/1ps
module a_mult_tb;
        reg [31:0] OP1;
        reg [31:0] OP2;
        wire [31:0] HI;
        wire [31:0] LO;
        MULT32 m(HI, LO, OP1, OP2);
        initial begin
                #5;
                #5 OP1='b0; OP2='b0;
                #5 $write("%d * %d = %d\n", OP1, OP2, LO);
                #5;
                #5 OP1='b1; OP2='b0;
                #5 $write("%d * %d = %d\n", OP1, OP2, LO);
                #5;
                #5 OP1='b0; OP2='b1;
                #5 $write("%d * %d = %d\n", OP1, OP2, LO);
                #5;
                #5 OP1='b1; OP2='b1;
                #5 $write("%d * %d = %d\n", OP1, OP2, LO);
                #5;
                #5 OP1='b11; OP2='b1;
                #5 $write("%d * %d = %d\n", OP1, OP2, LO);
                #5;
                #5 OP1='b11111111111111111111111111111111; OP2='b1;
                #5 $write("%d * %d = %d\n", OP1, OP2, LO);
                #5;
                #5 OP1='hffffffff; OP2='b1;
                #5 $write("%d * %d = %d\n", OP1, OP2, LO);
                #5;
                #5 OP1='b10; OP2='b10;
                #5 $write("%d * %d = %d\n", OP1, OP2, LO);
                #5;
                #5 OP1='b11; OP2='b11;
                #5 $write("%d * %d = %d\n", OP1, OP2, LO);
                #5;
                #5 OP1=-1; OP2=1;
                #5 $write("%d * %d = %d\n", OP1, OP2, LO);
                #5;
                #5 OP1=-1; OP2=-1;
                #5 $write("%d * %d = %d\n", OP1, OP2, LO);
                #5;

        end

endmodule
```

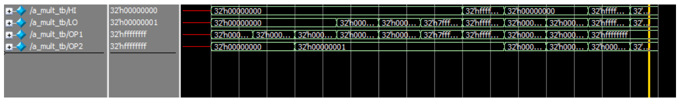
OutPut:



```
#          0 *          0 =          0
#          1 *          0 =          0
#          0 *          1 =          0
#          1 *          1 =          1
#          3 *          1 =          3
# 2147483647 *          1 = 2147483647
# 4294967295 *          1 = 4294967295
#          2 *          2 =          4
#          3 *          3 =          9
# 4294967295 *          1 = 4294967295
# 4294967295 * 4294967295 =          1

VSIM 292>
```

Here we can see that for positive numbers everything works according to plan but it gets a little messed up when we multiply negative numbers. However we know that the last test the operands are -1 and -1 and we can see that we get the correct answer of 1 meaning our multiplication is working correctly.

Wave:



In our wave we can see that the for negative numbers we get the correct LO and HI values but since they all have a 1 at the beginning of them our output puts a large number instead of the negative value. But we can get from our testing that everything is working correctly and no more work is needed on our multiplier.

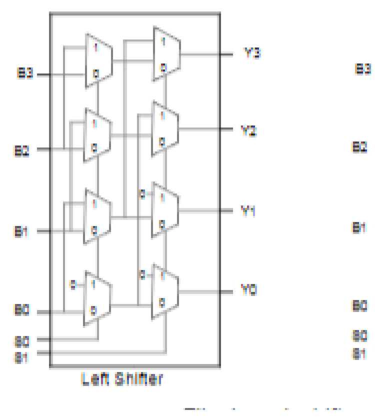
6. Barrel Shifter

a. Design

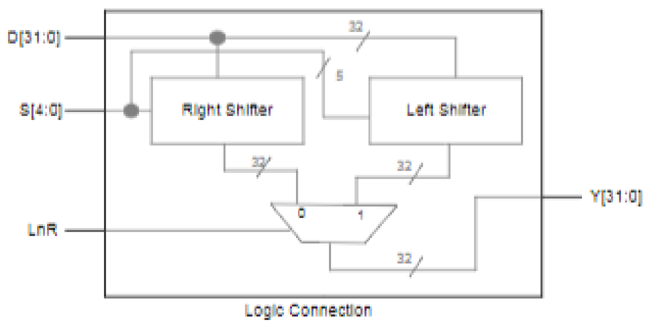
Our barrel shifter needs to be able to shift both left and right by a specified shift amount. In order to implement this we need a special input bit that will decide either left or right shifting, which we will call LnR. The overall design of our barrel shifter will have both a left and right shifter along with a multiplexer that will check if the shift amount is greater than 32 because then the amount will always be zero.

Left/Right Shifter:

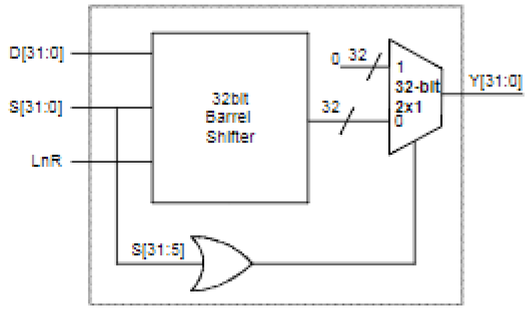
The only difference between this diagram and our shifter is that there will be five columns instead of 2 in order to be able to shift all 32 bits.



32-bit Barrel Shifter:



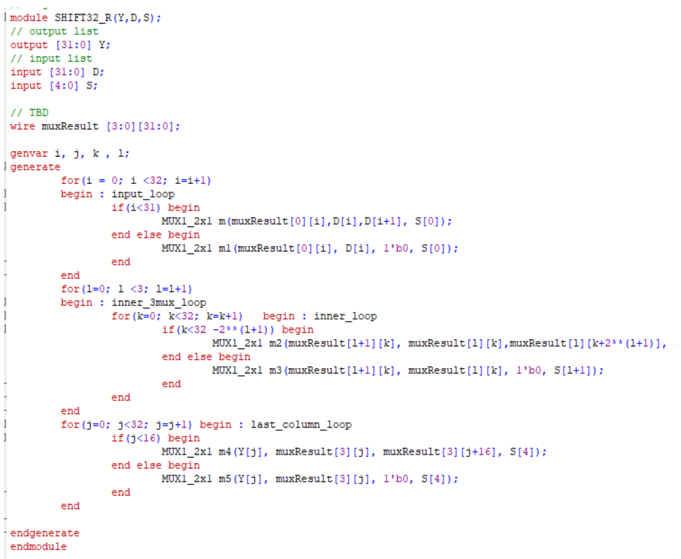
32-bit shamt shifter:



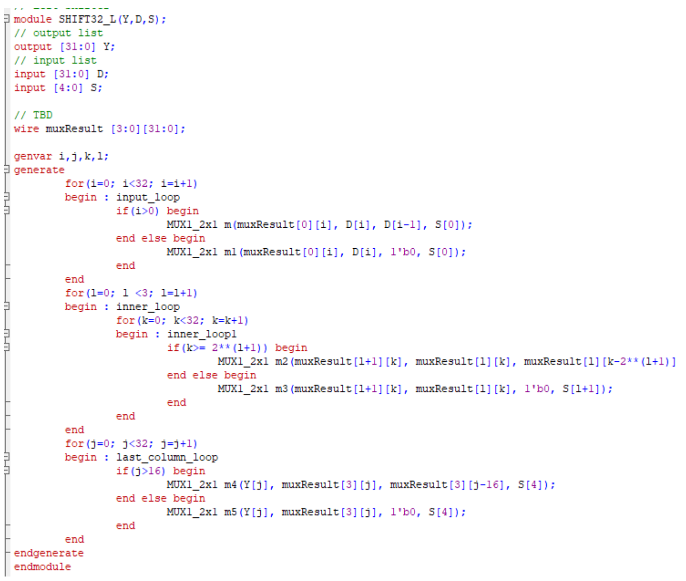
b. Implementation

Left/Right Shifter:

To implement our left right shifter we need 4 different for loops to create each column in our design. The first loop will only have one 0 shifted into the answer while the 3 middle loops will have 2^n 0's shifted into the answer. While the 5th loop will input 16 0's.



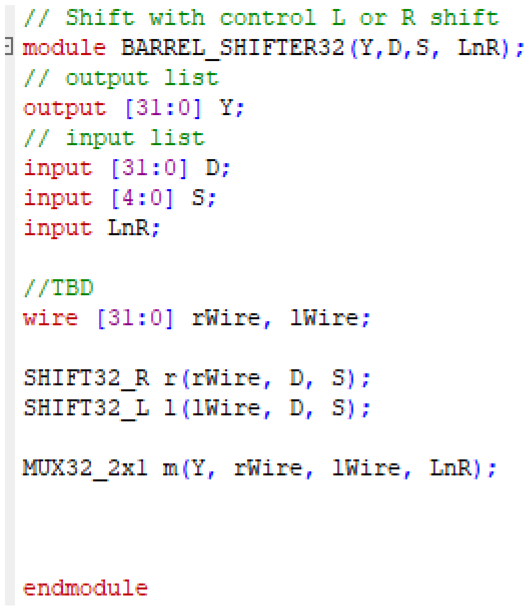
```
module SHIFT32_R(Y,D,S);
// output list
output [31:0] Y;
// input list
input [31:0] D;
input [4:0] S;

// TBD
wire muxResult [3:0][31:0];

genvar i, j, k , l;
generate
        for(i = 0; i <32; i=i+1)
        begin : input_loop
                if(i<31) begin
                        MUX1_2x1 m(muxResult[0][i],D[i],D[i+1], S[0]);
                end else begin
                        MUX1_2x1 m1(muxResult[0][i], D[i], 1'b0, S[0]);
                end
        end
        for(l=0; l <3; l=l+1)
        begin : inner_3mux_loop
                for(k=0; k<32; k=k+1)   begin : inner_loop
                        if(k<32 -2**(l+1)) begin
                                MUX1_2x1 m2(muxResult[l+1][k], muxResult[l][k],muxResult[l][k+2**(l+1)],
                        end else begin
                                MUX1_2x1 m3(muxResult[l+1][k], muxResult[l][k], 1'b0, S[l+1]);
                        end
                end
        end
        for(j=0; j<32; j=j+1) begin : last_column_loop
                if(j<16) begin
                        MUX1_2x1 m4(Y[j], muxResult[3][j], muxResult[3][j+16], S[4]);
                end else begin
                        MUX1_2x1 m5(Y[j], muxResult[3][j], 1'b0, S[4]);
                end
        end

endgenerate
endmodule
```



```
module SHIFT32_L(Y,D,S);
// output list
output [31:0] Y;
// input list
input [31:0] D;
input [4:0] S;

// TBD
wire muxResult [3:0][31:0];

genvar i,j,k,l;
generate
        for(i=0; i<32; i=i+1)
        begin : input_loop
                if(i>0) begin
                        MUX1_2x1 m(muxResult[0][i], D[i], D[i-1], S[0]);
                end else begin
                        MUX1_2x1 m1(muxResult[0][i], D[i], 1'b0, S[0]);
                end
        end
        for(l=0; l <3; l=l+1)
        begin : inner_loop
                for(k=0; k<32; k=k+1)
                begin : inner_loop1
                        if(k>= 2**(l+1)) begin
                                MUX1_2x1 m2(muxResult[l+1][k], muxResult[l][k], muxResult[l][k-2**(l+1)]
                        end else begin
                                MUX1_2x1 m3(muxResult[l+1][k], muxResult[l][k], 1'b0, S[l+1]);
                        end
                end
        end
        for(j=0; j<32; j=j+1)
        begin : last_column_loop
                if(j>16) begin
                        MUX1_2x1 m4(Y[j], muxResult[3][j], muxResult[3][j-16], S[4]);
                end else begin
                        MUX1_2x1 m5(Y[j], muxResult[3][j], 1'b0, S[4]);
                end
        end
endgenerate
endmodule
```

Barrel Shifter:

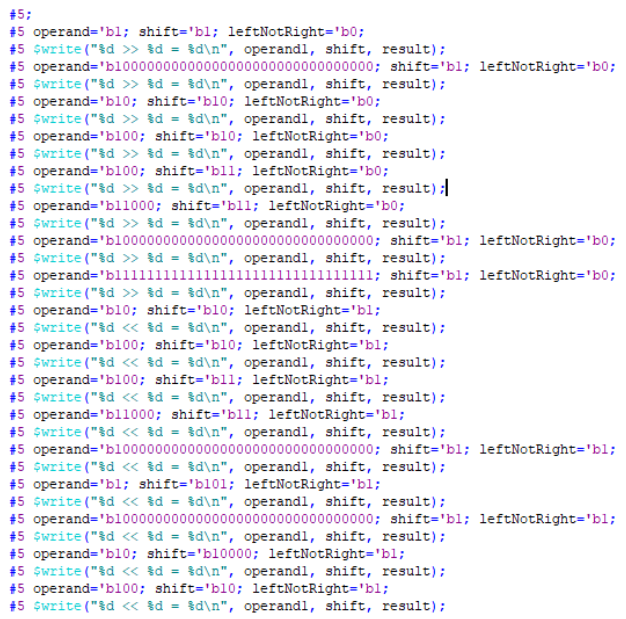
Our barrel shifter is simply combing our left and right shifters into one component.



```
// Shift with control L or R shift
module BARREL_SHIFTER32(Y,D,S, LnR);
// output list
output [31:0] Y;
// input list
input [31:0] D;
input [4:0] S;
input LnR;

//TBD
wire [31:0] rWire, lWire;

SHIFT32_R r(rWire, D, S);
SHIFT32_L l(lWire, D, S);

MUX32_2x1 m(Y, rWire, lWire, LnR);



endmodule
```

c. Testing

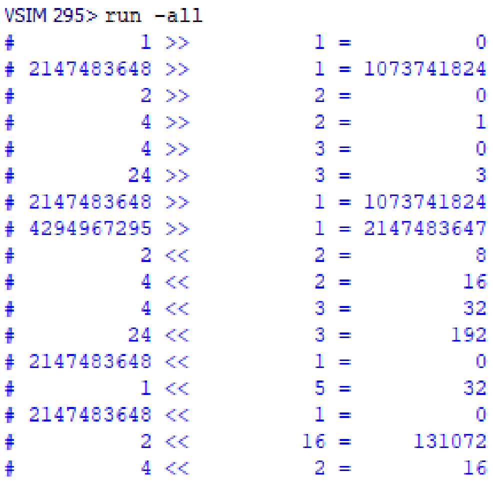
To test our barrel shifter I created multiple different scenarios with different inputs, shift amounts, and shift types. I also write the answer out to check if its correct as well as observing the wave.



```
#5;
#5 operand='b1; shift='b1; leftNotRight='b0;
#5 $write("%d >> %d = %d\n", operand1, shift, result);
#5 operand='b10000000000000000000000000000000; shift='b1; leftNotRight='b0;
#5 $write("%d >> %d = %d\n", operand1, shift, result);
#5 operand='b10; shift='b10; leftNotRight='b0;
#5 $write("%d >> %d = %d\n", operand1, shift, result);
#5 operand='b100; shift='b10; leftNotRight='b0;
#5 $write("%d >> %d = %d\n", operand1, shift, result);
#5 operand='b100; shift='b11; leftNotRight='b0;
#5 $write("%d >> %d = %d\n", operand1, shift, result);
#5 operand='b11000; shift='b11; leftNotRight='b0;
#5 $write("%d >> %d = %d\n", operand1, shift, result);
#5 operand='b10000000000000000000000000000000; shift='b1; leftNotRight='b0;
#5 $write("%d >> %d = %d\n", operand1, shift, result);
#5 operand='b11111111111111111111111111111111; shift='b1; leftNotRight='b0;
#5 $write("%d >> %d = %d\n", operand1, shift, result);
#5 operand='b10; shift='b10; leftNotRight='b1;
#5 $write("%d << %d = %d\n", operand1, shift, result);
#5 operand='b100; shift='b10; leftNotRight='b1;
#5 $write("%d << %d = %d\n", operand1, shift, result);
#5 operand='b100; shift='b11; leftNotRight='b1;
#5 $write("%d << %d = %d\n", operand1, shift, result);
#5 operand='b11000; shift='b11; leftNotRight='b1;
#5 $write("%d << %d = %d\n", operand1, shift, result);
#5 operand='b10000000000000000000000000000000; shift='b1; leftNotRight='b1;
#5 $write("%d << %d = %d\n", operand1, shift, result);
#5 operand='b1; shift='b101; leftNotRight='b1;
#5 $write("%d << %d = %d\n", operand1, shift, result);
#5 operand='b10000000000000000000000000000000; shift='b1; leftNotRight='b1;
#5 $write("%d << %d = %d\n", operand1, shift, result);
#5 operand='b10; shift='b10000; leftNotRight='b1;
#5 $write("%d << %d = %d\n", operand1, shift, result);
#5 operand='b100; shift='b10; leftNotRight='b1;
#5 $write("%d << %d = %d\n", operand1, shift, result);
```

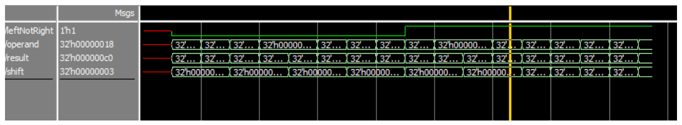
OutPut:

As we can see all outputs are correct and working



```
VSIM 295> run -all
#          1 >>          1 =          0
# 2147483648 >>          1 = 1073741824
#          2 >>          2 =          0
#          4 >>          2 =          1
#          4 >>          3 =          0
#         24 >>          3 =          3
# 2147483648 >>          1 = 1073741824
# 4294967295 >>          1 = 2147483647
#          2 <<          2 =          8
#          4 <<          2 =         16
#          4 <<          3 =         32
#         24 <<          3 =        192
# 2147483648 <<          1 =          0
#          1 <<          5 =         32
# 2147483648 <<          1 =          0
#          2 <<         16 =     131072
#          4 <<          2 =         16
```

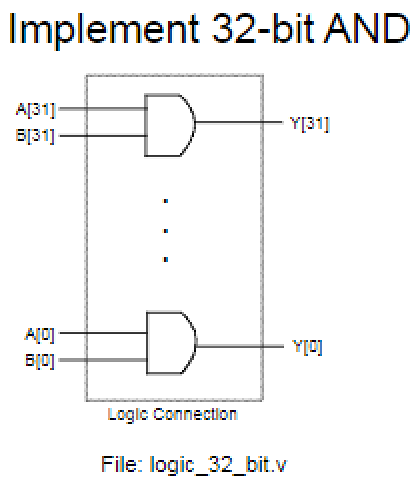
Wave:



7. AND, OR, NOR

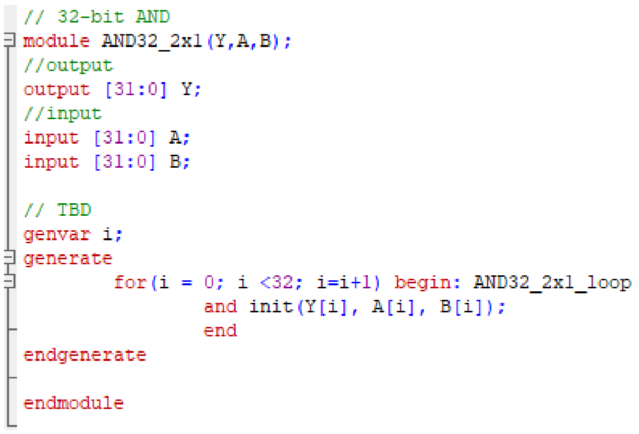
a. Design

We need to create a module that will perform a logic gate on a 32 bit input. Each module will be created using the same concept displayed in the diagram below except each with their respective logic gate.



b. Implementation

To implement this we simply create a for loop that will have 32 iterations that will perform the logic gate on each bit of input.



```
// 32-bit AND
module AND32_2x1(Y,A,B);
//output
output [31:0] Y;
//input
input [31:0] A;
input [31:0] B;

// TBD
genvar i;
generate
        for(i = 0; i <32; i=i+1) begin: AND32_2x1_loop
                and init(Y[i], A[i], B[i]);
                end
endgenerate

endmodule
```

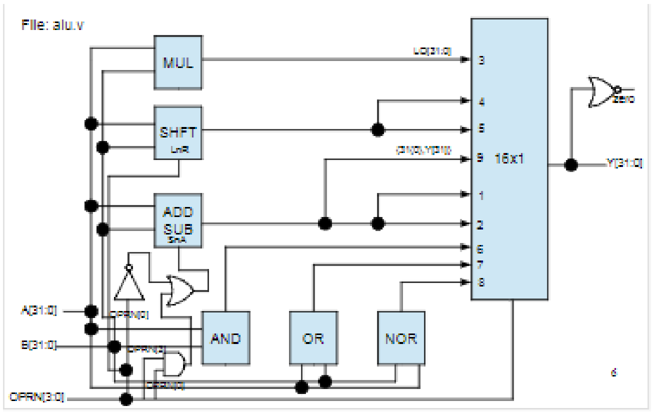
c. Testing

We test this in our ALU testing which will call each of these modules.

8. ALU

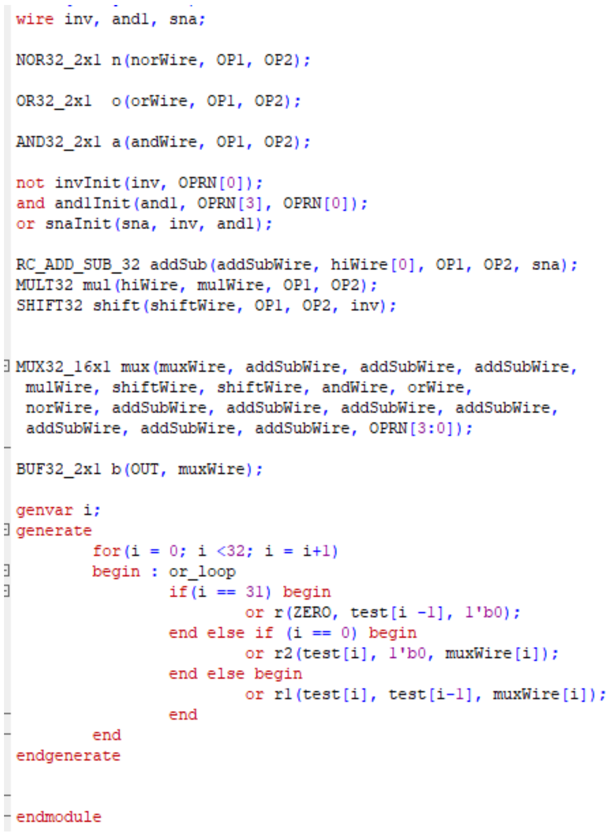
a. Design

ALU will take in 3 inputs two 32 bit operators A and B and a 4-bit operand which will be used to determine what operation we want to take. The operand will be sent to a multiplexer which will determine which operation we set equal to Y and our zero as well.



b. Implementation

To implement our ALU we create each module with the correct Inputs including the separate OR and AND. then we send them to our multiplexer in the correct slots.

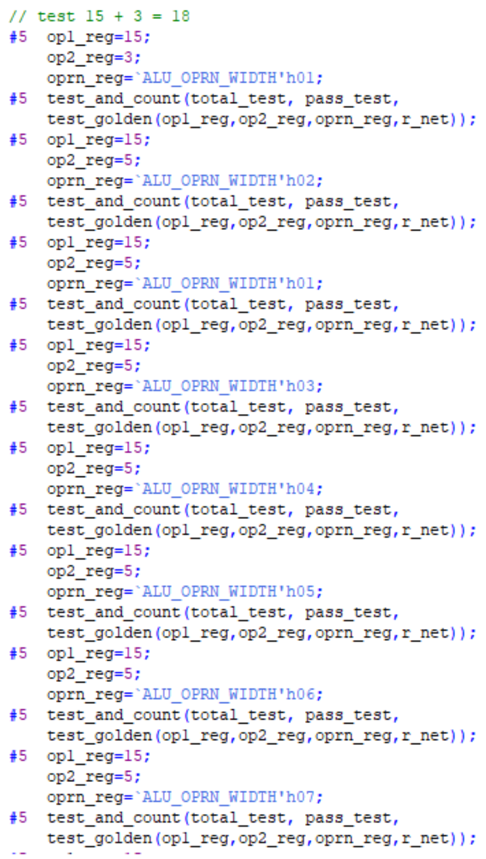


```
wire inv, andl, sna;

NOR32_2x1 n(norWire, OP1, OP2);

OR32_2x1  o(orWire, OP1, OP2);

AND32_2x1 a(andWire, OP1, OP2);

not invInit(inv, OPRN[0]);
and andlInit(andl, OPRN[3], OPRN[0]);
or snaInit(sna, inv, andl);

RC_ADD_SUB_32 addSub(addSubWire, hiWire[0], OP1, OP2, sna);
MULT32 mul(hiWire, mulWire, OP1, OP2);
SHIFT32 shift(shiftWire, OP1, OP2, inv);


MUX32_16x1 mux(muxWire, addSubWire, addSubWire, addSubWire,
 mulWire, shiftWire, shiftWire, andWire, orWire,
 norWire, addSubWire, addSubWire, addSubWire, addSubWire,
 addSubWire, addSubWire, addSubWire, OPRN[3:0]);

BUF32_2x1 b(OUT, muxWire);

genvar i;
generate
        for(i = 0; i <32; i = i+1)
        begin : or_loop
                if(i == 31) begin
                        or r(ZERO, test[i -1], 1'b0);
                end else if (i == 0) begin
                        or r2(test[i], 1'b0, muxWire[i]);
                end else begin
                        or r1(test[i], test[i-1], muxWire[i]);
                end
        end
endgenerate

endmodule
```

c. Testing

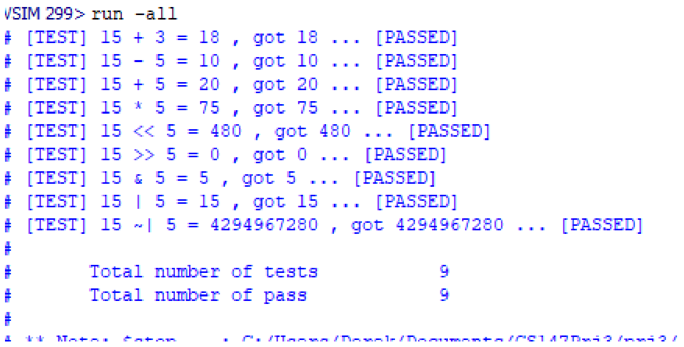
To test the ALU we need to test the different operands in our instruction set. We don't need a lot of different operators because we have already tested our individual modules we only need to test the ALU and its ability to execute the correct instruction.

TB:



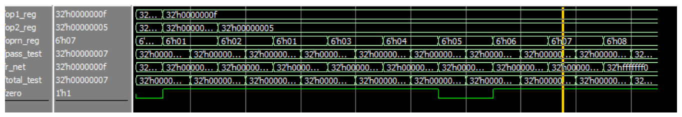
```
// test 15 + 3 = 18
#5  op1_reg=15;
    op2_reg=3;
    oprn_reg=`ALU_OPRN_WIDTH'h01;
#5  test_and_count(total_test, pass_test,
    test_golden(op1_reg,op2_reg,oprn_reg,r_net));
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h02;
#5  test_and_count(total_test, pass_test,
    test_golden(op1_reg,op2_reg,oprn_reg,r_net));
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h01;
#5  test_and_count(total_test, pass_test,
    test_golden(op1_reg,op2_reg,oprn_reg,r_net));
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h03;
#5  test_and_count(total_test, pass_test,
    test_golden(op1_reg,op2_reg,oprn_reg,r_net));
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h04;
#5  test_and_count(total_test, pass_test,
    test_golden(op1_reg,op2_reg,oprn_reg,r_net));
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h05;
#5  test_and_count(total_test, pass_test,
    test_golden(op1_reg,op2_reg,oprn_reg,r_net));
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h06;
#5  test_and_count(total_test, pass_test,
    test_golden(op1_reg,op2_reg,oprn_reg,r_net));
#5  op1_reg=15;
    op2_reg=5;
    oprn_reg=`ALU_OPRN_WIDTH'h07;
#5  test_and_count(total_test, pass_test,
    test_golden(op1_reg,op2_reg,oprn_reg,r_net));
```

OutPut:



```
VSIM 299> run -all
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10 ... [PASSED]
# [TEST] 15 + 5 = 20 , got 20 ... [PASSED]
# [TEST] 15 * 5 = 75 , got 75 ... [PASSED]
# [TEST] 15 << 5 = 480 , got 480 ... [PASSED]
# [TEST] 15 >> 5 = 0 , got 0 ... [PASSED]
# [TEST] 15 & 5 = 5 , got 5 ... [PASSED]
# [TEST] 15 | 5 = 15 , got 15 ... [PASSED]
# [TEST] 15 ~| 5 = 4294967280 , got 4294967280 ... [PASSED]
#
#       Total number of tests          9
#       Total number of pass           9
#
# ** Note: $stop    : C:/Users/Derek/Documents/CS147Proj3/proj3/
```

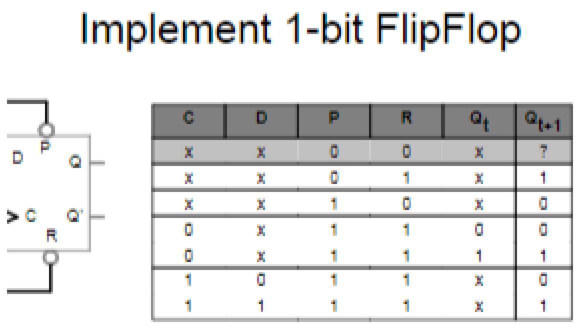
Wave:



9. Flip Flop

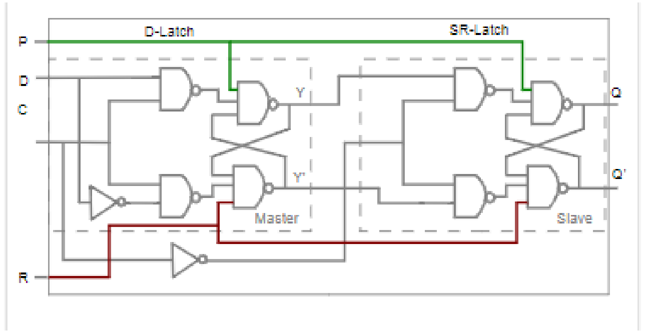
a. Design

Flip Flop is created by combing first a D-latch and then an SR-Latch along with a preset signal and reset signal. However we need to implement a 3 input AND instead of the regular 2 input AND.



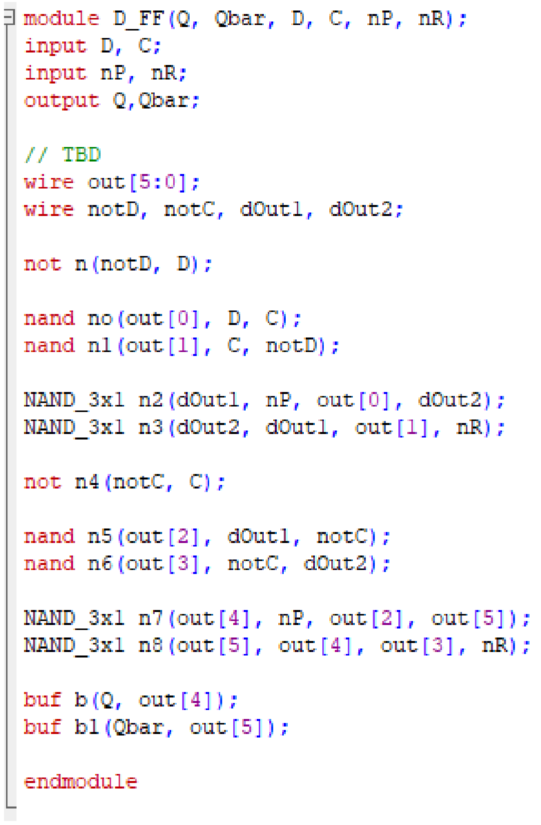
Implement 1-bit FlipFlop

C	D	P	R	Q_t	Q_{t+1}
x	x	0	0	x	?
x	x	0	1	x	1
x	x	1	0	x	0
0	x	1	1	0	0
0	x	1	1	1	1
1	0	1	1	x	0
1	1	1	1	x	1



b. Implementation

Instead of using our SR-Latch and D-latch models I decided to implement my own Flip Flop without using the pre existing modules. So I manually connected all in the inputs to the correct gates.



```
module D_FF(Q, Qbar, D, C, nP, nR);
input D, C;
input nP, nR;
output Q,Qbar;

// TBD
wire out[5:0];
wire notD, notC, dOut1, dOut2;

not n(notD, D);

nand no(out[0], D, C);
nand n1(out[1], C, notD);

NAND_3x1 n2(dOut1, nP, out[0], dOut2);
NAND_3x1 n3(dOut2, dOut1, out[1], nR);

not n4(notC, C);

nand n5(out[2], dOut1, notC);
nand n6(out[3], notC, dOut2);

NAND_3x1 n7(out[4], nP, out[2], out[5]);
NAND_3x1 n8(out[5], out[4], out[3], nR);

buf b(Q, out[4]);
buf b1(Qbar, out[5]);

endmodule
```

c. Testing

To test the flip flop we change the input values and check to see we get the correct output values according to the chart in our design section.

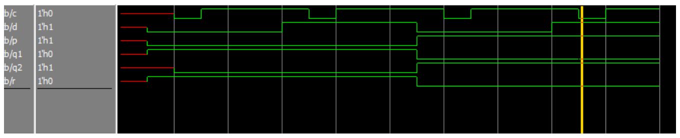
TB:

```
module a_bitFlipFlop_tb;
        reg p, d, c, r;
        wire q1, q2;
        D_FF b(q1, q2, d, c, p, r);
        initial begin

                #5 d='b0; p='b0; r='b1;
                #5 c='b0;
                #5 c='b1;
                #5 golden(q1, 'b1, c, d, p, r);
                #5;
                #5 d='b1; p='b0; r='b1;
                #5 c='b0;
                #5 c='b1;
                #5 golden(q1, 'b1, c, d, p, r);
                #5;
                #5 d='b0; p='b1; r='b0;
                #5 c='b0;
                #5 c='b1;
                #5 golden(q1, 'b0, c, d, p, r);
                #5;
                #5 d='b1; p='b1; r='b0;
                #5 c='b0;
                #5 c='b1;
                #5 golden(q1, 'b0, c, d, p, r);
                #5;
        end
```

OutPut:

```
VSIM 303> run -all
# d:0 p:0 r:1= Exp-q:1 and Act-q:1[PASSED]
# Ran bitFlipFlop TB
# d:1 p:0 r:1= Exp-q:1 and Act-q:1[PASSED]
# Ran bitFlipFlop TB
# d:0 p:1 r:0= Exp-q:0 and Act-q:0[PASSED]
# Ran bitFlipFlop TB
# d:1 p:1 r:0= Exp-q:0 and Act-q:0[PASSED]
# Ran bitFlipFlop TB
```

Wave:

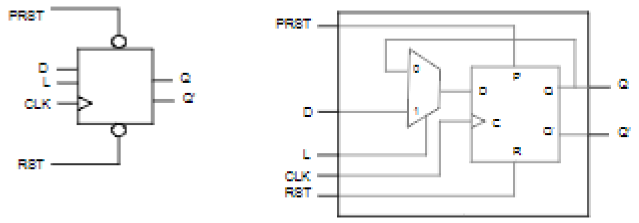


10. Register

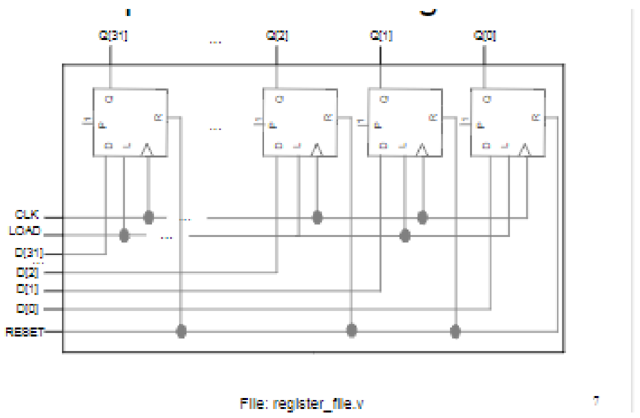
a. Design

For our 1 bit Register we use a multiplexer between our D input and the output of a Flip Flop. This is a sequential circuit dependent on the outcome of each other. Our Load value is what decides the D input of our flip flop. While 32-bit register is just 32 1-bit registers combined together

1-bit register:



32-bit register:



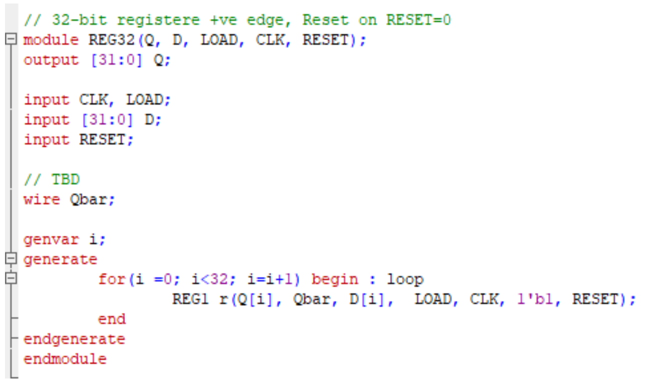
b. Implementation

To implement our one bit register we create a flip flop with the correct input values and a multiplexer that takes in D and Q and L as its control.

1bit register:

```
module REG1(Q, Qbar, D, L, C, nP, nR);
input D, C, L;
input nP, nR;
output Q,Qbar;

// TBD
wire muxResult, ffResult;

MUX1_2x1 m(muxResult, ffResult, D, L);

D_FF f(ffResult, Qbar, muxResult, C, nP, nR);

buf(Q, ffResult);

endmodule
```

32Bit register:

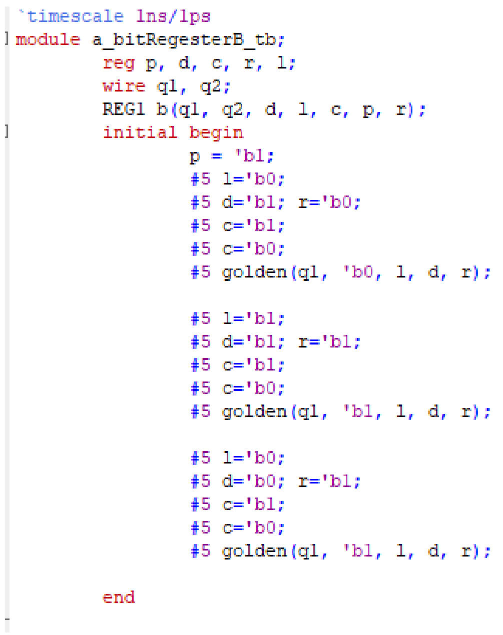


```
// 32-bit registere +ve edge, Reset on RESET=0
module REG32(Q, D, LOAD, CLK, RESET);
output [31:0] Q;

input CLK, LOAD;
input [31:0] D;
input RESET;

// TBD
wire Qbar;

genvar i;
generate
        for(i =0; i<32; i=i+1) begin : loop
                REG1 r(Q[i], Qbar, D[i],  LOAD, CLK, 1'b1, RESET);
        end
endgenerate
endmodule
```

c. Testing

We can test the register with different inputs and check against our chart to see that we get the correct output.

TB:

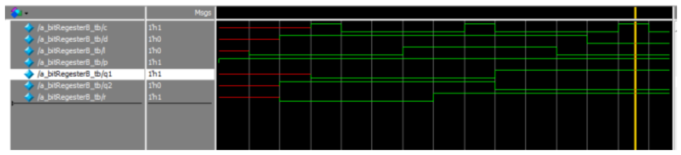


```
`timescale 1ns/1ps
module a_bitRegesterB_tb;
        reg p, d, c, r, l;
        wire q1, q2;
        REG1 b(q1, q2, d, l, c, p, r);
        initial begin
                p = 'b1;
                #5 l='b0;
                #5 d='b1; r='b0;
                #5 c='b1;
                #5 c='b0;
                #5 golden(q1, 'b0, l, d, r);

                #5 l='b1;
                #5 d='b1; r='b1;
                #5 c='b1;
                #5 c='b0;
                #5 golden(q1, 'b1, l, d, r);

                #5 l='b0;
                #5 d='b0; r='b1;
                #5 c='b1;
                #5 c='b0;
                #5 golden(q1, 'b1, l, d, r);

        end
```

OutPut:

```
VSIM 307> run -all
# l:0 d:1 r:0= Exp-q:0 and Act-q:0[PASSED]
# Ran bitReg TB
# l:1 d:1 r:1= Exp-q:1 and Act-q:1[PASSED]
# Ran bitReg TB
# l:0 d:1 r:1= Exp-q:1 and Act-q:1[PASSED]
# Ran bitReg TB
```

Wave:

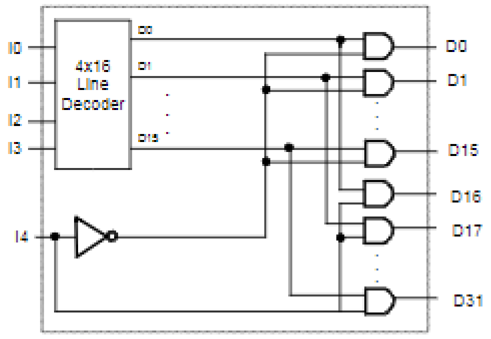


Msgs
/a_bitRegesterB_tb/c 1'h1
/a_bitRegesterB_tb/d 1'h0
/a_bitRegesterB_tb/l 1'h0
/a_bitRegesterB_tb/p 1'h1
/a_bitRegesterB_tb/q1 1'h1
/a_bitRegesterB_tb/q2 1'h0
/a_bitRegesterB_tb/r 1'h1

11. Line Decoder

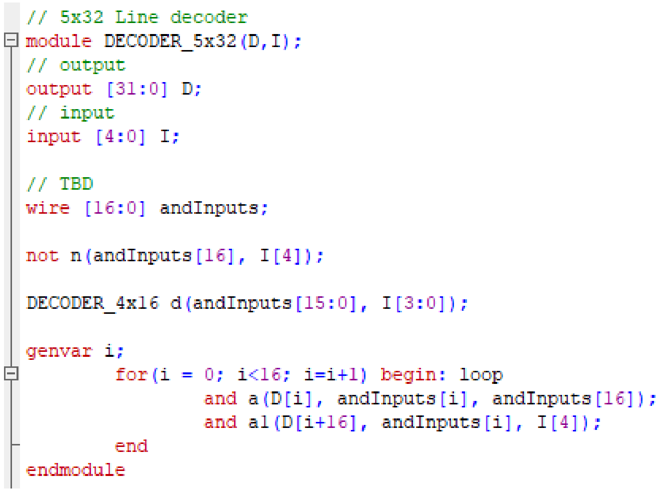
a. Design

Our line decoder will take in 5 inputs and output 32 bits. Its created by using lower level decoders and combining the output with an AND and the last input.



b. Implementation

We simply call our lower level decoder and use a for loop to AND all of the outputs.

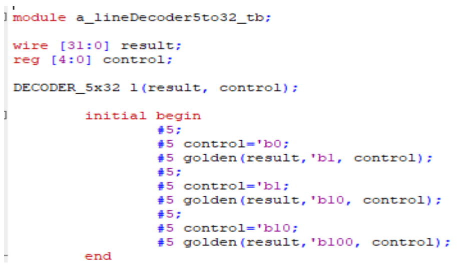


```
// 5x32 Line decoder
module DECODER_5x32(D,I);
// output
output [31:0] D;
// input
input [4:0] I;

// TBD
wire [16:0] andInputs;

not n(andInputs[16], I[4]);

DECODER_4x16 d(andInputs[15:0], I[3:0]);

genvar i;
        for(i = 0; i<16; i=i+1) begin: loop
                and a(D[i], andInputs[i], andInputs[16]);
                and a1(D[i+16], andInputs[i], I[4]);
        end
endmodule
```

c. Testing

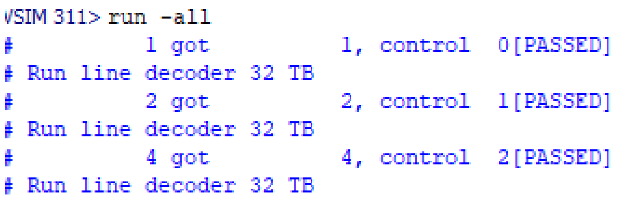
To test the line decoder all we need to test is the 5x32. If that works then we can assume that all the other decoders are implemented correctly.

TB:



```
module a_lineDecoder5to32_tb;

wire [31:0] result;
reg [4:0] control;

DECODER_5x32 l(result, control);

        initial begin
                #5;
                #5 control='b0;
                #5 golden(result,'b1, control);
                #5;
                #5 control='b1;
                #5 golden(result,'b10, control);
                #5;
                #5 control='b10;
                #5 golden(result,'b100, control);
        end
```

OutPut:



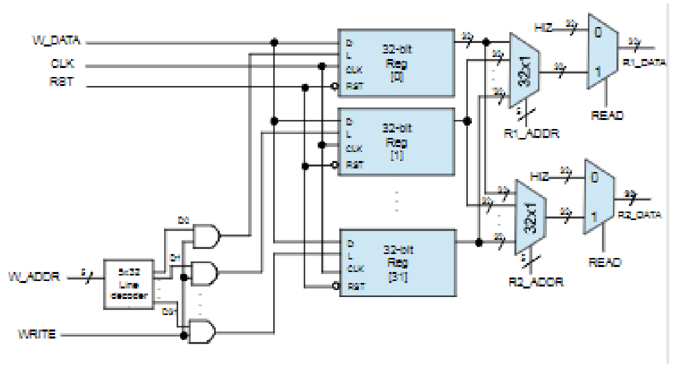
```
VSIM 311> run -all
#            1 got           1, control   0[PASSED]
# Run line decoder 32 TB
#            2 got           2, control   1[PASSED]
# Run line decoder 32 TB
#            4 got           4, control   2[PASSED]
# Run line decoder 32 TB
```

Wave:

12. *Register File*

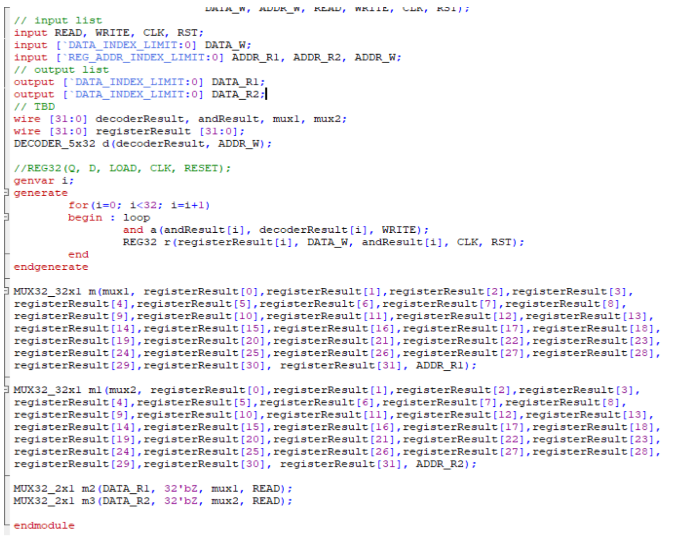
a. Design

Register File is 32 32-bit registers combined together with 4 multiplexers to decide which data is the correct register.



b. Implementation

I used a for loop to create each register along with a line decoder and AND gate with the write signal to create our L value.

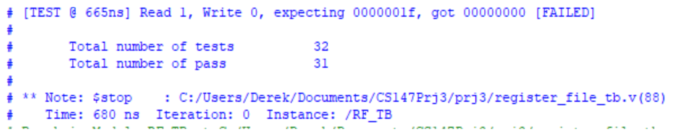


```
                          DATA_W, ADDR_W, READ, WRITE, CLK, RST);
// input list
input READ, WRITE, CLK, RST;
input [`DATA_INDEX_LIMIT:0] DATA_W;
input [`REG_ADDR_INDEX_LIMIT:0] ADDR_R1, ADDR_R2, ADDR_W;
// output list
output [`DATA_INDEX_LIMIT:0] DATA_R1;
output [`DATA_INDEX_LIMIT:0] DATA_R2;
// TBD
wire [31:0] decoderResult, andResult, mux1, mux2;
wire [31:0] registerResult [31:0];
DECODER_5x32 d(decoderResult, ADDR_W);

//REG32(Q, D, LOAD, CLK, RESET);
genvar i;
generate
        for(i=0; i<32; i=i+1)
        begin : loop
                and a(andResult[i], decoderResult[i], WRITE);
                REG32 r(registerResult[i], DATA_W, andResult[i], CLK, RST);
        end
endgenerate

MUX32_32x1 m(mux1, registerResult[0],registerResult[1],registerResult[2],registerResult[3],
registerResult[4],registerResult[5],registerResult[6],registerResult[7],registerResult[8],
registerResult[9],registerResult[10],registerResult[11],registerResult[12],registerResult[13],
registerResult[14],registerResult[15],registerResult[16],registerResult[17],registerResult[18],
registerResult[19],registerResult[20],registerResult[21],registerResult[22],registerResult[23],
registerResult[24],registerResult[25],registerResult[26],registerResult[27],registerResult[28],
registerResult[29],registerResult[30], registerResult[31], ADDR_R1);

MUX32_32x1 m1(mux2, registerResult[0],registerResult[1],registerResult[2],registerResult[3],
registerResult[4],registerResult[5],registerResult[6],registerResult[7],registerResult[8],
registerResult[9],registerResult[10],registerResult[11],registerResult[12],registerResult[13],
registerResult[14],registerResult[15],registerResult[16],registerResult[17],registerResult[18],
registerResult[19],registerResult[20],registerResult[21],registerResult[22],registerResult[23],
registerResult[24],registerResult[25],registerResult[26],registerResult[27],registerResult[28],
registerResult[29],registerResult[30], registerResult[31], ADDR_R2);

MUX32_2x1 m2(DATA_R1, 32'bZ, mux1, READ);
MUX32_2x1 m3(DATA_R2, 32'bZ, mux2, READ);

endmodule
```

c. Testing

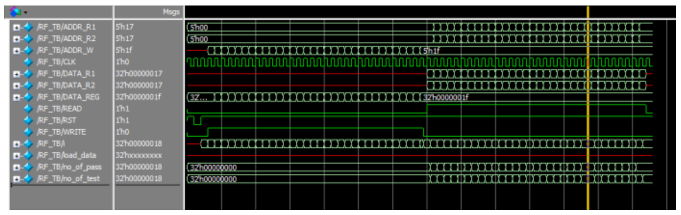
To test the register file we were given a test bench and currently it passes all of the test except for the last value. Everything is correct except that the value in the multiplexer is wrong and I cant figure out why only the 31 output is wrong.

OutPut:



```
# [TEST @ 665ns] Read 1, Write 0, expecting 0000001f, got 00000000 [FAILED]
#
#       Total number of tests          32
#       Total number of pass           31
#
# ** Note: $stop    : C:/Users/Derek/Documents/CS147Prj3/prj3/register_file_tb.v(88)
#    Time: 680 ns  Iteration: 0  Instance: /RF_TB
```

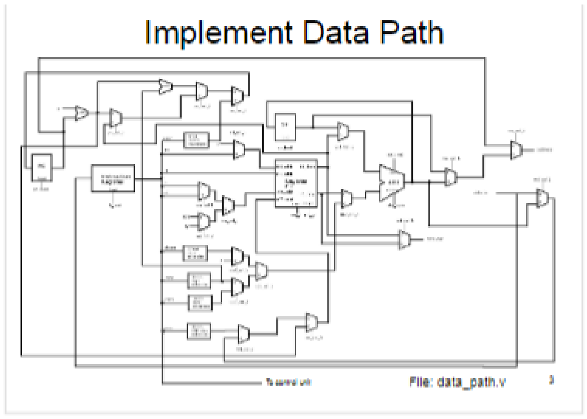
Wave:



13. *Data Path*

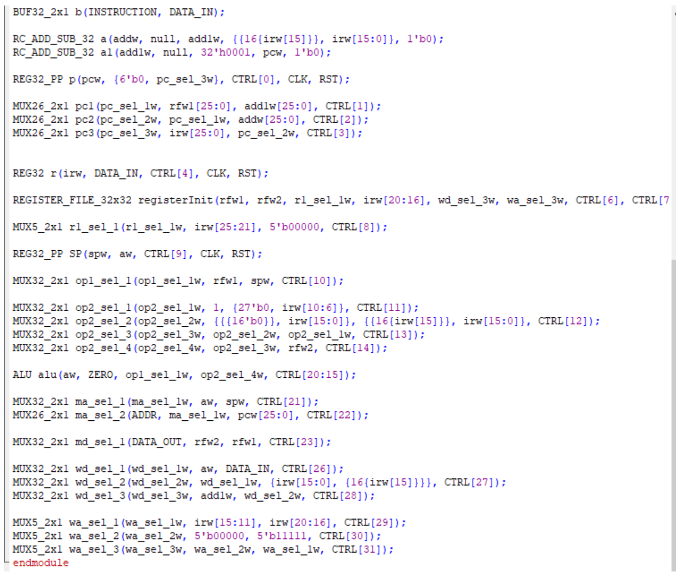
a. Design

The data path is created based on the diagram below. It is responsible for handling the flow of data in our system.



b. Implementation

We implement our data path using basically every component we previously created. It has multiplexers, ALU, and a register file. Then we simply connect all the paths together to the correct component. Here we also set up how our control signal is going to be used and what bits are going to be used for the different signals.

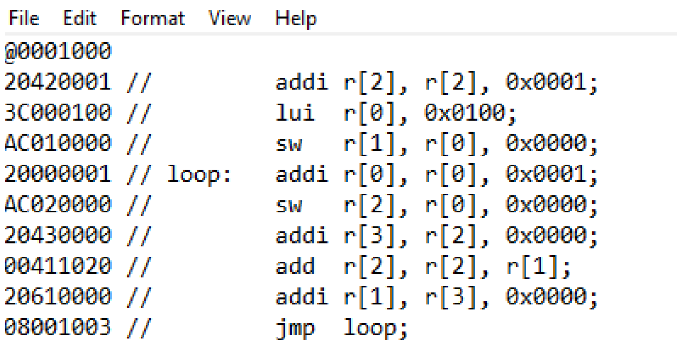


```
BUF32_2x1 b(INSTRUCTION, DATA_IN);

RC_ADD_SUB_32 a(addw, null, addlw, {{16{irw[15]}}, irw[15:0]}, 1'b0);
RC_ADD_SUB_32 al(addlw, null, 32'h0001, pcw, 1'b0);

REG32_PP p(pcw, {6'b0, pc_sel_3w}, CTRL[0], CLK, RST);

MUX26_2x1 pcl(pc_sel_lw, rfwl[25:0], addlw[25:0], CTRL[1]);
MUX26_2x1 pc2(pc_sel_2w, pc_sel_lw, addw[25:0], CTRL[2]);
MUX26_2x1 pc3(pc_sel_3w, irw[25:0], pc_sel_2w, CTRL[3]);


REG32 r(irw, DATA_IN, CTRL[4], CLK, RST);

REGISTER_FILE_32x32 registerInit(rfwl, rfw2, rl_sel_lw, irw[20:16], wd_sel_3w, wa_sel_3w, CTRL[6], CTRL[7

MUX5_2x1 rl_sel_l(rl_sel_lw, irw[25:21], 5'b00000, CTRL[8]);

REG32_PP SP(spw, aw, CTRL[9], CLK, RST);

MUX32_2x1 opl_sel_l(opl_sel_lw, rfwl, spw, CTRL[10]);

MUX32_2x1 op2_sel_l(op2_sel_lw, 1, {27'b0, irw[10:6]}, CTRL[11]);
MUX32_2x1 op2_sel_2(op2_sel_2w, {{{16'b0}}, irw[15:0]}, {{16{irw[15]}}, irw[15:0]}, CTRL[12]);
MUX32_2x1 op2_sel_3(op2_sel_3w, op2_sel_2w, op2_sel_lw, CTRL[13]);
MUX32_2x1 op2_sel_4(op2_sel_4w, op2_sel_3w, rfw2, CTRL[14]);

ALU alu(aw, ZERO, opl_sel_lw, op2_sel_4w, CTRL[20:15]);

MUX32_2x1 ma_sel_l(ma_sel_lw, aw, spw, CTRL[21]);
MUX26_2x1 ma_sel_2(ADDR, ma_sel_lw, pcw[25:0], CTRL[22]);

MUX32_2x1 md_sel_l(DATA_OUT, rfw2, rfwl, CTRL[23]);

MUX32_2x1 wd_sel_l(wd_sel_lw, aw, DATA_IN, CTRL[26]);
MUX32_2x1 wd_sel_2(wd_sel_2w, wd_sel_lw, {irw[15:0], {16{irw[15]}}}, CTRL[27]);
MUX32_2x1 wd_sel_3(wd_sel_3w, addlw, wd_sel_2w, CTRL[28]);

MUX5_2x1 wa_sel_l(wa_sel_lw, irw[15:11], irw[20:16], CTRL[29]);
MUX5_2x1 wa_sel_2(wa_sel_2w, 5'b00000, 5'b11111, CTRL[30]);
MUX5_2x1 wa_sel_3(wa_sel_3w, wa_sel_2w, wa_sel_lw, CTRL[31]);
endmodule
```

c. Testing

To test the data path I simply ran the Fibonacci test to see if it output correctly. It did so I'm assuming everything is working as it should but I didn't really test it as extensively as my lower level components.

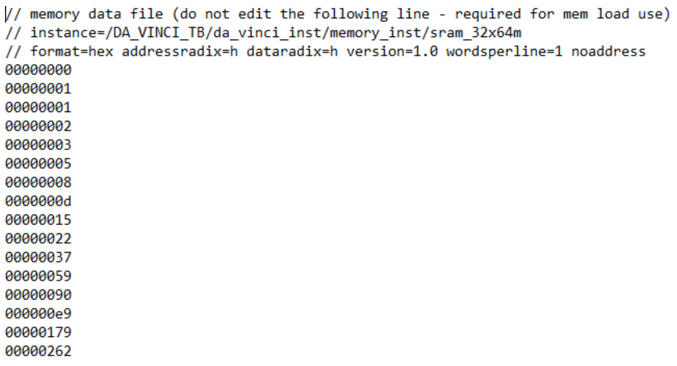
Fib Input:



File Edit Format View Help

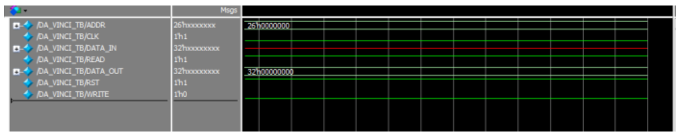
```
@0001000
20420001 //         addi r[2], r[2], 0x0001;
3C000100 //         lui  r[0], 0x0100;
AC010000 //         sw   r[1], r[0], 0x0000;
20000001 // loop:   addi r[0], r[0], 0x0001;
AC020000 //         sw   r[2], r[0], 0x0000;
20430000 //         addi r[3], r[2], 0x0000;
00411020 //         add  r[2], r[2], r[1];
20610000 //         addi r[1], r[3], 0x0000;
08001003 //         jmp  loop;
```

Fib OutPut:



```
// memory data file (do not edit the following line - required for mem load use)
// instance=/DA_VINCI_TB/da_vinci_inst/memory_inst/sram_32x64m
// format=hex addressradix=h dataradix=h version=1.0 wordsperline=1 noaddress
00000000
00000001
00000001
00000002
00000003
00000005
00000008
0000000d
00000015
00000022
00000037
00000059
00000090
000000e9
00000179
00000262
```

Wave:



```
/DA_VINCI_TB/ADDR       26'hxxxxxxx     26'h0000000
/DA_VINCI_TB/CLK        1'h1
/DA_VINCI_TB/DATA_IN    32'hxxxxxxxx
/DA_VINCI_TB/READ       1'h1
/DA_VINCI_TB/DATA_OUT   32'hxxxxxxxx    32'h00000000
/DA_VINCI_TB/RST        1'h1
/DA_VINCI_TB/WRITE      1'h0
```

I didn't have enough time to fully test and debug my data path and control path because I spent more time on lower level components and making sure they all worked correctly. But I think that im using the wrong multiplexers. I got a lot of warnings about the ports not being the same size so I believe that's why my wave is showing nothing.

14. Control Path

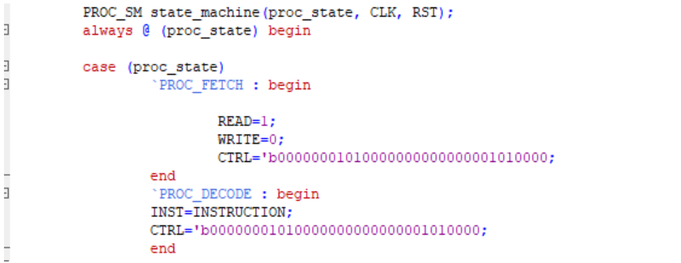
a. Design

Our control path contains the control unit as well as the state machine. It has five different stages FETCH, DECODE, EXECUTE,MEMORY, and WB. We need to change the ctrl value in order to tell the data path what executions it needs to do.

b. Implementation

To implement this we use a state machine to switch between states and our control unit which will take action depending on the state the system is currently in. Action is basically changing the ctrl to correctly tell the data path what to do.

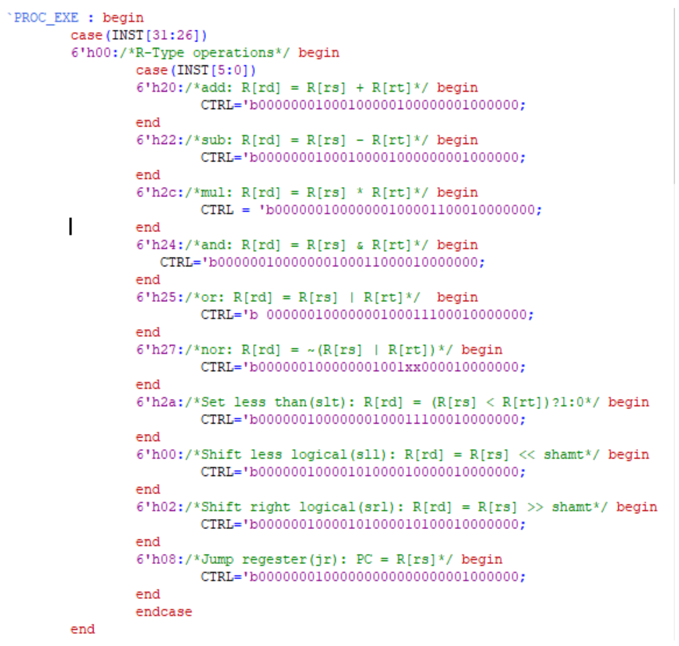
Fetch/Decode:



```
        PROC_SM state_machine(proc_state, CLK, RST);
        always @ (proc_state) begin

        case (proc_state)
                `PROC_FETCH : begin

                        READ=1;
                        WRITE=0;
                        CTRL='b00000001010000000000000001010000;
                end
                `PROC_DECODE : begin
                INST=INSTRUCTION;
                CTRL='b00000001010000000000000001010000;
                end
```

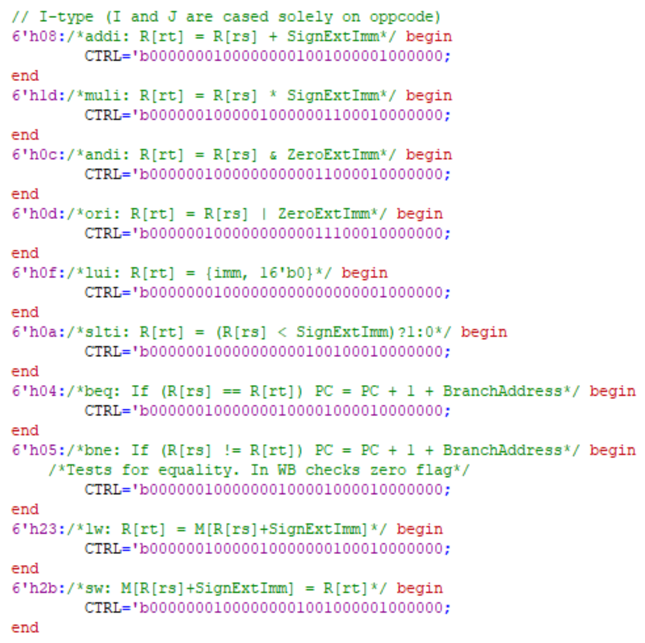
EXE:

r-type



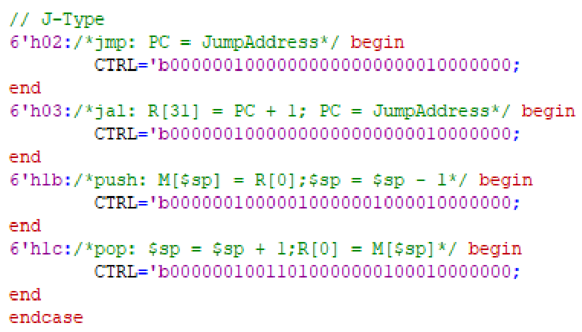
```
`PROC_EXE : begin
        case(INST[31:26])
        6'h00:/*R-Type operations*/ begin
                case(INST[5:0])
                6'h20:/*add: R[rd] = R[rs] + R[rt]*/ begin
                        CTRL='b00000001000100000100000001000000;
                end
                6'h22:/*sub: R[rd] = R[rs] - R[rt]*/ begin
                        CTRL='b00000001000100001000000001000000;
                end
                6'h2c:/*mul: R[rd] = R[rs] * R[rt]*/ begin
                        CTRL = 'b00000010000000100001100010000000;
                end
                6'h24:/*and: R[rd] = R[rs] & R[rt]*/ begin
                   CTRL='b00000010000000100011000010000000;
                end
                6'h25:/*or: R[rd] = R[rs] | R[rt]*/  begin
                        CTRL='b 00000010000000100011100010000000;
                end
                6'h27:/*nor: R[rd] = ~(R[rs] | R[rt])*/ begin
                        CTRL='b000000100000001001xx000010000000;
                end
                6'h2a:/*Set less than(slt): R[rd] = (R[rs] < R[rt])?1:0*/ begin
                        CTRL='b00000010000000100011100010000000;
                end
                6'h00:/*Shift less logical(sll): R[rd] = R[rs] << shamt*/ begin
                        CTRL='b00000010000101000010000010000000;
                end
                6'h02:/*Shift right logical(srl): R[rd] = R[rs] >> shamt*/ begin
                        CTRL='b00000010000101000010100010000000;
                end
                6'h08:/*Jump regester(jr): PC = R[rs]*/ begin
                        CTRL='b00000001000000000000000001000000;
                end
                endcase
        end
```

i-type:



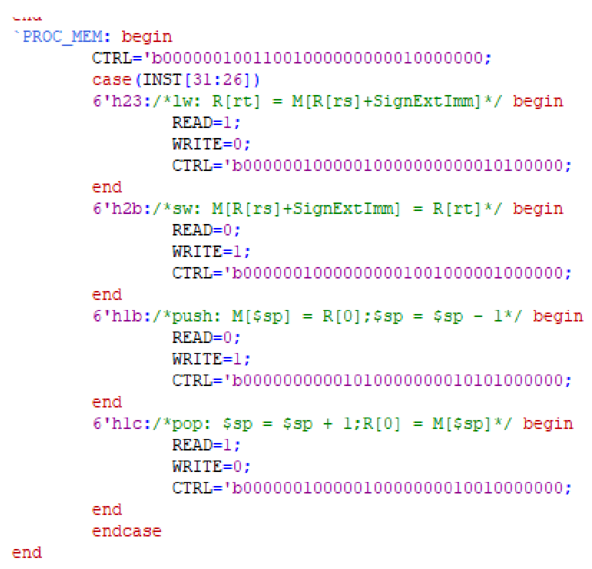
```
// I-type (I and J are cased solely on oppcode)
6'h08:/*addi: R[rt] = R[rs] + SignExtImm*/ begin
        CTRL='b00000001000000001001000001000000;
end
6'h1d:/*muli: R[rt] = R[rs] * SignExtImm*/ begin
        CTRL='b00000010000010000001100010000000;
end
6'h0c:/*andi: R[rt] = R[rs] & ZeroExtImm*/ begin
        CTRL='b00000010000000000011000010000000;
end
6'h0d:/*ori: R[rt] = R[rs] | ZeroExtImm*/ begin
        CTRL='b00000010000000000011100010000000;
end
6'h0f:/*lui: R[rt] = {imm, 16'b0}*/ begin
        CTRL='b00000001000000000000000001000000;
end
6'h0a:/*slti: R[rt] = (R[rs] < SignExtImm)?1:0*/ begin
        CTRL='b00000010000000000100100010000000;
end
6'h04:/*beq: If (R[rs] == R[rt]) PC = PC + 1 + BranchAddress*/ begin
        CTRL='b00000010000000100001000010000000;
end
6'h05:/*bne: If (R[rs] != R[rt]) PC = PC + 1 + BranchAddress*/ begin
    /*Tests for equality. In WB checks zero flag*/
        CTRL='b00000010000000100001000010000000;
end
6'h23:/*lw: R[rt] = M[R[rs]+SignExtImm]*/ begin
        CTRL='b00000010000010000000100010000000;
end
6'h2b:/*sw: M[R[rs]+SignExtImm] = R[rt]*/ begin
        CTRL='b00000001000000001001000001000000;
end
```

j-type:



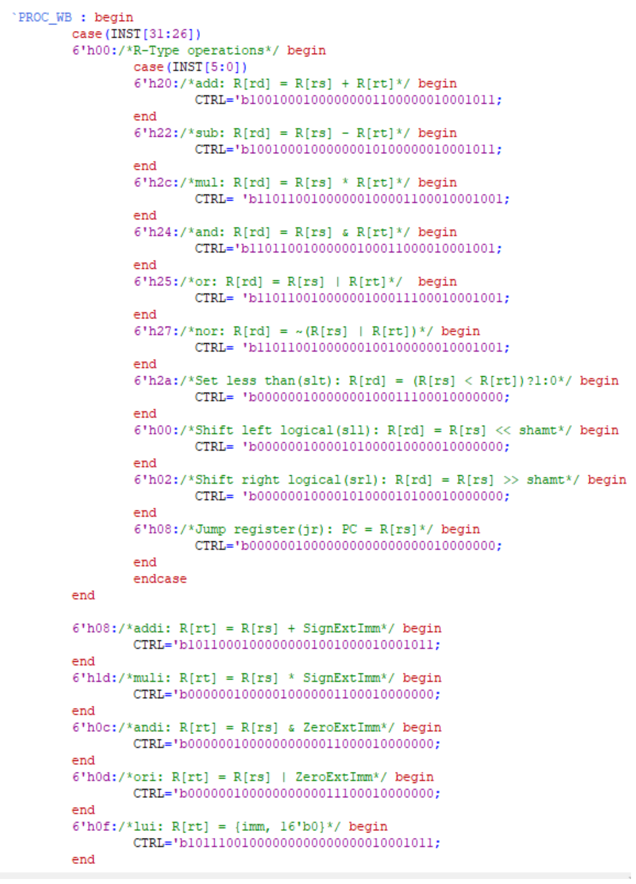
```
// J-Type
6'h02:/*jmp: PC = JumpAddress*/ begin
        CTRL='b00000010000000000000000010000000;
end
6'h03:/*jal: R[31] = PC + 1; PC = JumpAddress*/ begin
        CTRL='b00000010000000000000000010000000;
end
6'h1b:/*push: M[$sp] = R[0];$sp = $sp - 1*/ begin
        CTRL='b00000010000010000001000010000000;
end
6'h1c:/*pop: $sp = $sp + 1;R[0] = M[$sp]*/ begin
        CTRL='b00000010011010000000100010000000;
end
endcase
```

MEM:

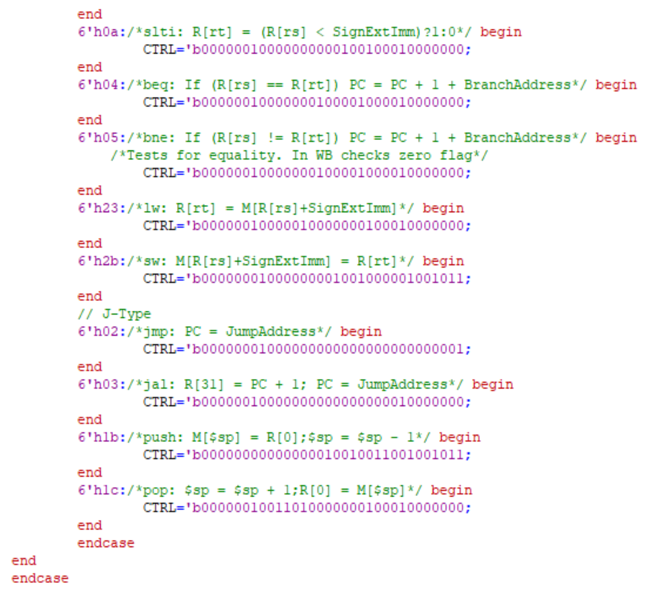


```
`PROC_MEM: begin
        CTRL='b00000010011001000000000010000000;
        case(INST[31:26])
        6'h23:/*lw: R[rt] = M[R[rs]+SignExtImm]*/ begin
                READ=1;
                WRITE=0;
                CTRL='b00000010000010000000000010100000;
        end
        6'h2b:/*sw: M[R[rs]+SignExtImm] = R[rt]*/ begin
                READ=0;
                WRITE=1;
                CTRL='b00000010000000001001000001000000;
        end
        6'h1b:/*push: M[$sp] = R[0];$sp = $sp - 1*/ begin
                READ=0;
                WRITE=1;
                CTRL='b00000000000101000000000101000000;
        end
        6'h1c:/*pop: $sp = $sp + 1;R[0] = M[$sp]*/ begin
                READ=1;
                WRITE=0;
                CTRL='b00000010000010000000010010000000;
        end
        endcase
end
```

WB:



```
`PROC_WB : begin
        case(INST[31:26])
        6'h00:/*R-Type operations*/ begin
                case(INST[5:0])
                6'h20:/*add: R[rd] = R[rs] + R[rt]*/ begin
                        CTRL='b10010001000000001100000010001011;
                end
                6'h22:/*sub: R[rd] = R[rs] - R[rt]*/ begin
                        CTRL='b10010001000000001010000010001011;
                end
                6'h2c:/*mul: R[rd] = R[rs] * R[rt]*/ begin
                        CTRL= 'b11011001000000100001100010001001;
                end
                6'h24:/*and: R[rd] = R[rs] & R[rt]*/ begin
                        CTRL='b11011001000000100011000010001001;
                end
                6'h25:/*or: R[rd] = R[rs] | R[rt]*/  begin
                        CTRL= 'b11011001000000100011100010001001;
                end
                6'h27:/*nor: R[rd] = ~(R[rs] | R[rt])*/ begin
                        CTRL= 'b11011001000000100100000010001001;
                end
                6'h2a:/*Set less than(slt): R[rd] = (R[rs] < R[rt])?1:0*/ begin
                        CTRL= 'b00000010000000100011100010000000;
                end
                6'h00:/*Shift left logical(sll): R[rd] = R[rs] << shamt*/ begin
                        CTRL= 'b00000010000101000010000010000000;
                end
                6'h02:/*Shift right logical(srl): R[rd] = R[rs] >> shamt*/ begin
                        CTRL= 'b00000010000101000010100010000000;
                end
                6'h08:/*Jump register(jr): PC = R[rs]*/ begin
                        CTRL='b00000010000000000000000010000000;
                end
                endcase
        end

        6'h08:/*addi: R[rt] = R[rs] + SignExtImm*/ begin
                CTRL='b10110001000000001001000010001011;
        end
        6'h1d:/*muli: R[rt] = R[rs] * SignExtImm*/ begin
                CTRL='b00000010000010000001100010000000;
        end
        6'h0c:/*andi: R[rt] = R[rs] & ZeroExtImm*/ begin
                CTRL='b00000010000000000011000010000000;
        end
        6'h0d:/*ori: R[rt] = R[rs] | ZeroExtImm*/ begin
                CTRL='b00000010000000000011100010000000;
        end
        6'h0f:/*lui: R[rt] = {imm, 16'b0}*/ begin
                CTRL='b10111001000000000000000010001011;
        end
```



```
        end
        6'h0a:/*slti: R[rt] = (R[rs] < SignExtImm)?1:0*/ begin
                CTRL='b00000010000000000010010001000000;
        end
        6'h04:/*beq: If (R[rs] == R[rt]) PC = PC + 1 + BranchAddress*/ begin
                CTRL='b00000010000000100001000010000000;
        end
        6'h05:/*bne: If (R[rs] != R[rt]) PC = PC + 1 + BranchAddress*/ begin
            /*Tests for equality. In WB checks zero flag*/
                CTRL='b00000010000000100001000010000000;
        end
        6'h23:/*lw: R[rt] = M[R[rs]+SignExtImm]*/ begin
                CTRL='b00000010000010000000100010000000;
        end
        6'h2b:/*sw: M[R[rs]+SignExtImm] = R[rt]*/ begin
                CTRL='b00000001000000000100100000100101l;
        end
        // J-Type
        6'h02:/*jmp: PC = JumpAddress*/ begin
                CTRL='b00000001000000000000000000000001;
        end
        6'h03:/*jal: R[31] = PC + 1; PC = JumpAddress*/ begin
                CTRL='b00000010000000000000000010000000;
        end
        6'h1b:/*push: M[$sp] = R[0];$sp = $sp - 1*/ begin
                CTRL='b00000000000000001001001100100101l;
        end
        6'h1c:/*pop: $sp = $sp + 1;R[0] = M[$sp]*/ begin
                CTRL='b00000010011010000000100010000000;
        end
        endcase
    end
    endcase
```

The state machine is the same as the previous project so no changes have been made to that.

c. Testing

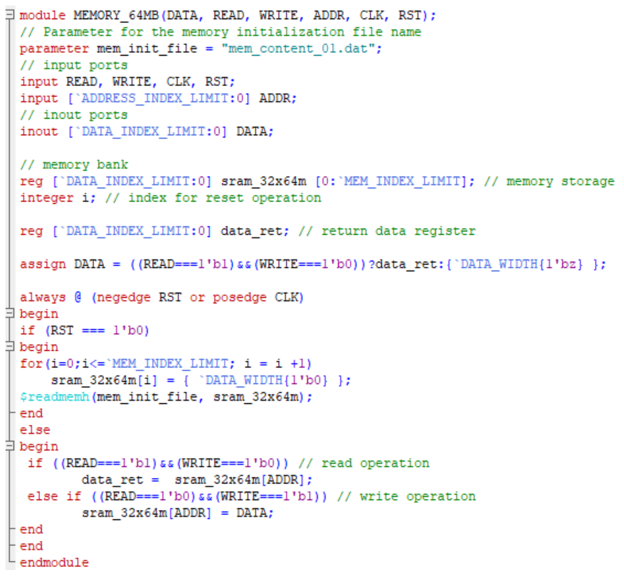
I tested the control path essentially the same way as the data path because it is the last step of our system. It doesn't really need a lot of testing because it is almost the same as our project 2 control unit.

15. Memory

a. Design

The memory was already implemented for us and didn't need any modifications.

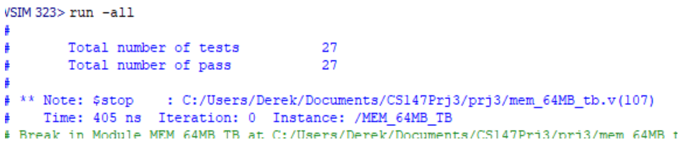
b. Implementation



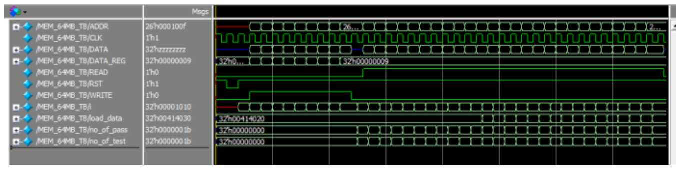
```
module MEMORY_64MB(DATA, READ, WRITE, ADDR, CLK, RST);
// Parameter for the memory initialization file name
parameter mem_init_file = "mem_content_01.dat";
// input ports
input READ, WRITE, CLK, RST;
input [`ADDRESS_INDEX_LIMIT:0] ADDR;
// inout ports
inout [`DATA_INDEX_LIMIT:0] DATA;

// memory bank
reg [`DATA_INDEX_LIMIT:0] sram_32x64m [0:`MEM_INDEX_LIMIT]; // memory storage
integer i; // index for reset operation

reg [`DATA_INDEX_LIMIT:0] data_ret; // return data register

assign DATA = ((READ===1'b1)&&(WRITE===1'b0))?data_ret:{`DATA_WIDTH{1'bz} };

always @ (negedge RST or posedge CLK)
begin
if (RST === 1'b0)
begin
for(i=0;i<=`MEM_INDEX_LIMIT; i = i +1)
    sram_32x64m[i] = { `DATA_WIDTH{1'b0} };
$readmemh(mem_init_file, sram_32x64m);
end
else
begin
 if ((READ===1'b1)&&(WRITE===1'b0)) // read operation
        data_ret =  sram_32x64m[ADDR];
 else if ((READ===1'b0)&&(WRITE===1'b1)) // write operation
        sram_32x64m[ADDR] = DATA;
end
end
endmodule
```

c. Testing

We run the memory test bench given to us and we can see that it passes all of our tests.



```
VSIM 323> run -all
#
#       Total number of tests           27
#       Total number of pass            27
#
# ** Note: $stop    : C:/Users/Derek/Documents/CS147Prj3/prj3/mem_64MB_tb.v(107)
#    Time: 405 ns  Iteration: 0  Instance: /MEM_64MB_TB
# Break in Module MEM_64MB_TB at C:/Users/Derek/Documents/CS147Prj3/prj3/mem_64MB_t
```



```
/MEM_64MB_TB/ADDR          26'h000100f
/MEM_64MB_TB/CLK           1'h1
/MEM_64MB_TB/DATA          32'hzzzzzzzz
/MEM_64MB_TB/DATA_REG      32'h00000009
/MEM_64MB_TB/READ          1'h0
/MEM_64MB_TB/RST           1'h1
/MEM_64MB_TB/WRITE         1'h0
/MEM_64MB_TB/i             32'h00001010
/MEM_64MB_TB/load_data     32'h00414030
/MEM_64MB_TB/no_of_pass    32'h0000001b
/MEM_64MB_TB/no_of_test    32'h0000001b
```

III. Conclusion

In this report I've shown how I implemented our davinci system using gate level modeling. In order to create a system that works it's very important that your base components are working properly or it will be almost impossible to test and debug the entire system since there are so many components. I was able to do this but it seems I was unable to correctly implement the data path.