# Gate Level Modeling for the DaVinciv1.0m Computer System

Michael Wong
Computer Science Department, San Jose State University
California United States
michael.j.wong@sjsu.edu

***Abstract*— In this project, I will explore the gate level modeling of a DaVinci computer system which supports the CS 147DV instruction set. The gate level modeling will cover numerous components, big and small, that make up the functionality of the computer system. This report will demonstrate my process in creating this computer system and testing for its proper functionality.**

## I. INTRODUCTION

Using the HDL simulator, ModelSim, I will build upon the previous project by adding to its structural components. The task in the second project was to design the CS 147DV instruction set. For Project 3 however, I will be taking it a step further by implementing its computer system, using gate level modeling to obtain a greater understanding of how the processor and memory works together. The purpose of this project is to create a fully functional processor, memory model, and system to integrate the two to work together. Testing will then be used to validate the correctness of my design.

## II. REQUIREMENTS

Much like many of the other processors out in the open market, the processor that I am tasked with creating in this project requires similar components. Those requirements will be detailed in this section.

### *A. ALU*

All processors require a type of ALU in order to perform arithmetic and logical operations. This particular ALU will support the following operations: addition, subtraction, multiplication, integer shift left, integer shift right, and logical, or logical, nor logical, and set less than. These nine operations are required to be implemented in this processor's ALU for it properly yield correct results. Inside the ALU will be universal logic gates, multiplexers and a functional ripple carry adder/subtractor. The ALU must be 32-bit.

### *B. Register File*

Another key element that is required for any processor is the register file component. The purpose of the register file is to act as a fast storage element that the CPU can access quickly. Also, it is the place where the instructions, such as for the various ALU operations, are stored. The specifications for the register file that must be implemented for this project are that it must be contain 32 registers that can store a 32-bit word. This register must also have a dual read register. Components such as decoders and multiplexer will be required to create the register file.

### *C. Memory*

The memory model requirements for this project is that the memory needs to be 256MB. Adding on to the 256MB requirement, the memory needs to be double word, 32-bit, addressable 64M address. This component will act as a higher-level storage component that exist away from the processor. The processor will be able to access this memory and perform the necessary tasks, reading or writing, when prompted to.

### *D. Control Unit and Data Path*

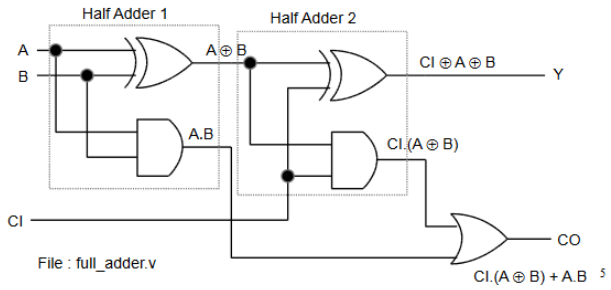
Much like Project 2, this project requires the usage of a control unit to dictate different states of the processor. Just like the previous project, the processor states that this project will follow are as follows: Instruction Fetch, Instruction Decode, Execution, Memory Access, and Write Back. In addition, a data path will be necessary to regulate all the actions of the computer system consisting of all the elements named prior

## III. Design and Implementation

Now with the requirements of this computer system established, I can now move on to discussing the design and implementation of the computer system. This section will first discuss the minor components of the processor, such as logic gates, multiplexers, and decoders, and then move on to the major aspects of the system. The smaller components will be designed and implemented which will then be used in the much larger and significant components that make up the system.

### *A. Full Adder*

Before I can discuss the ripple carry adder/subtractor, I need to address the design and implementation of the of the full adder. A critical element of the adder/subtractor. The design of the full adder is to take two half adders and merge them together to create a full adder. The diagram below details the full adder's design.

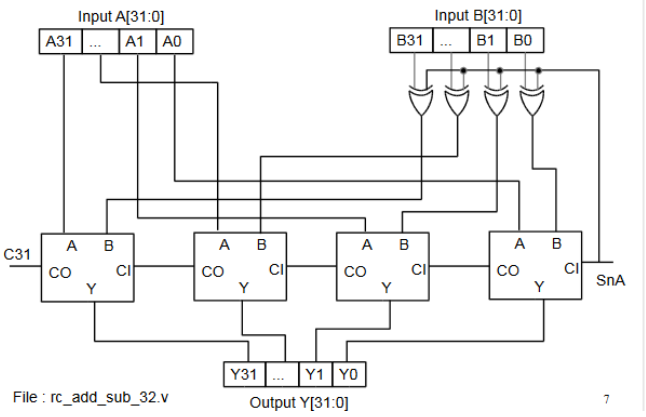



By observing the diagram provided to me through lab slides, I am able to see that implementing the half adder will require an xor and and gate. Verilog supports one-bit logic gates, so by utilizing the gates provided, I was able to follow the circuit's design to implement the half adder. Once the half adder is created, incorporate the or gate to produce a carry out bit. The full adder takes in two values and a carry in. Its output is the sum/difference of the two values passed through and the carry out bit from the resulting operation. Create two half adders to produce the result and an or gate completed the full adder.

```
module FULL_ADDER(S,CO,A,B, CI);
output S,CO;
input A,B, CI;


//TBD
wire firstAddSum;
wire firstCI;
wire secCI;

//first half adder
//xor inst1(S, A, B);
//and inst2(CO, A, B);
HALF_ADDER instFirst(.Y(firstAddSum), .C(firstCI), .A(A), .B(B));


//second half adder
//and inst3(CI, CI, S);
//or  inst4(CO, CO, CI);
HALF_ADDER instSec(.Y(S), .C(secCI), .A(firstAddSum), .B(CI));

//get the final carryout
or instCO(CO, firstCI, secCI);

endmodule;
```

### *B. Ripple Carry Adder/Subtractor*

The ripple carry adder/subtractor will be used in the ALU to produce proper results regarding addition or subtraction operations. The adder will use the full adder to achieve its design goal. Since this adder is 32-bit, it will take in two 32-bit buses of input and calculate the output from the data held in those two 32-bit buses. The full adder that was created in the previous section only supports 1-bit input and outputs. Hench, the design for this adder requires 32 full adders to achieve its goal. The figure below displays the design for the adder/subtractor.




Some details to note are the xor gates attached to the second inputs before being passed into the full adders. The purpose of those xor gates are to pass in the appropriate value for an addition or subtraction operation. The other input being passed into the xor gates are the SnA signal. The SnA signal determines if the operation being passed is addition

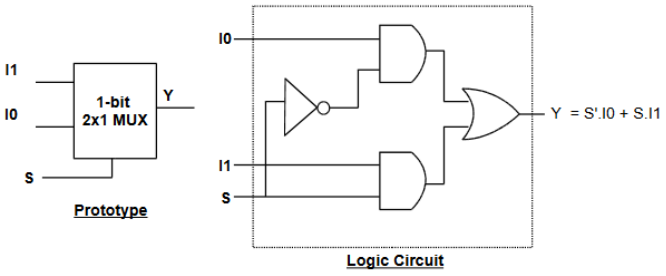
or subtraction. If SnA is 0, then input B is unchanged, otherwise the inverted bit is passed for subtraction.

To implement, I used the genvar variable offered in Verilog to loop through 31 full adders and set their output to the output of the ripple carry adder/subtractor. The first full adder takes in a unique carry in input, SnA, which is why it is outside the generated loop. To keep track of the carry in's and the output of the xor gates, I used wires to store that data. Wires act as physical wires that will consistently hold data and can also be altered on demand. It prevents confusion if one is trying to set an output or input over and over.

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

// TBD
wire [`DATA_INDEX_LIMIT:0] BthruX;      //holds the value of passing B and SnA through an xor gate
wire [`DATA_INDEX_LIMIT:0] nextCI;      //holds the value of for the CO to be passed into CI at the next adder

xor firstints(BthruX[0], B[0], SnA);
FULL_ADDER inst1(.S(Y[0]), .CO(nextCI[0]), .A(A[0]), .B(BthruX[0]), .CI(SnA));

genvar i;
generate
for(i = 1; i < `DATA_WIDTH; i = i+1)
begin
     xor intsxor(BthruX[i], SnA, B[i]);
     FULL_ADDER inst2(.S(Y[i]), .CO(nextCI[i]), .A(A[i]), .B(BthruX[i]), .CI(nextCI[i-1]));
end
endgenerate

//overflow detector
xor(CO, nextCI[31], nextCI[30]);

endmodule
```

## C. *Multiplexers*

The next component that I will discuss are multiplexers. Multiplexers are a device that take in n number of inputs and depending on an inputted signal, select one of those n inputs to output. This computer system utilizes many multiplexers, some which were used to simply build off one another. In the interest of length, I will present only a few of the multiplexers that were implemented in this project. The first is the 1-bit 2x1 multiplexer that underlies all the other bigger multiplexers. The figure beneath shows its design.



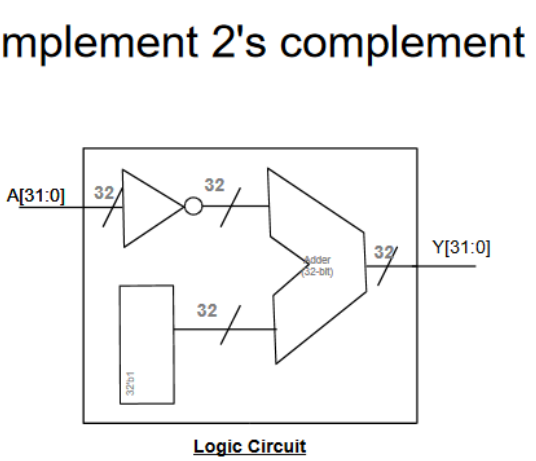

File : mux.v

To start creating such a multiplexer, I created a 32-bit 4x1 multiplexer that relies on two of its predecessors and then passes the result of the two into a 32-bit 2x1 multiplexer to determine its final decision. The lower bits of the signal dictate the first two multiplexer's output and the highest bit for the signal determines the last multiplexer's output. From 4, one will proceed to 8, then 16 and finally 32. Following the implementation pattern, the 32x1 multiplexer will require the usage of two 16x1 multiplexers and then passing those two results into a 2x1 multiplexer to yield the desired output.

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

// TBD
genvar i;
generate
for(i = 0; i < 32; i = i + 1)
begin
        MUX1_2x1 mux(.Y(Y[i]), .I0(I0[i]), .I1(I1[i]), .S(S));

end
endgenerate

endmodule

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

// TBD
wire firstAnd;
wire secAnd;
wire notS;

not inst2(notS, S);
and inst1(firstAnd, notS, I0);
and inst3(secAnd, S, I1);
or  inst4(Y, secAnd, firstAnd);

endmodule
```

## D. *Two's Complement*

2's complement is used to produce the signed value of binary number. To implement a component to output a 2's complement number from a binary number, I followed the simple circuit below.

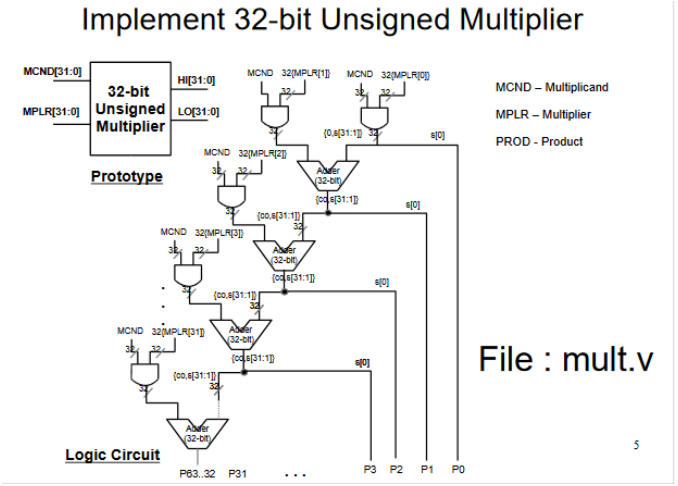



It simply passes an inverted input and a 32-bit binary 1 into the ripple carry adder/subtractor. The adder will take in the two inputs and produce the two's complement form of the binary number. Here's the code.

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

// TBD
wire [31:0] invA;
wire CO;                //pointless for the twos complement

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

genvar i;
generate
for(i = 0; i < 32; i = i+1)
begin
     not inst1(invA[i], A[i]);
end
endgenerate

RC_ADD_SUB_32 inst(.Y(Y), .CO(CO), .A(invA), .B(one), .SnA(signal));

endmodule
```

## *E. Multiplier*

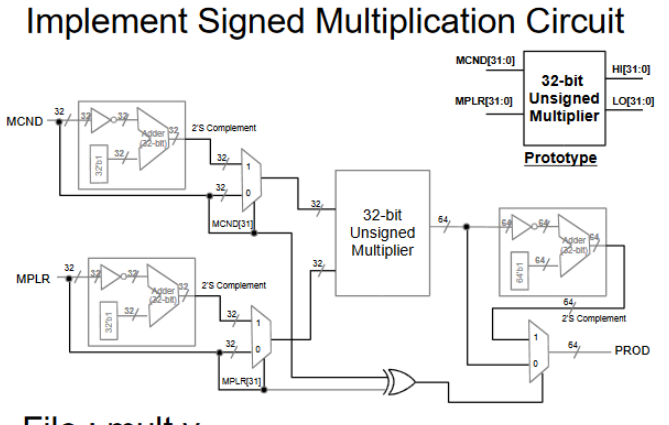
There are two version for the multiplier, signed and an unsigned version. Both are 32-bit, but the signed multiplier relies on the unsigned multiplier to calculate its result. The unsigned multiplier is a sequential circuit that takes in two inputs and follows through a series of adders and and gates. The circuit is detailed in the diagram below.




Implementation follows the design circuit which requires 32 and gates and adders to implement. I placed these components inside a generated loop to create and pass these arguments down the series of adders and and gates. The initial case has to be handled outside the generated loop, similar to ripple carry adder's implementation since it requires a unique input. Here's the code

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

// TBD
wire carryO [31:0];             //represents the carry out output from the adder
wire [31:0] OP2 [31:0];         //the second operation to go into the adders, need 32 32 bit operands

AND32_2x1 firstAnd(.Y(OP2[0]), .A(A), .B({32{B[0]}}));  //that thing at the end there is to make it 32 bit replicated
buf b0 (carryO[0], 1'b0);
buf b1(LO[0], OP2[0][0]);

//loop to move down the circuit and get a result
genvar r;
generate
for(r = 1; r < 32; r = r + 1)
begin
        wire [31:0] OP1;        //the first operand to go into the adder
        AND32_2x1 leap(.Y(OP1), .A(A), .B({32{B[r]}}));
        RC_ADD_SUB_32 calcu(.Y(OP2[r]), .CO(carryO[r]), .A(OP1), .B({carryO[r-1], (OP2[r-1][31:1])}), .SnA(1'b0));
        buf b2(LO[r], OP2[r][0]);
end
endgenerate

//assign the hi registers to the last carryO value and op2 value
BUF32_1x1 instbuff(.Y(HI), .A({carryO[31], (OP2[31][31:1])}));

endmodule
```

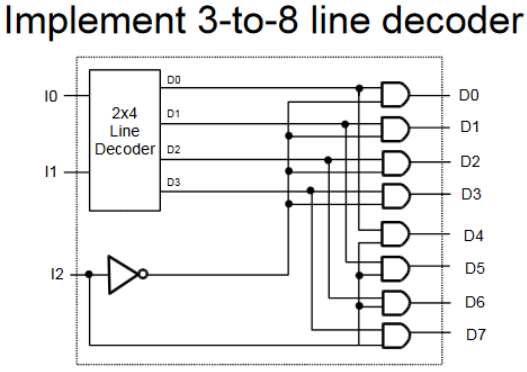
The design for the signed multiplier is easier to follow than the unsigned. The figure below displays its circuit.



Since this multiplier needs to handle signed and unsigned operands, the implementation can be viewed as a split between signed and unsigned. The multiplexer handles that split. The xor gate remembers the sign of both operands and its output is used as the signal when deciding to use the signed or unsigned version of the product. Regardless if the values will be in two's complement form or not, they can still be passed into the unsigned multiplier because unsigned numbers have a larger range than unsigned. Also if needed, the product will be converted to the two's complement form after computation.

*F. Decoder*

A decoder is a digital component that takes in a specific value of data and outputs that data along with their inverted versions. For example, passing two data fields, A and B will produce outputs A and B along with A bar and B bar. For this project, I first examined the design of a 2x4 line decoder. [design]
From there I created 3x8 line decoders to 5x32 line decoders. The designs build off of each other in order to limit the number of gates in their respected design.




File: logic.v

Implementation follows the design circuits of each line decoder and which requires the implementation of their predecessor. The number of inputs determine the number of outputs that the decoder will have. In the instance of a 5x32 line decoder, 5 bits can make up 32 unique bits of data, hence why there are 32 bits of output. Code for the decoders can be found below.

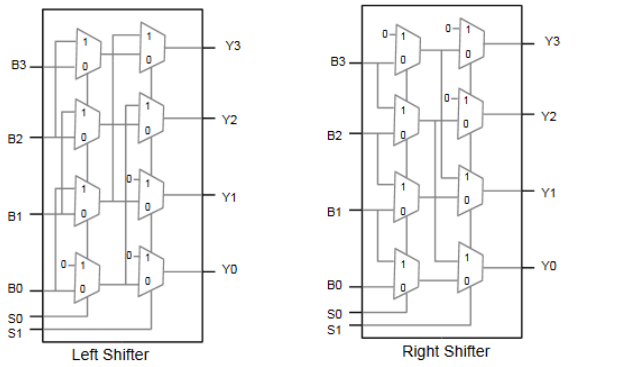
```
// 2x4 Line decoder
module DECODER_2x4(D,I);
// output
output [3:0] D;
// input
input [1:0] I;

// TBD

wire invI0;
wire invI1;

not(invI0, I[0]);
not(invI1, I[1]);

and inst0(D[0], invI0, invI1);
and inst1(D[1], invI0, I[1]);
and inst2(D[2], I[0], invI1);
and inst3(D[3], I[0], I[1]);


endmodule
```

```
// 4x16 Line decoder
module DECODER_4x16(D,I);
// output
output [15:0] D;
// input
input [3:0] I;

// TBD
wire invI3;
wire [7:0] out;
wire [2:0] in = I[2:0];
not(invI3, I[3]);

DECODER_3x8(.D(out), .I(in));

and(D[0], out[0], invI3);
and(D[1], out[1], invI3);
and(D[2], out[2], invI3);
and(D[3], out[3], invI3);
and(D[4], out[4], invI3);
and(D[5], out[5], invI3);
and(D[6], out[6], invI3);
and(D[7], out[7], invI3);
and(D[8], out[0], I[3]);
and(D[9], out[1], I[3]);
and(D[10], out[2], I[3]);
and(D[11], out[3], I[3]);
and(D[12], out[4], I[3]);
and(D[13], out[5], I[3]);
and(D[14], out[6], I[3]);
and(D[15], out[7], I[3]);

endmodule
```

## G. *Shifters*

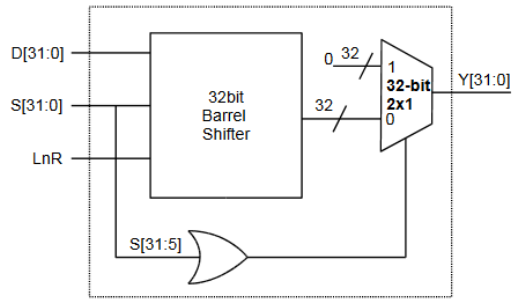
The shifter that Project 3 will utilize is a 32-bit barrel shifter. This barrel shifter will consist of two individual shifters: a 32-bit right shifter and a 32-bit left shifter. These two shifters working together with the assistance of other components like multiplexers and or gates will yield the desired 32-bit shifter for this project. The designs for these shifter components can be found below. Please note that the left and right shifters are only 4-bit shifters. In order for those shifters to be compatible with the 32-bit shifter that we wish to implement, they must be extended to 32-bits.

Extend 4-bit Barrel Shifter to 32-bit




File: barrel_shifter.v 3

Implement 32-bit Barrel Shifter




File: barrel_shifter.v 5

My implementation approach was to generate the left and right shifters first and then incorporate them into the other shifter working up to the final barrel shifter. Prior implementing the two shifters, it was suggested to me that I use the genvar aspect of Verilog to populate all the rows and columns of the left and right shifters. I found this approach to be very confusing and hard to implement so I looked to use the brute force method. This being hard code every multiplexer in the shifters which adds up to about 260 plus multiplexers. Seeing that this approach would be very time consuming and strenuous, I opted to take a hybrid of the two. I hard coded the multiplexers that required one of their inputs to be a bitwise 0. Once those multiplexers where initialized, I used the genvar generation blocks to handle the rest of the multiplexer initializations. I found this approach to be the easiest to code and debug if problems persisted. Here's the code.

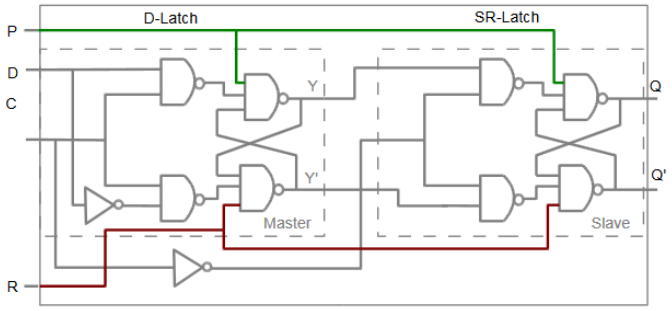
```
module SHIFT32_L(Y,D,S);
// output list
output [31:0] Y;
// input list
input [31:0] D;
input [4:0] S;

// TBD
wire [31:0] result1, result2, result3, result4, result5;
wire [31:0] prev;

genvar j, m, n, k, p, q;
generate
//first column of the shifter
MUX1_2x1 firstI0(.Y(result1[0]), .I0(D[0]), .I1(1'b0), .S(S[0]));
for(j = 1; j < 32; j = j + 1)
begin
        MUX1_2x1 first(.Y(result1[j]), .I0(D[j]), .I1(D[j-1]), .S(S[0]));
end

//BUF32_1x1 inst1(prev, result);

//second column of the shifter
MUX1_2x1 secI0(.Y(result2[0]), .I0(result1[0]), .I1(1'b0), .S(S[1]));
MUX1_2x1 secI1(.Y(result2[1]), .I0(result1[1]), .I1(1'b0), .S(S[1]));
for(m = 2; m < 32; m = m + 1)
begin
        MUX1_2x1 second(.Y(result2[m]), .I0(result1[m]), .I1(result1[m-2]), .S(S[1]));
end

//BUF32_1x1 inst2(prev, result);

//third colum of the shifter
MUX1_2x1 thirdI0(.Y(result3[0]), .I0(result2[0]), .I1(1'b0), .S(S[2]));
MUX1_2x1 thirdI1(.Y(result3[1]), .I0(result2[1]), .I1(1'b0), .S(S[2]));
```

```
module SHIFT32(Y,D,S, LnR);
// output list
output [31:0] Y;
// input list
input [31:0] D;
input [31:0] S;
input LnR;

// TBD
wire [31:0] shiftCheck;
wire [31:0] result;
wire signal;

//the 32 bit barrel shifter
BARREL_SHIFTER32 barrelbarrel(.Y(result), .D(D), .S(S[4:0]), .LnR(LnR));

//multiplexer to choose decide if the shift amount is valid or not
OR32_2x1 inst(.Y(shiftCheck), .A({5'b0,{S[31:5]}}), .B(32'b0));

or inst3(signal, shiftCheck[29], shiftCheck[31]);       //this might not be right

MUX32_2x1 out(.Y(Y), .I0(result), .I1(32'b0), .S(signal));

endmodule
```

## H. *Flip Flop and Register*

Flip flops are a key digital component in regards to the design and implementation of registers. Flip flops are tasked with the sustaining two states by the usage of latches. Registers use the flip flops to flip between different states so that it knows what path it will send data through. Flip flops follow the convention of a master and slave model. In this case, the slave is a SR Latch and the master being a D Latch. The designs of the flip flop applying this convention can be found below.

## Implement 1-bit FlipFlop




Originally, the designs for the D and SR Latches did not contain a third input for the second column of and gates. Those inputs are the reset and preset signals of the flip flop. The design of the latches did not accommodate for those additional inputs, so I had to modify my initial created latches to take those inputs into consideration. Following the design paths, I implemented the flip flops in this fashion

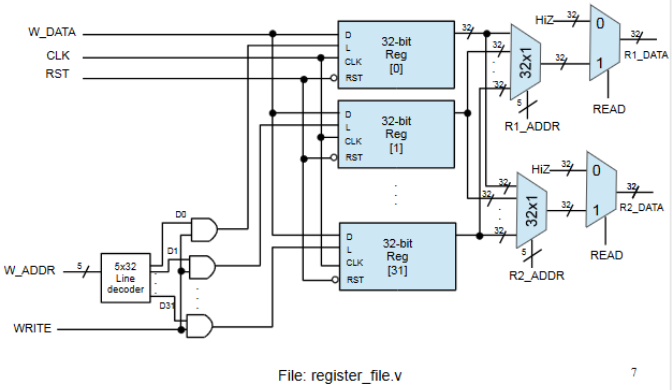
```
module D_LATCH(Q, Qbar, D, C, nP, nR);
input D, C;
input nP, nR;
output Q,Qbar;

// TBD
wire Dbar, firstRes, secRes, resQ, resQBAR;
not(Dbar, D);
nand(firstRes, D, C);
nand(secRes, C, Dbar);

//check this, is this the proper way of redirecting
nand(resQ, firstRes, resQBAR, nP);
nand(resQBAR, secRes, resQ, nR);

buf(Q, resQ);
buf(Qbar, resQBAR);

endmodule

// 1 bit SR latch
// Preset on nP=0, nR=1, reset on nP=1, nR=0;
// Undefined nP=0, nR=0
// normal operation nP=1, nR=1
module SR_LATCH(Q,Qbar, S, R, C, nP, nR);
input S, R, C;
input nP, nR;
output Q,Qbar;

// TBD
wire firstRes, secRes, resQ, resQBAR, nCLK;

not inv(nCLK, C);

nand first(firstRes, S, nCLK);
nand second(secRes, nCLK, R);
//check to make sure this is correct way of redirection
nand third(resQ, firstRes, resQBAR, nP);
nand fourth(resQBAR, secRes, resQ, nR);

buf(Q, resQ);
buf(Qbar, resQBAR);

endmodule
```

One aspect that I was unable to adapt to was the activation toggle of the latches. It was brought to my attention that the designs in the lab were negative edge triggered whereas the project was supposed to be positive edge triggers. It became too late to try to make the proper adjustments. My attempt to fulfill this was instead of passing positive clock to the latches, I inverted them and passed the inverted clock instead.

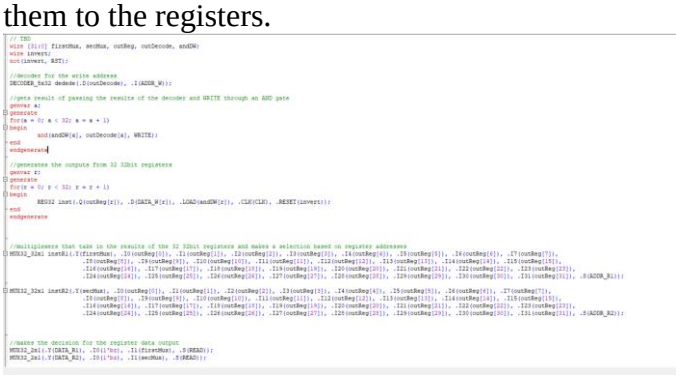
### *I. Register File*

Much like the ALU, the register file is not less important to the functionality of a computer's processor. The purpose of the register file is to act as fast memory storage for the processor since it is right there for it. Instructions and data are stored for easy access for the processor. The design for the projects 32x32 registers file can be viewed in the diagram below.

## Implement 32x32-bit Register File

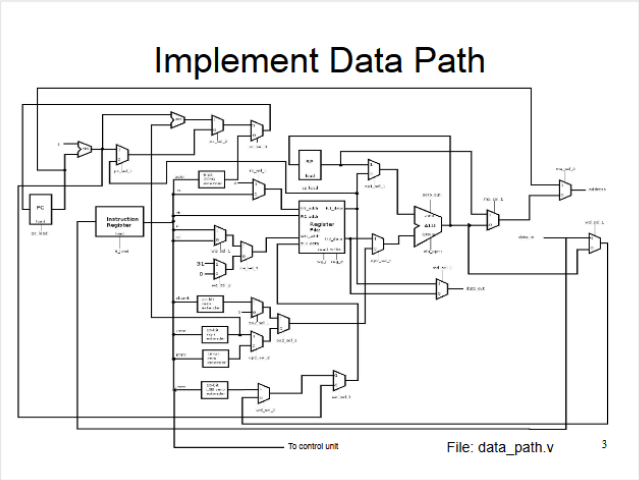



This register file consists of a 5x32 line decoder, 32 registers and some and gates and multiplexers. Having implemented these individual components earlier in the design and implementation process, the register file's implementation was a matter connecting the correct wires from each input to their corresponding objects and following through them to the correct outputs. Generation blocks were used to handle the multiple registers and and gates. Viewing the design, I assumed that the unfilled circle before the rest signals for the register files indicated an inverted input reset signal for each of the registers. Following this thought, I proceeded to make all the reset inputs inverted before passing them to the registers.



### J. *Data Path and Control Unit*

So far all I designed and implemented were smaller digital components that work their way into a bigger picture of a computer system. The data path and control unit are the ultimate destination for all of the components that were developed thus far. The control unit follows a behavior model since it is much too complicated to implement. The data path on the other hand can be designed and implemented.



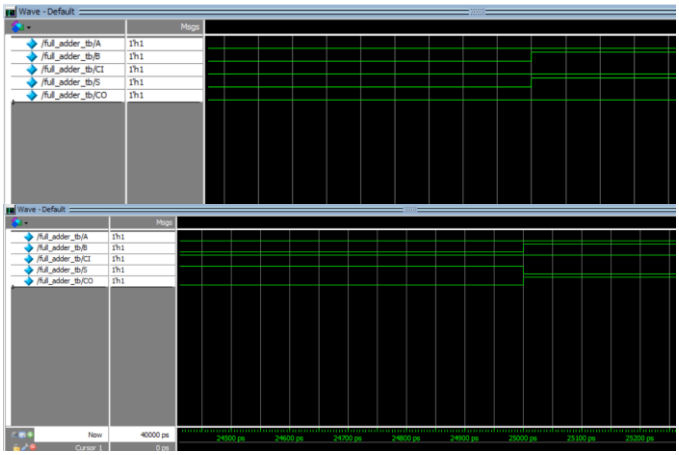

Unfortunately, I was unable to implement data path or the behavioral model for the control unit. I was unable to even make an attempt or start these two major elements of the computer system. So, I will keep this section short and move on to testing the components that I was able to design and hopefully implement properly and correctly.

## IV. Testing

In order to test the various components that I have created, I have developed a number of testbenches. These testbenches were designed to test for fundamental functionality so that when larger components such as the ALU and register file uses them, it will be easier to debug the source of the problem.

### A. *Full Adder*

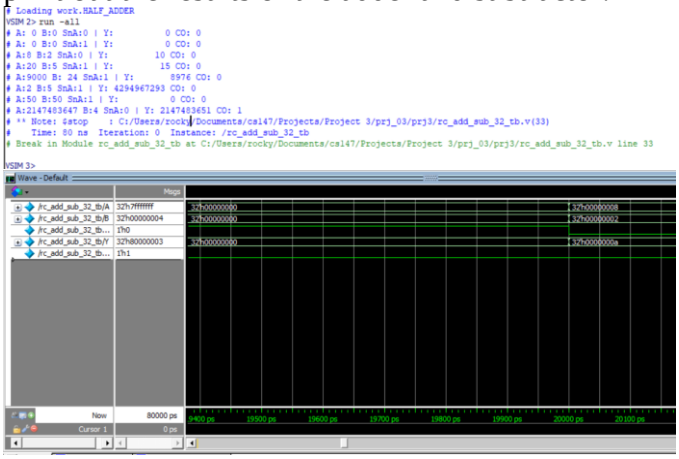
The full adder was tested by feeding input values to the full adder and then observing the wave forms that it produces. The wave forms can be found below.



As evident by these wave forms I can observe that the full adder is working properly. Following the value of the inputs in correspondence to a full adder truth table I can see that the full adder is working properly.

### B. *Ripple Carry Adder/Subtractor*

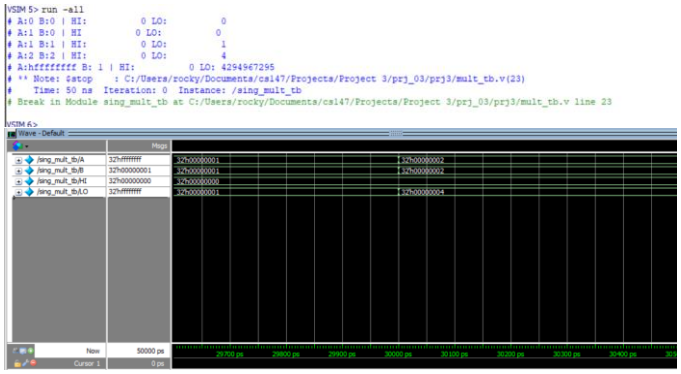
With a functional full adder, I can proceed to test the ripple carry adder/subtractor. Similar to the full adder, I can insert inputs to the adder, but instead of relying solely on the wave forms I can print out the results of the adder and subtractor.



From my test, I can see that my ripple carry adder and subtractor are working properly. Inputting simple arithmetic operations yielded easy to understand results to compare and check. Thus I convinced myself that I have a functional ripple carry adder/subtractor.

### C. *Multiplier*

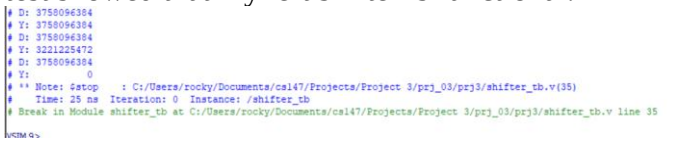
For the multiplier, I did not test both the unsigned and the sign separately. Since they rely on each other, I decided to test the signed multiplier only.




The results that appeared from the multiplier were a little different from what I was expecting. My testbench reached actual integer values and not the bitwise equivalents. I wasn't sure if this was the intention or not by regardless the answers were mathematically correct. One problem I found in the multiplier was inability to carry data in the HI output. One of the tests was multiplying a very large number by another number to produce a value in the HI output. This was my last test case, but the HI output remained 0. A possible reason for why this is may be the case is because I am not buffering to the HI data lines at the end of the signed multiplier correctly.

### *D. Shifter*

I was unable to test the full functionality of the barrel shifter, but I was able to specifically test the left shifter. To test the left shifter, I inserted 32-bit values into the D input and for the shift amount, 0, 1 and 2. These basic shift amounts were easy to compute myself so I used my own judgement to check the actual results to my expected results. The test showed that my left shifter is functional.




Since, left shift works I assumed right shift works as well since they are nearly identical except for the direction of shift. However, I believe that the entire barrel shifter is not properly working. I believe that the cause for its faulty performance is in the single input or gate. I do not understand how a one input or gate functions and thus proceeded to use a 32-bit 2x1 or gate instead. The ranged input was unclear to me and caused confusion. I think that this was the source of the problem and caused its incorrect functionality.

### *E. ALU*

Knowing that the shifter does not work, I expected that the ALU will also not work. My intuition was correct. My testbench has a total of 12 test cases and the 3 of the 4 that failed were shifting operation. This confirms that my shifter is inadequate to perform. Other than that, the rest of the test cases passed. The testbench was the same one I used in previous projects.

```
# Expect performance to be adversely affected.
VSIM 11> run -all
# [TEST] 15 + 3 = 18, 0 : got 18, 1 ... [PASSED]
# [TEST] 15 - 5 = 10, 0 : got 75, 0 ... [FAILED]
# [TEST] 5 - 5 = 0, 1 : got 25, 0 ... [FAILED]
# [TEST] 24 * 0 = 0, 1 : got 0, 1 ... [PASSED]
# [TEST] 15 >> 4 = 0, 1 : got X, X ... [FAILED]
# [TEST] 12 << 3 = 96, 0 : got 96, 1 ... [PASSED]
# [TEST] 3 & 5 = 1, 0 : got 7, 0 ... [FAILED]
# [TEST] 7 | 9 = 15, 0 : got 15, 0 ... [PASSED]
# [TEST] 0 ~| 0 = 4294967295, 0 : got 4294967295, 0 ... [PASSED]
# [TEST] 2147483647 ~| 102 = 2147483648, 0 : got 2147483648, 1 ... [PASSED]
# [TEST] 12 < 36 = 1, 0 : got 1, 0 ... [PASSED]
# [TEST] 99 < 9 = 0, 1 : got 0, 1 ... [PASSED]
#
```

### *F. Unable to Test*

As evident of a small testing section of this report, there were some components that I was unable to test some of the components such as the register file. On the issue of the register file, there were numerous warnings that needed to be handled, but unfortunately, I ran out of time. Also, since I could not implement the data path, I could not test the entire computer system as well. Some components I did not feel necessary to test like the 32-bit logic gates. Their implementation was simple and it felt like I would not be using my time wisely by testing them. The decoders and multiplexers were implemented by strictly following their circuits so I felt as long as there was not an obvious error, they would be fine. As a result I was unable to test some of the bigger components and the main data path due to time constraints and simulation warnings.

## V. Conclusion

In conclusion, I was unable to complete the objectives set forward by the project. I was able to create the components consisting of the processor, but not the data path or control unit. Some of the components that I've implemented in this project

were able to pass my test benches while others did not. The components that were unable to pass my testbenches contained bugs or design flaws that I was unable to overcome. They were still implemented, but their functionality is not correct. From this project, I've learned that gate modeling can be a very tedious and hard task to complete. Provided a design, one can replicate that digital circuit into code and generate a prototype of the circuit to be ran on a simulation. It is a powerful tool to know and I hope to learn from the struggles I've had on this project and apply them in future opportunities.