DaVinci v1.0m: Mixed Model of a Computer System

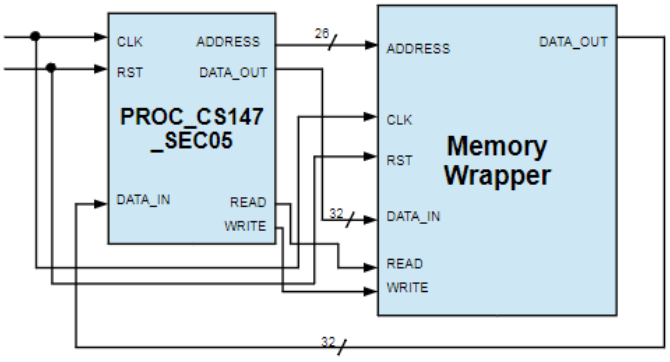
CS147 Project 3: Implementing DaVinci v1.0 at the gate level

Ryan Tran
College of Science: Computer Science
San José State University
San José, United States
tranryanp@gmail.com

***Abstract*—This report explains the creation of the DaVinci v1.0m mixed model computer system using Verilog. Supporting the previous DaVinci v1.0 behavioral model, the DaVinci v1.0m is an extension of project two. The following sections will cover each component individually and then later demonstrating how all the components work together to simulate a working DaVinci v1.0m system. This system requires setup and knowledge from the first and second CS147 projects.**

I. Introduction

This project is the third and final assignment of CS147. The objective is to implement a mixed model of a computer system that supports the 'CS147DV' instruction set named DaVinci v1.0m. This system's backbone is the DaVinci v1.0 behavioral model, however, some of which being implemented at a gate level, resulting in a mix of behavioral and gate level implementation, hence the term mixed model. Implementation at the gate level is done through using logic gates, the foundation of all hardware. Shown below is the completed DaVinci v1.0m mixed model system:



The results of this project are dependent on the first and second CS147 projects, in which understanding ModelSim and the components of the DaVinci v1.0 system is vital and will not be reintroduced. This report contains the requirements, designs, and test results of such tasks.

II. Project Requirements

Understanding and having completed the setup of the first and second projects are necessary in proceeding with this project. Without it, understanding of ModelSim, debugging, DaVinci v1.0 components, and implementation will be very difficult to complete, if not impossible. The components that need to be completed are:

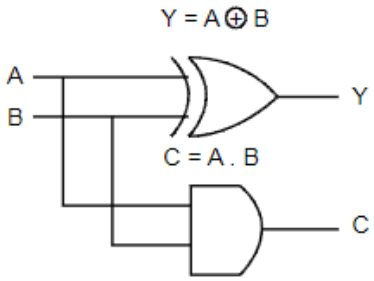
1) Half Adder
2) Full Adder
3) Ripple-Carry Adder/Subtractor
4) Multiplexer
5) Unsigned & Signed Multiplier
6) Barrel Shifter
7) Logic gates: AND, NOR, INV, OR
8) ALU
9) SR-Latch
10) D-Latch
11) FlipFlop
12) Decoder
13) Register
14) Data Path (Optional)
15) Control Unit (Optional)
16) Memory Wrapper (Optional)
17) Processor (Optional)
18) System (Optional)

Each of these many components work together to make up the higher level components, such as the ALU, control unit, and processor. All other code has already been provided in order to avoid tedious work of connecting ports and wires.

III. Implementation

A. Half Adder

A half adder implements "half addition," which is only two bits being added together and no carry in-bit involved. The result is a 1-bit sum and a 1-bit carry-out, also known as the overflow bit. The follow schematic for the half adder is shown below:



As depicted, ‘Y’ and ‘C’ are respectively the sum and carry out. As a relatively simple component, as well as the foundation of more complex components, the implementation is simple.

```
module HALF_ADDER(Y,C,A,B);
output Y,C;
input A,B;
// half adder
// Y = A XOR B
// C = A AND B

xor xor_inst1(Y, A, B);
and and_inst1(C, A, B);

endmodule
```

The two bits to be added, ‘A’ and ‘B’, are XOR’ed together to receive ‘Y’. Then ‘A’ and ‘B’ are AND’ed together to receive ‘C’. With that, the half adder has been implemented. Despite its quick implementation, testing is still important as this component will be used again in higher-level components.

```
module HALF_ADDER_TB;
reg A, B;
wire Y, C;

HALF_ADDER HA_inst1(.Y(Y), .C(C), .A(A), .B(B));

initial
begin
        A=0; B=0;
        #5 $write("A:%d, B:%d, Y (Sum):%d, C (Carry):%d\n", A, B, Y, C);
        #5 A=1; B=0;
        #5 $write("A:%d, B:%d, Y (Sum):%d, C (Carry):%d\n", A, B, Y, C);
        #5 A=0; B=1;
        #5 $write("A:%d, B:%d, Y (Sum):%d, C (Carry):%d\n", A, B, Y, C);
        #5 A=1; B=1;
        #5 $write("A:%d, B:%d, Y (Sum):%d, C (Carry):%d\n", A, B, Y, C);
end

endmodule
```

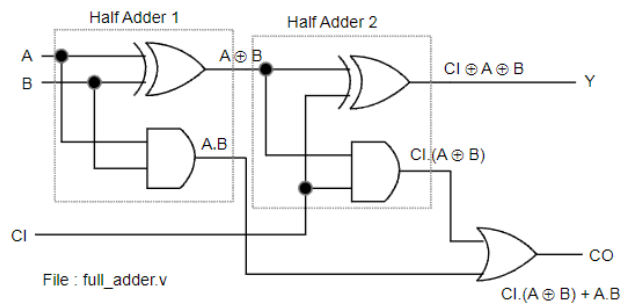
The testbench code first connects and initializes the necessary wires and registers, as well as the half adder itself. Once complete, the values of ‘A’ and ‘B’ are changed to test different sums, as well as every possible combination of bits. The results are printed to the Transcript for checking. Testing will be further detailed in the later “Testing” section.

B. Full Adder

A full adder is an upgraded version of the half adder, now supporting a carry-in bit while still retaining the same results from the half adder, sum and carry-out. However, carry-out has now been renamed from ‘C’ to ‘CO’ for better clarity alongside carry-in’s ‘CI’. A schematic of the full adder is shown:

$Y = CI \oplus (A \oplus B)$

$CO = CI.(A \oplus B) + A.B$



As expected, the full adder consists of two half adders. This connection of the two half adders allows for addition with a carry-in value. For a 1-bit adder, the reason behind this isn’t clear at this step, but the next component will explain the purpose of having the carry-in.

```
module FULL_ADDER(S,CO,A,B, CI);
output S,CO;
input A,B, CI;
wire Y, C1, C2;
// for half adder
// Y = A XOR B
// C1 = A AND B

//for full adder
// S = Y XOR CI
// C2 = Y AND CI
// CO = C1 OR C2

HALF_ADDER ha_inst_1(.Y(Y), .C(C1), .A(A), .B(B));
HALF_ADDER ha_inst_2(.Y(S), .C(C2), .A(Y), .B(CI));

or or_inst1(CO, C1, C2);

endmodule
```

Following along with the schematic, two half adders are initialized, though with slightly different fields. The first half adder takes in the standard inputs of ‘Y’, ‘A’, ‘B’, but also ‘C1’, which has the ‘1’ to denote that it is the carry-out of the first half adder. The second half adder then takes in ‘Y’ again, but as the ‘A’ value instead, ‘CI’ as the carry-in value, and ‘C2’ which is the same as ‘C1’, having the ‘2’ to denote that it is the carry-out of the second half adder. After going through both half-adders, the final results are stored in ‘S’, the sum, and ‘CO’, the final carry-out. For testing, the code is similar that of the half adder’s testbench.

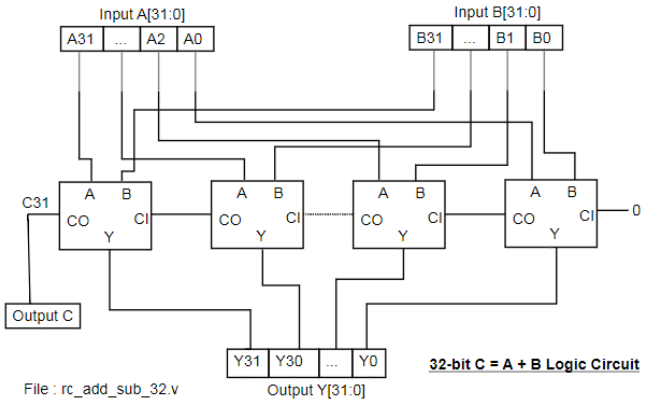
```
FULL_ADDER FA_inst1(.S(S), .CO(CO), .A(A), .B(B), .CI(CI));

initial
begin
        A = 0; B = 0; CI = 0;
        #5 $write("A:%d, B:%d, CI (Carry in):%d, S (Sum):%d, CO (Carry-out):%d\n", A, B, CI, S, CO);
        #5 A = 0; B = 0; CI = 1;
        #5 $write("A:%d, B:%d, CI (Carry in):%d, S (Sum):%d, CO (Carry-out):%d\n", A, B, CI, S, CO);
        #5 A = 0; B = 1; CI = 0;
        #5 $write("A:%d, B:%d, CI (Carry in):%d, S (Sum):%d, CO (Carry-out):%d\n", A, B, CI, S, CO);
        #5 A = 0; B = 1; CI = 1;
        #5 $write("A:%d, B:%d, CI (Carry in):%d, S (Sum):%d, CO (Carry-out):%d\n", A, B, CI, S, CO);
        #5 A = 1; B = 0; CI = 0;
        #5 $write("A:%d, B:%d, CI (Carry in):%d, S (Sum):%d, CO (Carry-out):%d\n", A, B, CI, S, CO);
        #5 A = 1; B = 0; CI = 1;
        #5 $write("A:%d, B:%d, CI (Carry in):%d, S (Sum):%d, CO (Carry-out):%d\n", A, B, CI, S, CO);
        #5 A = 1; B = 1; CI = 0;
        #5 $write("A:%d, B:%d, CI (Carry in):%d, S (Sum):%d, CO (Carry-out):%d\n", A, B, CI, S, CO);
        #5 A = 1; B = 1; CI = 1;
        #5 $write("A:%d, B:%d, CI (Carry in):%d, S (Sum):%d, CO (Carry-out):%d\n", A, B, CI, S, CO);
end

endmodule
```

Just like the half adder’s code, ‘A’ and ‘B’ are set to the values that are intended to be tested. The new addition is the ‘CI’ value which will also be changed to test all possible bit combinations.

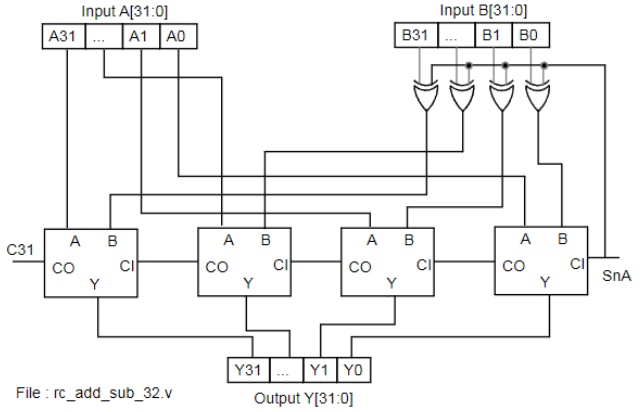
C. Ripple-Carry Adder / Subtractor

The Binary Ripple Carry Adder is a system in which multiple full adders are combined together to sum multiple bit numbers at a time, specifically 32-bit and 64-bit, rather than just 1-bit. This system requires the usage of the carry-in value because the next adder requires the previous adder’s overflow bit to determine how it should calculate the next bit’s sum. The

carry-in bit is then “rippled” from the lower bit adders to the higher bit adders, giving it its name. This can be understood from the simple example of adding the binary values 1 + 1 + 1, where the final ‘1’ is the carry-in while the first two values are A and B (the ‘+’ is not the OR operation in this example). It would be (1 + 1) + 1 = (10) + 1 = 11. This is the same concept as adding the decimal values 8 + 9 by hand and “carrying over” the next tenth digit. The schematic for this component is shown below:



As shown, the initial carry-in value is 0. Although the carry in is only necessary for the subsequent full adders, the implementation of all full adders have a carry-in, so the first one must be set to 0. The carry-in is then computed to determine the carry-out, which is then used as the next full adder’s carry-in. This cycle repeats until the very last full adder, in which the final carry-out is the overflow bit. Otherwise, each corresponding pair of bits from ‘A’ and ‘B’ are input into the full adder to compute ‘Y’, which is then stored in a 32-bit register. However, this Binary Ripple-Carry Adder can be extended to subtraction too. The new schematic is as shown below:



Due to the mechanics of 2’s complement, subtracting A – B is equal to A + INV(B) + 1. As such, the way to subtract B is to invert each individual bit and then add 1 to the inverted value. However, this can be further improved and generalized for both addition and subtraction because the result of the inversion and adding 1 is equal to the XORing of ‘B’’s bit’s and the initial carry-in value. With the carry-in value now named ‘SnA’, when it is 0, the operation is addition. When ‘SnA’ is 0, the XORing of ‘B’’s bits will not have any affect. However, when ‘SnA’ is 1 and the operation is subtraction, ‘B’’s bits will be inverted, along with ‘SnA’ being passed as the carry-in of value 1 to complete the 2’s complement inversion.

```
// additional wires to contain sum and carry results
wire [63:0] sum;
// one more carry than sum results because there is an initial carry
wire [64:0] carry;

// assign carry to CO since carries are the previous CO
assign CO = carry[64];
// the first carry is dependent on if it is addition or subtraction
assign carry[0] = SnA;

// use generate to generate a new instance of the full adder per iteration
genvar i;
generate
	for (i = 0; i < 64; i = i + 1)
	begin: FA_64bit_generation_loop
		xor xor_inst1(sum[i], B[i], SnA);
		FULL_ADDER FA_inst1(.S(Y[i]), .CO(carry[i + 1]), .A(A[i]), .B(sum[i]), .CI(carry[i]));
	end
endgenerate

endmodule
```

Two wires are first initialized for the sum and carry-in results from the adders. Since it is 32-bit addition and subtraction, ‘sum’ is 32-bits while ‘carry’ is 33-bits. The ‘carry’ wire is one larger than the ‘sum’ as there must be the initial carry-in value ‘SnA’ that determines the operation being performed. The first ‘carry’ element is then set to ‘SnA’.

Once that is done, each of the 32 bits are looped through. For B, each bit is first XOR’ed with ‘SnA’ to get the appropriate value depending on the operation. After that, a full adder instantiation is created to receive and store that corresponding pair of bits’ sum and carry-out. As seen in the instantiation, the carry-out value is saved at ‘i+1’ position of ‘carry’ because of the initial ‘SnA’ value located at the position 0 of ‘carry’.

This looping is done through a standard for loop, but the additional ‘generate’ keyword. Generating allows for several instantiations of the full adder at the same time. The reason this is necessary is because there are 32 full adders working together to acquire the final output. Though it is possible to achieve this with only a for loop and no generation, this is the most effective method to follow the schematic’s design.

The Ripple-Carry Adder / Subtractor can also be extended to handle 64-bits. The implementation is identical other than a minor change.

```
// additional wires to contain sum and carry results
wire [63:0] sum;
// one more carry than sum results because there is an initial carry
wire [64:0] carry;

// assign carry to CO since carries are the previous CO
assign CO = carry[64];
// the first carry is dependent on if it is addition or subtraction
assign carry[0] = SnA;

// use generate to generate a new instance of the full adder per iteration
genvar i;
generate
	for (i = 0; i < 64; i = i + 1)
	begin : FA_64bit_generation_loop
		xor xor_inst1(sum[i], B[i], SnA);
		FULL_ADDER FA_inst1(.S(Y[i]), .CO(carry[i + 1]), .A(A[i]), .B(sum[i]), .CI(carry[i]));
	end
endgenerate
```

As shown, the code is the same as the 32-bit version, except that the wires ‘sum’ and ‘carry’ can now contain 64 bits and 65 bits respectively. With that, the Ripple-Carry Adder / Subtractor is complete. Testing for both 32-bit and 64-bit is very similar to the full adder. The testbench code is as shown:

```
module RC_ADD_SUB_32_TB;
wire [`DATA_INDEX_LIMIT:0] Y;
wire CO;
reg [`DATA_INDEX_LIMIT:0] A, B;
reg SnA;

RC_ADD_SUB_32 RC_add_sub_32_inst1(.Y(Y), .CO(CO), .A(A), .B(B), .SnA(SnA));

initial
begin
A = 0; B = 0; SnA = 0;
#5 $write("A:%d, B:%d, SnA:%d, Y (Sum):%d, CO (Final carry-out/overflow bit):%d\n", A, B, SnA, Y, CO);
#5 A = 32; B = 32; SnA = 1;
#5 $write("A:%d, B:%d, SnA:%d, Y (Sum):%d, CO (Final carry-out/overflow bit):%d\n", A, B, SnA, Y, CO);
#5 A = 32; B = -32; SnA = 0;
#5 $write("A:%d, B:%d, SnA:%d, Y (Sum):%d, CO (Final carry-out/overflow bit):%d\n", A, B, SnA, Y, CO);
#5 A = 32; B = 16; SnA = 0;
#5 $write("A:%d, B:%d, SnA:%d, Y (Sum):%d, CO (Final carry-out/overflow bit):%d\n", A, B, SnA, Y, CO);
#5 A = 16; B = -32; SnA = 1;
#5 $write("A:%d, B:%d, SnA:%d, Y (Sum):%d, CO (Final carry-out/overflow bit):%d\n", A, B, SnA, Y, CO);
#5 A = 0; B = 32; SnA = 0;
#5 $write("A:%d, B:%d, SnA:%d, Y (Sum):%d, CO (Final carry-out/overflow bit):%d\n", A, B, SnA, Y, CO);
#5 A = 2147483647; B = 2147483647; SnA = 0;
#5 $write("A:%d, B:%d, SnA:%d, Y (Sum):%d, CO (Final carry-out/overflow bit):%d\n", A, B, SnA, Y, CO);
#5 A = 2147483647; B = 2147483647; SnA = 1;
#5 $write("A:%d, B:%d, SnA:%d, Y (Sum):%d, CO (Final carry-out/overflow bit):%d\n", A, B, SnA, Y, CO);
#5 A = 2147483647; B = -2147483648; SnA = 0;
#5 $write("A:%d, B:%d, SnA:%d, Y (Sum):%d, CO (Final carry-out/overflow bit):%d\n", A, B, SnA, Y, CO);
end
```

Just like the full adder, there are three fields to be changed: ‘A’, ‘B’, and ‘SnA’, previously known as ‘CI’ in the full adder testbench. However, the new change is that ‘A’ and ‘B’ are no longer 1-bit values, but 32-bit values instead. This allows for a greater range of potential test options, but unfortunately not possible for all combinations to be tested. ‘SnA’ remains 1-bit as it is a binary value that determines if it is addition or subtraction.

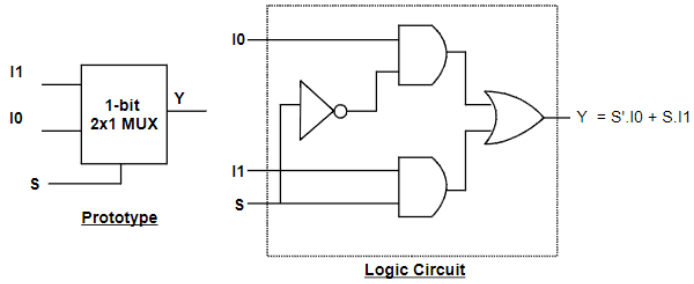
The code shown is the 32-bit testbench code, as shown by the 32-bit Ripple-Carry Adder / Subtractor instantiation. To test 64-bit, the instantiation just needs to be edited to be ‘64’ instead of ‘32’. Testing results will be provided in the following section.

D. Multiplexer

A multiplexer (MUX) is a digital logic circuit component. It is used to select input data depending on the selection input. It can be considered the digital logic circuit that represents an “if then else” statement in high-level programming. For any MUX, there is only one output. However, the relationship between input and select is that there are at most 2 to the number of selection wires of inputs. For example, if there are 2 selection wires, there can be at most 4 inputs.

1) 1-bit 2x1 MUX

In the case of a 2x1 multiplexer, in which the notation means 2 inputs to 1 output, there is only 1 selection wire. While there are multiple different ways to implement a MUX, this project’s MUX will follow this schematic:



As shown, ‘I0’ and ‘I1’ are the inputs while ‘S’ is the selection. The final result is ‘Y’, which is noted to be equal to ‘S’.I0 + SI1’. Using this, the code can be implemented in the same fashion.

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

wire s_inv, and_1, and_2;

not not_inst1(s_inv, S);
and and_inst1(and_1, s_inv, I0);
and and_inst2(and_2, S, I1);
or or_inst1(Y, and_1, and_2);
```

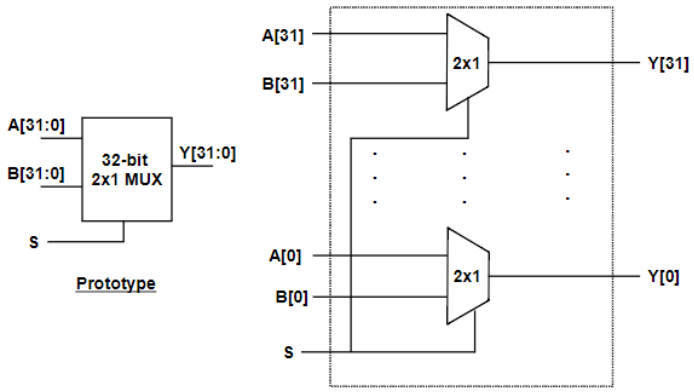
The name of the inputs and outputs match those of the schematic. However, additional wires are instantiated in order to pass in the updated values to the next logic gates. Following along with the equation for ‘Y’, ‘S’ is first inverted and stored on a wire. ‘S’.I0’ and ‘S.I1’ are performed, ending with the two being OR’ed together to be output as ‘Y’. To test the 1-bit 2x1 MUX, the same approach used for the full adder can be used.

```
module MUX1_2x1_TB;
reg I0, I1, S;
wire Y;

MUX1_2x1 mux1_2x1_inst1(.Y(Y), .I0(I0), .I1(I1), .S(S));

initial
begin
#5 I0 = 0; I1 = 0; S = 0;
#5 $write("I0:%d, I1:%d, S:%d, Y:%d\n", I0, I1, S, Y);
#5 I0 = 0; I1 = 0; S = 1;
#5 $write("I0:%d, I1:%d, S:%d, Y:%d\n", I0, I1, S, Y);
#5 I0 = 0; I1 = 1; S = 0;
#5 $write("I0:%d, I1:%d, S:%d, Y:%d\n", I0, I1, S, Y);
#5 I0 = 0; I1 = 1; S = 1;
#5 $write("I0:%d, I1:%d, S:%d, Y:%d\n", I0, I1, S, Y);
#5 I0 = 1; I1 = 0; S = 0;
#5 $write("I0:%d, I1:%d, S:%d, Y:%d\n", I0, I1, S, Y);
#5 I0 = 1; I1 = 0; S = 1;
#5 $write("I0:%d, I1:%d, S:%d, Y:%d\n", I0, I1, S, Y);
#5 I0 = 1; I1 = 1; S = 0;
#5 $write("I0:%d, I1:%d, S:%d, Y:%d\n", I0, I1, S, Y);
#5 I0 = 1; I1 = 1; S = 1;
#5 $write("I0:%d, I1:%d, S:%d, Y:%d\n", I0, I1, S, Y);
end

endmodule
```

Similar to the full adder test, there are three 1-bit values that can be changed. Every combination will be tested to check for full functionality. Testing will be detailed later.

2) 32-bit 2x1 MUX

This MUX can be further upgraded to support 32-bits rather than just 1. Not to be confused with 32 inputs, this MUX will be able to handle two 32-bit inputs, still retaining a 2x1 structure rather than a 32x1. The following schematic demonstrates this design:



The schematic follows the same structure as the 1-bit 2x1 MUX. Two inputs, one selection, and 1 output, with the exception that all but the selection are now 32-bit. This upgrade is done through the usage of 32 1-bit 2x1 MUX's within this system. Essentially, there is a 1-bit 2x1 MUX for each and every one of the 32 bits. This can be done relatively simply through looping.

```
// generates 32 1-bit 2x1 multiplexers
genvar i;
generate
        for (i = 0; i < 32; i = i + 1)
        begin: mux1_2x1_generation_loop
                MUX1_2x1 mux1_2x1_inst1(Y[i], I0[i], I1[i], S);
        end
endgenerate
```

Similar to the Ripple-Carry Adder / Subtractor, the 32-bit 2x1 MUX also benefits from the usage of generating multiple instances. By generating 32 of the 1-bit 2x1 MUX's, the schematic is properly implemented. As explained earlier, it can be seen that the 1-bit 2x1 MUX is initialized 32 times for each of the corresponding bits. Testing will be done similarly to the 1-bit 2x1 MUX by providing 'I0', 'I1', and 'S' values.

```
module MUX32_2x1_TB;
reg [31:0] I0, I1;
reg S;
wire [31:0] Y;

MUX32_2x1 mux32_2x1_inst1(.Y(Y), .I0(I0), .I1(I1), .S(S));

initial
begin
I0 = 0; I1 = 0; S = 0;
#5 $write("I0:%d, I1:%d, S:%d, Y:%d\n", I0, I1, S, Y);
#5 I0 = 1431655700; I1 = 1431655701; S = 0;
#5 $write("I0:%d, I1:%d, S:%d, Y:%d\n", I0, I1, S, Y);
#5 I0 = 1431655700; I1 = 1431655701; S = 1;
#5 $write("I0:%d, I1:%d, S:%d, Y:%d\n", I0, I1, S, Y);
end

endmodule
```

Because the range of 32-bit values are expansive, it is difficult to test all combinations. Instead, testing two 32-bit values with different select values will suffice. If testing succeeds for the whole 32-bit values, then it will work for any other smaller bit value as well. The 'I0' and 'I1' values tested in this testbench are inverted of one another to ensure that every bit of the 32-bit 2x1 MUX functions. Test results will be detailed later.

3) 64-bit 2x1 MUX

Though not required, the implementation of a 64-bit 2x1 MUX will be very helpful in future components. There is no schematic for this MUX, but it is very similar to the 32-bit 2x1 MUX, using 64 1-bit 2x1 MUX's instead of 32.

```
// generates 64 1-bit 2x1 multiplexers
genvar i;
generate
        for (i = 0; i < 64; i = i + 1)
        begin : mux1_2x1_generation_loop
                MUX1_2x1 mux1_2x1_inst1(Y[i], I0[i], I1[i], S);
        end
endgenerate
```

This code is identical to the implementation of the 32-bit 2x1 MUX. The only difference is that the loop runs 64 times rather than 32 in order to generate 64 1-bit 2x1 MUX's. The same test-code as the 32-bit 2x1 MUX will be used other than the instantiation of the MUX as a 64-bit 2x1 instead. Test results will be seen later.

4) 5-bit 2x1 MUX

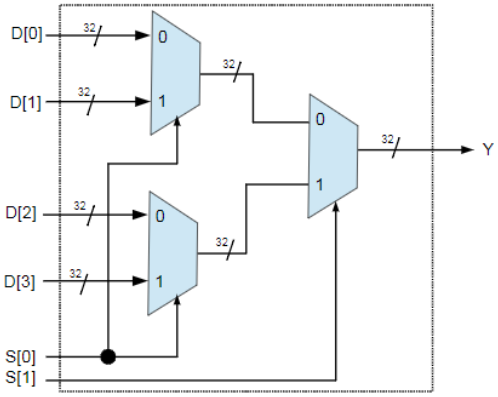
An additional useful MUX that will be used later on is 5-bit 2x1 MUX. It implemented the same way as the 32-bit and 64-bit 2x1 MUX, except only 5 1-bit 2x1 MUX's are generated.

```
// 5-bit mux
module MUX5_2x1(Y, I0, I1, S);
// output list
output [4:0] Y;
//input list
input [4:0] I0;
input [4:0] I1;
input S;

genvar i;
generate
    for (i = 0; i < 5; i = i + 1)
    begin : mux1_2x1_gen_loop
        MUX1_2x1 mux1_2x1_inst1(Y[i], I0[i], I1[i], S);
    end
endgenerate

endmodule
```

Following the same approach, a generation loop is used to generate the 5-bit 2x1 MUX. Testing will not be performed on this case as the 32-bit and 64-bit testing will suffice and by connection, will mean that the 5-bit 2x1 MUX also functions.

5) 32-bit 4x1 MUX

Being further upgraded, the 32-bit 2x1 MUX can be used to implement a 32-bit 4x1 MUX, allowing for more input options, all of 32-bit size. Below is the new schematic:



The 32-bit 4x1 MUX consists of three 32-bit 2x1 MUX's. An extra selection wire has been implemented to support the greater amount of inputs. The results of the left two MUX's are wired as the inputs of the final MUX.

```
wire [31:0] MUX_result_1;
wire [31:0] MUX_result_2;

MUX32_2x1 mux32_2x1_inst1(.Y(MUX_result_1), .I0(I0), .I1(I1), .S(S[0]));
MUX32_2x1 mux32_2x1_inst2(.Y(MUX_result_2), .I0(I2), .I1(I3), .S(S[0]));
MUX32_2x1 mux32_2x1_inst3(.Y(Y), .I0(MUX_result_1), .I1(MUX_result_2), .S(S[1]));
```

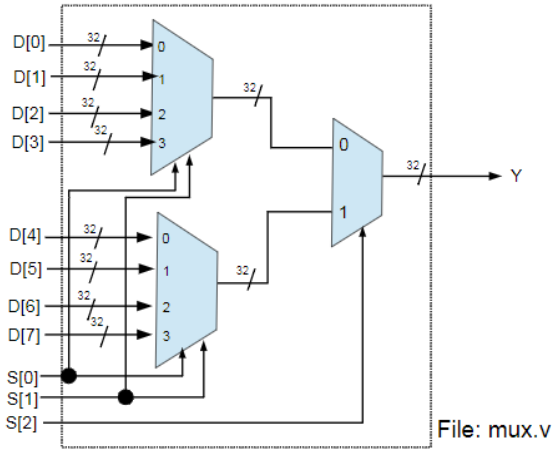
Two wires are first created to represent the results of the left two MUX's. They are then passed into 32-bit 2x1 MUX instances that will calculate and store the result on the wires. After that, the third and final MUX takes the previous two results, as well as the last selection input, to output the result of the 32-bit 4x1 MUX. The structure of the test code for the 32-bit 4x1 MUX will be similar to the previous MUX's, but there will now be four inputs to choose from, rather than just two.

```
module MUX32_4x1_TB;
reg [31:0] I0, I1, I2, I3;
reg [1:0] S;
wire [31:0] Y;

MUX32_4x1 mux32_4x1_inst1(.Y(Y), .I0(I0), .I1(I1), .I2(I2), .I3(I3), .S(S));

initial
begin
I0 = 0; I1 = 0; I2 = 0; I3 = 0; S = 0;
#5 $write("I0:%d, I1:%d, I2:%d, I3:%d, S:%d Y:%d\n", I0, I1, I2, I3, S, Y);
I0 = 1431655700; I1 = 1431655701; I2 = 1431655702; I3 = 1431655703; S = 0;
#5 $write("I0:%d, I1:%d, I2:%d, I3:%d, S:%d Y:%d\n", I0, I1, I2, I3, S, Y);
I0 = 1431655700; I1 = 1431655701; I2 = 1431655702; I3 = 1431655703; S = 1;
#5 $write("I0:%d, I1:%d, I2:%d, I3:%d, S:%d Y:%d\n", I0, I1, I2, I3, S, Y);
I0 = 1431655700; I1 = 1431655701; I2 = 1431655702; I3 = 1431655703; S = 2;
#5 $write("I0:%d, I1:%d, I2:%d, I3:%d, S:%d Y:%d\n", I0, I1, I2, I3, S, Y);
I0 = 1431655700; I1 = 1431655701; I2 = 1431655702; I3 = 1431655703; S = 3;
#5 $write("I0:%d, I1:%d, I2:%d, I3:%d, S:%d Y:%d\n", I0, I1, I2, I3, S, Y);
end

endmodule
```

As there are more and more inputs, there will need to be more tests per selection possibility. With the selection input now being a range from 0-3, four tests must be completed. The values that are input as the choices end in two digits that will make it easier to determine if the correct result was output. If selection '0' was chosen, the value ending in '00' should be output. If selection '1', '01', and etc. Testing will be detailed later in the report.

6) 32-bit 8x1 MUX

As a pattern is now emerging, the previous 32-bit 4x1 MUX can be used to implement a 32-bit 8x1 MUX. There are gradually more and more input options that are able to be handled. The following schematic is the newer model:



File: mux.v

Similar to the 4x1 MUX, the 8x1 MUX uses the two of the previous MUX's, as well as a 2x1 MUX. So in this case, there are two 4x1 MUX's and one 2x1 MUX, as well as yet another selection input.

```
wire [31:0] MUX_result_1;
wire [31:0] MUX_result_2;

MUX32_4x1 mux32_4x1_inst1(.Y(MUX_result_1), .I0(I0), .I1(I1), .I2(I2), .I3(I3), .S(S[1:0]));
MUX32_4x1 mux32_4x1_inst2(.Y(MUX_result_2), .I0(I4), .I1(I5), .I2(I6), .I3(I7), .S(S[1:0]));
MUX32_2x1 mux32_2x1_inst1(.Y(Y), .I0(MUX_result_1), .I1(MUX_result_2), .S(S[2]));
```

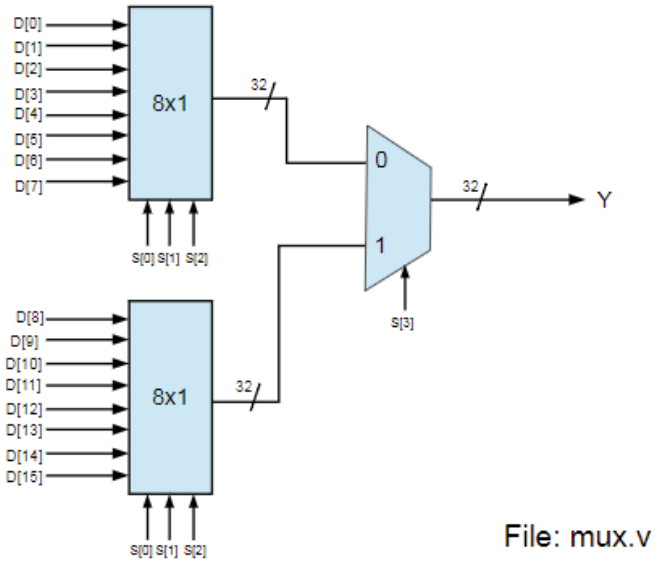
The approach is as straightforward as implementing the 4x1 MUX. Two 4x1 MUX's are instantiated and the results are calculated onto wires. The wires are then both pathed into the 2x1 MUX to acquire the final result of the 8x1 MUX. Similar to the 32-bit 4x1 MUX testbench code, the testbench for the 8x1 MUX will be the same, except expanded to test eight inputs.

```
module MUX32_8x1_TB;
reg [31:0] I0, I1, I2, I3, I4, I5, I6, I7;
reg [2:0] S;
wire [31:0] Y;

MUX32_8x1 mux32_8x1_inst1(.Y(Y), .I0(I0), .I1(I1), .I2(I2), .I3(I3),
                          .I4(I4), .I5(I5), .I6(I6), .I7(I7), .S(S));

initial
begin
I0 = 0; I1 = 0; I2 = 0; I3 = 0; I4 = 0; I5 = 0; I6 = 0; I7 = 0; S = 0;
#5 $write("I0:%d, I1:%d, I2:%d, I3:%d, I4:%d, I5:%d, I6:%d, I7:%d, S:%d Y:%d\n",
I0, I1, I2, I3, I4, I5, I6, I7, S, Y);
I0 = 1431655700; I1 = 1431655701; I2 = 1431655702; I3 = 1431655703;
I4 = 1431655704; I5 = 1431655705; I6 = 1431655706; I7 = 1431655707; S = 0;
#5 $write("I0:%d, I1:%d, I2:%d, I3:%d, I4:%d, I5:%d, I6:%d, I7:%d, S:%d Y:%d\n",
I0, I1, I2, I3, I4, I5, I6, I7, S, Y);
I0 = 1431655700; I1 = 1431655701; I2 = 1431655702; I3 = 1431655703;
I4 = 1431655704; I5 = 1431655705; I6 = 1431655706; I7 = 1431655707; S = 1;
#5 $write("I0:%d, I1:%d, I2:%d, I3:%d, I4:%d, I5:%d, I6:%d, I7:%d, S:%d Y:%d\n",
I0, I1, I2, I3, I4, I5, I6, I7, S, Y);
I0 = 1431655700; I1 = 1431655701; I2 = 1431655702; I3 = 1431655703;
I4 = 1431655704; I5 = 1431655705; I6 = 1431655706; I7 = 1431655707; S = 2;
#5 $write("I0:%d, I1:%d, I2:%d, I3:%d, I4:%d, I5:%d, I6:%d, I7:%d, S:%d Y:%d\n",
I0, I1, I2, I3, I4, I5, I6, I7, S, Y);
I0 = 1431655700; I1 = 1431655701; I2 = 1431655702; I3 = 1431655703;
I4 = 1431655704; I5 = 1431655705; I6 = 1431655706; I7 = 1431655707; S = 3;
#5 $write("I0:%d, I1:%d, I2:%d, I3:%d, I4:%d, I5:%d, I6:%d, I7:%d, S:%d Y:%d\n",
I0, I1, I2, I3, I4, I5, I6, I7, S, Y);
I0 = 1431655700; I1 = 1431655701; I2 = 1431655702; I3 = 1431655703;
I4 = 1431655704; I5 = 1431655705; I6 = 1431655706; I7 = 1431655707; S = 4;
#5 $write("I0:%d, I1:%d, I2:%d, I3:%d, I4:%d, I5:%d, I6:%d, I7:%d, S:%d Y:%d\n",
I0, I1, I2, I3, I4, I5, I6, I7, S, Y);
I0 = 1431655700; I1 = 1431655701; I2 = 1431655702; I3 = 1431655703;
I4 = 1431655704; I5 = 1431655705; I6 = 1431655706; I7 = 1431655707; S = 5;
#5 $write("I0:%d, I1:%d, I2:%d, I3:%d, I4:%d, I5:%d, I6:%d, I7:%d, S:%d Y:%d\n",
I0, I1, I2, I3, I4, I5, I6, I7, S, Y);
I0 = 1431655700; I1 = 1431655701; I2 = 1431655702; I3 = 1431655703;
I4 = 1431655704; I5 = 1431655705; I6 = 1431655706; I7 = 1431655707; S = 6;
#5 $write("I0:%d, I1:%d, I2:%d, I3:%d, I4:%d, I5:%d, I6:%d, I7:%d, S:%d Y:%d\n",
I0, I1, I2, I3, I4, I5, I6, I7, S, Y);
I0 = 1431655700; I1 = 1431655701; I2 = 1431655702; I3 = 1431655703;
I4 = 1431655704; I5 = 1431655705; I6 = 1431655706; I7 = 1431655707; S = 7;
#5 $write("I0:%d, I1:%d, I2:%d, I3:%d, I4:%d, I5:%d, I6:%d, I7:%d, S:%d Y:%d\n",
I0, I1, I2, I3, I4, I5, I6, I7, S, Y);
end

endmodule
```

Now with eight inputs, there must be eight tests to complete. Once again, like the previous MUX test, the last two digits of the input values will correspond with the selection input for ease of checking.

7) 32-bit 16x1 MUX

Moving up further by a multiple of two each time, the 32-bit 8x1 MUX will now be upgraded to a 32-bit 16x1 MUX. Each time the number of inputs is doubled, another selection wire can be expected as well. The following schematic of the 16x1 MUX is:



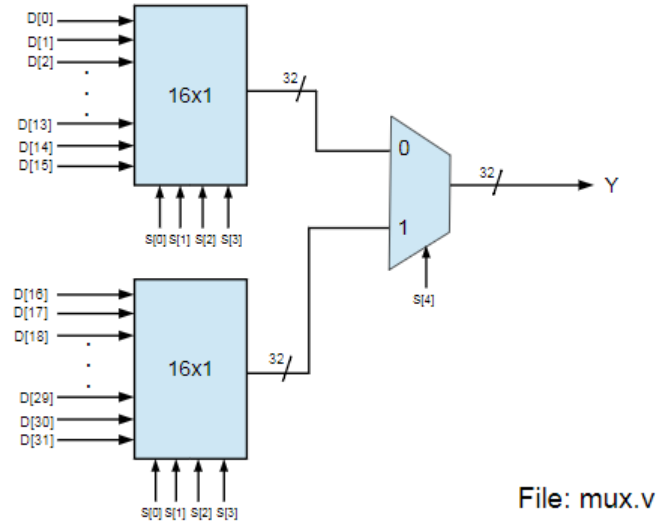
Using two previous MUX's and one 2x1 MUX, the process should be very simple now.

```
wire [31:0] MUX_result_1;
wire [31:0] MUX_result_2;

MUX32_8x1 mux_32_8x1_inst1(.Y(MUX_result_1), .I0(I0), .I1(I1),.I2(I2),
        .I3(I3), .I4(I4), .I5(I5), .I6(I6), .I7(I7), .S(S[2:0]));
MUX32_8x1 mux_32_8x1_inst2(.Y(MUX_result_2), .I0(I8), .I1(I9), .I2(I10),
        .I3(I11), .I4(I12), .I5(I13), .I6(I14), .I7(I15), .S(S[2:0]));
MUX32_2x1 mux_32_2x1_inst1(.Y(Y), .I0(MUX_result_1), .I1(MUX_result_2), .S(S[3]));
```

Aside from the much larger range of inputs, the instantiation of two 8x1 MUX's and their results are stored. Then those results are wired into the final 2x1 MUX to complete the 32-bit 16x1 MUX. By now, the approach for testing MUX's should be understandable. Using the same structure as the previous two testbenches, sixteen tests will be run for the 16x1 MUX. Due to the length of this test code, it will not be pasted in the report. The testing results will be still shown in the testing section.

8) 32-bit 32x1 MUX

As the final upgrade, the 32-bit 16x1 MUX's will be used to create a 32-bit 32x1 MUX. Jumping right into it, the schematic is as shown:



Once again using the previous MUX's and a 2x1 MUX, the 32-bit 32x1 MUX is no different. The code is intuitive.

```
wire [31:0] MUX_result_1;
wire [31:0] MUX_result_2;

MUX32_16x1 mux32_16x1_inst1(.Y(MUX_result_1), .I0(I0), .I1(I1), .I2(I2), .I3(I3),
        .I4(I4), .I5(I5), .I6(I6), .I7(I7), .I8(I8), .I9(I9), .I10(I10),
        .I11(I11), .I12(I12), .I13(I13), .I14(I14), .I15(I15), .S(S[3:0]));
MUX32_16x1 mux32_16x1_inst2(.Y(MUX_result_2), .I0(I16), .I1(I17), .I2(I18), .I3(I19), .I4(I20),
        .I5(I21), .I6(I22), .I7(I23), .I8(I24), .I9(I25), .I10(I26),
        .I11(I27), .I12(I28), .I13(I29), .I14(I30), .I15(I31), .S(S[3:0]));
MUX32_2x1 mux32_2x1_inst1(.Y(Y), .I0(MUX_result_1), .I1(MUX_result_2), .S(S[4]));
```

Using two 16x1 MUX's to wire their results into a 2x1 MUX, the 32-bit 32x1 MUX has been successfully implemented, using dozens of smaller MUX's in the process. Because of this reliance on several smaller components, it is vital to test the MUX's at every level, from the bottom-up. Testbenches have been written for each MUX, so the 32-bit 32x1 MUX is the final one. Testing for this mux is the same as the previous ones, except that thirty-two tests must be run, each with thirty-two inputs and a selection choice. Because of the length of this code, it will not be included in this report. Its test results, however, will still be included in the testing section.

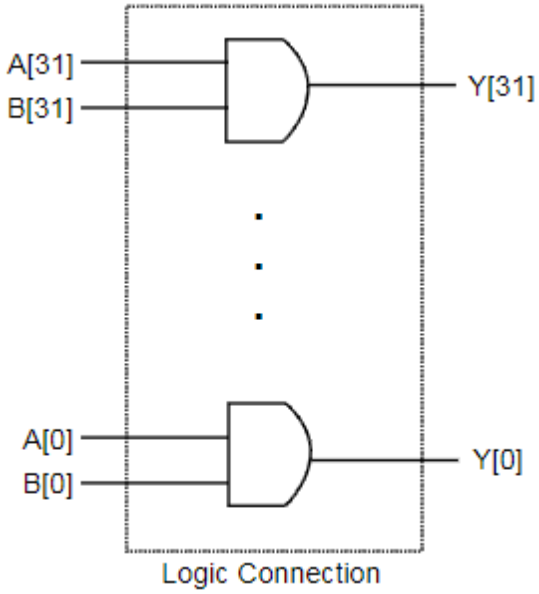
E. Logic

For any device, its logic is the most vital feature, allowing for the device have a variety of features. In the DaVinci v1.0m system, logic gates and 2's complement are important logical components.

Starting with logic gates, they are what allow the system to alter values, one bit at a time. While there are many logic gates, only a few of them will require an upgrade in order to be able to handle 32-bit inputs and outputs. Each of the following logic gates will receive this upgrade.

1) AND

The AND gate receives two inputs and results in one output. Below is a schematic of a 32-bit design:



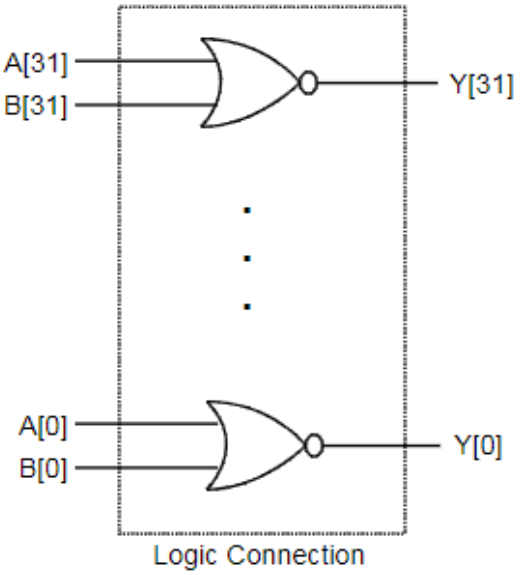
Similar to previous components where several components are created at the same time, it can be understood that a generation loop will be used.

```
// generates 32 AND's
genvar i;
generate
        for (i = 0; i < 32; i = i + 1)
        begin: and_32bit_generation_loop
                and and_inst1(Y[i], A[i], B[i]);
        end
endgenerate
```

As seen, the generation loop runs 32 times, each time instantiating a new AND gate. Using 'i' as the index, each AND gate will store the AND gate result of the i'th bit of A and B into the i'th bit of Y.

2) NOR

Similar to the AND gate, the NOR gate also receives two inputs and results in one output. Below is the schematic of a 32-bit design:



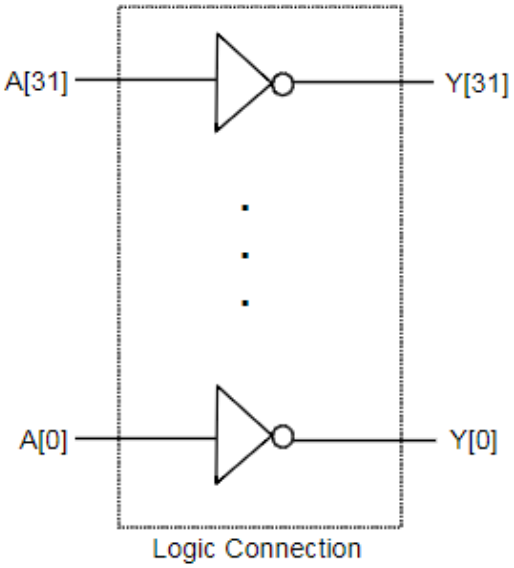
The 32-bit NOR gate will follow the same approach of the 32-bit AND gate through the usage of a generation loop.

```
// generates 32 NOR's
genvar i;
generate
        for (i = 0; i < 32; i = i + 1)
        begin: nor_32bit_generation_loop
                nor nor_inst1(Y[i], A[i], B[i]);
        end
endgenerate
```

Generating 32 NOR gates and storing the results of each corresponding pair of bits in 'A' and 'B' in the same index of 'Y', the 32-bit NOR gate has been implemented

3) INV

The INV gate receives one input and results in one output. Below is the schematic of a 32-bit design:



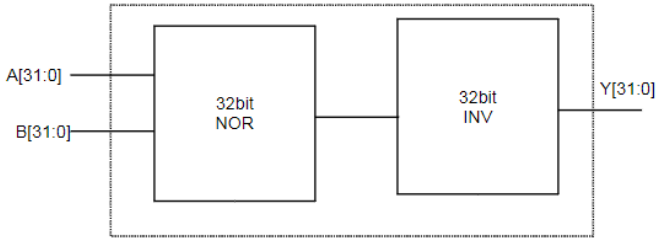
Once again, the schematic shows that a generation loop can be used for this 32-bit gate.

```
// generates 32 NOT's
genvar i;
generate
        for (i = 0; i < 32; i = i + 1)
        begin: inv_32bit_generation_loop
                not inv_inst1(Y[i], A[i]);
        end
endgenerate
```

Unlike AND and NOR, inverting with NOT only requires one input and one output. Generating 32 NOT's and providing an input and output bit completes the 32-bit INV's implementation.

4) OR

Lastly, the OR gate takes in two inputs and results in one output. It is special in the fact that it can be created through the usage of NOR and INV, two components also upgraded. Below is the schematic of a 32-bit design using this trait:



Different from the previous schematics, the 32-bit OR gate will not require any generation loop. Instead, a simple combinational circuit is all that is necessary.

```
// 32-bit OR
module OR32_2x1(Y,A,B);
//output
output [31:0] Y;
//input
input [31:0] A;
input [31:0] B;

wire [31:0] nor_result;

NOR32_2x1 nor32_2x1_inst1(.Y(nor_result), .A(A), .B(B));
INV32_1x1 inv32_1x1_inst1(.Y(Y), .A(nor_result));

endmodule
```

Taking advantage of the previously implemented 32-bit NOR gate and 32-bit INV gate, the schematic can easily be implemented. By NOR'ing the two 32-bit values and then INV'ing the result of the NOR gate, the 32-bit OR gate has been quickly implemented.

5) BUF

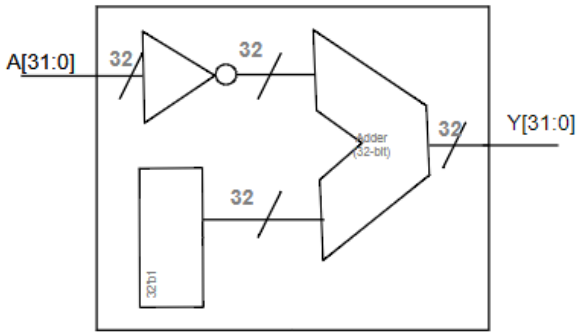
Though unnecessary for the DaVinci v1.0m system, having a 32-bit BUF gate will be very helpful in loading values. Rather than using a for-loop every time, BUF gates can be used to directly load values onto inputs, outputs, or wires. Though there is no schematic, the 32-bit is the same as the 32-bit INV gate, but with BUF gates instead.

```
// generates 32 BUF's
genvar i;
generate
        for (i = 0; i < 32; i = i + 1)
        begin: buf_32bit_generation_loop
                buf buf_inst1(Y[i], A[i]);
        end
endgenerate
```

Using the same exact same structure as the 32-bit INV gate, except with the instantiation of the BUF gates, the 32-bit BUF gate is implemented.

6) 2's complement

2's complement is a format for binary numbers in which the largest bit represents if it is a negative number or not. If the bit is a 0, then it is positive; else 1 is negative. A useful feature of 2's complement is that adding or subtracting 2's complement numbers is easily handled and functions the same as unsigned addition or subtraction. For this reason, the ability to convert unsigned numbers into 2's complement numbers is very useful. Provided is a schematic of a component that will convert from unsigned to 2's complement:



Logic Circuit

With 32-bit value 'A' as the unsigned input number and 32 bit value 'Y' as the 2's complemented output number, the process in-between is fairly simple. First, 'A' is inverted using the previously implemented 32-bit INV. Then the now inverted 'A' is put into a 32-bit adder that will add a 32-bit value of all 1's to it. The output of the addition is the resultant 2's complement form of 'A'.

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

wire [31:0] A_inv;
INV32_1x1 inv32_inst1(A_inv, A);

wire CO;
RC_ADD_SUB_32 RC_add_sub_32_inst1(.Y(Y), .CO(CO), .A(A_inv), .B(32'b1), .SnA(1'b0));

endmodule
```

A wire for the inverted 'A' is created for later connecting into the adder. 'A' is then inverted and stored. Once that is done, all that is left is adding. Since the number is 32-bit, the Ripple-Carry Adder will be used. Taking in 'A_inv', a 32-bit value of all 1's, and the 'SnA' value of 0, the 2's complemented number is calculated and stored in the output 'Y'. To test the 2's complement component, a value must be passed into the component and checked to see if each of its bits is inverted.

```
module TWOS_COMP32_TB;
//output list
wire [31:0] Y;
//input list
reg [31:0] A;

TWOSCOMP32 twoscomp32_inst1(.Y(Y), .A(A));

initial
begin
A = 0;
#5 $write("A:%b, Y:%b\n", A, Y);
A = 1;
#5 $write("A:%b, Y:%b\n", A, Y);
A = -1;
#5 $write("A:%b, Y:%b\n", A, Y);
A = 1431655765;
#5 $write("A:%b, Y:%b\n", A, Y);
A = -1431655766;
#5 $write("A:%b, Y:%b\n", A, Y);
end

endmodule
```

As shown by the code, it is a simple instantiation of the 2's complement component, in which different values are passed into it to check its outputs.

It is also useful to be able to convert unsigned 64-bit values to 2's complement as well. The schematic and design is the same, except that it will handle 64 bits instead now.

```
// 64-bit two's complement
module TWOSCOMP64(Y,A);
//output list
output [63:0] Y;
//input list
input [63:0] A;

wire [63:0] A_inv;
INV32_1x1 inv32_inst1(A_inv[31:0], A[31:0]);
INV32_1x1 inv32_inst2(A_inv[63:32], A[63:32]);

wire CO;
RC_ADD_SUB_64 rc_add_sub_64_inst1(.Y(Y), .CO(CO), .A(A_inv), .B(64'b1), .SnA(1'b0));

endmodule
```

Using two 32-bit INV's, the first and second 32-bits of 'A' are inverted and stored on a wire. The 64-bit Ripple Carry Adder is then instantiated, taking in the same inputs as the 32-bit Ripple Carry Adder. Once complete, the newly 2's complemented 64-bit value is stored in 'Y'. Testing will not be performed for the 64-bit 2's complement component as the implementation of it is essentially identical to that of the 32-bit.

F. Multiplier

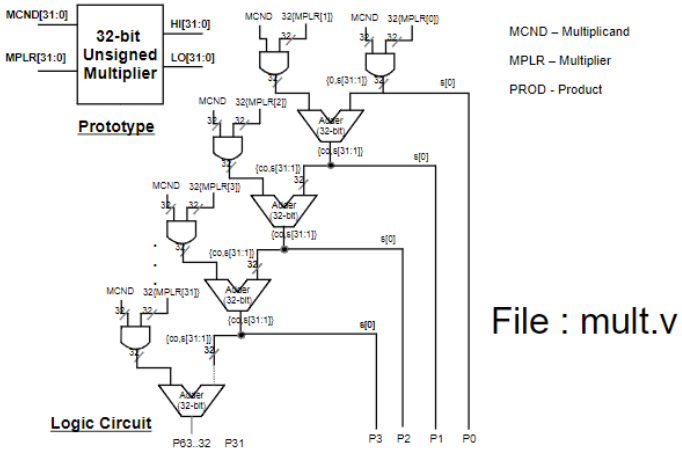
Multiplication can be understood as an extension of addition. When calculating multiplication on paper, the process uses multiplication and then addition to acquire the final answer. An example is as shown:

```
      1000  <- Multiplicand
    X 1001  <- Multiplier
-------------
      1000
+    00000
+   000000
+  1000000
-------------
   1001000  <- Product
```

Using this same strategy, multiplication can be implemented in this way. However, unlike addition and subtraction which uses 2's complement, not all multiplication performed is signed.

1) 32-bit Unsigned Multiplier

Taking in two inputs that are both 32-bits, the result will then be calculated and stored in two 32-bit registers, 'high', and 'low', as a 32-bit value times another 32-bit value may potentially result in a 64-bit value. This must be stored across two 32-bit registers. Shown below is the schematic of such an unsigned multiplication component that takes advantage of the previously introduced addition mechanic:



File : mult.v

The schematic depicts a stacked usage of Ripple Carry adders to perform unsigned multiplication. Implementing unsigned multiplication will require many AND's and 32-bit adders, meaning generation should be used in this scenario as well.

```
wire carry [31:0];
wire [31:0] result [31:0];

AND32_2x1 and32_2x1_inst1(result[0], A, {32{B[0]}});
buf buf_inst1(carry[0], 1'b0);
buf buf_inst2(LO[0], result[0][0]);

genvar i;
generate
        for (i = 1; i < 32; i = i + 1)
        begin : adder_generation_loop
                wire [31:0] and_res;
                AND32_2x1 and32_2x1_inst(and_res, A, {32{B[i]}});
                RC_ADD_SUB_32 rc_add_sub_32_inst(.Y(result[i]), .CO(carry[i]),
                .A(and_res), .B({carry[i - 1], result[i - 1][31:1]}), .SnA(1'b0));
                buf buf_inst(LO[i], result[i][0]);
        end
endgenerate

BUF32_2x1 buf32_2x1_inst1(.Y(HI), .A({carry[31], result[31][31:1]}));
```

Creating two wires, 'carry' and 'result' for addition results, the first layer of 'result' is first set to be the 32-bit AND result of 'A' and 32 bits of 'B''s first bit. The first value of 'carry' is then set to '0' while the first bit of 'LO' is set to 'result[0][0]' in preparation for the multiplication process.

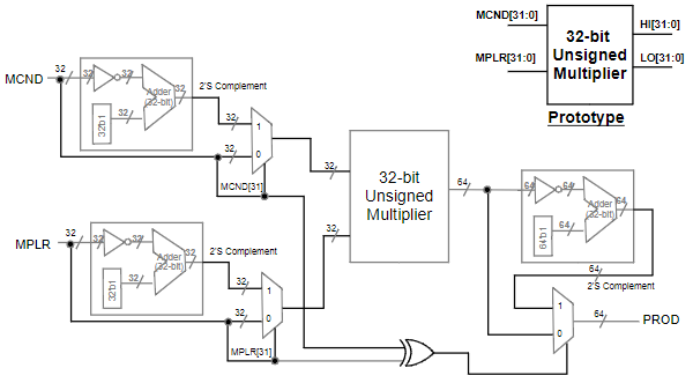
Once this is complete, the generation loop begins. Within the loop that runs thirty-one times, since the first bit is calculated without any adders, a 32-bit AND operation is performed, followed by an adder. Wires and 'buf' are used to contain and store the results. Through this process, the chained AND and adders are created with every loop, fulfilling the schematic's design. To test the unsigned multiplier, it needs to be instantiated and given the appropriate fields: the multiplicand, multiplier, 'HI' output wire, and 'LO' output wire.

```
module MULT32_U_TB;
// output list
wire [31:0] HI;
wire [31:0] LO;
// input list
reg [31:0] A;
reg [31:0] B;

MULT32_U mult32_u_inst1(.HI(HI), .LO(LO), .A(A), .B(B));

initial
begin
A = 0; B = 0;
#5 $write("A:%d, B:%d, HI:%d, LO:%d\n", A, B, HI, LO);
A = 2; B = 5;
#5 $write("A:%d, B:%d, HI:%d, LO:%d\n", A, B, HI, LO);
A = 5; B = 2;
#5 $write("A:%d, B:%d, HI:%d, LO:%d\n", A, B, HI, LO);
A = 3781; B = 7132;
#5 $write("A:%d, B:%d, HI:%d, LO:%d\n", A, B, HI, LO);
A = 4294967295; B = 4294967295;
#5 $write("A:%d, B:%d, HI:%d, LO:%d\n", A, B, HI, LO);
end

endmodule
```

As described, the unsigned multiplier is instantiated while different 'A' and 'B' values are tested to be multiplied to one another. It is important to note that all these values are unsigned, positive values. Including a negative value will produce incorrect results. Testing will be shown in the later section.

2) 32-bit Signed Multiplier

Having the same inputs and registers as the unsigned multiplier, the only difference is that the numbers being handle are now signed. This introduces the new issue of handling negative and positive numbers, especially since there are two result registers, 'high' and 'low'. The following schematic shows the layout of the 32-bit signed multiplier:



The 32-bit signed multiplier has a very simple implementation, relying on several previously implemented components. All that must be done is correctly wiring each component to the next to achieve the correct multiplication output.

```
// 2's complement multiplicand and multiplier
wire [31:0] twos_mcnd;
wire [31:0] twos_mplr;
TWOSCOMP32 twoscomp32_inst1(.Y(twos_mcnd), .A(A));
TWOSCOMP32 twoscomp32_inst2(.Y(twos_mplr), .A(B));

// MUX to choose between non-complemented or 2's complemented based on original first bits
wire [31:0] mux_mcnd;
wire [31:0] mux_mclr;
MUX32_2x1 mux32_2x1_inst1(.Y(mux_mcnd), .I0(A), .I1(twos_mcnd), .S(A[31]));
MUX32_2x1 mux32_2x1_inst2(.Y(mux_mclr), .I0(B), .I1(twos_mplr), .S(B[31]));

// multiply unsigned
wire [63:0] multu_result;
MULT32_U mult32_u_inst1(.HI(multu_result[63:32]), .LO(multu_result[31:0]), .A(mux_mcnd), .B(mux_mclr));

// 2's complement of unsigned multiplication result
wire [63:0] mult_result;
TWOSCOMP64 twoscomp64_inst1(.Y(mult_result), .A(multu_result));

// XOR the original first bits to choose signed or unsignedd
wire xor_result;
xor xor_inst1(xor_result, A[31], B[31]);

// MUX to choose complement or non-complement for multiplication result
wire [63:0] mux_result;
MUX64_2x1 mux64_2x1_inst1(.Y(mux_result), .I0(multu_result), .I1(mult_result), .S(xor_result));

// store first 32 bits in HI and last 32 bits in LO
BUF32_2x1 buf32_inst1(.Y(HI), .A(mux_result[63:32]));
BUF32_2x1 buf32_inst2(.Y(LO), .A(mux_result[31:0]));
```

For the signed multiplier, implementation is a matter of connecting the correct wires to the correct components, rather than the unsigned multiplier's need for the generation loop to complete the schematic. Many wires are instantiated in order to achieve this schematic's design.

The multiplicand 'A' and multiplier 'B' are first 2's complemented and passed through a MUX, along with their non-complemented form as the second input option. The MUX's selection wire is dependent on the last bit of 'A' and 'B' respectively, as having the first bit as a '1' denotes that the number is signed, so the 2's complemented form should be selected.

The two fields, now potentially 2's complemented or not, are passed into the 32-bit unsigned multiplier. The 64-bit result is then taken and complemented with the 64-bit 2's complement component. Once done, it is passed into a 64-bit 2x1 MUX that decides between the original unsigned multiplier result or the newly 2's complemented result. This decision is based on the selection wire, which is calculated from the XOR'ing of the first bit of both 'A' and 'B'. Finally, the result of multiplication will be BUF'ed into 'HI' and 'LO' because setting 'HI' and 'LO' equal to the result does not work. Instead, BUF essentially copies the results to the outputs. For testing the 32-bit signed multiplier, the same approach can be taken as the unsigned multiplier, except that negative values can now be tested.

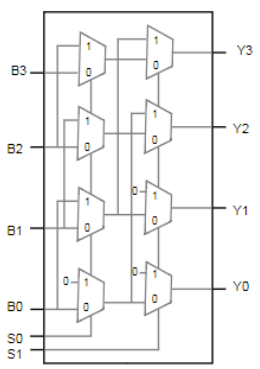
```
module MULT32_TB;
// output list
wire [31:0] HI;
wire [31:0] LO;
// input list
reg [31:0] A;
reg [31:0] B;

MULT32 mult32_inst1(.HI(HI), .LO(LO), .A(A), .B(B));

initial
begin
A = 0; B = 0;
#5 $write("A:%d, B:%d, HI:%d, LO:%d\n", A, B, HI, LO);
A = 2; B = 5;
#5 $write("A:%d, B:%d, HI:%d, LO:%d\n", A, B, HI, LO);
A = 5; B = 2;
#5 $write("A:%d, B:%d, HI:%d, LO:%d\n", A, B, HI, LO);
A = 3781; B = 7132;
#5 $write("A:%d, B:%d, HI:%d, LO:%d\n", A, B, HI, LO);
A = 4294967295; B = 4294967295;
#5 $write("A:%d, B:%d, HI:%d, LO:%d\n", A, B, HI, LO);
A = -2; B = 5;
#5 $write("A:%d, B:%d, HI:%d, LO:%d\n", A, B, HI, LO);
A = -5; B = 2;
#5 $write("A:%d, B:%d, HI:%d, LO:%d\n", A, B, HI, LO);
A = 3781; B = -7132;
#5 $write("A:%d, B:%d, HI:%d, LO:%d\n", A, B, HI, LO);
A = -4294967295; B = -4294967295;
#5 $write("A:%d, B:%d, HI:%d, LO:%d\n", A, B, HI, LO);
end

endmodule
```

Reusing the previous test cases from the unsigned multiplier to ensure that the results are still correct, the test cases are also given negatives in order to test the new capability. One, both, or none of the arguments are given negatives to check all possible positive/negative input combinations. The test results will be detailed later in the report.

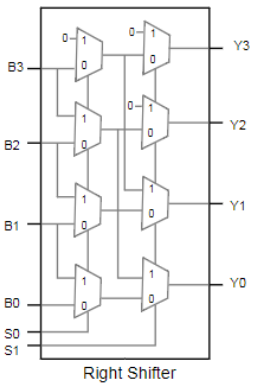
G. Shifter

A shifter is a component that takes the input number and shifts it by an amount dictated. To create a shifter that can shift both right and left, multiple components must first be implemented.

In order to shift 32-bits, a 32-bit shifter for both right and left are required. Though the schematic for the 32-bit component is not provided, below is the schematic for a 4-bit that can be edited and expanded to handle 32-bits:



Left Shifter



Right Shifter

As seen in the schematic, four bits, ‘B0’ to ‘B3’, are used. Along with those four bits, there are two selection wires, allowing for four possible choices. For each bit, there are two MUX’s. The MUX inputs that are just ‘0’ are dependent on if it is a left or right shift. With this knowledge, it will be expanded to handle 32-bits.

1) Right barrel shift

The right barrel shift moves the value ‘n’ bits to the right, placing 0’s at the front where the previous bits once were. Following the schematic, the 32-bit code is:

```
wire [31:0] result [3:0];

genvar i, j;
generate
        for (i = 0; i < 32; i = i + 1)
        begin : mux_generation_loop_1
                if (i == 31)
                begin
                        MUX1_2x1 mux_inst_1(.Y(result[0][i]), .I0(D[i]), .I1(1'b0), .S(S[0]));
                end
                else
                begin
                        MUX1_2x1 mux_inst_2(.Y(result[0][i]), .I0(D[i]), .I1(D[i + 1]), .S(S[0]));
                end
        end

        for (i = 1; i < 4; i = i + 1)
        begin : mux_generation_loop_2
                for (j = 0; j < 32; j = j + 1)
                begin : mux_generation_loop_2_inner
                        if (j > 31 - (2 ** i))
                        begin
                                MUX1_2x1 mux_inst_3(.Y(result[i][j]), .I0(result[i - 1][j]), .I1(1'b0), .S(S[i]));
                        end
                        else
                        begin
                                MUX1_2x1 mux_inst_4(.Y(result[i][j]), .I0(result[i - 1][j]), .I1(result[i - 1][j + 2 ** i]),
                        end
                end
        end

        for (i = 0; i < 32; i = i + 1)
        begin : mux_generation_loop_3
                if (i > 15)
                begin
                        MUX1_2x1 mux_inst_5(.Y(Y[i]), .I0(result[3][i]), .I1(1'b0), .S(S[4]));
                end
                else
                begin
                        MUX1_2x1 mux_inst_6(.Y(Y[i]), .I0(result[3][i]), .I1(result[3][i + 16]), .S(S[4]));
                end
        end
endgenerate
```

As seen in the schematic, there are multiple layers to barrel shifters. For 4-bit shifting, there are 2 layers. Following this logic, for a 32-bit barrel shifter, there will be 5 layers. From each layer, the MUX output is sent to the next layer’s MUX, to shift by the corresponding number of bits for that layer. To test the results of the right shift, values are passed in with a shift amount. The output shifted value will be checked against the two inputs to validate accuracy.

```
module SHIFT32_R_TB;
// output list
wire [31:0] Y;
// input list
reg [31:0] D;
reg [4:0] S;

SHIFT32_R shift32_r_inst1(.Y(Y), .D(D), .S(S));

initial
begin
D = 0; S = 0;
#5 $write("D:%d, S:%d, Y:%d\n", D, S, Y);
D = 1; S = 1;
#5 $write("D:%d, S:%d, Y:%d\n", D, S, Y);
D = 15; S = 2;
#5 $write("D:%d, S:%d, Y:%d\n", D, S, Y);
D = 200; S = 3;
#5 $write("D:%d, S:%d, Y:%d\n", D, S, Y);
D = 2147483647; S = 10;
#5 $write("D:%d, S:%d, Y:%d\n", D, S, Y);
end

endmodule
```

The testbench code instantiates the right shift and gives a variety of values and shift amounts to test. Test results will be shown later.

2) Left barrel shift

The left barrel shift is similar to the right shift. For the code, it is essentially the same, except for where the ‘0’ input for the MUX will be.

```
wire [31:0] result [3:0];

genvar i, j;
generate
        for (i = 0; i < 32; i = i + 1)
        begin : mux_generation_loop_1
                if (i == 0)
                begin
                        MUX1_2x1 mux_inst_1(.Y(result[0][i]), .I0(D[i]), .I1(1'b0), .S(S[0]));
                end
                else
                begin
                MUX1_2x1 mux_inst_2(.Y(result[0][i]), .I0(D[i]), .I1(D[i - 1]), .S(S[0]));
                end
        end

        for (i = 1; i < 4; i = i + 1)
        begin : mux_generation_loop_2
                for (j = 0; j < 32; j = j + 1)
                begin : mux_generation_loop_2_inner
                        if (j < 2 ** i)
                        begin
                                MUX1_2x1 mux_inst_3(.Y(result[i][j]), .I0(result[i - 1][j]), .I1(1'b0), .S(S[i]));
                        end
                        else
                        begin
                                MUX1_2x1 mux_inst_4(.Y(result[i][j]), .I0(result[i - 1][j]), .I1(result[i - 1][j - 2 ** i]), .S(S[i]));
                        end
                end
        end

        for (i = 0; i < 32; i = i + 1)
        begin : mux_generation_loop_3
                if (i < 16)
                begin
                        MUX1_2x1 mux_inst_5(.Y(Y[i]), .I0(result[3][i]), .I1(1'b0), .S(S[4]));
                end
                else
                begin
                        MUX1_2x1 mux_inst_6(.Y(Y[i]), .I0(result[3][i]), .I1(result[3][i - 16]), .S(S[4]));
                end
        end
endgenerate
```

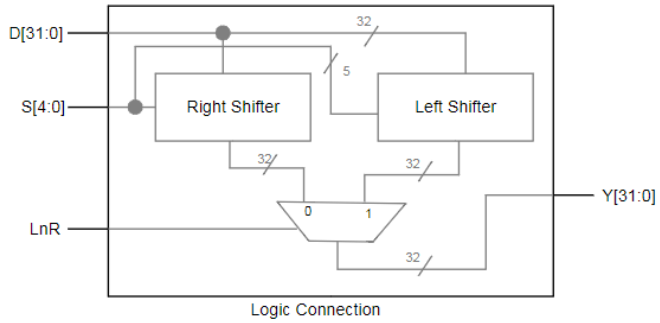
The left shift code is essentially a mirror of the right shift code. The only difference is where the ‘0’ bit input is placed into the multiplexers. Since it is left shift, the ‘0’’s will be on the right side of the bits, since those are the bits that are displaced. Same as the right shift testbench, the left shift will also test cases and check to make sure the result is correct.

```
module SHIFT32_L_TB;
// output list
wire [31:0] Y;
// input list
reg [31:0] D;
reg [4:0] S;

SHIFT32_L shift32_l_inst1(.Y(Y), .D(D), .S(S));

initial
begin
D = 0; S = 0;
#5 $write("D:%d, S:%d, Y:%d\n", D, S, Y);
D = 1; S = 1;
#5 $write("D:%d, S:%d, Y:%d\n", D, S, Y);
D = 15; S = 2;
#5 $write("D:%d, S:%d, Y:%d\n", D, S, Y);
D = 200; S = 3;
#5 $write("D:%d, S:%d, Y:%d\n", D, S, Y);
D = 2147483647; S = 10;
#5 $write("D:%d, S:%d, Y:%d\n", D, S, Y);
end

endmodule
```

Using the same test cases as the right shift, the test is to now check if the same values are able to be computed with a left shift instead. The test results will be detailed later.

3) 32-bit barrel shifter with Left or Right control

Combining the previous two shifting components results in a 32-bit barrel shifter that has Left or Right control. The schematic that makes use of the left and right shift is as shown:



Logic Connection

The schematic shows the usage of the right shifter, left shifter, and a 32-bit 2x1 MUX. This implementation will only require some instantiations and wiring.

```
wire [31:0] shift32r_result;
wire [31:0] shift32l_result;

SHIFT32_R shift32_r_inst1(.Y(shift32r_result), .D(D), .S(S));
SHIFT32_L shift32_l_inst1(.Y(shift32l_result), .D(D), .S(S));
MUX32_2x1 mux32_2x1_inst1(.Y(Y), .I0(shift32r_result), .I1(shift32l_result), .S(LnR));
```

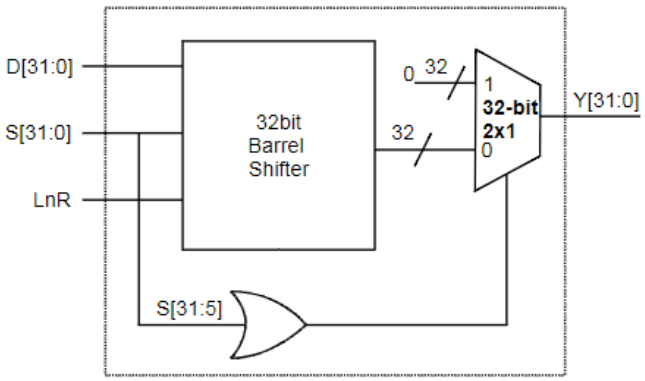
Two wires are created for the result of the right shift and left shift. Once those results are stored on the wires, they are passed into the MUX. In order to select which shift result as the output result, 'LnR' is passed in as the selection wire. Once the MUX chooses a result, the component is complete. As for testing, instead of testing only right or left shift, the testbench for the 32-bit barrel shifter with Left or Right control will try both shifts.

```
module BARREL_SHIFTER32_TB;
// output list
wire [31:0] Y;
// input list
reg [31:0] D;
reg [4:0] S;
reg LnR;

BARREL_SHIFTER32 barrel_shifter32_inst1(.Y(Y), .D(D), .S(S), .LnR(LnR));

initial
begin
D = 0; S = 0; LnR = 0;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 0; S = 0; LnR = 1;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 1; S = 1; LnR = 0;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 1; S = 1; LnR = 1;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 15; S = 2; LnR = 0;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 15; S = 2; LnR = 1;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 200; S = 3; LnR = 0;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 200; S = 3; LnR = 1;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 2147483647; S = 10; LnR = 0;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 2147483647; S = 10; LnR = 1;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
end
```

Once again reusing the same test cases from the right and left shift, this testbench tests both shifting right and left. If the results are the same as the results seen for the right and left shift individually, then the component function correctly. Test results are posted in the later section.

4) 32-bit shifter

The 32-bit shifter is composed of the previously implemented 32-bit barrel shifter and a 32-bit 2x1 MUX. The difference between this shifter and the previous barrel shifter is its ability to provide the output of a 32-bit value of all '0's if the shifting amount is more than '32'. Since all 32-bit values shifted by 32 or higher result in '0', the MUX will simply choose the 32-bit '0' value if that is the case. The schematic is as shown:



Calculating the 32-bit barrel shift as usual, its results are then wired to MUX input '0'. Input '1' is a 32-bit '0' value. The selection of the MUX is the OR of the 6th through 32nd bits of 'S'.

```
wire [31:0] shift_result;
BARREL_SHIFTER32 barrel_shifter32_inst(.Y(shift_result), .D(D), .S(S[4:0]), .LnR(LnR));

wire [31:5] or_result;
or or_inst1(or_result[5], S[5], S[5]);
genvar i;
generate
        for (i = 6; i <= 31; i = i + 1)
        begin : or_generation_loop
                or or_inst2(or_result[i], S[i], or_result[i - 1]);
        end
endgenerate

MUX32_2x1 mux32_2x1_inst1(.Y(Y), .I0(shift_result), .I1(32'b0), .S(or_result[31]));
```

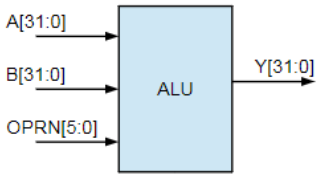
The 32-bit barrel shifter with left and right control is first instantiated with a wire to contain its results. Once done, the next step is to get the result of OR'ing the 6th to 32nd bit of 'S'. This is performed by performing an initial 'OR', followed by a generation loop of OR's. Once complete, the result of the OR is passed into a 32-bit 2x1 MUX as the selection wire, while the two inputs are the barrel shift result and a 32-bit '0' value. To test the 32-bit shifter, the same test cases from the 32-bit barrel shifter will be used since the results will be the same if implemented correctly.

```
module SHIFT32_TB;
// output list
wire [31:0] Y;
// input list
reg [31:0] D;
reg [31:0] S;
reg LnR;

SHIFT32 shift32_inst1(.Y(Y), .D(D), .S(S), .LnR(LnR));

initial
begin
D = 0; S = 0; LnR = 0;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 0; S = 0; LnR = 1;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 1; S = 1; LnR = 0;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 1; S = 1; LnR = 1;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 15; S = 2; LnR = 0;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 15; S = 2; LnR = 1;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 200; S = 3; LnR = 0;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 200; S = 3; LnR = 1;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 2147483647; S = 10; LnR = 0;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
D = 2147483647; S = 10; LnR = 1;
#5 $write("D:%d, S:%d, LnR:%d, Y:%d\n", D, S, LnR, Y);
end

endmodule
```

Using the same test cases, the results of the test will be detailed later in the report.

H. Arithmatic Logic Unit (ALU)

The ALU is the culmination of all the components implemented thus far. It is able to handle adding, subtracting, multiplication, shifting, AND, OR, NOR, and SLT. In this project, the ALU has a sequential digital circuit design. Below is a diagram that explains the details of the ALU:

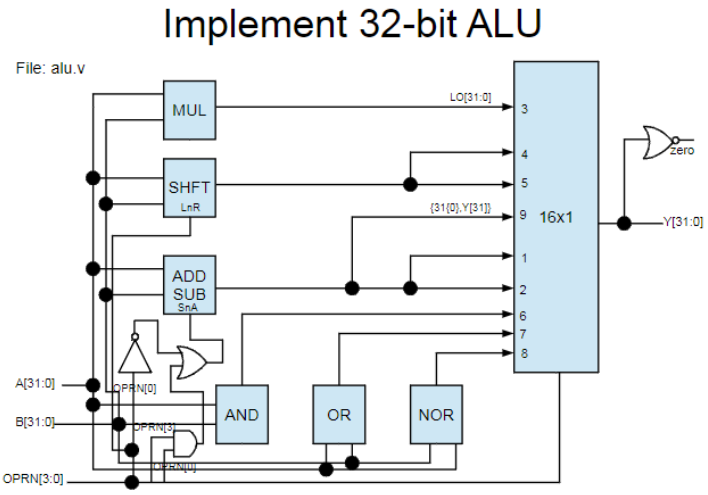


TYPE	OPRN CODE
add	xx0001
sub	xx0010
mul	xx0011
Shift R	xx0100
Shift L	xx0101
and	xx0110
or	xx0111
nor	xx1000
slt	xx1001

Control Signals:

- For Adder-Subtractor & SLT → SnA : OPRN[0]' + OPRN[3].OPRN[0]
- For shifter → LnR : OPRN[0]

As shown, there are specific operation codes per function. Depending on the function, the value that must be used is a very specific bit or bits. For example, if it is the shift operation, determining if it is a left or right shift is dependent on the 'OPRN''s first bit. The full schematic of the 32-bit ALU is shown below:



Using all previous components: adding/subtraction, 32-bit AND, 32-bit OR, 32-bit NOR, signed/unsigned multiplication, left/right shifting, and a multiplexer to choose the final result, the ALU combines all individual components into one, functioning logic unit. At the very end, the 'ZERO' field, returning the project two, is the NOR of the entire result.

```
wire oprn0_inv;
not not_inst1(oprn0_inv, OPRN[0]);

wire oprn0and3;
and and_inst1(oprn0and3, OPRN[3], OPRN[0]);

wire SnA;
or or_inst1(SnA, oprn0_inv, oprn0and3);

wire [`DATA_INDEX_LIMIT:0] ripple_carry;
wire [`DATA_INDEX_LIMIT:0] NA; // to be used for irrelevant fields

// use NA[0] as CO
RC_ADD_SUB_32 rc_add_sub_32_inst1(.Y(ripple_carry), .CO(NA[0]), .A(OP1), .B(OP2), .SnA(SnA));

// multiply, but only 'lo' required
wire [`DATA_INDEX_LIMIT:0] mul_lo;
MULT32 mult32_inst1(.HI(NA), .LO(mul_lo), .A(OP1), .B(OP2));

wire [`DATA_INDEX_LIMIT:0] shift_result;
SHIFT32 shift32_inst1(.Y(shift_result), .D(OP1), .S(OP2), .LnR(OPRN[0]));

wire [`DATA_INDEX_LIMIT:0] and_result;
AND32_2x1 and32_2x1_inst1(.Y(and_result), .A(OP1), .B(OP2));

wire [`DATA_INDEX_LIMIT:0] or_result;
OR32_2x1 or32_2x1_inst1(.Y(or_result), .A(OP1), .B(OP2));

wire [`DATA_INDEX_LIMIT:0] nor_result;
NOR32_2x1 nor32_2x1_inst1(.Y(nor_result), .A(OP1), .B(OP2));

// place NA in any spots of the MUX that is unused
MUX32_16x1 mux32_16x1_inst1(.Y(OUT), .I0(NA), .I1(ripple_carry), .I2(ripple_carry), .I3(mul_lo),
                            .I4(shift_result), .I5(shift_result), .I6(and_result), .I7(or_result),
                            .I8(nor_result), .I9({31'b0, ripple_carry[31]}), .I10(NA), .I11(NA),
                            .I12(NA), .I13(NA), .I14(NA), .I15(NA), .S(OPRN[3:0]));

// OR then NOT to get ZERO field from OUT
wire [31:0] zero_result;
or or_inst2(zero_result[0], OUT[0], OUT[0]);

genvar i;
generate
        for (i = 1; i < 31; i = i + 1)
        begin : or_generation_loop
                or or_inst3(zero_result[i], OUT[i], zero_result[i - 1]);
        end
endgenerate

wire zero_inv;
or or_inst4(zero_inv, OUT[31], zero_result[30]);
not not_inst2(ZERO, zero_inv);
```

To begin, the 0th bit of 'OPRN' is inverted while the 0th and 3rd bit of 'OPRN' are AND'ed together. Once done, the two resultant bits are OR'ed together to acquire the 'SnA' for the adder/subtractor. Two wires are then created: 'ripple_carry' and 'NA'. They are used to store the result of the add/subtract operation and to store unnecessary values, respectively. For the adder/subtractor, the 'CO' field is not required, so 'NA[0]' is used there. They are then passed into the Ripple-Carry Adder/Subtractor instantiation.

Next is multiplication. It requires the full 'OP1' and 'OP2' values to be multiplied together. However, only the 'LO' bits are required, so 'NA' is filled in the 'HI' field while a newly created wire is used to store the 'LO' results.

For shifting, the barrel-shifter instantiation is created, passing in both 'OP1' and 'OP2', however using 'OPRN[0]' as the 'LnR' field. These fields are then passed into the barrel shifter instantiation.

For the 32-bit AND, OR, and NOR, wires are created for each operation, passing in the wires along with 'OP1' and 'OP2' into their respective 32-bit logic gates.

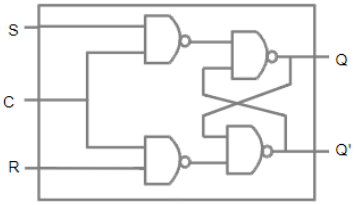
Coming to the 32-bit 16x1 MUX, following the schematic will help determine which result of which component should go into which input slot of the MUX. Whenever a value that is not listed on the schematic appears, 'NA' is used. For the selection wire, it is the first four bits of 'OPRN'. Once instantiated, the result 'Y' is produced from all the inputs.

At the very end, the 'ZERO' field is calculated by OR'ing 'Y' with itself. This is done with an initial OR with the 0th bit and itself, following with a generation loop to OR the rest of bits of 'Y'. Once completed, the value is then inverted onto the output 'ZERO' to complete the ALU.

To test the ALU, the same test bench from the second project will be reused. Testing results will be shown in the later section.

I. SR-Latch

An SR-Latch is a sequential component that uses two logic gates that have mutual feedback. There are two inputs, 'S' and 'R', and two results, 'Q' and 'Q''. SR-Latches can be created using either NOR or NAND, but not both. Some implementations also include an additional input, known as the control. The SR-Latch to be completed in this project is as shown in the schematic:



Truth Table

C	S	R	Q_T	Q_{T+1}	Q'_{T+1}
0	x	x	0	0	1
0	x	x	1	1	0
1	1	0	X	1	0
1	0	1	X	0	1
1	0	0	0	0	1
1	0	0	1	1	0
1	1	1	X	1	1

The SR-Latch shown is the NAND SR-Latch with Control. The left two NAND's are the Control portion of the component while the right two NAND's are the NAND SR-Latch. Implementing this component will be tricky due to its sequential dependency on itself.

```
wire nand_SC;
nand nand_sc_inst(nand_SC, S, C);

wire nand_RC;
nand nand_rc_inst(nand_RC, R, C);

wire nand_sQbar, nand_rQ;

wire and_1, and_2;
and and_inst1(and_1, nP, nand_SC);
and and_inst2(and_2, and_1, nand_rQ);
not not_inst1(nand_sQbar, and_2);

wire and_3, and_4;
and and_inst3(and_3, nR, nand_RC);
and and_inst4(and_4, and_3, nand_sQbar);
not not_inst2(nand_rQ, and_4);

buf buf_inst1(Q, nand_sQbar);
buf buf_inst2(Qbar, nand_rQ);
```

Following the schematic, from left to right, there are first two NAND gates, one between 'S' and 'C' and another between 'C' and 'R'. Each NAND is performed and stored on its respective wire.

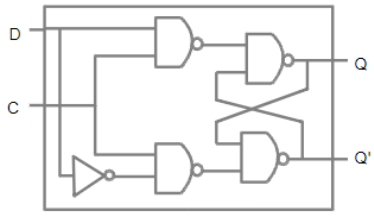
After that, handling the second layer of NANDs is dependent on the inputted 'nP' and 'nR', which are the initial NAND gate values. What happens next is essentially a 3x1 NAND, except handling it 2 inputs at a time, leading to 2 AND's and 1 NOT to complete the 3x1 NAND. This will result in the result 'Q'. Likewise, the same 3x1 NAND is preformed to obtain the result 'Qbar'. These results are then BUF'ed into the outputs. To test the SR-Latch, the truth table can be used to check if every input case leads to the correct output case.

```
// Set
#5 S = 1'b1; R = 1'b0; C = 1'b1; nP = 1'b1; nR = 1'b1;
#5 $write("S:%d, R:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", S, R, C, nP, nR, Q, Qbar);

// Control off, retain values = 1
#5 S = 1'bx; R = 1'bx; C = 1'b0; nP = 1'b1; nR = 1'b1;
#5 $write("S:%d, R:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", S, R, C, nP, nR, Q, Qbar);

// S = R = 0, retain values = 1
#5 S = 1'b0; R = 1'b0; C = 1'b1; nP = 1'b1; nR = 1'b1;
#5 $write("S:%d, R:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", S, R, C, nP, nR, Q, Qbar);

// Reset
#5 S = 1'b0; R = 1'b1; C = 1'b1; nP = 1'b1; nR = 1'b1;
#5 $write("S:%d, R:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", S, R, C, nP, nR, Q, Qbar);

// Control off, retain values = 0
#5 S = 1'bx; R = 1'bx; C = 1'b0; nP = 1'b1; nR = 1'b1;
#5 $write("S:%d, R:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", S, R, C, nP, nR, Q, Qbar);

// S = R = 0, retain values = 0
#5 S = 1'b0; R = 1'b0; C = 1'b1; nP = 1'b1; nR = 1'b1;
#5 $write("S:%d, R:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", S, R, C, nP, nR, Q, Qbar);

// Preset
#5 S = 1'bx; R = 1'bx; C = 1'bx; nP = 1'b0; nR = 1'b1;
#5 $write("S:%d, R:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", S, R, C, nP, nR, Q, Qbar);

// Reset
#5 S = 1'bx; R = 1'bx; C = 1'bx; nP = 1'b1; nR = 1'b0;
#5 $write("S:%d, R:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", S, R, C, nP, nR, Q, Qbar);
```

The code changes the values of ‘S’, ‘R’, ‘C’, ‘nP’, and ‘nR’, checking the results of ‘Q’ and ‘Qbar’. If the results match up, then the component has been correctly implemented.

J. D-Latch

A D-Latch is a sequential component that shares its structure with the SR-Latch. The only difference is that there is only one input, ‘D’, while SR-Latches have two inputs, ‘S’ and ‘R’. This components schematic is as shown:



Truth Table

C	D	Q_T	Q_{T+1}	Q'_{T+1}
0	x	0	0	1
0	x	1	1	0
1	1	X	1	0
1	0	X	0	1

The right two NAND’s make up a typical SR-Latch without Control. However, the left half has been changed due to having only one input. In the SR-Latch with Control, the first layer of AND gates use ‘S’ and ‘R’, the two inputs. In order to have two inputs to use AND gates, ‘D’ and its inverted form will be used.

```
wire d_inv;
not not_inst(d_inv, D);

SR_LATCH sr_latch_inst1(.Q(Q), .Qbar(Qbar), .S(D), .R(d_inv), .C(C), .nP(nP), .nR(nR));

endmodule
```

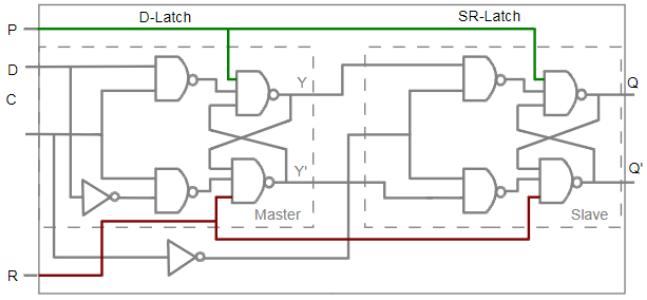
Implementing the D-Latch only requires the inverted ‘D’ value and an instantiation of the SR-Latch with Control. Once ‘D’ is NOT’ed into ‘d_inv’, the two fields are passed into the SR-Latch with Control’s ‘S’ and ‘R’ fields. To test the D-Latch, the truth table will be used to ensure that the inputs lead to its respective output.

```
// Set
#5 D = 1'b1; C = 1'b1; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", D, C, nP, nR, Q, Qbar);

// Control off, retain values = 1
#5 D = 1'bx; C = 1'b0; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", D, C, nP, nR, Q, Qbar);

// Reset
#5 D = 1'b0; C = 1'b1; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", D, C, nP, nR, Q, Qbar);

// Control off, retain values = 0
#5 D = 1'bx; C = 1'b0; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", D, C, nP, nR, Q, Qbar);

// Preset
#5 D = 1'bx; C = 1'b1; nP = 1'b0; nR = 1'b1;
#5 $write("D:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", D, C, nP, nR, Q, Qbar);

// Reset
#5 D = 1'bx; C = 1'b1; nP = 1'b1; nR = 1'b0;
#5 $write("D:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", D, C, nP, nR, Q, Qbar);
```

The code changes the values of ‘D’, ‘C’, ‘nP’, and ‘nR’, checking the results of ‘Q’ and ‘Qbar’. Same as the SR-Latch, if the results match up, then the D-Latch has been correctly implemented.

K. FlipFlop

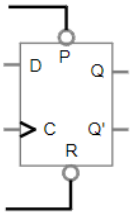
A FlipFlop, or D-FlipFlop, is a sequential component that tracks the input. It makes use of both the D-Latch and SR-Latch with Control. The schematic below shows this relationship:



As shown, the first half is a D-Latch while the second half is the SR-Latch with Control. It can be seen, however, that the control ‘C’ is inverted and used in the SR-Latch (the ‘C’ on the left side of the schematic is meant to be next to the middle wire).

```
wire c_inv;
not not_inst1(c_inv, C);

wire Y, Ybar;
D_LATCH d_latch_inst(.Q(Y), .Qbar(Ybar), .D(D), .C(c_inv), .nP(nP), .nR(nR));
SR_LATCH sr_latch_inst(.Q(Q), .Qbar(Qbar), .S(Y), .R(Ybar), .C(C), .nP(nP), .nR(nR));

endmodule
```

‘C’ is first inverted to ‘c_inv’ in preparation for its usage in the SR-Latch. Once complete, two wires are created for the results of the D-Latch. They are passed into the D-Latch instantiation, in which the results of the D-Latch are then wired into the SR-Latch to acquire the final result of the FlipFlop. Following the same testing approach as the SR-Latch and D-Latch, the FlipFlop will be tested according to its truth table as well. Its truth table is as shown:



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

```
// Negative edge to set D-Latch = 1
#5 D = 1'b1; C = 1'b0; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", D, C, nP, nR, Q, Qbar);

// Set
#5 D = 1'b1; C = 1'b1; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", D, C, nP, nR, Q, Qbar);

// Control off, retain values = 1
#5 D = 1'bx; C = 1'b0; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", D, C, nP, nR, Q, Qbar);

// Negative edge to set D-Latch = 0
#5 D = 1'b0; C = 1'b0; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", D, C, nP, nR, Q, Qbar);

// Reset
#5 D = 1'b0; C = 1'b1; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", D, C, nP, nR, Q, Qbar);

// Control off, retain values = 0
#5 D = 1'bx; C = 1'b0; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", D, C, nP, nR, Q, Qbar);

// Preset
#5 D = 1'bx; C = 1'b1; nP = 1'b0; nR = 1'b1;
#5 $write("D:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", D, C, nP, nR, Q, Qbar);

// Reset
#5 D = 1'bx; C = 1'b1; nP = 1'b1; nR = 1'b0;
#5 $write("D:%d, C:%d, nP:%d, nR:%d, Q:%d, Qbar:%d\n", D, C, nP, nR, Q, Qbar);
```

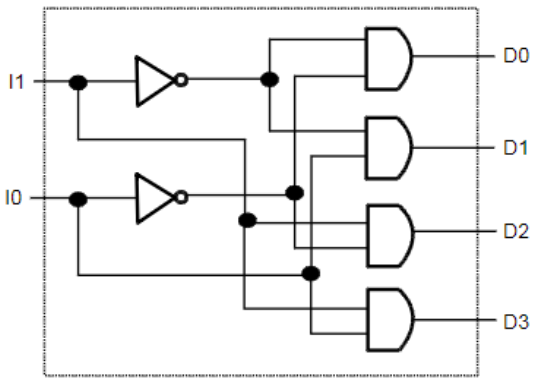
The code changes the values of 'D', 'C', 'nP', and 'nR', once again checking the results of 'Q' and 'Qbar'. If the results match up, then the FlipFlop has been implemented correctly. Testing results will be shown in the later section.

L. Decoder

A decoder is a component that takes an input a number of bits and outputs a larger number of bits. The output bits are one-hot, which means that only one of the bits is '1' while the rest are all '0'. For example, a 2-to-4 line decoder will take in 2 bits and output 4 bits, of which only one bit is '1'. Using the format of "n-to-m line decoder," a line decoder will convert n-bit into m-bit. The size of m-bit is in the range of $n \leq m \leq 2^n$. Several line decoders will need to be implemented.

1) 2-to-4 line decoder

Previously used as an example, this line decoder will convert 2 bits into 4 bits. Shown is a schematic of how this will be implemented:



Requiring the usage of NOR and AND gates, the process of implementing the 2-to-4 line decoder is straightforward and simple.

```
// 2x4 Line decoder
module DECODER_2x4(D,I);
// output
output [3:0] D;
// input
input [1:0] I;

wire [1:0] I_inv;

not not_inst1(I_inv[0], I[0]);
not not_inst2(I_inv[1], I[1]);

and and_inst1(D[0], I_inv[1], I_inv[0]);
and and_inst2(D[1], I_inv[1], I[0]);
and and_inst3(D[2], I[1], I_inv[0]);
and and_inst4(D[3], I[1], I[0]);
```

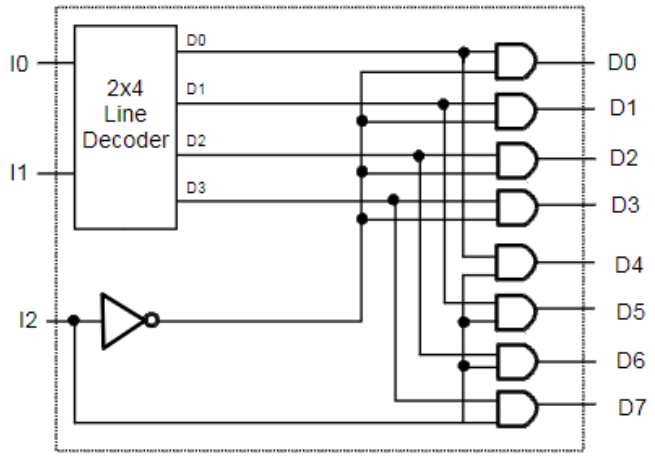
A wire of width 2 is created for the inverted I's. After that, the rest of the AND gates simply follow the schematic. Though it is a bit of slow, tedious work tracing the wires in the schematic, this approach should not be used for larger line decoders. To test the 2x4 line decoder, all possible 'I' values will be passed into the decoder instantiation.

```
module DECODER_2x4_TB;
// output
wire [3:0] D;
// input
reg [1:0] I;

DECODER_2x4 decoder_2x4_inst1(.D(D), .I(I));

initial
begin
I = 0;
#5 $write("I:%d, D:%b\n", I, D);
I = 1;
#5 $write("I:%d, D:%b\n", I, D);
I = 2;
#5 $write("I:%d, D:%b\n", I, D);
I = 3;
#5 $write("I:%d, D:%b\n", I, D);
end

endmodule
```

Once the 2x4 line decoder has been instantiated, 'I' ascends consecutively, testing all possible values of 'I'. Since the component is a 2x4 line decoder, the second value, '4', is the number of 'I' values available, from 0-3. The resultant 'D' value should be a 4-bit value with only one bit being a '1'. Testing results will be shown in the 'Testing' section.

2) 3-to-8 line decoder

Converting 3 bits into 8 bits, the 3-to-8 line decoder builds on top of the 2-to-4 line decoder. The follow schematic shows how this is done:



Using a 2-to-4 line decoder along with another ‘I’ value, more outputs can be achieved. Looking at the schematic, a pattern can be found within the multiple AND gates. It can be seen that each output ‘D0’ is used in intervals of 4. For example, output of the line decoder ‘D0’ is used as the 3-to-8 line decoder output ‘D0’ and the output four above it, ‘D4’. Other pairs also follow this pattern, up till ‘D3’, in which it reaches the final AND gate. For the first half of the AND gates, ‘D0’ to ‘D3’, their second wire is ‘I2’ inverted. The second half of the AND gates, ‘D4’ to ‘D7’, will take in regular ‘I2’. This pattern can be taken advantage of in the generation loop of the AND gates.

```
// 3x8 Line decoder
module DECODER_3x8(D,I);
// output
output [7:0] D;
// input
input [2:0] I;

wire I2_inv;
not not_inst1(I2_inv, I[2]);

wire [3:0] decoder_2x4_result;
DECODER_2x4 decoder_2x4_inst1(.D(decoder_2x4_result), .I(I[1:0]));

// pattern of wire connections for decoders
genvar i;
generate
        for (i = 0; i < 4; i = i + 1)
        begin : and_generation_loop
                and and_inst1(D[i], decoder_2x4_result[i], I2_inv);
                and and_inst2(D[i + 4], decoder_2x4_result[i], I[2]);
        end
endgenerate

endmodule
```

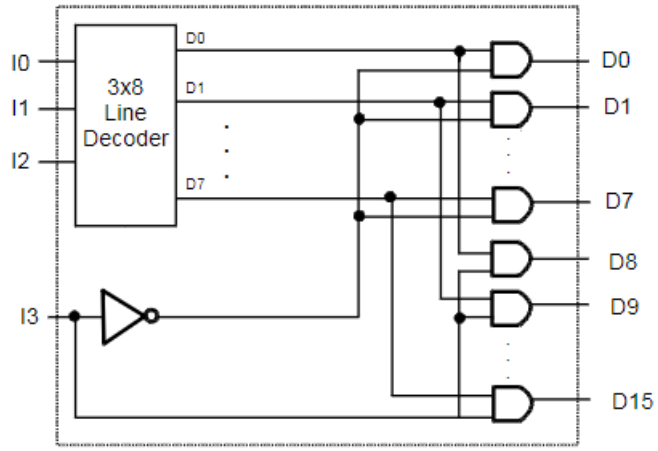
A wire for the inverted ‘I2’ is created and the inverted value is stored on it. An additional wire is created for the result of the initial 2-to-4 line decoder. Once that is done, 8 AND gates must be instantiated through a generation loop. Relying on the previously found pattern, it is apparent that there need to be only half the number of loops compared to the number of outputs. In each loop, there will be 2 AND gates to make up for the halved loop count. Using ‘i’ and the interval of 4 allows the pattern to be used to calculate the final result of the 3-to-8 line decoder. To test the 3-to-8 line decoder, the same test will be completed as the 2-to-4 line decoder, except with a larger range.

```
module DECODER_3x8_TB;
// output
wire [7:0] D;
// input
reg [2:0] I;

DECODER_3x8 decoder_3x8_inst1(.D(D), .I(I));

initial
begin
I = 0;
#5 $write("I:%d, D:%b\n", I, D);
I = 1;
#5 $write("I:%d, D:%b\n", I, D);
I = 2;
#5 $write("I:%d, D:%b\n", I, D);
I = 3;
#5 $write("I:%d, D:%b\n", I, D);
I = 4;
#5 $write("I:%d, D:%b\n", I, D);
I = 5;
#5 $write("I:%d, D:%b\n", I, D);
I = 6;
#5 $write("I:%d, D:%b\n", I, D);
I = 7;
#5 $write("I:%d, D:%b\n", I, D);
end

endmodule
```

Since it is a 3-to-8 line decoder, there are 8 possible ‘I’ values, from 0-7. The resultant ‘D’ will be an 8-bit value with only one bit being a ‘1’. Testing results will be detailed further in the report.

3) 4-to-16 line decoder

Just like upgrading the MUX’s using a previous version, the same is done for line decoders. Using the 3-to-8 line decoder, a 4-to-16 line decoder can be created. The schematic is very similar to the 3-to-8 line decoder and is as shown:



Using the previous line decoder with another ‘I’ value, the number of outputs can be doubled. Once again, a pattern can be derived from the schematic. Similar to the outputs being doubled, the interval pattern from the 2-to-4 line decoder doubles as well. It can be seen that the previous interval of 4 is

now 8, using 3-to-8 line decoder output ‘D0’ in ‘D0’ and ‘D8’. AND gates for decoder outputs The first half of AND gates take their second wire as the inverted ‘I3’ while the second half of AND gates take their second wire as regular ‘I3’, following the same pattern as the 3-to-8 line decoder.

```
// 3x8 Line decoder
module DECODER_3x8(D,I);
// output
output [7:0] D;
// input
input [2:0] I;

wire I2_inv;
not not_inst1(I2_inv, I[2]);

wire [3:0] decoder_2x4_result;
DECODER_2x4 decoder_2x4_inst1(.D(decoder_2x4_result), .I(I[1:0]));

// pattern of wire connections for decoders
genvar i;
generate
        for (i = 0; i < 4; i = i + 1)
        begin : and_generation_loop
                and and_inst1(D[i], decoder_2x4_result[i], I2_inv);
                and and_inst2(D[i + 4], decoder_2x4_result[i], I[2]);
        end
endgenerate

endmodule
```

The exact same approach as the 3-to-8 line decoder is taken. The only differences are that ‘I3’ is the new last wire that will be decoded, a 3-to-8 line decoder will be used instead of a 2-to-4, and the interval is changed in the AND loop generation. This approach allows the pattern to be used, while keeping a similar structure that will be reused in the next line decoder. No different than the other line decoders, the same test of all ‘I’ values will be performed.

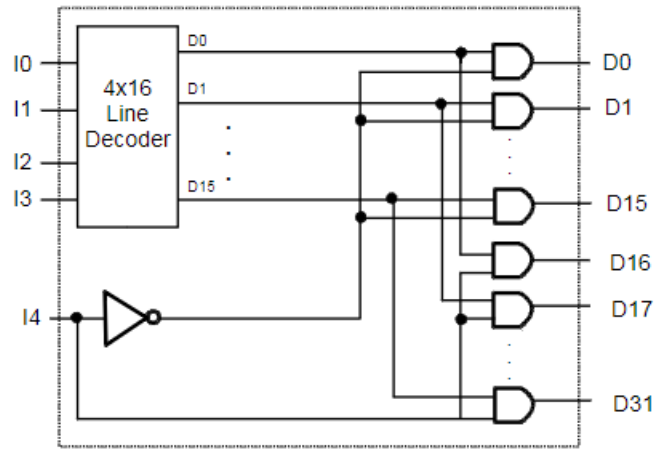
```
module DECODER_4x16_TB;
// output
wire [15:0] D;
// input
reg [3:0] I;

DECODER_4x16 decoder_4x16_inst1(.D(D), .I(I));

initial
begin
I = 0;
#5 $write("I:%d, D:%b\n", I, D);
I = 1;
#5 $write("I:%d, D:%b\n", I, D);
I = 2;
#5 $write("I:%d, D:%b\n", I, D);
I = 3;
#5 $write("I:%d, D:%b\n", I, D);
I = 4;
#5 $write("I:%d, D:%b\n", I, D);
I = 5;
#5 $write("I:%d, D:%b\n", I, D);
```

… etc.

Since it is a 4-to-16 line decoder, there are 16 possible ‘I’ values, from 0-15. Because of this, only a portion of the testbench code can be shown to save space. The resultant ‘D’ value will be a 16-bit value with only one bit as a ‘1’. The results will be in the ‘Testing’ section.

4) 5-to-32 line decoder

Lastly, the 4-to-16 line decoder can be used to create a 5-to 32 line decoder. The schematic is shown below:



Aside from having another ‘I’ value like the previous line decoders, yet another pattern can be derived from the schematic. By following the pattern seen thus far, the interval is now doubled once again to 16, in which the second wire for the ‘D0’ to ‘D15’ AND gates are inverted ‘I4’ while the second half of AND gates, ‘D16’ to ‘D31’, have a wire for regular ‘I4’.

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

wire I4_inv;
not not_inst1(I4_inv, I[4]);

wire [15:0] decoder_4x16_result;
DECODER_4x16 decoder_4x16_inst1(.D(decoder_4x16_result), .I(I[3:0]));

genvar i;
generate
        for (i = 0; i < 16; i = i + 1)
        begin : and_generation_loop
                and and_inst1(D[i], decoder_4x16_result[i], I4_inv);
                and and_inst2(D[i + 16], decoder_4x16_result[i], I[4]);
        end
endgenerate

endmodule
```

The 5-to-32 line decoder utilizes the same structure as the previous line decoders, making the implementation very simple. With that, all necessary line decoders have been implemented. To test this decoder, the same approach is taken.

```
module DECODER_5x32_TB;
// output
wire [31:0] D;
// input
reg [4:0] I;

DECODER_5x32 decoder_5x32_inst1(.D(D), .I(I));

initial
begin
I = 0;
#5 $write("I:%d, D:%b\n", I, D);
I = 1;
#5 $write("I:%d, D:%b\n", I, D);
I = 2;
#5 $write("I:%d, D:%b\n", I, D);
I = 3;
#5 $write("I:%d, D:%b\n", I, D);
I = 4;
#5 $write("I:%d, D:%b\n", I, D);
I = 5;
#5 $write("I:%d, D:%b\n", I, D);
```

… etc.

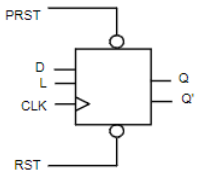
As a 5-to-32 line decoder, there are 32 ‘I’ values, from 0 31. The resultant ‘D’ will be a 32-bit value that has only one bit of ‘1’. Testing results will be detailed later.

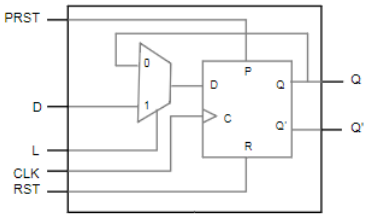
M. Register

Registers are sequential components that store bit values. These values can be stored and loaded into the register when instructed.

1) 1-bit Register

As the first register to be created, 1-bit register is a fairly straightforward sequential component. The schematic is as shown:





Requiring only a D-FlipFlop and a 1-bit 2x1 MUX, the implementation is simple instantiation and wiring.

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

wire mux_result, q_d_ff;
MUX1_2x1 mux1_2x1_inst1(.Y(mux_result), .I0(q_d_ff), .I1(D), .S(L));

D_FF d_ff_inst1(.Q(q_d_ff), .Qbar(Qbar), .D(mux_result), .C(C), .nP(nP), .nR(nR));

buf buf_inst1(Q, q_d_ff);

endmodule
```

Two wires are first instantiated for the result of the MUX and for ‘Q’ of the D-FlipFlop. Once complete, the MUX is instantiated, with the result of the MUX then being passed into the D-FlipFlop as the ‘D’ field, along with the other given wires, such as ‘CLK’ and ‘RST’. The ‘Q’ of the D-FlipFlop is then BUF’ed to the final output ‘Q’ to complete the 1-bit register. To test the 1-bit register, different input fields will be tested.

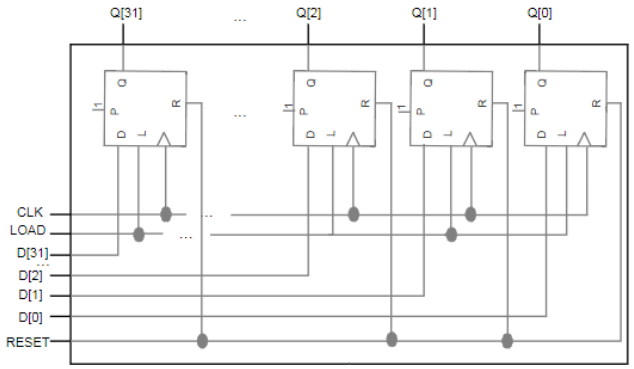
```
/* L = 1 */

// Negative edge to set D-Latch = 1
#5 D = 1'b1; C = 1'b0; L = 1'b1; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d C:%d L:%d nP:%d nR:%d Q:%d Qbar:%d\n", D, C, L, nP, nR, Q, Qbar);

// Set
#5 D = 1'b1; C = 1'b1; L = 1'b1; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d C:%d L:%d nP:%d nR:%d Q:%d Qbar:%d\n", D, C, L, nP, nR, Q, Qbar);

// Control off, retain values = 1
#5 D = 1'bx; C = 1'b0; L = 1'b1; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d C:%d L:%d nP:%d nR:%d Q:%d Qbar:%d\n", D, C, L, nP, nR, Q, Qbar);

// Negative edge to set D-Latch = 0
#5 D = 1'b0; C = 1'b0; L = 1'b1; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d C:%d L:%d nP:%d nR:%d Q:%d Qbar:%d\n", D, C, L, nP, nR, Q, Qbar);

// Reset
#5 D = 1'b0; C = 1'b1; L = 1'b1; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d C:%d L:%d nP:%d nR:%d Q:%d Qbar:%d\n", D, C, L, nP, nR, Q, Qbar);

// Control off, retain values = 0
#5 D = 1'bx; C = 1'b0; L = 1'b1; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d C:%d L:%d nP:%d nR:%d Q:%d Qbar:%d\n", D, C, L, nP, nR, Q, Qbar);

// Preset
#5 D = 1'bx; C = 1'b1; L = 1'b1; nP = 1'b0; nR = 1'b1;
#5 $write("D:%d C:%d L:%d nP:%d nR:%d Q:%d Qbar:%d\n", D, C, L, nP, nR, Q, Qbar);

// Reset
#5 D = 1'bx; C = 1'b1; L = 1'b1; nP = 1'b1; nR = 1'b0;
#5 $write("D:%d C:%d L:%d nP:%d nR:%d Q:%d Qbar:%d\n", D, C, L, nP, nR, Q, Qbar);

/* L = 0 */

// Negative edge to set D-Latch = 1
#5 D = 1'b1; C = 1'b0; L = 1'b0; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d C:%d L:%d nP:%d nR:%d Q:%d Qbar:%d\n", D, C, L, nP, nR, Q, Qbar);

// Set
#5 D = 1'b1; C = 1'b1; L = 1'b0; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d C:%d L:%d nP:%d nR:%d Q:%d Qbar:%d\n", D, C, L, nP, nR, Q, Qbar);

// Control off, retain values = 1
#5 D = 1'bx; C = 1'b0; L = 1'b0; nP = 1'b1; nR = 1'b1;
#5 $write("D:%d C:%d L:%d nP:%d nR:%d Q:%d Qbar:%d\n", D, C, L, nP, nR, Q, Qbar);
```

… etc.

Following the schematic, it can be seen that there are five input fields to be changed: ‘D’, ‘C’, ‘L’, ‘nP’, and ‘nR’ since the 1-bit register uses the FlipFlop. Compared to the FlipFlop, only ‘L’ is the new field. This doubles the amount of tests to check, as compared to the FlipFlop. Test results will be detailed later in the report.

2) 32-bit Register

Multi-bit registers can be produced by chaining together multiple 1-bit registers. For 32-bit registers, 32 1-bit registers are chained together. The chaining implementation is as shown in the schematic:



All the registers share the same ‘CLK’, ‘LOAD’, and ‘RESET’. The only difference is that they receive their respective bit. It can be seen from the schematic that multiple registers need to be created, meaning generation is necessary.

```
wire NA; // for unnecessary Qbar in register

// generates 32 1-bit registers
genvar i;
generate
        for (i = 0; i < 32; i = i + 1)
        begin : reg1_generation_loop
                REG1 reg1_inst1(.Q(Q[i]), .Qbar(NA), .D(D[i]), .L(LOAD), .C(CLK), .nP(1'b1), .nR(RE
        end
endgenerate
```

A wire ‘NA’ is first created to store unnecessary results. Because 1-bit registers use D-FlipFlops, Q and Q’ are produced. However, only Q is used, while Q’ is discarded. Wire ‘NA’ will be used to fulfill the instantiation, while ignoring the value that it receives. After that, the generation loop will create 32 1-bit registers, sharing their ‘Qbar’, ‘LOAD’, ‘CLOCK’, and ‘RESET’. Otherwise, ‘Q’ and ‘D’ take their respective bits. Testing the 32-bit register is the same as testing the 1-bit register, except that 32-bit values can be loaded instead. This makes it so that not all cases can be tested, but there are ways to work around that.

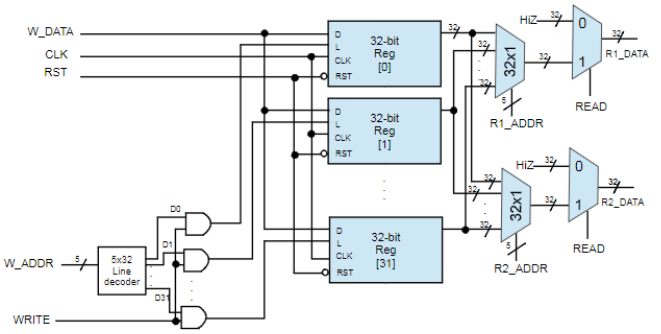
```
/* LOAD = 1 */

// Negative edge to set D-Latch = 1
#5 D = 32'hffffffff; LOAD = 1'b1; CLK = 1'b0; RESET = 1'b1;
#5 $write("D:%d LOAD:%d CLK:%d RESET:%d Q:%d\n", D, LOAD, CLK, RESET, Q);

// Set
#5 D = 32'hffffffff; LOAD = 1'b1; CLK = 1'b1; RESET = 1'b1;
#5 $write("D:%d LOAD:%d CLK:%d RESET:%d Q:%d\n", D, LOAD, CLK, RESET, Q);

// Control off, retain values = 1
#5 D = 32'bx; LOAD = 1'b1; CLK = 1'b0; RESET = 1'b1;
#5 $write("D:%d LOAD:%d CLK:%d RESET:%d Q:%d\n", D, LOAD, CLK, RESET, Q);

// Reset
#5 D = 32'bx; LOAD = 1'b1; CLK = 1'b1; RESET = 1'b0;
#5 $write("D:%d LOAD:%d CLK:%d RESET:%d Q:%d\n", D, LOAD, CLK, RESET, Q);

// Negative edge to set D-Latch = 1
#5 D = 32'hffffffff; LOAD = 1'b1; CLK = 1'b0; RESET = 1'b1;
#5 $write("D:%d LOAD:%d CLK:%d RESET:%d Q:%d\n", D, LOAD, CLK, RESET, Q);

// Set
#5 D = 32'hffffffff; LOAD = 1'b1; CLK = 1'b1; RESET = 1'b1;
#5 $write("D:%d LOAD:%d CLK:%d RESET:%d Q:%d\n", D, LOAD, CLK, RESET, Q);

// Negative edge to set D-Latch = 0
#5 D = 32'b0; LOAD = 1'b1; CLK = 1'b0; RESET = 1'b1;
#5 $write("D:%d LOAD:%d CLK:%d RESET:%d Q:%d\n", D, LOAD, CLK, RESET, Q);

// Reset
#5 D = 32'b0; LOAD = 1'b1; CLK = 1'b1; RESET = 1'b1;
#5 $write("D:%d LOAD:%d CLK:%d RESET:%d Q:%d\n", D, LOAD, CLK, RESET, Q);

// Control off, retain values = 0
#5 D = 32'bx; LOAD = 1'b1; CLK = 1'b0; RESET = 1'b1;
#5 $write("D:%d LOAD:%d CLK:%d RESET:%d Q:%d\n", D, LOAD, CLK, RESET, Q);

/* L = 0 */

// Negative edge to set D-Latch = 1
#5 D = 32'hffffffff; LOAD = 1'b0; CLK = 1'b0; RESET = 1'b1;
#5 $write("D:%d LOAD:%d CLK:%d RESET:%d Q:%d\n", D, LOAD, CLK, RESET, Q);
```

… etc.

The only difference between this testbench and the 1-bit register testbench is that ‘D’ is now a 32-bit value. Since not all 32-bit values can be tested, there are some 32-bit unknown values that can be used to simulate random values. Test results will be shown in the ‘Testing’ section.

3) 32x32-bit Register

Further expanding on the previous registers is the 32x32-bit register. Not only are there more bits that it can hold, but it has additional functionalities to be able to choose from one of the 32 32-bit registers, with the usage of the previously implemented line decoder and MUX’s. Below is the complete schematic and design:



The 32x32-bit register requires a 5x32 line decoder, 32 32 bit registers, two 32x1 MUX’s, and two 32-bit 2x1 MUX’s. As shown by the multiple registers, a generation loop will be used to create them. Aside from instantiation of other components, wiring is all that is left.

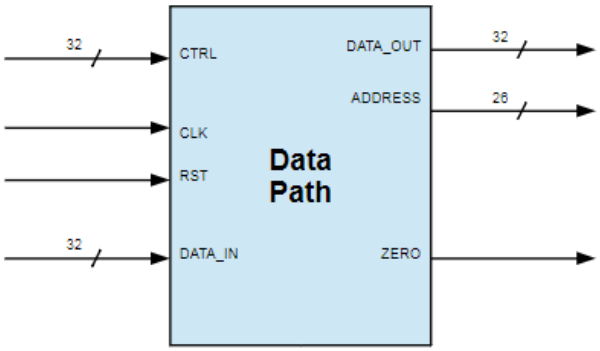
```
wire [`DATA_INDEX_LIMIT:0] decoder_result;
DECODER_5x32 decoder_5x32_inst1(.D(decoder_result), .I(ADDR_W));

wire [`DATA_INDEX_LIMIT:0] and_result;
wire [`DATA_INDEX_LIMIT:0] reg1 [`DATA_INDEX_LIMIT:0];

// generates 32 32-bit registers
genvar i;
generate
        for (i = 0; i <= `DATA_INDEX_LIMIT; i = i + 1)
        begin : reg32_generation_loop
                and and_write_inst(and_result[i], decoder_result[i], WRITE);
                REG32 reg32_inst(.Q(reg1[i]), .D(DATA_W), .LOAD(and_result[i]), .CLK(CLK), .RESET(RST));
        end
endgenerate

wire [`DATA_INDEX_LIMIT:0] mux_result1, mux_result2;

MUX32_32x1 mux32_32x1_inst_1(.Y(mux_result1), .I0(reg1[0]), .I1(reg1[1]), .I2(reg1[2]), .I3(reg1[3]), .I4(reg1[4]), .I5(reg1[5]),
                             .I6(reg1[6]), .I7(reg1[7]), .I8(reg1[8]), .I9(reg1[9]), .I10(reg1[10]), .I11(reg1[11]), .I12(reg1[12]),
                             .I13(reg1[13]), .I14(reg1[14]), .I15(reg1[15]), .I16(reg1[16]), .I17(reg1[17]), .I18(reg1[18]), .I19(reg1[19]),
                             .I20(reg1[20]), .I21(reg1[21]), .I22(reg1[22]), .I23(reg1[23]), .I24(reg1[24]), .I25(reg1[25]), .I26(reg1[26]),
                             .I27(reg1[27]), .I28(reg1[28]), .I29(reg1[29]), .I30(reg1[30]), .I31(reg1[31]), .S(ADDR_R1));

MUX32_32x1 mux32_32x1_inst_2(.Y(mux_result2), .I0(reg1[0]), .I1(reg1[1]), .I2(reg1[2]), .I3(reg1[3]), .I4(reg1[4]), .I5(reg1[5]),
                             .I6(reg1[6]), .I7(reg1[7]), .I8(reg1[8]), .I9(reg1[9]), .I10(reg1[10]), .I11(reg1[11]), .I12(reg1[12]),
                             .I13(reg1[13]), .I14(reg1[14]), .I15(reg1[15]), .I16(reg1[16]), .I17(reg1[17]), .I18(reg1[18]), .I19(reg1[19]),
                             .I20(reg1[20]), .I21(reg1[21]), .I22(reg1[22]), .I23(reg1[23]), .I24(reg1[24]), .I25(reg1[25]), .I26(reg1[26]),
                             .I27(reg1[27]), .I28(reg1[28]), .I29(reg1[29]), .I30(reg1[30]), .I31(reg1[31]), .S(ADDR_R2));

MUX32_2x1 mux32_2x1_inst_1(.Y(DATA_R1), .I0({`DATA_WIDTH{1'bz}}), .I1(mux_result1), .S(READ));
MUX32_2x1 mux32_2x1_inst_2(.Y(DATA_R2), .I0({`DATA_WIDTH{1'bz}}), .I1(mux_result2), .S(READ));
```

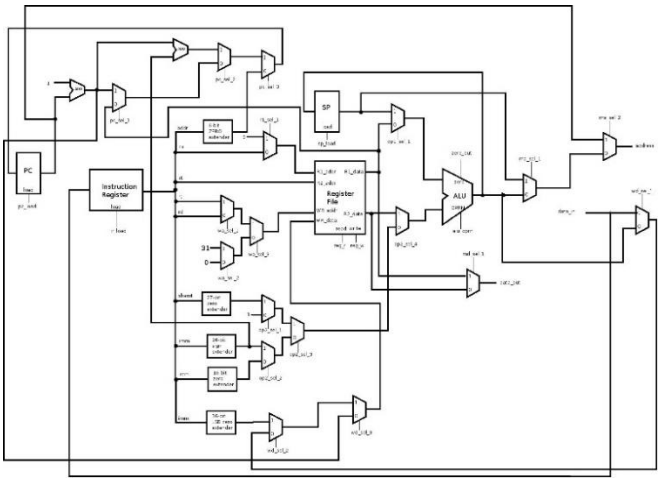
The very small and dense code begins by instantiating a 5x32 line decoder with a 32-bit wire and ‘DATA_W’, as shown by the schematic. Once done, the registers must be generated. Using a generation loop, it starts by performing AND with the result of the decoder and ‘WRITE’ to determine if the register should be written to or not. Each register then receives the same ‘DATA_W’, ‘CLK, and ‘RESET’, but are given their respective output register and respective AND result previously calculated.

Once done with the generation loop, two 32x1 MUX’s are instantiating along with wires to store their results. The inputs of the MUX’s are each index of the register. The selection is ‘ADDR_R1’ and ‘ADDR_R2’ respectively. Once the 32x1 MUX’s are done, the results are then passed into 2x1 MUX’s. The selection is ‘READ’, determining if the output of the 32x32-bit register should be the data from the previous 32x1 MUX or a 32-bit unknown. The testbench for the 32x32-bit register has already been included. Testing results will be shown further on.

N. Data Path

The data path is a vital component which links all the previous components together. It is in charge of making sure that the system is able to access whatever components and information it may need, whether already processed or needing to be processed. The schematics are as shown below:





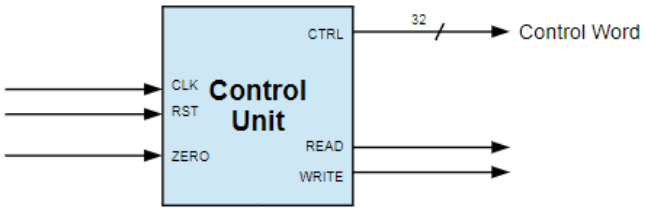
As shown, it is a very length mapping, requiring many components and wires. Though not a difficult task, it will take time to wire the path together.

```
wire [`DATA_INDEX_LIMIT:0] R1_data, R2_data, alu_out, add_res_1, add_res_2, op1_sel_1, op2_sel_1,
                           op2_sel_2, op2_sel_3, op2_sel_4, wd_sel_1, wd_sel_2, wd_sel_3, ma_sel_2,
                           ma_sel_1, pc_out, ir_res, sp_res, pc_sel_1, pc_sel_2, pc_sel_3;
wire [4:0] r1_sel_1, wa_sel_1, wa_sel_2, wa_sel_3;
wire NA;

BUF32_2x1 buf_inst(.Y(INSTRUCTION), .A(DATA_IN));

RC_ADD_SUB_32 rc_add_sub_inst1(.Y(add_res_1), .CO(NA), .A(32'b1), .B(pc_out), .SnA(1'b0));
RC_ADD_SUB_32 rc_add_sub_inst2(.Y(add_res_2), .CO(NA), .A(add_res_1), .B({{16{ir_res[15]}}, ir_res[15:0]}), .SnA(1'b0));

defparam pc_inst.PATTERN = `INST_START_ADDR;
REG32_PP pc_inst1(.Q(pc_out), .D(pc_sel_3), .LOAD(CTRL[0]), .CLK(CLK), .RESET(RST));

MUX32_2x1 mux_pc_sel_1(.Y(pc_sel_1), .I0(R1_data), .I1(add_res_1), .S(CTRL[1]));
MUX32_2x1 mux_pc_sel_2(.Y(pc_sel_2), .I0(pc_sel_1), .I1(add_res_2), .S(CTRL[2]));
MUX32_2x1 mux_pc_sel_3(.Y(pc_sel_3), .I0({6'b0, ir_res[25:0]}), .I1(pc_sel_2), .S(CTRL[3]));

REG32 ir_inst(.Q(ir_res), .D(DATA_IN), .LOAD(CTRL[4]), .CLK(CLK), .RESET(RST));

MUX5_2x1 mux_r1_sel_1(.Y(r1_sel_1), .I0(ir_res[25:21]), .I1(5'b00000), .S(CTRL[5]));

REGISTER_FILE_32x32 rf_32x32_inst1(.DATA_R1(R1_data), .DATA_R2(R2_data), .ADDR_R1(r1_sel_1),
                                   .ADDR_R2(ir_res[20:16]), .DATA_W(wd_sel_3), .ADDR_W(wa_sel_3),
                                   .READ(CTRL[6]), .WRITE(CTRL[7]), .CLK(CLK), .RST(RST));

defparam sp_inst.PATTERN = `INIT_STACK_POINTER;
REG32_PP sp_inst1(.Q(sp_res), .D(alu_out), .LOAD(CTRL[8]), .CLK(CLK), .RESET(RST));

MUX32_2x1 mux_op1_sel_1(.Y(op1_sel_1), .I0(R1_data), .I1(sp_res), .S(CTRL[9]));

MUX32_2x1 mux_op2_sel_1(.Y(op2_sel_1), .I0(32'b1), .I1({27'b0, ir_res[10:6]}), .S(CTRL[10]));
MUX32_2x1 mux_op2_sel_2(.Y(op2_sel_2), .I0({16'b0, ir_res[15:0]}), .I1({{16{ir_res[15]}}, ir_res[15:0]}), .S(CTRL[11]));
MUX32_2x1 mux_op2_sel_3(.Y(op2_sel_3), .I0(op2_sel_2), .I1(op2_sel_1), .S(CTRL[12]));
MUX32_2x1 mux_op2_sel_4(.Y(op2_sel_4), .I0(op2_sel_3), .I1(R2_data), .S(CTRL[13]));

ALU alu_inst(.OUT(alu_out), .ZERO(ZERO), .OP1(op1_sel_1), .OP2(op2_sel_4), .OPRN(CTRL[19:14]));

MUX32_2x1 mux_ma_sel_1(.Y(ma_sel_1), .I0(alu_out), .I1(sp_res), .S(CTRL[20]));
MUX32_2x1 mux_ma_sel_2(.Y(ADDR), .I0(ma_sel_1), .I1(pc_out), .S(CTRL[21]));

MUX32_2x1 mux_md_sel_1(.Y(DATA_OUT), .I0(R2_data), .I1(R1_data), .S(CTRL[22]));

MUX32_2x1 mux_wd_sel_1(.Y(wd_sel_1), .I0(alu_out), .I1(DATA_IN), .S(CTRL[23]));
MUX32_2x1 mux_wd_sel_2(.Y(wd_sel_2), .I0(wd_sel_1), .I1({ir_res[15:0], 16'b0}), .S(CTRL[24]));
MUX32_2x1 mux_wd_sel_3(.Y(wd_sel_3), .I0(add_res_1), .I1(wd_sel_2), .S(CTRL[25]));

MUX5_2x1 mux_wa_sel_1(.Y(wa_sel_1), .I0(ir_res[15:11]), .I1(ir_res[20:16]), .S(CTRL[26]));
MUX5_2x1 mux_wa_sel_2(.Y(wa_sel_2), .I0(5'b00000), .I1(5'b11111), .S(CTRL[27]));
MUX5_2x1 mux_wa_sel_3(.Y(wa_sel_3), .I0(wa_sel_2), .I1(wa_sel_1), .S(CTRL[28]));
```

To briefly sum it up, all the components have been wired together in the correct fashion, including the previously introduced, but unused 5-bit 2x1 MUX. The data path has been completed.

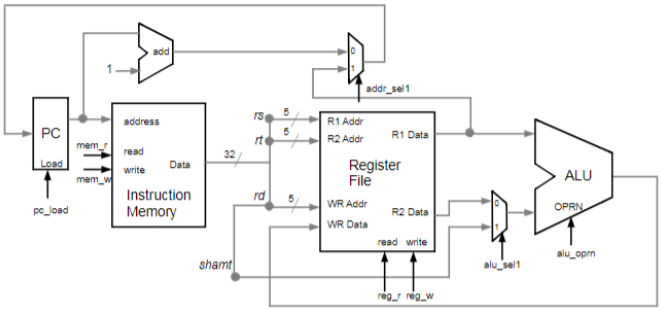
O. Control Unit

The control unit is in charge of handling the state machine and instructions. Similar to the control unit in project two, this control unit shares the same state machine, but has an altered instruction handling unit. The schematic is as shown:

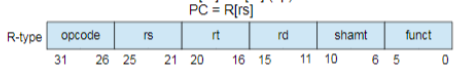


Essentially, the difference between this control unit and project two's control unit is that this control unit will be handled at a lower level, rather than the second project's behavioral model approach. Provided are schematics for each type of instruction:

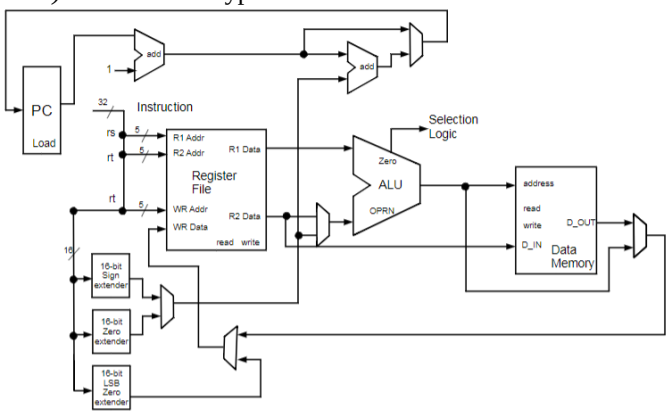
1) Combined R-Type Instruction Control



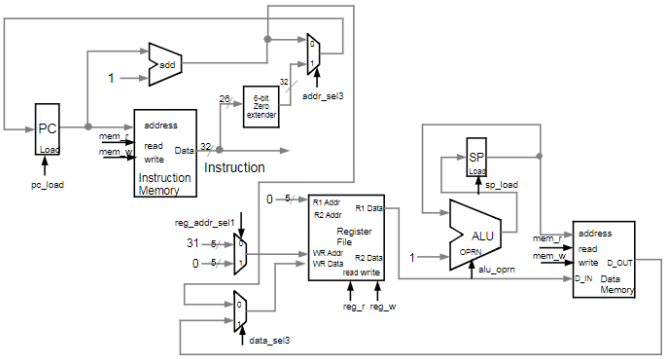
Operation: R[rd] = R[rs] (op) R[rt]
R[rd] = R[rs] (op) shamt
PC = R[rs]



2) Combined I-Type Instruction Control



3) Combined J-Type Instruction Control



These schematics handle all of their respective instructions within one connected diagram. For more smaller, in-depth schematics that make up the larger combined one, check the references at the end of the report for the “Putting Together a Microprocessor I” lecture.

```
reg [`CTRL_WIDTH_INDEX_LIMIT:0]  CTRL;
reg READ, WRITE;
reg [`DATA_INDEX_LIMIT:0] INSTR;

wire [2:0] proc_state;
PROC_SM state_machine(.STATE(proc_state), .CLK(CLK), .RST(RST));

always @ (proc_state)
begin
        if (proc_state === `PROC_FETCH)
        begin
                READ = 1'b1;
                WRITE = 1'b0;
                CTRL = `CTRL_WIDTH'h00200000;
        end

        else if (proc_state === `PROC_DECODE)
        begin
                INSTR = INSTRUCTION;
                if (INSTR[31:26] === 6'h1b) // push
                begin
                        CTRL = `CTRL_WIDTH'h00000070;
                end
                else if (INSTR[31:26] === 6'h0f || INSTR[31:26] === 6'h02 ||
                        INSTR[31:26] === 6'h03 || INSTR[31:26] === 6'h1c) // lui, jmp, jal, pop
                begin
                        CTRL = `CTRL_WIDTH'h00000010;
                end
                else
                begin
                        CTRL = `CTRL_WIDTH'h00000050; // everything else
                end
        end
        else if (proc_state === `PROC_EXE)
        begin
                // R-Types
                if (INSTR[31:26] === 6'h00)
                begin
                        if (INSTR[5:0] === 6'h20) // add
                        begin
                                CTRL = `CTRL_WIDTH'h00006000;
                        end
                        else if (INSTR[5:0] === 6'h22) // sub
                        begin
                                CTRL = `CTRL_WIDTH'h0000A000;
                        end
                        else if (INSTR[5:0] === 6'h2c) // mul
                        begin
                                CTRL = `CTRL_WIDTH'h0000E000;
                        end
                        else if (INSTR[5:0] === 6'h24) // and
                        begin
                                CTRL = `CTRL_WIDTH'h0001A000;
                        end
                        else if (INSTR[5:0] === 6'h25) // or
                        begin
                                CTRL = `CTRL_WIDTH'h0001E000;
                        end
                        else if (INSTR[5:0] === 6'h27) // nor
                        begin
                                CTRL = `CTRL_WIDTH'h00022000;
                        end
                        else if (INSTR[5:0] === 6'h2a) // slt
                        begin
                                CTRL = `CTRL_WIDTH'h00026000;
                        end
                        else if (INSTR[5:0] === 6'h01) // sll
                        begin
                                CTRL = `CTRL_WIDTH'h00015400;
                        end
                        else if (INSTR[5:0] === 6'h02) // srl
                        begin
                                CTRL = `CTRL_WIDTH'h00011400;
                        end
                        else if (INSTR[5:0] === 6'h08) // jr
                        begin
                                CTRL = `CTRL_WIDTH'h00000000;
                        end
                end
```

```
        else
        // I-Types and J-Types
        begin
                /* I-Type */
                if (INSTR[31:26] === 6'h08) // addi
                begin
                        CTRL = `CTRL_WIDTH'h00004800;
                end
                else if (INSTR[31:26] === 6'h1d) // muli
                begin
                        CTRL = `CTRL_WIDTH'h0000C800;
                end
                else if (INSTR[31:26] === 6'h0c) // andi
                begin
                        CTRL = `CTRL_WIDTH'h00018000;
                end
                else if (INSTR[31:26] === 6'h0d) // ori
                begin
                                CTRL = `CTRL_WIDTH'h0001C000;
                end
                else if (INSTR[31:26] === 6'h0f) // lui
                begin
                        CTRL = `CTRL_WIDTH'h00000000;
                end
                else if (INSTR[31:26] === 6'h0a) // stli
                begin
                        CTRL = `CTRL_WIDTH'h00024800;
                end
                else if (INSTR[31:26] === 6'h04) // beq
                begin
                        CTRL = `CTRL_WIDTH'h0000A000;
                end
                else if (INSTR[31:26] === 6'h05) // bne
                begin
                        CTRL = `CTRL_WIDTH'h0000A000;
                end
                else if (INSTR[31:26] === 6'h23) // lw
                begin
                        CTRL = `CTRL_WIDTH'h00004800;
                end
                else if (INSTR[31:26] === 6'h2b) // sw
                begin
                        CTRL = `CTRL_WIDTH'h00004800;
                end
                // J-Types
                else if (INSTR[31:26] === 6'h02) // jmp
                begin
                        CTRL = `CTRL_WIDTH'h00000000;
                end
                else if (INSTR[31:26] === 6'h03) // jal
                begin
                        CTRL = `CTRL_WIDTH'h00000000;
                end
                else if (INSTR[31:26] === 6'h1b) // push
                begin
                        CTRL = `CTRL_WIDTH'h00009200;
                end
                else if (INSTR[31:26] === 6'h1c) // pop
                begin
                        CTRL = `CTRL_WIDTH'h00005300;
                end
        end
end

else if (proc_state === `PROC_MEM)
begin
        // R-Types
        if (INSTR[31:26] === 6'h00)
        begin
                if (INSTR[5:0] === 6'h20) // add
                begin
                        CTRL = `CTRL_WIDTH'h00006000;
                end
                else if (INSTR[5:0] === 6'h22) // sub
                begin
                        CTRL = `CTRL_WIDTH'h0000A000;
                end
                else if (INSTR[5:0] === 6'h2c) // mul
                begin
                        CTRL = `CTRL_WIDTH'h0000E000;
                end
                else if (INSTR[5:0] === 6'h24) // and
                begin
                        CTRL = `CTRL_WIDTH'h0001A000;
                end
                else if (INSTR[5:0] === 6'h25) // or
                begin
                        CTRL = `CTRL_WIDTH'h0001E000;
                end
                else if (INSTR[5:0] === 6'h27) // nor
                begin
                        CTRL = `CTRL_WIDTH'h00022000;
                end
                else if (INSTR[5:0] === 6'h2a) // slt
                begin
                        CTRL = `CTRL_WIDTH'h00026000;
                end
                else if (INSTR[5:0] === 6'h01) // sll
                begin
                        CTRL = `CTRL_WIDTH'h00015400;
                end
                else if (INSTR[5:0] === 6'h02) // srl
                begin
                        CTRL = `CTRL_WIDTH'h00011400;
                end
                else if (INSTR[5:0] === 6'h08) // jr
                begin
                        CTRL = `CTRL_WIDTH'h00000000;
                end
        end
```

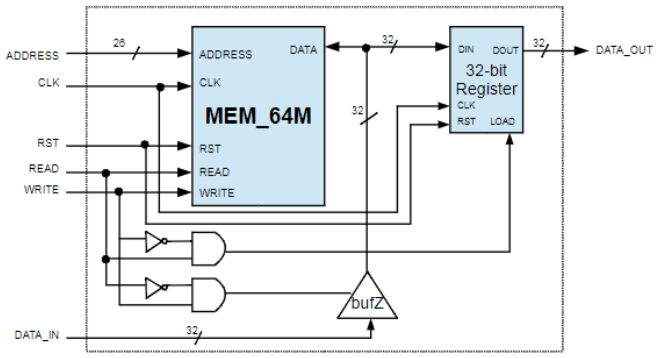
```
        else
        // I-Type and J-Type
        begin
                // I-Types
                if (INSTR[31:26] === 6'h08) // addi
                begin
                        CTRL = `CTRL_WIDTH'h00004800;
                end
                else if (INSTR[31:26] === 6'h1d) // muli
                begin
                        CTRL = `CTRL_WIDTH'h0000C800;
                end
                else if (INSTR[31:26] === 6'h0c) // andi
                begin
                        CTRL = `CTRL_WIDTH'h00018000;
                end
                else if (INSTR[31:26] === 6'h0d) // ori
                begin
                        CTRL = `CTRL_WIDTH'h0001C000;
                end
                else if (INSTR[31:26] === 6'h0f) // lui
                begin
                        CTRL = `CTRL_WIDTH'h00000000;
                end
                else if (INSTR[31:26] === 6'h0a) // stli
                begin
                        CTRL = `CTRL_WIDTH'h00024800;
                end
                else if (INSTR[31:26] === 6'h04) // beq
                begin
                        CTRL = `CTRL_WIDTH'h0000A000;
                end
                else if (INSTR[31:26] === 6'h05) // bne
                begin
                        CTRL = `CTRL_WIDTH'h0000A000;
                end
                else if (INSTR[31:26] === 6'h23) // lw
                begin
                        READ = 1'b1;
                        WRITE = 1'b0;
                        CTRL = `CTRL_WIDTH'h00004800;
                end
                else if (INSTR[31:26] === 6'h2b) // sw
                begin
                        READ = 1'b0;
                        WRITE = 1'b1;
                        CTRL = `CTRL_WIDTH'h00004800;
                end
                // J-Types
                else if (INSTR[31:26] === 6'h02) // jmp
                begin
                        CTRL = `CTRL_WIDTH'h00000000;
                end
                else if (INSTR[31:26] === 6'h03) // jal
                begin
                        CTRL = `CTRL_WIDTH'h00000000;
                end
                else if (INSTR[31:26] === 6'h1b) // push
                begin
                        READ = 1'b0;
                        WRITE = 1'b1;
                        CTRL = `CTRL_WIDTH'h00509200;
                end
                else if (INSTR[31:26] === 6'h1c) // pop
                begin
                        READ = 1'b1;
                        WRITE = 1'b0;
                        CTRL = `CTRL_WIDTH'h00100000;
                end
        end
end

else if (proc_state === `PROC_WB)
begin
        // R-Types
        if (INSTR[31:26] === 6'h00)
        begin
                if (INSTR[5:0] === 6'h20) // add
                begin
                        CTRL = `CTRL_WIDTH'h1200608B;
                end
                else if (INSTR[5:0] === 6'h22) // sub
                begin
                        CTRL = `CTRL_WIDTH'h1200A08B;
                end
                else if (INSTR[5:0] === 6'h2c) // mul
                begin
                        CTRL = `CTRL_WIDTH'h1200E08B;
                end
                else if (INSTR[5:0] === 6'h24) // and
                begin
                        CTRL = `CTRL_WIDTH'h1201A08B;
                end
                else if (INSTR[5:0] === 6'h25) // or
                begin
                        CTRL = `CTRL_WIDTH'h1201E08B;
                end
                else if (INSTR[5:0] === 6'h27) // nor
                begin
                        CTRL = `CTRL_WIDTH'h1202208B;
                end
                else if (INSTR[5:0] === 6'h2a) // slt
                begin
                        CTRL = `CTRL_WIDTH'h1202608B;
                end
                else if (INSTR[5:0] === 6'h01) // sll
                begin
                        CTRL = `CTRL_WIDTH'h1201548B;
                end
                else if (INSTR[5:0] === 6'h02) // srl
                begin
                        CTRL = `CTRL_WIDTH'h1201148B;
                end
                else if (INSTR[5:0] === 6'h08) // jr
                begin
                        CTRL = `CTRL_WIDTH'h00000009;
                end
        end
```

```
        else
        // I-Type and J-Type
        begin
                // I-Types
                if (INSTR[31:26] === 6'h08) // addi
                begin
                        CTRL = `CTRL_WIDTH'h1600488B;
                end
                else if (INSTR[31:26] === 6'h1d) // muli
                begin
                        CTRL = `CTRL_WIDTH'h1600C88B;
                end
                else if (INSTR[31:26] === 6'h0c) // andi
                begin
                        CTRL = `CTRL_WIDTH'h1601808B;
                end
                else if (INSTR[31:26] === 6'h0d) // ori
                begin
                        CTRL = `CTRL_WIDTH'h1601C08B;
                end
                else if (INSTR[31:26] === 6'h0f) // lui
                begin
                        CTRL = `CTRL_WIDTH'h1700008B;
                end
                else if (INSTR[31:26] === 6'h0a) // stli
                begin
                        CTRL = `CTRL_WIDTH'h1602088B;
                end
                else if (INSTR[31:26] === 6'h04) // beq
                begin
                        if (ZERO === 1'b0)
                        begin
                                CTRL = `CTRL_WIDTH'h0000A00D;
                        end
                        else
                        begin
                                CTRL = `CTRL_WIDTH'h0000A00B;
                        end
                end
                else if (INSTR[31:26] === 6'h05) // bne
                begin
                        if (ZERO !== 1'b0)
                        begin
                                CTRL = `CTRL_WIDTH'h0000A00D;
                        end
                        else
                        begin
                                CTRL = `CTRL_WIDTH'h0000A00B;
                        end
                end
                else if (INSTR[31:26] === 6'h23) // lw
                begin
                        CTRL = `CTRL_WIDTH'h1680008B;
                end
                else if (INSTR[31:26] === 6'h2b) // sw
                begin
                        CTRL = `CTRL_WIDTH'h0000000B;
                end

                // J-Types
                else if (INSTR[31:26] === 6'h02) // jmp
                begin
                        CTRL = `CTRL_WIDTH'h00000001;
                end
                else if (INSTR[31:26] === 6'h03) // jal
                begin
                        CTRL = `CTRL_WIDTH'h08000081;
                end
                else if (INSTR[31:26] === 6'h1b) // push
                begin
                        CTRL = `CTRL_WIDTH'h0000930B;
                end
                else if (INSTR[31:26] === 6'h1c) // pop
                begin
                        CTRL = `CTRL_WIDTH'h0280008B;
                end
        end
    end
end
```

To sum up the very lengthy control unit code, it checks each instruction on a case-by-case basis. Each instruction's opcode is checked, and function code as well if it is an R-Type, and handled appropriately depending on the given instruction. The code that is performed is also dependent on the current state of the state machine. Only applicable instructions that are run on that current state are handled. The control unit is tested alongside the entire system, which will be detailed in the later 'Testing' section.

P. Memory Wrapper

The memory wrapper is a vital component for the system, handling any sort of data. It is composed of a memory unit and register component within itself. The schematic is shown below:



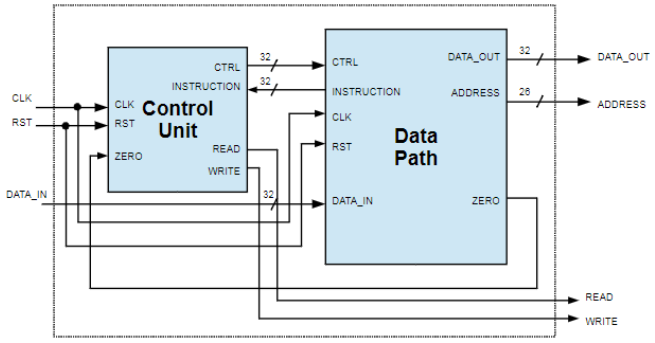
The code for the memory has already been included, but it is not immediately ready for usage in the memory wrapper. Having only one 'DATA' field, 'MEM_64MB', or 'MEM_64M' as shown in the schematic, must be expanded to handle 'DATA_IN' and 'DATA_OUT'.

```
module MEMORY_64MB(DATA_OUT, DATA_IN, 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
input [`DATA_INDEX_LIMIT:0] DATA_IN;
output [`DATA_INDEX_LIMIT:0] DATA_OUT;

// 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_OUT = ((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_IN;
end
end
endmodule
```

The original code's instantiation parameters had only one 'DATA' input. Expanding it to both 'OUT' and 'IN', along with assigning 'DATA_OUT' depending on the operation, the change to include both new 'DATA' fields is very neat. In only two instances does the original 'DATA' field appear in the code. Following the 'memory.v' code completed in project two, it is simple to appropriately change it to the correct new 'DATA' field.

Q. Processor

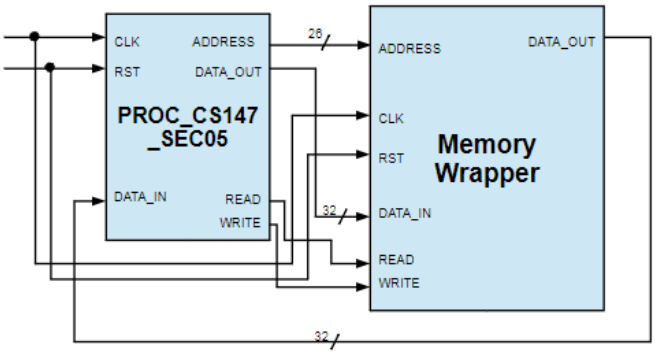
The processor is a vital component, alongside the memory wrapper. It is in charge of handling the instantiation of the control unit and data path, other necessary components for the system's functionality. The schematic of the processor is as shown:



The code for the processor has already been included, so no work is necessary for this component.

R. System

The system is the final "component" of project three. Rather than calling it a component, it is the entirety of all previous components, creating the entire DaVinci v1.0m system. The schematic is as follows:



The system essentially just links up the previous two components. Since the two components have already linked up all other smaller components, this final step simple ties every component together. Once again, this step has already been completed, so all that is left now is to test all the components.

IV. Testing

Finally, testing is extremely vital, as the DaVinci v1.0m system is made up of dozens of components that build upon one another. To ensure the DaVinci v1.0m system works as intended, it is necessary to test the components individually to ensure functionality, as well as together to ensure compatability. Testing will be done from the ground up to check that the base components are functional before the higher-level components are. However, not all components will require testing, such as components in which code has already been completed. From the bottom up:

A. Half Adder

Due to the half adder being 1-bit, all combinations of 'A' and 'B' can be tested. The results are as shown:

```
# A:0, B:0, Y (Sum):0, C (Carry):0
# A:1, B:0, Y (Sum):1, C (Carry):0
# A:0, B:1, Y (Sum):1, C (Carry):0
# A:1, B:1, Y (Sum):0, C (Carry):1
```

Checking each of these cases, all of them are correct, showing that the half adder works for all possible cases.

B. Full Adder

Following the pattern of the half adder, all combinations of the full adder's 'A', 'B', and 'CI' are tested. The following results are:

```
# A:0, B:0, CI (Carry in):0, S (Sum):0, CO (Carry-out):0
# A:0, B:0, CI (Carry in):1, S (Sum):1, CO (Carry-out):0
# A:0, B:1, CI (Carry in):0, S (Sum):1, CO (Carry-out):0
# A:0, B:1, CI (Carry in):1, S (Sum):0, CO (Carry-out):1
# A:1, B:0, CI (Carry in):0, S (Sum):1, CO (Carry-out):0
# A:1, B:0, CI (Carry in):1, S (Sum):0, CO (Carry-out):1
# A:1, B:1, CI (Carry in):0, S (Sum):0, CO (Carry-out):1
# A:1, B:1, CI (Carry in):1, S (Sum):1, CO (Carry-out):1
```

Once again checking each case, it can be seen that all cases have correctly been calculated. The full adder has been correctly implemented.

C. Ripple-Carry Adder/Subtractor

Moving on to the Ripple-Carry Adder/Subtractor, not all cases can be checked due to having 32-bit capability for inputs and outputs. Random cases will be checked to see if the results hold up. The results are:

```
# A:          0, B:          0, SnA:0, Y (Sum):          0, CO (Final carry-out/overflow bit):0
# A:         32, B:         32, SnA:1, Y (Sum):          0, CO (Final carry-out/overflow bit):1
# A:         32, B:4294967264, SnA:0, Y (Sum):          0, CO (Final carry-out/overflow bit):1
# A:         32, B:         16, SnA:0, Y (Sum):         48, CO (Final carry-out/overflow bit):0
# A:         16, B:4294967264, SnA:1, Y (Sum):         48, CO (Final carry-out/overflow bit):0
# A:          0, B:         32, SnA:0, Y (Sum):         32, CO (Final carry-out/overflow bit):0
# A:2147483647, B:2147483647, SnA:0, Y (Sum):4294967294, CO (Final carry-out/overflow bit):0
# A:2147483647, B:2147483647, SnA:1, Y (Sum):          0, CO (Final carry-out/overflow bit):1
# A:2147483647, B:2147483648, SnA:0, Y (Sum):4294967295, CO (Final carry-out/overflow bit):0
```

Double-checking the calculation along with the 'SnA' field, the Ripple-Carry Adder/Subtractor has been shown in work in all cases, even detecting when an overflow occurs, showing that the Ripple-Carry Adder/Subtractor is properly implemented.

D. Multiplexer

There are multiple MUX's to check. As stated in the previous section, the testbench for the 32-bit MUX's has been coded to have the last two digits match the selection wire in order to make confirming the results easier. The 1-bit MUX's will test all combinations due to their low amount of combination.

1) 1-bit 2x1 MUX

The three fields, 'I0', 'I1', and 'S' will be set to '0' or '1' in all possible combinations. 'Y' will then be checked to see if the correct value has been chosen.

```
# I0:0, I1:0, S:0, Y:0
# I0:0, I1:0, S:1, Y:0
# I0:0, I1:1, S:0, Y:0
# I0:0, I1:1, S:1, Y:1
# I0:1, I1:0, S:0, Y:1
# I0:1, I1:0, S:1, Y:0
# I0:1, I1:1, S:0, Y:1
# I0:1, I1:1, S:1, Y:1
```

As shown, the correct 'Y' output appears for its respective 'S' field. When 'S' is '0', 'Y' is equal to 'I0', and vice versa. When both '' and '' are '0' or '1', 'Y' is the same.

2) 32-bit 2x1 MUX

The same is done for the 32-bit 2x1 MUX, except that once again, not all combinations can be checked. Following what was outlined earlier, the last two digits will confirm if the MUX works.

```
# I0:          0, I1:          0, S:0, Y:          0
# I0:1431655700, I1:1431655701, S:0, Y:1431655700
# I0:1431655700, I1:1431655701, S:1, Y:1431655701
```

As seen, when 'S' is '0', the last two digits of the output 'Y' are '00'. When 'S' is '1', the last two digits of 'Y' are '01'. This test confirms that the MUX is able to select between two 32-bit values without issue.

3) 64-bit 2x1 MUX

The 64-bit version of the 2x1 MUX follows the same approach as the 32-bit testbench. Due to the limitations of ModelSim, literal values larger than 32-bit cannot be input. Instead, the same test cases will be reused from the 32-bit 2x1 MUX test.

```
# I0:                  0, I1:                  0, S:0, Y:                  0
# I0:         1431655700, I1:         1431655701, S:0, Y:         1431655700
# I0:         1431655700, I1:         1431655701, S:1, Y:         1431655701
```

Since the results are the same as the 32-bit 2x1 MUX, it has been shown that the 64-bit 2x1 MUX is fully functional as well.

4) 32-bit 4x1 MUX

The 4x1 MUX will now select from four inputs, all of which are 32-bit. The results are as shown:

```
# I0:          0, I1:          0, I2:          0, I3:          0, S:0 Y:          0
# I0:1431655700, I1:1431655701, I2:1431655702, I3:1431655703, S:0 Y:1431655700
# I0:1431655700, I1:1431655701, I2:1431655702, I3:1431655703, S:1 Y:1431655701
# I0:1431655700, I1:1431655701, I2:1431655702, I3:1431655703, S:2 Y:1431655702
# I0:1431655700, I1:1431655701, I2:1431655702, I3:1431655703, S:3 Y:1431655703
```

It can be seen that for the selection input, the corresponding 'Y' output is correct. This can be seen from the correct last two digits that match up with the selection wire. The 4x1 MUX is properly implemented.

5) 32-bit 8x1 MUX

As the MUX gets larger, so does the number of inputs. Now checking eight inputs, the results are:

```
S:0 Y:          0
S:0 Y:1431655700
S:1 Y:1431655701
S:2 Y:1431655702
S:3 Y:1431655703
S:4 Y:1431655704
S:5 Y:1431655705
S:6 Y:1431655706
S:7 Y:1431655707
```

Due to the length of the resultant transcript, all test results from this one forward will only display the selection input and the 'Y' output. However, once again, it can be seen that the last two digits are aligning with its selection input, showing that the 32-bit 8x1 MUX is also correctly implemented.

6) 32-bit 16x1 MUX

With sixteen inputs now, from selection 0-15, the results are:

```
S: 0 Y:          0
S: 0 Y:1431655700
S: 1 Y:1431655701
S: 2 Y:1431655702
S: 3 Y:1431655703
S: 4 Y:1431655704
S: 5 Y:1431655705
S: 6 Y:1431655706
S: 7 Y:1431655707
S: 8 Y:1431655708
S: 9 Y:1431655709
S:10 Y:1431655710
S:11 Y:1431655711
S:12 Y:1431655712
S:13 Y:1431655713
S:14 Y:1431655714
S:15 Y:1431655715
```

Looking at the last two digits, they are counting up, matching the selection wires that are also counting up. The implementation of the 32-bit 16x1 MUX is successful.

7) 32-bit 32x1 MUX

Finally, the 32-bit 32x1 MUX has thirty-two inputs to check, making for a very lengthy testbench result.

```
S: 0 Y:          0
S: 0 Y:1431655700
S: 1 Y:1431655701
S: 2 Y:1431655702
S: 3 Y:1431655703
S: 4 Y:1431655704
S: 5 Y:1431655705
S: 6 Y:1431655706
S: 7 Y:1431655707
S: 8 Y:1431655708
S: 9 Y:1431655709
S:10 Y:1431655710
S:11 Y:1431655711
S:12 Y:1431655712
S:13 Y:1431655713
S:14 Y:1431655714
S:15 Y:1431655715
S:16 Y:1431655716
S:17 Y:1431655717
S:18 Y:1431655718
S:19 Y:1431655719
S:20 Y:1431655720
S:21 Y:1431655721
S:22 Y:1431655722
S:23 Y:1431655723
S:24 Y:1431655724
S:25 Y:1431655725
S:26 Y:1431655726
S:27 Y:1431655727
S:28 Y:1431655728
S:29 Y:1431655729
S:30 Y:1431655730
S:31 Y:1431655731
```

From this, it can be seen that even with thirty-two inputs, the MUX is functioning as intended. With this, the 32-bit 32x1 MUX, along with all previous MUX's are successfully implemented.

E. Unsigned & Signed Multiplier

1) Unsigned multiplier

Multiplying unsigned test values and checking their results:

```
# A:          0, B:          0, HI:          0, LO:          0
# A:          2, B:          5, HI:          0, LO:         10
# A:          5, B:          2, HI:          0, LO:         10
# A:       3781, B:       7132, HI:          0, LO:   26966092
# A:4294967295, B:4294967295, HI:4294967294, LO:          1
```

Checking each of these cases, the resultant 'HI' and 'LO' outputs are correct. It can be seen that for large number multiplication, the 'HI' output is used, and with the correct output also. Though the answer may look strange for the last test case, it is simply due to the binary value being printed as decimal. Converting it back to binary shows that the answer is correct, proving that unsigned multiplication is complete.

2) Signed multiplier

Multiplying signed test values and checking their results:

```
# A:          0, B:          0, HI:          0, LO:          0
# A:          2, B:          5, HI:          0, LO:         10
# A:          5, B:          2, HI:          0, LO:         10
# A:       3781, B:       7132, HI:          0, LO:   26966092
# A:4294967295, B:4294967295, HI:          0, LO:          1
# A:4294967294, B:          5, HI:4294967295, LO:4294967286
# A:4294967291, B:          2, HI:4294967295, LO:4294967286
# A:       3781, B:4294960164, HI:4294967295, LO:4268001204
# A:          1, B:          1, HI:          0, LO:          1
```

The results show that unsigned multiplication still functions as intended. However, the unsigned multiplication is shown to work as well, despite the large values. Once again converting the decimal values back to binary and combining 'HI' and 'LO' together to acquire one 64-bit value, the signed multiplication results are correct. The signed multiplication component has been completed.

F. Barrel Shifter

The complete barrel shifter with left and right control, as well as shift amount control, is made up of several, lower-level barrel shifters, all of which require testing.

1) Right barrel shift

Firstly, right barrel shifting is tested by inputting random test cases along with the shift amount. The results of the test cases are as shown:

```
# D:          0, S: 0, Y:          0
# D:          1, S: 1, Y:          0
# D:         15, S: 2, Y:          3
# D:        200, S: 3, Y:         25
# D:2147483647, S:10, Y:    2097151
```

From these results, it can be seen and confirmed that each 'D' value is being shifted right by the number of 'S' to obtain the correct 'Y' output. The right shift has been correctly implemented.

2) Left barrel shift

Next, the left barrel shift will reuse the same test cases as the right shift in order to check that the same test cases can also be shifted left without issue.

```
# D:          0, S: 0, Y:          0
# D:          1, S: 1, Y:          2
# D:         15, S: 2, Y:         60
# D:        200, S: 3, Y:       1600
# D:2147483647, S:10, Y:4294966272
```

As seen by the results once again, the value of 'Y' is the correct result of 'D' being left shifted by the number of 'S'. The left shift is functioning correctly.

3) 32-bit barrel shifter with Left or Right control

This component is the same as the left and right shift, except that it can decide which direction it shifts with the 'LnR' field. The same test cases from left and right shift will be tested again in this barrel shifter.

```
# D:          0, S: 0, LnR:0, Y:          0
# D:          0, S: 0, LnR:1, Y:          0
# D:          1, S: 1, LnR:0, Y:          0
# D:          1, S: 1, LnR:1, Y:          2
# D:         15, S: 2, LnR:0, Y:          3
# D:         15, S: 2, LnR:1, Y:         60
# D:        200, S: 3, LnR:0, Y:         25
# D:        200, S: 3, LnR:1, Y:       1600
# D:2147483647, S:10, LnR:0, Y:    2097151
# D:2147483647, S:10, LnR:1, Y:4294966272
```

Since the results of the shifting match up with the left and right shift, respectively, it is shown that the 32-bit barrel shifter with Left and Right control is correctly implemented.

4) 32-bit shifter

Finally, the 32-bit shifter tests the same cases as the 32-bit barrel shifter.

```
# D:          0, S:          0, LnR:0, Y:          0
# D:          0, S:          0, LnR:1, Y:          0
# D:          1, S:          1, LnR:0, Y:          0
# D:          1, S:          1, LnR:1, Y:          2
# D:         15, S:          2, LnR:0, Y:          3
# D:         15, S:          2, LnR:1, Y:         60
# D:        200, S:          3, LnR:0, Y:         25
# D:        200, S:          3, LnR:1, Y:       1600
# D:2147483647, S:         10, LnR:0, Y:    2097151
# D:2147483647, S:         10, LnR:1, Y:4294966272
```

Since the results are the same as the 32-bit barrel shifter, the 32-bit shifter, along with all the previous shifters, has been successfully implemented.

G. ALU

Using the same testbench as the previous projects, the results are as shown:

```
# [TEST] 15 + 3 = 18 , got 18 ... [PASSED]
# [TEST] 15 - 5 = 10 , got 10 ... [PASSED]
# [TEST] 15 + 5 = 20 , got 20 ... [PASSED]
# [TEST] 2 * 5 = 10 , got 10 ... [PASSED]
# [TEST] 4 * 0 = 0 , got 0 ... [PASSED]
# [TEST] 7 >> 1 = 3 , got 3 ... [PASSED]
# [TEST] 7 >> 2 = 1 , got 1 ... [PASSED]
# [TEST] 7 >> 3 = 0 , got 0 ... [PASSED]
# [TEST] 7 << 1 = 14 , got 14 ... [PASSED]
# [TEST] 7 << 2 = 28 , got 28 ... [PASSED]
# [TEST] 7 << 3 = 56 , got 56 ... [PASSED]
# [TEST] 15 & 5 = 5 , got 5 ... [PASSED]
# [TEST] 15 | 5 = 15 , got 15 ... [PASSED]
# [TEST] 15 ~| 5 = 4294967280 , got 4294967280 ... [PASSED]
# [TEST] 2 < 1 = 0 , got 0 ... [PASSED]
# [TEST] 1 < 2 = 1 , got 1 ... [PASSED]
#
#       Total number of tests          16
#       Total number of pass           16
```

Since all test cases have passed, it has been shown that the ALU has been correctly implemented and is fully functional.

H. SR-Latch

Comparing the SR-Latch results to its truth table:

```
# S:1, R:0, C:1, nP:1, nR:1, Q:1, Qbar:0
# S:x, R:x, C:0, nP:1, nR:1, Q:1, Qbar:0
# S:0, R:0, C:1, nP:1, nR:1, Q:1, Qbar:0
# S:0, R:1, C:1, nP:1, nR:1, Q:0, Qbar:1
# S:x, R:x, C:0, nP:1, nR:1, Q:0, Qbar:1
# S:0, R:0, C:1, nP:1, nR:1, Q:0, Qbar:1
# S:x, R:x, C:x, nP:0, nR:1, Q:1, Qbar:x
# S:x, R:x, C:x, nP:1, nR:0, Q:x, Qbar:1
```

It can be seen that the correct results for ‘Q’ and ‘Qbar’ are obtained with their corresponding inputs. The SR-Latch is correctly implemented and ready to be used.

I. D-Latch

Comparing the D-Latch test results to its truth table:

```
# D:1, C:1, nP:1, nR:1, Q:1, Qbar:0
# D:x, C:0, nP:1, nR:1, Q:1, Qbar:0
# D:0, C:1, nP:1, nR:1, Q:0, Qbar:1
# D:x, C:0, nP:1, nR:1, Q:0, Qbar:1
# D:x, C:1, nP:0, nR:1, Q:1, Qbar:x
# D:x, C:1, nP:1, nR:0, Q:x, Qbar:1
```

Once again, it can be seen that the outputs are correct for each given set of inputs. The D-Latch has been implemented correctly and is ready for usage.

J. FlipFlop

Comparing the FlipFlop test results to its truth table:

```
# D:1, C:0, nP:1, nR:1, Q:x, Qbar:x
# D:1, C:1, nP:1, nR:1, Q:1, Qbar:0
# D:x, C:0, nP:1, nR:1, Q:1, Qbar:0
# D:0, C:0, nP:1, nR:1, Q:1, Qbar:0
# D:0, C:1, nP:1, nR:1, Q:0, Qbar:1
# D:x, C:0, nP:1, nR:1, Q:0, Qbar:1
# D:x, C:1, nP:0, nR:1, Q:1, Qbar:0
# D:x, C:1, nP:1, nR:0, Q:0, Qbar:1
```

Checking each case and result, we can see that the FlipFlop passes all possible cases, meaning it is successfully implemented, along with the previous Latch components.

K. Decoder

Since decoder results are one-hot, meaning that only one bit is a ‘1’ and the rest of the bits are ‘0’’s, checking that the decoder works a simple matter.

1) 2-to-4 line decoder

The 2-to-4 line decoder takes in a 2-bit input, values 0-3, converting them into a 4-bit, one-hot value.

```
# I:0, D:0001
# I:1, D:0010
# I:2, D:0100
# I:3, D:1000
```

As seen in these results, there is only one ‘1’ per ‘D’ result. It can also been determined that the value of ‘I’ determines the bit location of the ‘1’. When ‘I’ is ‘0’, the ‘1’ is in the 0^{th} position, or the very lowest bit. From this, a pattern can be seen that as long as ‘I’ is ascending consecutively, the ‘1’’s will appear in a diagonal fashion, from the top-right to bottom-left. This pattern will be used to confirm the accuracy of the future decoder results. With this, however, the 2-to-4 line decoder has been shown to work successfully.

2) 3-to-8 line decoder

Now with a 3-bit input, values 0-7, there are 8-bit value results. They are as shown:

```
# I:0, D:00000001
# I:1, D:00000010
# I:2, D:00000100
# I:3, D:00001000
# I:4, D:00010000
# I:5, D:00100000
# I:6, D:01000000
# I:7, D:10000000
```

Checking the pattern found in the previous decoder, it can be seen that as ‘I’ ascends consecutively in value, the ‘1’’s show up in a diagonal pattern. This shows that the 3-to-8 line decoder works as intended.

3) 4-to-16 line decoder

From a 4-bit input, values 0-15, 16-bit values are the output result.

```
# I: 0, D:0000000000000001
# I: 1, D:0000000000000010
# I: 2, D:0000000000000100
# I: 3, D:0000000000001000
# I: 4, D:0000000000010000
# I: 5, D:0000000000100000
# I: 6, D:0000000001000000
# I: 7, D:0000000010000000
# I: 8, D:0000000100000000
# I: 9, D:0000001000000000
# I:10, D:0000010000000000
# I:11, D:0000100000000000
# I:12, D:0001000000000000
# I:13, D:0010000000000000
# I:14, D:0100000000000000
# I:15, D:1000000000000000
```

With the '1''s moving diagonally in the same pattern, the 4-to-16 line decoder has been proven to work.

4) 5-to-32 line decoder

Finally, the 5-to-32 line decoder takes in a 5-bit input, values 0-31, the resultant 'D' is a 32-bit value.

```
# I: 0, D:00000000000000000000000000000001
# I: 1, D:00000000000000000000000000000010
# I: 2, D:00000000000000000000000000000100
# I: 3, D:00000000000000000000000000001000
# I: 4, D:00000000000000000000000000010000
# I: 5, D:00000000000000000000000000100000
# I: 6, D:00000000000000000000000001000000
# I: 7, D:00000000000000000000000010000000
# I: 8, D:00000000000000000000000100000000
# I: 9, D:00000000000000000000001000000000
# I:10, D:00000000000000000000010000000000
# I:11, D:00000000000000000000100000000000
# I:12, D:00000000000000000001000000000000
# I:13, D:00000000000000000010000000000000
# I:14, D:00000000000000000100000000000000
# I:15, D:00000000000000001000000000000000
# I:16, D:00000000000000010000000000000000
# I:17, D:00000000000000100000000000000000
# I:18, D:00000000000001000000000000000000
# I:19, D:00000000000010000000000000000000
# I:20, D:00000000000100000000000000000000
# I:21, D:00000000001000000000000000000000
# I:22, D:00000000010000000000000000000000
# I:23, D:00000000100000000000000000000000
# I:24, D:00000001000000000000000000000000
# I:25, D:00000010000000000000000000000000
# I:26, D:00000100000000000000000000000000
# I:27, D:00001000000000000000000000000000
# I:28, D:00010000000000000000000000000000
# I:29, D:00100000000000000000000000000000
# I:30, D:01000000000000000000000000000000
# I:31, D:10000000000000000000000000000000
```

Seeing that the same pattern is intact in this final decoder, it proves that the 5-to-32 line decoder, and all other decoders prior, have been correctly implemented.

L. Register

1) 1-bit Register

The 1-bit register tests all combinations of inputs. The results are as shown:

```
# D:1 C:0 L:1 nP:1 nR:1 Q:x Qbar:x
# D:1 C:1 L:1 nP:1 nR:1 Q:1 Qbar:0
# D:x C:0 L:1 nP:1 nR:1 Q:1 Qbar:0
# D:0 C:0 L:1 nP:1 nR:1 Q:1 Qbar:0
# D:0 C:1 L:1 nP:1 nR:1 Q:0 Qbar:1
# D:x C:0 L:1 nP:1 nR:1 Q:0 Qbar:1
# D:x C:1 L:1 nP:0 nR:1 Q:1 Qbar:0
# D:x C:1 L:1 nP:1 nR:0 Q:0 Qbar:1
# D:1 C:0 L:0 nP:1 nR:1 Q:0 Qbar:1
# D:1 C:1 L:0 nP:1 nR:1 Q:0 Qbar:1
# D:x C:0 L:0 nP:1 nR:1 Q:0 Qbar:1
# D:0 C:0 L:0 nP:1 nR:1 Q:0 Qbar:1
# D:0 C:1 L:0 nP:1 nR:1 Q:0 Qbar:1
# D:x C:0 L:0 nP:1 nR:1 Q:0 Qbar:1
# D:x C:1 L:0 nP:0 nR:1 Q:1 Qbar:0
# D:x C:1 L:0 nP:1 nR:0 Q:0 Qbar:1
```

Checking each case, it can be seen that the correct 'Q' is obtained for every case. Since the register does not need 'Qbar', its result is not considered. The 1-bit register has been successfully implanted.

2) 32-bit Register

Following the same testing approach as the 1-bit register, except that 32-bit values are input instead, the results are:

```
# D:4294967295 LOAD:1 CLK:0 RESET:1 Q:          x
# D:4294967295 LOAD:1 CLK:1 RESET:1 Q:4294967295
# D:          x LOAD:1 CLK:0 RESET:1 Q:4294967295
# D:          x LOAD:1 CLK:1 RESET:0 Q:          0
# D:4294967295 LOAD:1 CLK:0 RESET:1 Q:          0
# D:4294967295 LOAD:1 CLK:1 RESET:1 Q:4294967295
# D:          0 LOAD:1 CLK:0 RESET:1 Q:4294967295
# D:          0 LOAD:1 CLK:1 RESET:1 Q:          0
# D:          x LOAD:1 CLK:0 RESET:1 Q:          0
# D:4294967295 LOAD:0 CLK:0 RESET:1 Q:          0
# D:4294967295 LOAD:0 CLK:1 RESET:1 Q:          0
# D:          x LOAD:0 CLK:0 RESET:1 Q:          0
# D:          x LOAD:0 CLK:1 RESET:0 Q:          0
# D:4294967295 LOAD:0 CLK:0 RESET:1 Q:          0
# D:4294967295 LOAD:0 CLK:1 RESET:1 Q:          0
# D:          0 LOAD:0 CLK:0 RESET:1 Q:          0
# D:          0 LOAD:0 CLK:1 RESET:1 Q:          0
# D:          x LOAD:0 CLK:0 RESET:1 Q:          0
```

Watching the 'LOAD', 'CLK', and 'RESET' signal values, it can be seen that values that are passed in are held until it is either reset or a new value is loaded in, following what a register should do. Looking through the results sequentially, it is clear that the 32-bit register has been correctly implemented.

3) 32x32-bit Register

Using the provided testbench, the results are:

```
#       Total number of tests          32
#       Total number of pass           32
```

Having successfully completed all given test cases, the 32x32-bit register has been successfully implemented, along with all previous register components.

M. Data Path

Because the data path requires all components, especially the control unit, to work individually, it does not need its own testbench. It is instead tested alongside the entire system.

N. System

At the very end, the last test is to run 'da_vinci_tb.v' and check its results. The testing is the same as project two's 'RevFib' and 'Fibonacci' test cases. If the output of both cases match with their respective 'golden' file, then the system has been correctly implemented.

Unfortunately, despite how far I have gotten on the project, I was unable to fully complete the DaVinci v1.0m system. The system is unable to load properly due to the data path not being configured correctly. Because of this, several other components such as the control unit are also unable to be tested.

V. Conclusion

At the end of this project and report, I have learned how to implement the arithmetic logic unit and register file at the gate level in Verilog. Additionally, I have completed the optional components leading up to the entire system, however not the system itself. Though unsuccessful in the full implementation, I am proud of my hard work attempting the optional requirements. With a 100% success rate (at least on my generated test cases) on all components that make up, and including, the ALU and register file, I have succeeded in every project of CS 147, fully learning how to implement the components of a computer at the gate level and behavioral model. With that, I have completed the third and final CS147 project in which I successfully implemented the DaVinci v1.0m mixed model computer system's ALU and register file through Verilog in ModelSim.

References

[1] K. Patra, Class Lecture, Topic: "Addition / Subtraction," San José State University, San José, Feb., 27, 2020.

[2] K. Patra, Class Lab, Topic: "Gate Level Modeling I," San José State University, San José, Feb., 24, 2020.

[3] K. Patra, Class Lecture, Topic: "Digital Synthesis, Number Representation," San José State University, San José, Feb., 4, 2020.

[4] K. Patra, Class Lecture, Topic: "Comb/Seq Circuits I," San José State University, San José, Feb., 13, 2020.

[5] K. Patra, Class Lab, Topic: "Gate Level Modeling III," San José State University, San José, Apr., 7, 2020.

[6] K. Patra, Class Lecture, Topic: "Sequential Logic Design, Common Digital Components," San José State University, San José, Feb., 20, 2020.

[7] K. Patra, Class Lecture, Topic: "Multiplication & Division," San José State University, San José, Mar., 3, 2020.

[8] K. Patra, Class Lab, Topic: "Gate Level Modeling II," San José State University, San José, Feb., 26, 2020.

[9] K. Patra, Class Lecture, Topic: "Putting Together a Microprocessor I," San José State University, San José, Mar., 5, 2020.

[10] K. Patra, Class Lab, Topic: "Gate Level Modeling IV," San José State University, San José, Apr., 9, 2020.

[11] K. Patra, Class Lecture, Topic: "Comb/Seq Circuits II," San José State University, San José, Feb., 18, 2020.

[12] K. Patra, Class Lab, Topic: "Gate Level Modeling V," San José State University, San José, Apr., 14, 2020.

[13] K. Patra, Class Lab, Topic: "Gate Level Modeling VI," San José State University, San José, Apr., 16, 2020.

[14] K. Patra, Class Lab, Topic: "Gate Level Modeling VII," San José State University, San José, Apr., 21, 2020.