CS 147 Da Vinci (CS147DV)
Jason Tang
San Jose State University
jason.t.tang@sjsu.edu

Abstract: **This report details the ModelSim simulation of a mixed model DaVinci computer system with a 32-bit processor and 256 MB memory that is double word (32-bit) addressable 64M address). The report will include:**

1) Installation of ModelSim and setup
2) Instruction set
3) System requirements
4) Implementation of the Processor Components
5) Implementation of test bench code
6) Simulation and output waveforms of all the components using the separate test benches
7) Explanation of missing components

General Information

Table 1.1: Table of Tools Used

Name	Company	Used For	Free (Y/N)
ModelSim	Mentor Graphics	Simulation and Test	Yes (for students only)

1. STEPS TO INSTALL THE SIMULATION TOOL 'MODELSIM':

1) Open http://www.mentor.com/company/higher_ed/modelsim-student-edition
2) Download Student Edition
3) Run the installation .exe file and complete the installation steps
4) Fill out the form that pops up in the browser with your name, address, phone number, email, university name, and other info, then click finish
5) In the email from ModelSim, locate the student_license.dat file
6) Save the license file to the installation directory for ModelSim PE Student Edition, the directory that contains win32pe_edu.
7) Run ModelSim PE Student Edition

2. STEPS OF THE SIMULATION PROJECT CREATION

In order to create the project and simulate the ALU using Verilog code, follow these steps.

1) Download the Project3 Zip file and extract the files

2) Open ModelSim and navigate to File->New->Project

- Open the Create project window and enter Project Name CS147_Project3_DaVinci

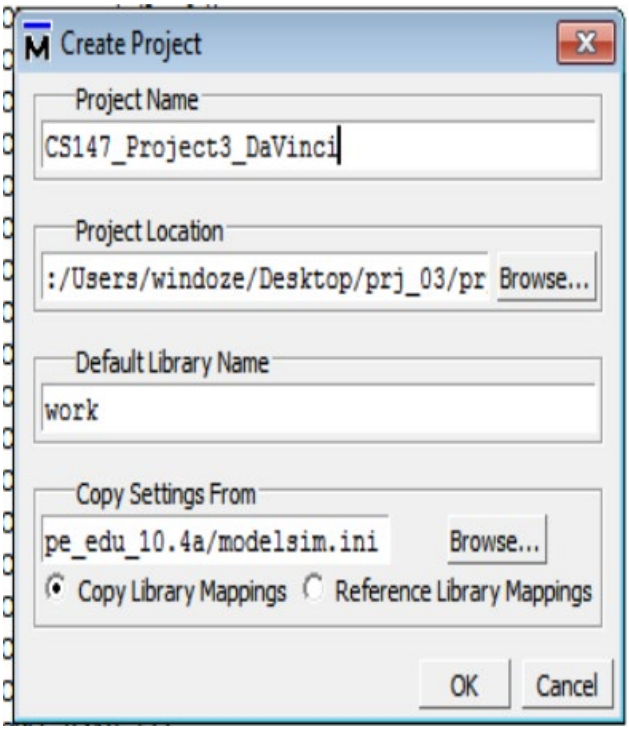


Figure 1 Create a project window

3) On the next window, select add existing file and browse to where the zipped project files are and add them in

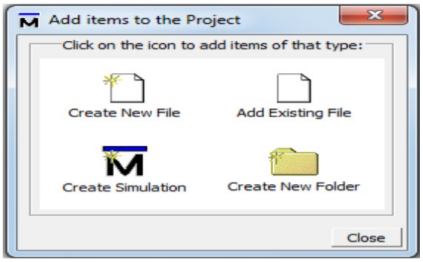


Figure 2 Add existing file window

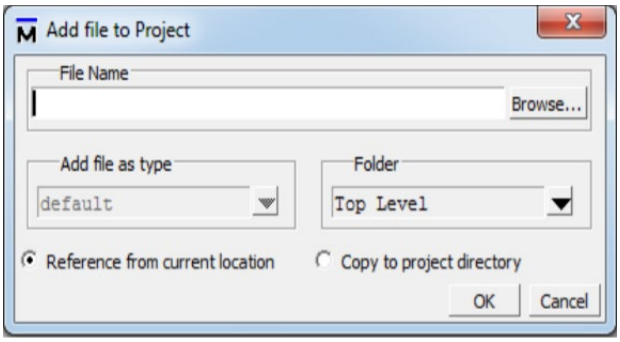


Figure 3 Add file to project

4) Add files into the project.

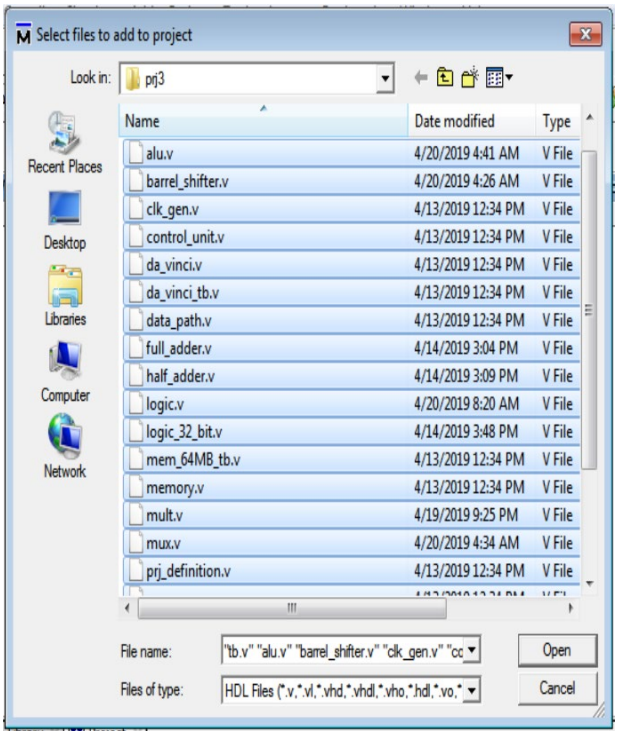


Figure 4 add files into project

5) Navigate to the Compile tab in the top of the window and then select all the files to compile

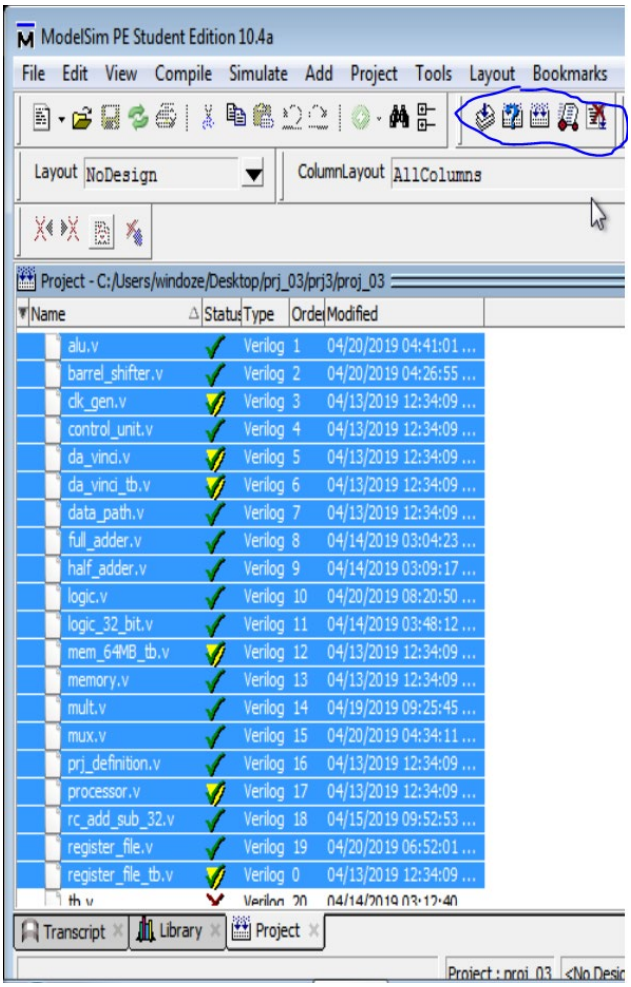


Figure 5 Compile all files

6) Navigate to the Library tab then select work ->prj_03_tb and right click and select simulate

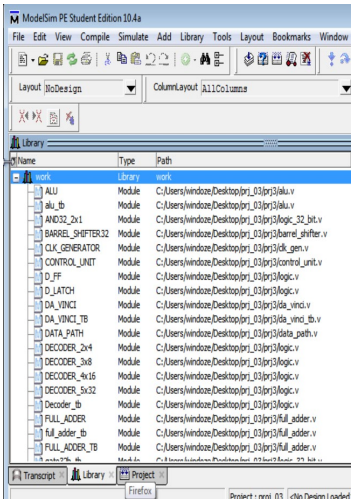


Figure 6 Select a test bench to start the simulation

7) Then in the sim tab, add your selected test bench to the wave. For this one we will be running DA_VINCI_TB

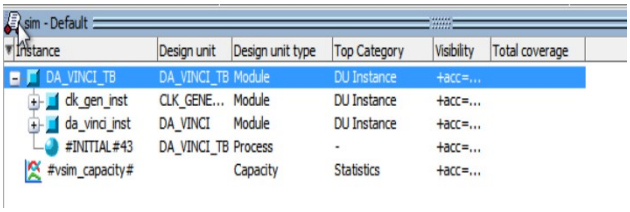


Figure 7 Add DA_VINCI_TB to the wave

8) In the Wave tab, the following objects should be present

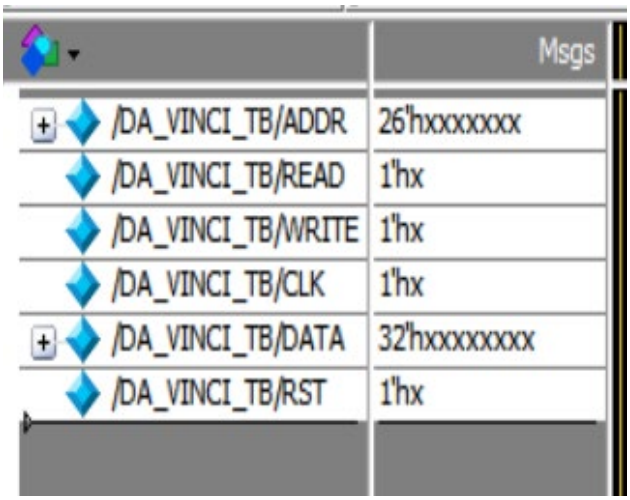


Figure 8 Added elements in the Wave

9) To run the simulation, change the runtime to 100000000 ps or an equivalent and select run from the navigation

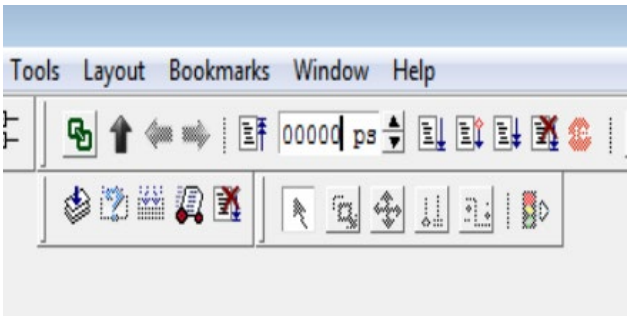


Figure 9 Run the simulation

3. REQUIREMENTS OF DAVINCI COMPUTER SYSTEM

A) The CS147DV Instruction Set Supports R-type, I-type, and J-type instructions with the following breakdown and instructions

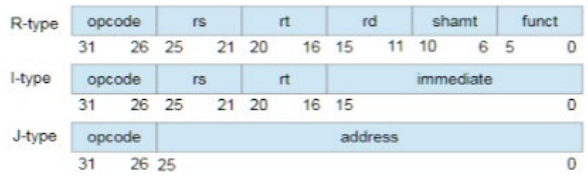


Figure 10 Instruction Format

An R type instruction is written with an OPCODE that is usually 0x00. The way that the R type format selects the instruction is through the FUNCT bits. The SHAMT is used for calculating how many bits to shift for Sll and Srl. RS, RT and RD, are the two operands and the destination register respectively. The following chart shows the operation that is done by each operation and its corresponding function. All of them use a RS and RT except for a few exceptions that will be discussed when the full R-type instruction set is introduced.

An I type instruction has an OPCODE and an RS, RT, and Immediate. The RT operand is used as the destination register for the operations done by the RS and Immediate function. There are exceptions that will be discussed after the full I instruction set is introduced.

A J-type instruction is made up of an OPCODE and Address which is then used to modify the PC register. With exceptions being the push and pop which will be discussed later when the full list of instructions is introduced.

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

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

Figure 11 R-type Instruction Format

The shift instructions which use the SHAMT for their second operand. The Jump Register

instruction uses the PC register as the storage for the RS register's value instead of the RD instruction.

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

Figure 12 I type Instruction Format

Branching works uses a comparison between the RS and RT to choose between the PC's new value. For loading and storing words, it uses the RT for the location. The RS and signed immediate are used to calculate the memory location. The I instructions for addi, muli, andi, and ori are like the R type instruction except the RS and the signExtendedImmediate are used as operands.

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

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

Figure 13 J-type Instruction Format

The push and pop functions use the stack pointer as a reference to Memory data. They both use R[0] as a way to store data that is going to be given to the memory or store data from the memory. A regular jump just sets the PC to the Jump Address. A Jump and Link sets R[31] to the PC +1 to store the location for recursive callbacks.

B) CS147 DaVinci

The CS147 DaVinci computer system uses a 32-bit processor, and 256 MB memory, with double word addressable 64M address. The processor supports the standard CS147DV instruction set. The ALU, register file, and data path need to be implemented at the gate level. The memory and control unit need to be implemented at the behavioral level

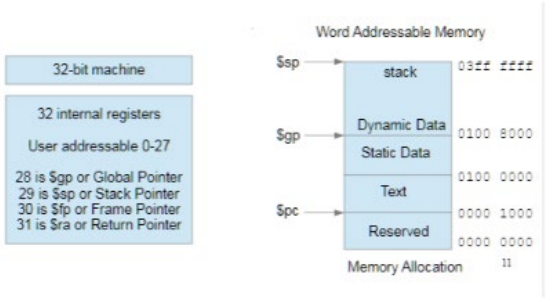


Figure 14 Storage Format

4. DESIGN AND IMPLEMENTATION

Below is the module design of each component used in the CS147 Da Vinci Computer system with hybrid gate and behavioral implementation.

A) Half Adder

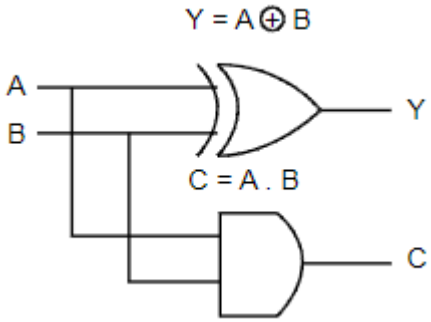


Figure 15 Half Adder Diagram

For the Half Adder, we simulate the A and B inputs by taking in the wire and attaching it to a XOR and NAND following the diagram. The Sum is Y and the Carry out bit is C.

B) Full Adder

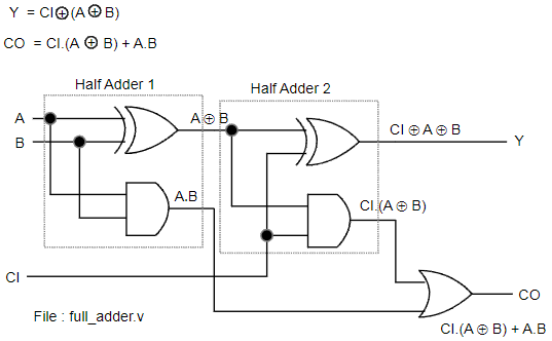


Figure 16 Full Adder Diagram

The Full Adder requires two half adders that feed into each other. For the second half adder, the A input is the Y output of the first half adder. The B input is the Carry In bit passed into the full adder. The Carry out is Or of the carry out of both half adders. The final sum is the A and B of the second half adder.

C) Binary Ripple Carry Adder/Subtractor

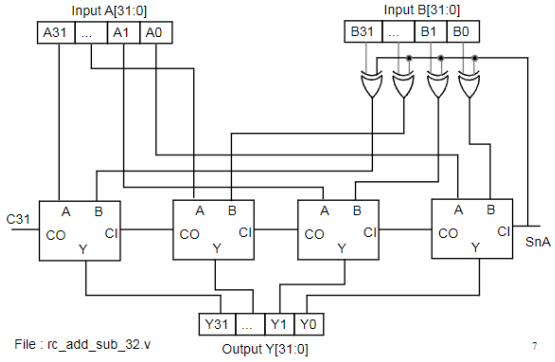


Figure 17 Binary Ripple Carry Adder/Subtractor

The binary ripple carry adder uses 32 full adders and passes in the nth bit from the A and B input into the nth full adder, saving its sum bit as the nth bit of the result. For subtraction, the circuit converts B to 2's bit complement adds A and B normally. To do that the B input is run through a XOR gate using the SnA signal which selects addition or subtraction. If the SnA bit is 1, the circuit is running subtraction, the XOR gate acts as an inverter, if it is 0, it does nothing. The SnA then adds one by being a 1 in the Carry In bit, thus fulfilling the 2's complement for the B input. For the first full adder, the Carry In is 0 and for all other n adders, Carry Ins it takes in the carry out of the n-1 adder. The 31st adder has its Carry out stored as Carry Out bit output for the whole ripple adder.

A 64 bit adder is also implemented the same way, except with 64 full adders instead of 32.

D) Unsigned Multiplier

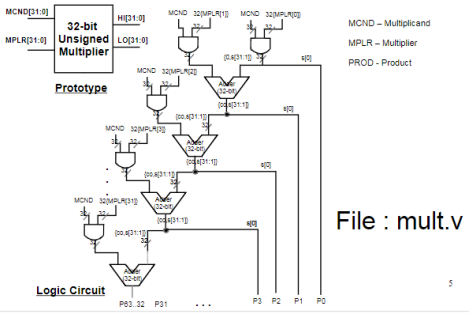


Figure 19 Unsigned Multiplier Diagram

The unsigned multiplier ANDs the Multiplicand and the nth bit of of the multiplier and runs a 32 bit adder on the Multiplicand and the n+1th bit of the multiplier. It then adds the {carry out, and the {31:1} bits of the result} of that addition to the Multiplicand ANDED to the next bit of the multiplier. The product is calculated as a LO and HI, for the 32 lower and higher bits respectively. For each adder and the first AND, the LSB result of those is taken as the nth bit of the lower product register. The 0 bit is the first and's LSB, the first Adder's LSB is the 1st bit, and it continues from there. For the HI register, the raw result of the last ADDER is taken as the HI register.

E) Signed Multiplier

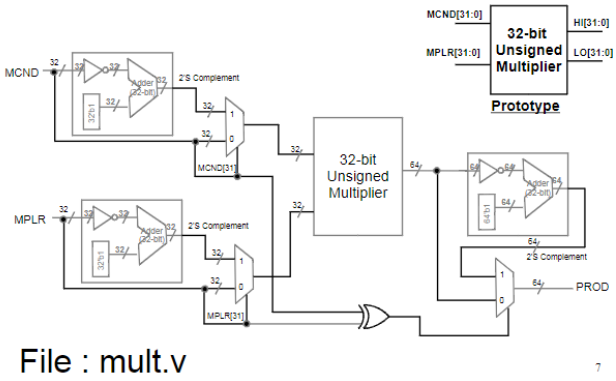


Figure 20 Signed Multiplier

The signed Multiplier uses an unsigned Multiplier and merely selects between the complemented and uncomplemented operands/results. To choose between the complemented and uncomplemented operands, the MSB is used as selection logic, 0 for uncomplemented, 1 for complemented. For the result a XOR of the MSB of the operands is taken. 1 for complemented, 0 for uncomplemented. The operands are complemented because the multiplier cannot handle signed numbers.

F) Multiplexers

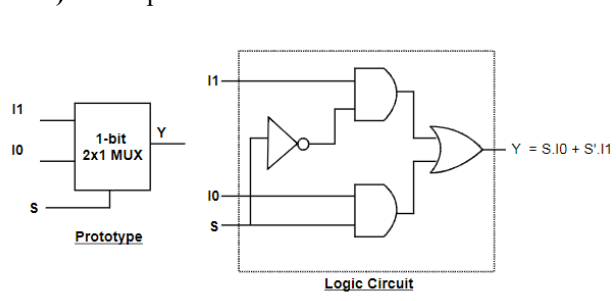


Figure 21 Incorrect 2x1 Multiplexer Diagram

The following diagram is correct as long as you flip the I1 and I0 outputs. The Selection logic bit S and I1 are ANDED together and the Selection logic and the NOTED I0 are ANDED together. Both results are ORED together to get the selected output.

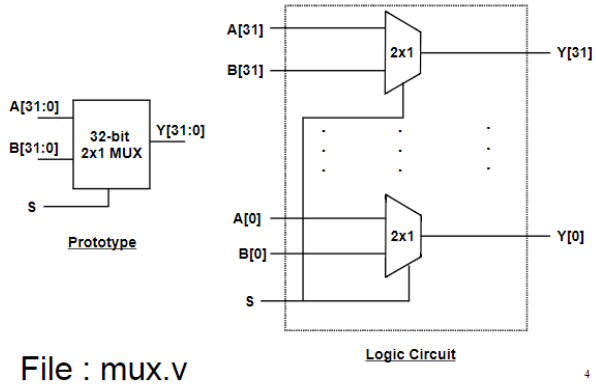


Figure 22 32-bit 2x1 Multiplexer

The 32-bit multiplexer runs a 1-bit 2x1 multiplexer for every bit of the output.

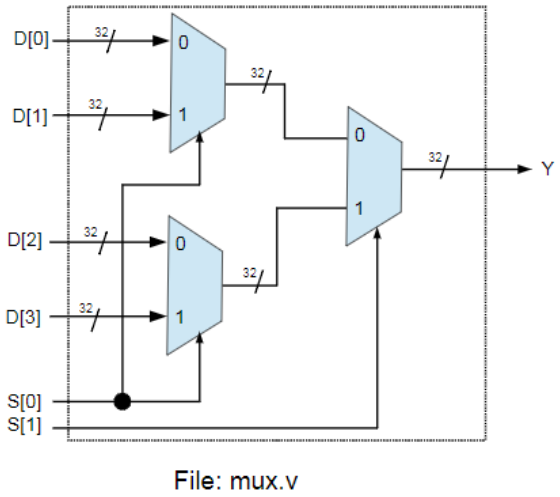


Figure 23 4x1 32-bit Multiplexer

The 4x1 32-bit Multiplexer takes in 2 selection bits and 4 inputs. It uses 2 2x1 32-bit Multiplexers that both use the first bit of the selection logic. It then runs the selected outputs into another 2x1 32-bit Multiplexer and selects using the second selection bit for the final output.

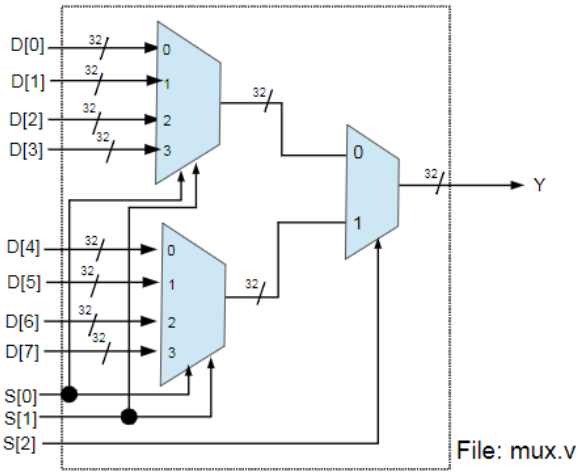


Figure 24 8x1 32-bit Multiplexer

The 8x1 32-bit Multiplexer uses 2 4x1 32 bit Multiplexers in the same way the 4x1 32 bit Multiplexer uses 2 2x1 32 bit Multiplexers. It splits the n inputs in half with the first half going into a 4x1 multiplexer and the second half going into another 4x1 multiplexer. The result of the two multiplexers is then selected by a 2x1 multiplexer using the MSB of the selection signal to select the output.

The rest of the multiplexers follow this pattern of using the (n/2)x1 multiplexer and all but the MSB of the selection signal and a 2x1 multiplexer with the MSB of the selection signal to select the result. The following are the larger multiplexer diagrams.

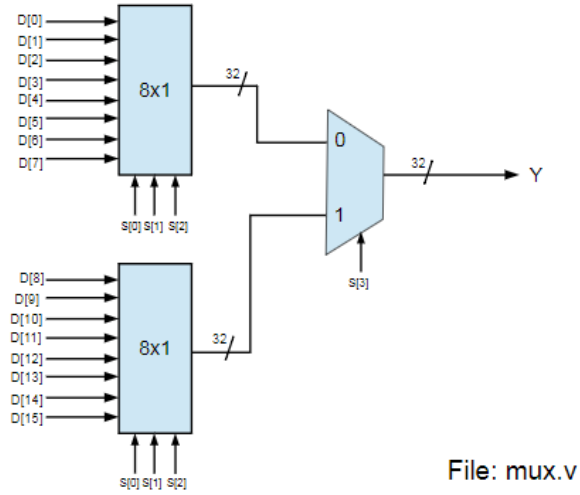


Figure 25 16x1 Multiplexer

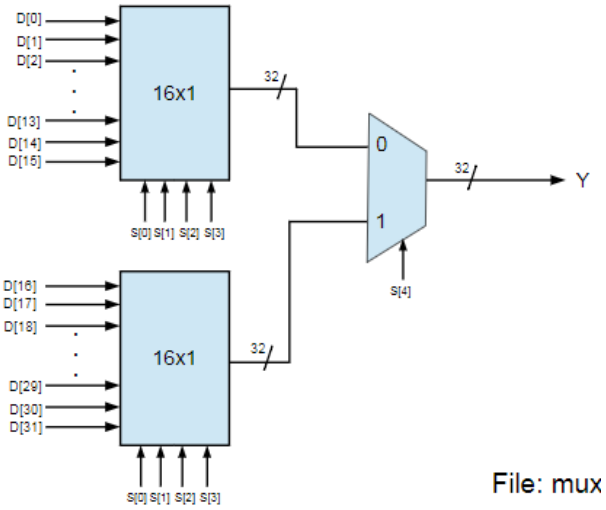


Figure 26 32x1 Multiplexer

It is also important to note that there is a 64bit 2x1 Multiplexer used in the multiplier. It follows the same logic as the 32-bit 2x1 multiplexer by making 64 1 bit 2x1 multiplexers.

G) 2's Complements

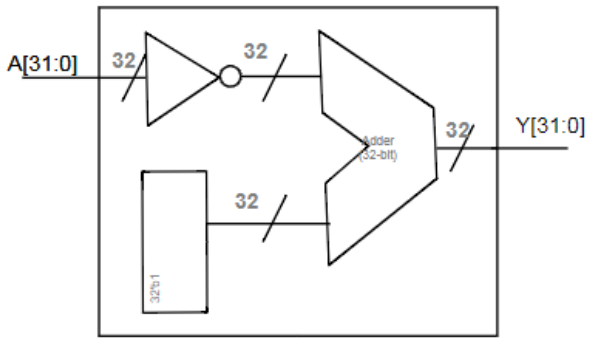


Figure 27 32 bit 2's complement

The 32-bit 2's complement follows the 2's complement formula to and NOTs the input and runs it through an adder with 32b'1 as its other operand. The addition result is taken as the 2's complemented version of the input.

A 64'b 2's complement follows the same logic except extended to 64 bits.

H) Barrel Shifter

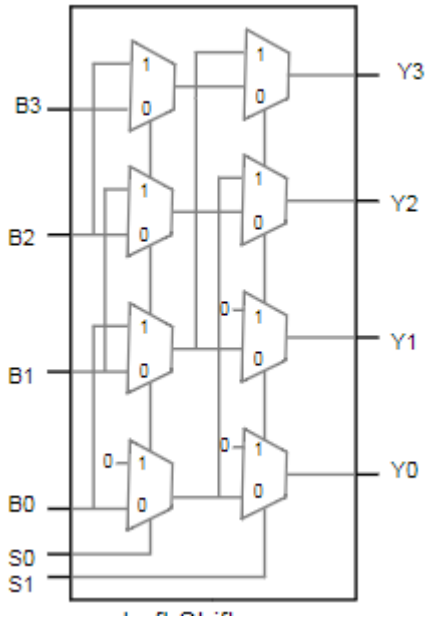


Figure 28 4-bit Barrel Shifter Diagram

The diagram is a 4-bit barrel shifter. The 32-bit barrel shifter works similarly except extended for 32 bits. A 32-bit barrel shifter has 5 bitts of selection logic and 32 multiplexers for each of the selection logic bits. Each nth starting with initial 0 stage has 2^n 0's it can put into the solution. Each stage feeds into the next stage into at two different locations. For example, the nth bit from the nth stage is fed into the nth bit of the n+1 stage and the n+ (number of 0's the stage offsets).

If a 0 is required, it is selected, and the nth bit is selected after the offset of 0's is selected since after the offset the n stage is fed into the 1 slot of the multiplexer. If it is 0, the multiplexers pass the n stage along without offsets.

In the module, this is done through a for loop with 32 bits being processed at once. Each stage is run one after the other with the initial stage's first multiplexer being run first. That first stage multiplexer then creates the output for the next stage multiplexer and so on.

The barrel shifter is split between a left and right shifter. The two of them have the same logic, except the right is completely inverted from the left. The loop starts from 31 and goes to 0, and each of the bounds are flipped and the number is decremented by 1.

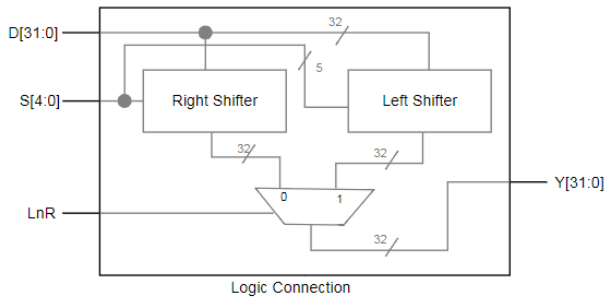


Figure 29 Barrel Shifter with control Diagram

The LnR signal controls which version is selected, because both values are automatically computed.

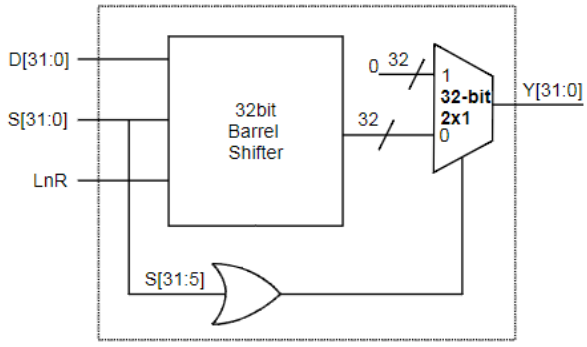


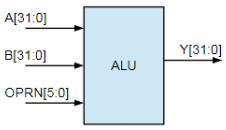
Figure 30 32-bit Shifter Diagram

A complete shifter will return 0's if there is a shift amount that is greater than 32. This is done by running the 31:5 bits of the Shift Amount into an OR.

I) ALU

Design Strategy:

The ALU is treated as a module with specific inputs and outputs. The ALU modules have three inputs and one output as outlined



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]

Figure 31 ALU module outline and OPRN signals diagram

The ALU OPRN has xx in the front because this sis only a simulation and those bits are unnecessary for the small amount of instructions that are implemented. The diagram also shows how the control signals are derived for the Shifter and the Adder/Subtractor and set less than operation.

The Gate Level implementation is below.

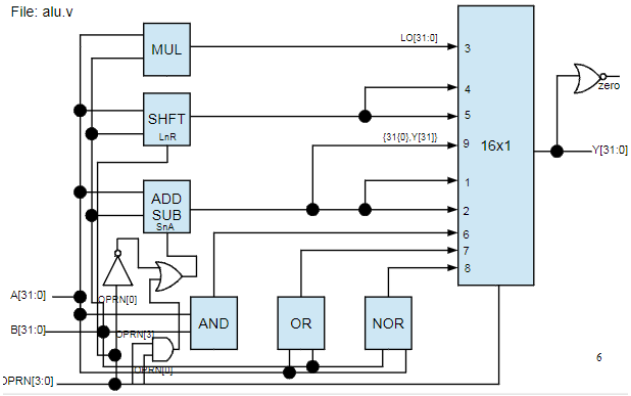


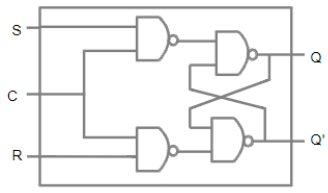
Figure 32 ALU diagram

For each component shown above, the A and B are sent to them and the control signals are derived as according to Figure 26. The OPRN's [3:0] bits are used as the selection logic for a 16x1 Multiplexer that gives the result since every operation is run at the same time. There is also a NOR gate that NORs the entire output and checks if there is only 0's.

J) Logic Gates

There are 32 bit AND, NOR, OR, and INV gates that contain 32 1-bit versions of those gates that work together to run the operation on every bit of the output. The exception is the OR gate which is the 32 bit NOR and INV gates result since OR is an inverted NOR.

K) SR-Latch



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

Figure 33 SR-Latch diagram and truth table

The SR latch has a preset-on P= and R=1. It sets the value and gets it ready for other operations. It is reset on P=1 and R=0. The value is undefined on P=0 and R=0 which is a problem that needs to be addressed

with a D-Latch. It operates normally on P=1 and R=1 with setting and resetting. The C control signal to control the changing of terms since the Q output is controlled by the Qt+1 output as well.

L) D-Latch

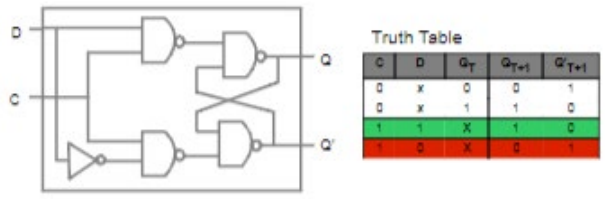


Figure 34 D -Latch diagram and truth table

The D latch is like an SR latch that replaces the S and R with a D input that is inverted. This is so all the functions of the S and R latch can be done using the inverter and it does not use the S=0 and R=0 data hold. It also eliminates the indeterministic state.

M) 1-bit Flip Flop

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

Figure 35 Truth Table for 1-bit Flip Flop

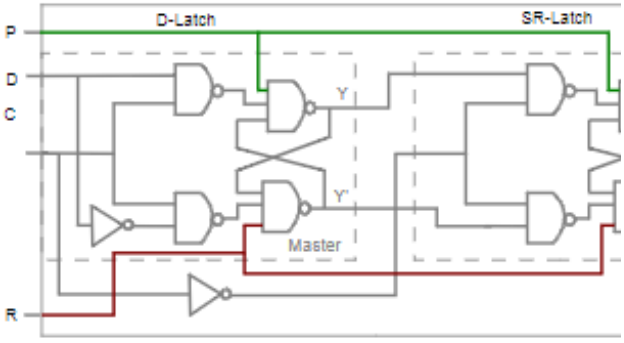


Figure 36 Negative edge Circuit Diagram for 1-Bit Flip Flop

The D-latch has a problem with latch time where gt+1 can be unstable when gt changes. The D flipflop makes sure that a D latch is used and connected to a SR latch to solve thie issue. During the latch time, the D latch gets the Data, but the SR is not saving it. During hold time, the SR latch is passed the D latch's data and the stable term is given. For the implementation of this project, a positive edge is used. The only difference is that the C signal is inverted when going to the D latch.

N) 1-bit Register

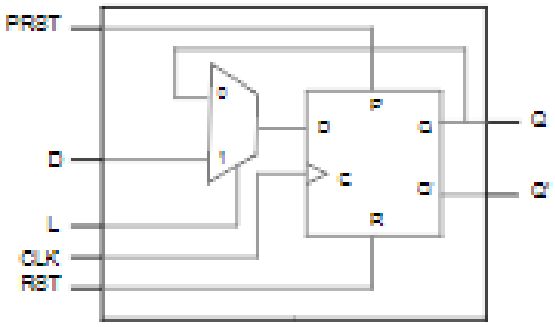


Figure 37 1-Bit Register Circuit Diagram

A 1-bit register runs off of a D flipflop with an Added L input with a multiplexer controlling the D input into the D flip flop. It chooses between Qt+1 and D input.

O) 32-bit Register

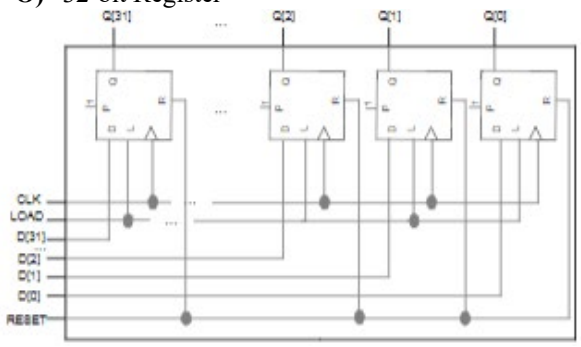


Figure 38 32-bit register Circuit Diagram

A 32-bit register has 32 1 bit registers put into it. Each register shares the clock and load signal. Each 1 bit register also gets its corresponding bit of the input data.

P) Decoders

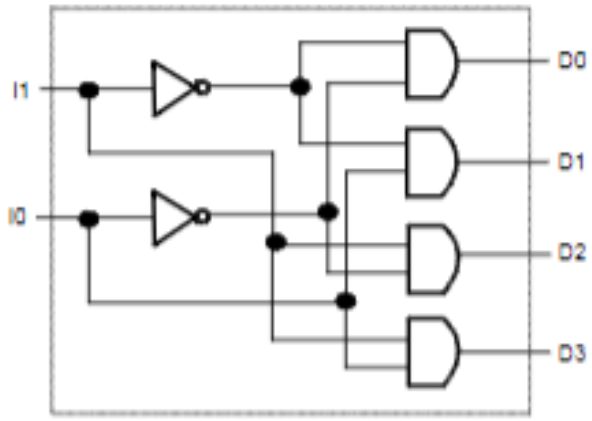


Figure 39 2x4 Decoder

The 2x4 1-bit line decoder has inputs 0 and 1 and outputs 0,1,2,3. It generates these outputs by ANDING the input bit 0 and input bit 1 as if they were the bits in a binary number.

For example, I0 is the first bit in the 2-bit pattern. I1 is the second bit in that bit pattern. A 0 for either input is considered the inverse of that input and a 1 is considered the input unchanged. Output D0 is AND of the inverse I0 and I1. Output D1 is the inverse of I1 ANDED with I0. The patter continues like that for the rest of the inputs.

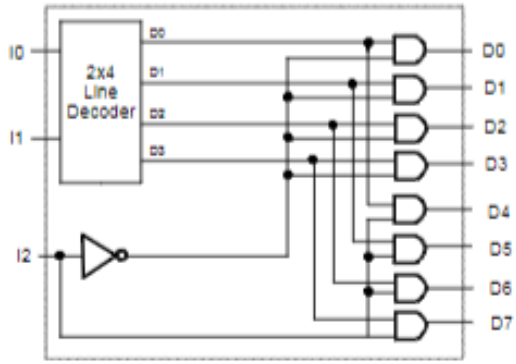


Figure 40 3x8 Decoder

A 3x8 Decoder uses a 2x4 decoder and adds I2 into the mix. Outputs D0 through D3 are ANDED with the inverse I2 input and outputs D4 through D7 are ANDED with I2 uninverted.

For the rest of the decoders, the patter continues where for an n x 2^n decoder, a n-1 x 2^(n-1) decoder is used in the same way, with the new selection bit ANDED with all the outputs of the smaller decoder when it is inverted and not inverted. The rest of the diagrams are show below.

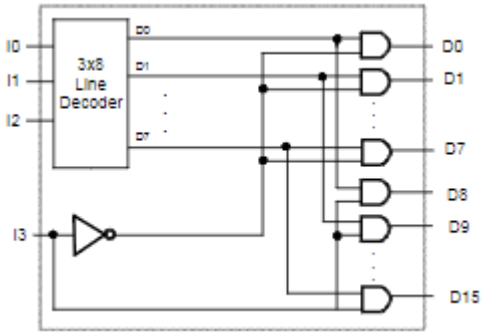


Figure 41 4x16 Decoder

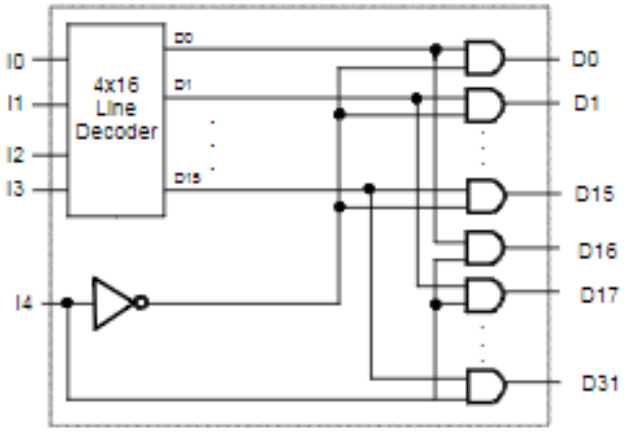


Figure 42 5x32 Decoder

Q) 32x32 bit Register File

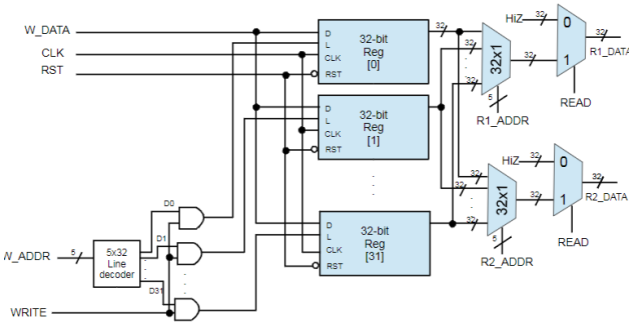


Figure 43 32x32 bit Register File Circuit Diagram

The 32x32 bit register file circuit takes in W_DATA as the 32-bit D input into all 32 registers. It still uses a Clock and Reset signal shared among all 32-

bit Registers. The WRITE and W_Addr are used to create the L signal in each of the 32-bit registers. First the W_Addr is taken in and run through a 5x32 decoder. Then the Write signal is Anded against all the outputs. Each ANDED output is then mapped to the corresponding 32-bit register. The 0 output of the W_Addr goes to the first register and so on and so forth.

At the end, the outputs of all the registers are fed into 32x1 multiplexers which are controlled by the R1 and R2 addresses respectively. The result of that is then fed into another multiplexer with the HiZ or the result of the previous multiplexer. The outputs are taken as the R1 and R2 data respectively.

R) Data Path

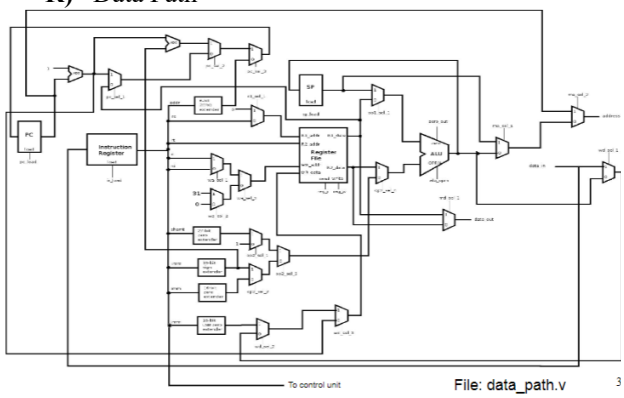


Figure 44 Data Path Diagram

The Data Path is implemented as above. The data path is created through the combination of all the data paths required by the different R,I, and J type instructions.

For the R type instructions, the register files and ALU are connected. The shift instructions force the ALU to draw from the shift amount as a operand and the Jump instruction uses the Program counter.

For I type arithmetic instructions, it is like the R type instructions except the ALU is connected to a 16-bit sign extender for the immediate values it processes. For the logical I type instructions, it is extended with 16 bits of 0. The LUI instruction path uses a 16-bit LSB zero extender. For branching, there is a 16-bit sign extender, PC counter connection and incrementation for when branch is not taken, and another multiplexer adder combination to pick between the two paths calculated. For LW the memory is connected with a sign extender, register, and ALU to calculate addresses and load them into registers. The SW operation uses similar components.

For J type instructions, the Instruction register, PC, and an adder for the PC and 6-bit zero extender are connected. The 6-bit zero extender and the result of the incremented PC are ran through a multiplexer. For the JAL, a register file is included to add a way to find the current PC to return to. For Stack instructions, the Data Memory, ALU, SP, and Register file are used.

For each of these instruction formats and types, there are numerous signals and conflicts that are fixed by adding in more control signals and multiplexers. All the control signals and multiplexer conflicts are rectified case by case and will not be covered in the report. The end result is the Data Path shown in the diagram.

S) Memory Wrapper

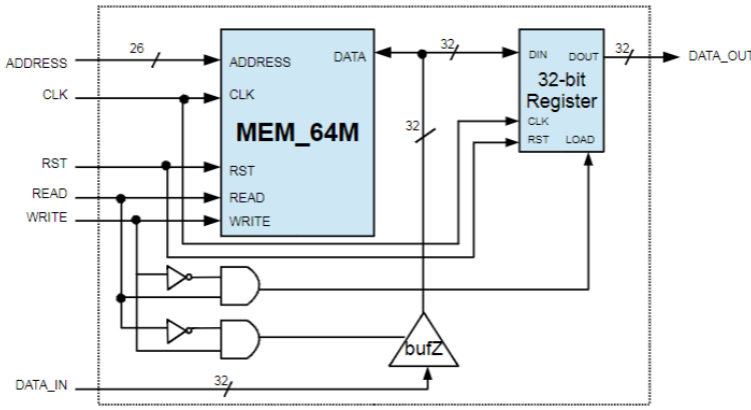


Figure 45 Memory Wrapper Diagram

The Memory Wrapper was not implemented in this project but operates as such. The Memory Wrapper is comprised of a 64-bit Memory unit that is much like a 64-bit version of the 32-bit register. The 32 64-bit registers are connected to 32-bit register which gives the data out for the whole module.

For actual processing, the Address is passed in as a 26-bit value and is taken in as address by the memory. The CLK and Reset are connected directly to the memory and register as seen in the diagram. The read and write for the Memory is passed as usual as well, but the Load and Data In for the register are different. There is an additional DATA_IN signal that is run through a bufZ and controlled by the inverted Read and Write signal run through an AND gate. If the read and write is not read=0 and write=1 the Data input of the

32-bit Register will be 32 bits of Z since the register should not be reading a value from the memory. For the Load of the register, if Read ANDED with inverted Write is 1, then it loads in the value of the memory. Both of these serve to invert the process of the memory and register.

When the memory is being read from, the register is writing and vice versa.

T) Control Unit

The control unit is a behavioral element that was not implemented in the project due to time constraints but outputs a 32-bit control signal as well as read and write signals when fed the CLK,RST,,and Instructions from the Data Path.

U) Processor

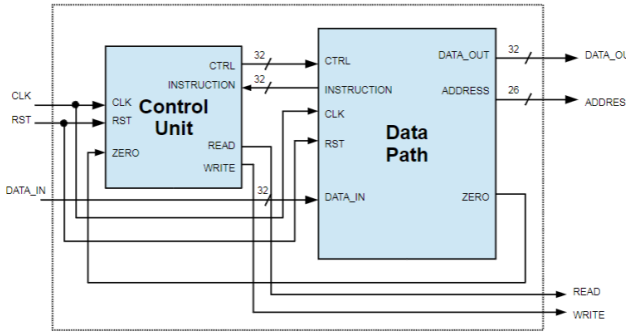


Figure 46 Processor Diagram

The Processor combines the Control Unit and the Data path with a similar CLK and RST instruction, having the control unit and Datapath exchange controls and instructions.

V) System

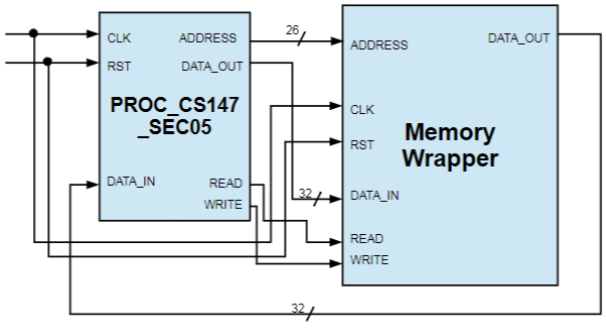


Figure 47 System Diagram

The completed system was not implemented in the project but operates as such. The CLK and RST are shared, the address, read, and write signals are sent from the Processor into the Memory wrapper, and the Data_Out of the Processor is taken as the Data_In of the Memory Wrapper. The Memory wrapper Data output is taken as the input into the Processor. The logic behind when each of these signals is actually used rather than just passed in and not used by control signals is different instruction by instruction.

5. TEST STRATEGY AND TEST IMPLEMENTATION

To Test the System, higher level testing was implemented. The main tests were run on the ALU and Register File. There is not a test for the data path and the Control Unit because those were not implemented in my project due to time constraints. Below, the high level tests of the ALU test is shown as well as the Register file test.

A) ALU

The ALU consists of the shifter, adder/subractor, AND,OR, and NOR gates as well as a 16x1 multiplexer. The testers for these specific components as well as the lower level multiplexers were not tested through a different test bench because every component has to work for the ALU to function. The tests have no proper output in the Transcript, so the test was run and verified completely in the Waveform, which was sufficient for my purposes.

```
// Name: alu.v
// Module: ALU
// Input: OP1[32] - operand 1
//        OP2[32] - operand 2
//        OPRN[6] - operation code
// Output: OUT[32] - output result for the operation
//
// Notes: 32 bit combinatorial ALU
//
// Supports the following functions
//      - Integer add (0x1), sub(0x2), mul(0x3)
//      - Integer shift_rigth (0x4), shift_left (0x5)
//      - Bitwise and (0x6), or (0x7), nor (0x8)
//  - set less than (0x9)
```

Figure 48 ALU inputs and operation logic

This shows the oprn passed into the ALU to make it run the correct functions. Waveform is given and will be explained case by case. Additional explanation will be given if the waveform explanation is insufficient.

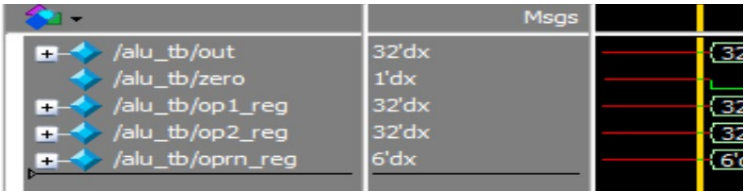


Figure 49 Waveform outputs

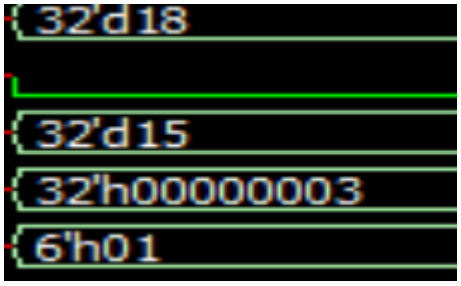


Figure 50 Test 15+3=18

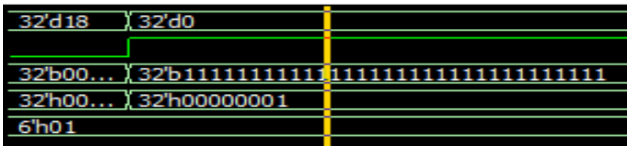


Figure 51 Test of addition overflow condition and 0 signal

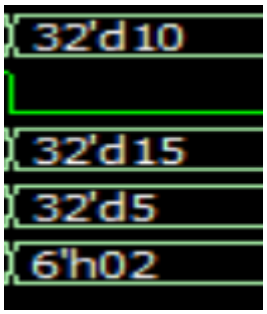


Figure 52 General Subtraction test

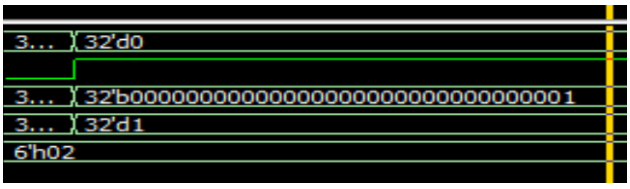


Figure 53 Subtraction overflow test

For this test, the first operand is actually given as -32'hFFFFFFFF. The two's complement of that is a negation, which returns all 0's, and an addition of one, wich results in 32'b00000….1. This results in the overflow result of 0 and triggers the zero signal.

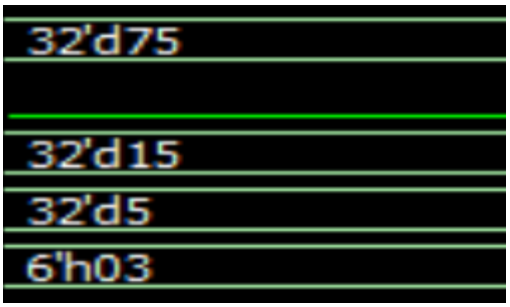


Figure 54 General multiplication test

Figure 55 Multiplication overflow test

In this case, the multiplication results in the bit pattern 1110 1111 1111 1111 1111 1111 1111 1111 0001. Since the output displayed is only the Lo register by design, the result is only 1111 1111 1111 1111 1111 1111 1111 1111 0001 as received.

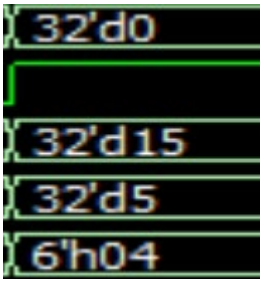


Figure 56 General Right shift and 0 flag

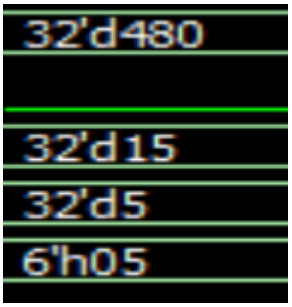


Figure 57 General left shift

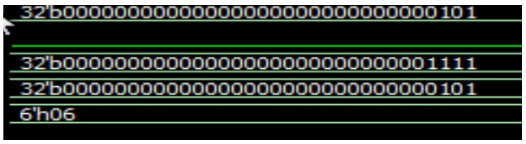


Figure 58 General And

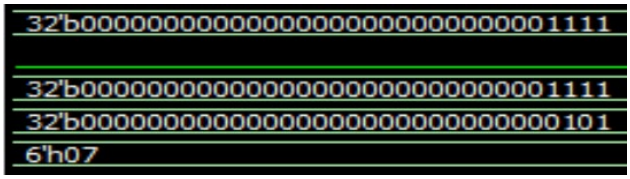


Figure 59 General Or

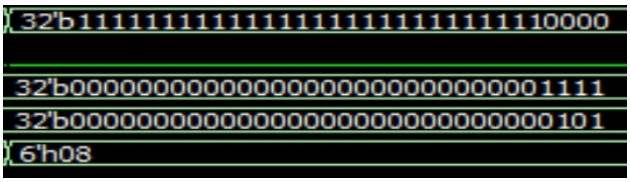


Figure 60 General Nor

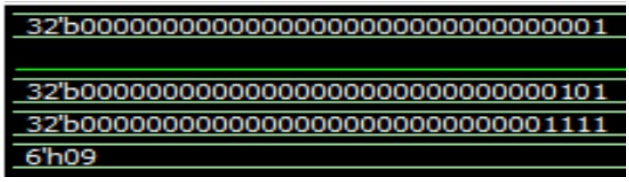


Figure 61 General SLT

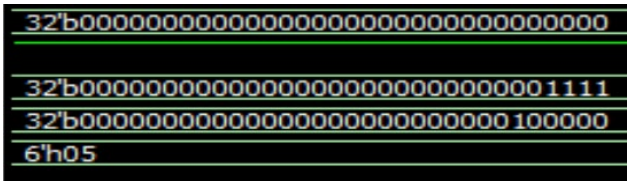


Figure 62 Shift amount is too great test

If the shift amount exceeds 32, the shift result is 0, regardless of the direction.

B) Register file

The register file has 32 32bit registers which in turn have 1-bit register copied over 32 times. The 1-bit register is made of a Flip Flop which consists of an SR-Latch connected to a D-latch with asynchronous control.

Since the register file was tested already, there is no need to test those components any further.

The testbench waveform and console output is shown. The testbench fills the register with data and then reads it, checking if the data is correct and counting passed tests if they pass. This process continues 32 times. For this test, the testbench was already included with the project.

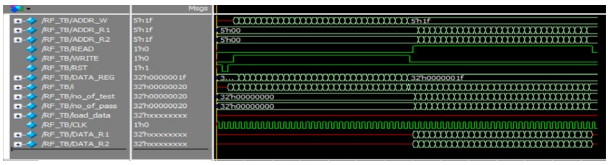


Figure 63 Register Tb waveform

Figure 64 Register TB output

C) Data Path

The Datapath was not implemented in my project due to time constraints.

D) Control Unit

The control unit was not implemented in my project due to time constraints.

7. Conclusion

From this project, I learnt how to install Modalism and simulate Verilog files. I also learned the basics of connecting registers, outputs, and filling out modules to work. I also learned how to navigate Verilog's outputs and wave form results and use a mem_data_dump. There was also a warning given by the simulator that my register file was outputting a 5-width output but that was being inputted into a 26-width output in processor. I did not touch it as it did not result in any change to the test result nor did it throw a compiler error because that would just mean some bits would not be used, which is ok for a simulation.

7. REFERENCES

1. Digital Design (4th edition) M. Moris Mano and Micheal D Ciletti (Dec 15 2006)
2. Verilog HDL: A guide to Digital Design and Synthesis, Second edition by Samir Palnitkar.