Arithmetic Logic Unit(ALU) Behavioural Modeling

Zachary Ian May
San Jose State University
Sunnyvale, CA
Zachary.May@sjsu.edu

***Abstract*—This Project deals with the creation of a behavioral model for a CPU. This model will support a basic instruction set following a RISC style.**

Keywords—ALU, Behavioral Modeling, CPU, Memory, Registers, Verilog

I. INTRODUCTION

The goal of this project is to model the behavior of a CPU using the ModelSim Verilog Simulation Program. Through this project, we should be able to implement the core functionality of a CPU using the given instruction set. This design will use a combinational ALU and will not have native support for division or floating-point values. Each instruction will be completed in 5 clock cycles.

II. SYSTEM REQUIREMENTS

The requirements for the system (DaVinci) were to support the CS147sec05 instruction set. This instruction set was given prior to the start of the project. The instruction set contains three types of instructions, R-type, which deal with registers. I-type, which deal with Immediates. And J-type, which deal with jumping and the stack. These different types of instructions feed the system instructions in different formats, as seen in fig. 2-1. Each instruction would be read from memory during the Instruction Fetch (IF) phase of the processor and decoded to send out different signals during the Instruction Decode (ID) phase of the processor. With this behavioral model, these two phases became rather small as most of the functionality occurred in the latter three phases: Execution (EXE), Memory access (MEM), and Write Back (WB). Through our instruction set we support most basic arithmetic operations (excluding division), some logical operations, and memory access operations, branch operations, and stack operations. The full list of supported operations can be seen in separated by type in figures 2-2 through 2-4.

The system would be divided into several modules to represent different components of a CPU. The primary functional components of DaVinci were the ALU, control unit, Register File, and the Memory. Of these modules, the Memory was provided to us, as were the modules used to link the functional modules to each other. The modules which required implementation (ALU, Control Unit, and Register File) were the primary focus of this project. The ALU required only minor adjustments from our previous project, as a result the ALU will mostly be left out of this report, except in cases where modifications occurred.

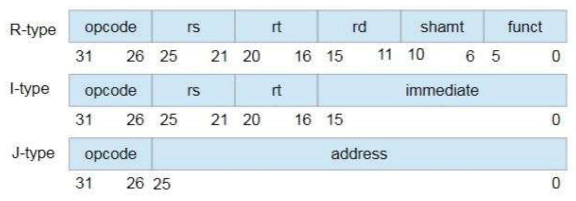
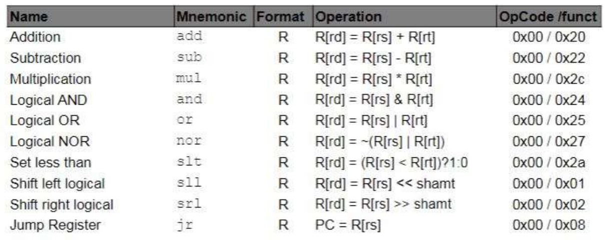
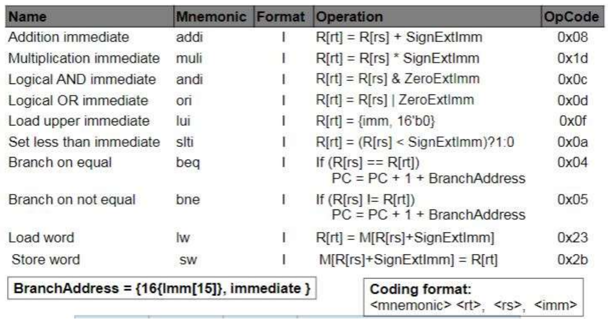


Fig 2-1. Instruction format according to instruction type [1]



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

Fig 2-2. R-type instructions [1]

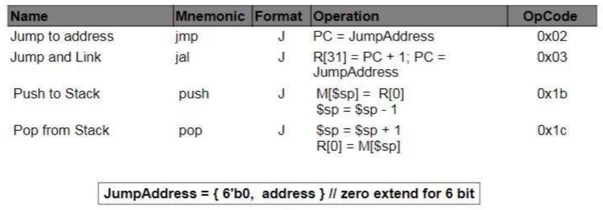


Name	Mnemonic	Format	Operation	OpCode
Addition immediate	addi	I	R[rt] = R[rs] + SignExtImm	0x08
Multiplication immediate	muli	I	R[rt] = R[rs] * SignExtImm	0x1d
Logical AND immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	0x0c
Logical OR immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	0x0d
Load upper immediate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

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

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

Fig 2-3. I-type instruction [1]



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

Fig 2-4. J-type instructions [1]

III. DESIGN AND IMPLEMNTATION OF MEMORY

The Memory's implementation was provided to us for this project. The memory model used a combined in/out data line instead of separate Read/Write lines. As a result, the implementation of the memory and the Control Unit required us to set the state to electrical isolation when not in use by one of the components in order to give the other component control of the line. The memory unit also provided a model for us to create our register file from. The memory model followed a layout as presented in figure 3-1.

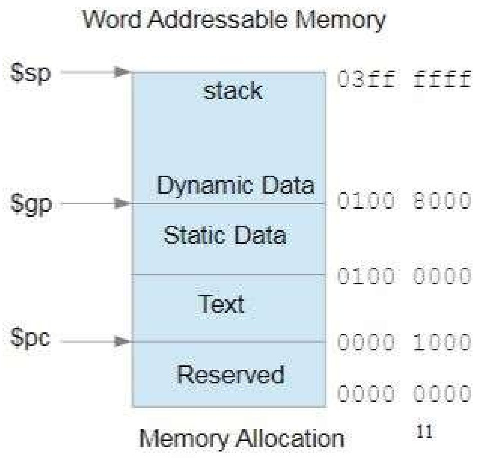


Fig 3-1. Memory Model [1]

IV. Design and implemnation of Processor

The processors design and implementation were the primary foci of this project. While the majority of the time spent on creating the DaVinci model processor came in the implementation of the Control Unit, the first part of the project was to create the Register File based of the Memory model we were given.

The Register File was to contain 32 32-bit registers. 28 of these registers were to be directly usable, and the remaining 4 to be reserved for specific functions. The implementation of the register was done primarily using registers in ModelSim, this type of register is more similar to a variable than to an actual register construction. The module would, upon being fed a read signal of 1 and a write signal of 0, send out the data stored in the two read addresses to the read data lines. When fed a write signal of 1 and a read signal of 0, it would write the data on the write data line to the location of the write address. If it were fed a read and write signal of both 1 or both 0, it would do nothing, simply holding the data that is already being fed out of the module. A diagram of the Register File can be seen in Figure 4-1.

The ALU had a minor change from the previous project, a ZERO line was added to signal if the result of the ALU was zero, this functionality was useful for the bne and beq instructions.

The largest part of the project, the Control Unit, was made up of the primary module and the state machine. The state machine was designed so that it would switch which state was output to the main module with each clock cycle. This part of the Control Unit was important to the functionality as a whole, this allowed the Control Unit to switch its behavior depending on which phase the processor was in.

The functionality of the Control unit would start with retrieving the instructions from the memory. From this state it would send the appropriate signals to the Memory and then proceed to the next phase. Here the unit would parse the instruction and generate the required control signals, however as we are only modeling the behavior of the CPU, this phase only parses the actual instruction and reads the registers provided in preparation for the next phase. The next phases differ between the instructions. In the Execution phase, the values from any provided registers, Immediates, or addresses are fed to the ALU along with the other required instructional data, to perform the calculations required of whichever instruction is being executed. The Memory access phase is only used by a few instructions, this phase is where those instructions send the signals required to the memory to write or read any data from memory. The final phase is where the Program Counter is incremented to the next instruction, or in the case of a jump or branch, to whichever address is specified in the instruction. This step is also responsible for writing the result of any calculations or memory read to the register file in the specified location. From this phase, the cycle starts over with the next instruction.

The other module involved in the Processor was the processor module itself, where all the functional modules were instantiated and linked to the proper ports such that the model would function properly. This module output the Memory read, write, address, and data lines to the memory. The end result of the project was the DaVinci module, which combines both the Memory and the Processor module into it's final instance. This final instance is where all testing of the functionality occurred.

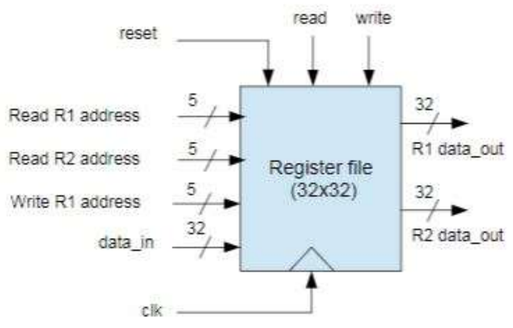
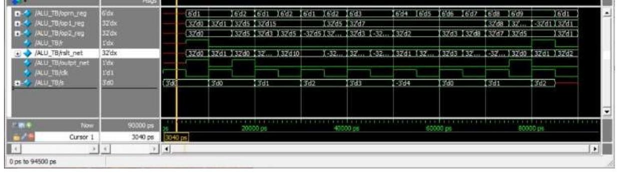


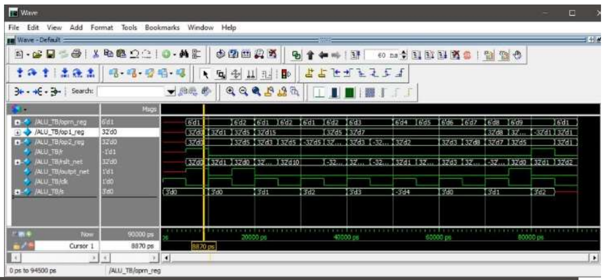
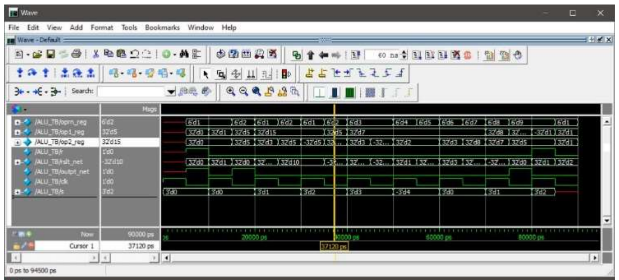
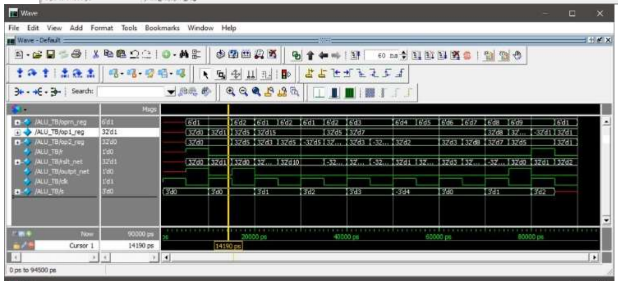
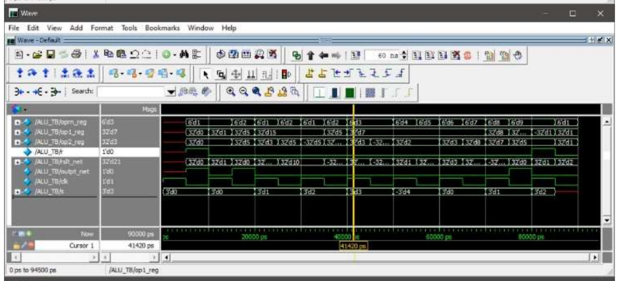
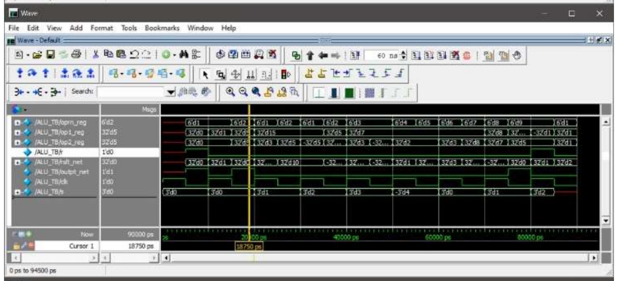
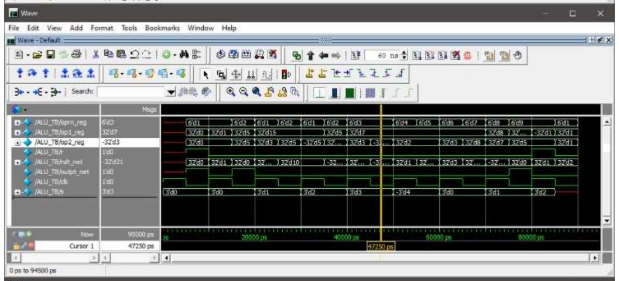
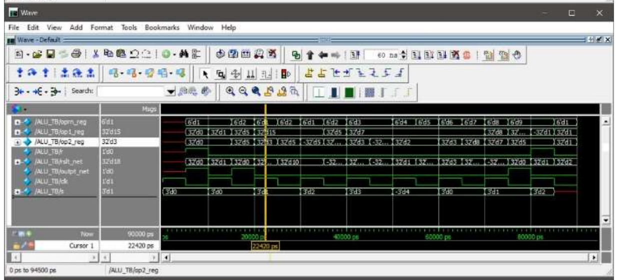
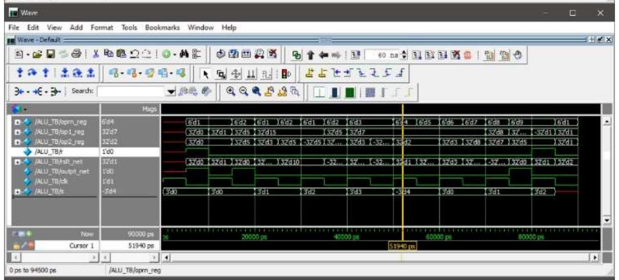
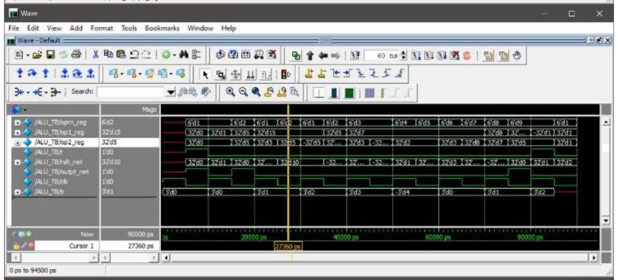
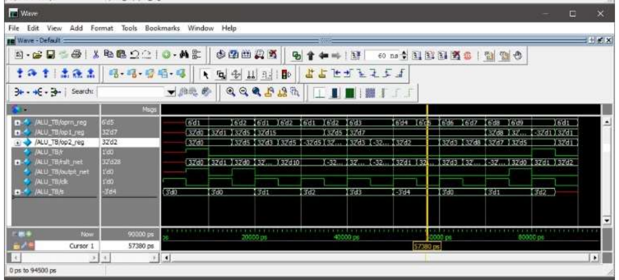
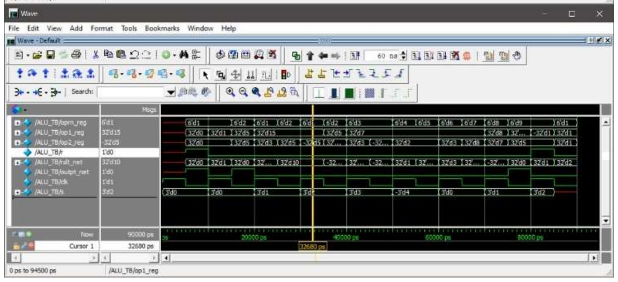
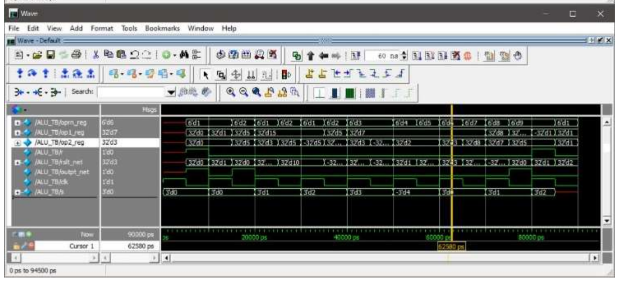
Fig 4-1. Register File Model [2]

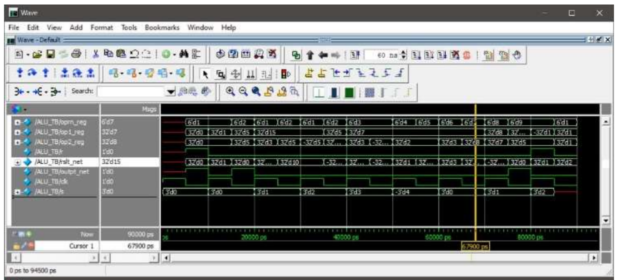
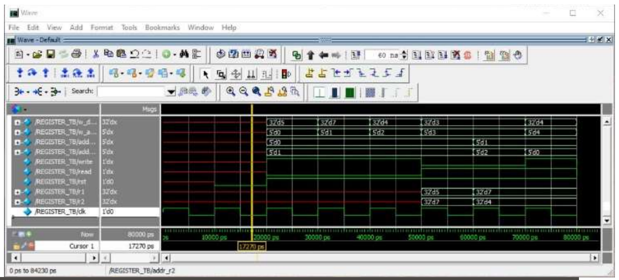
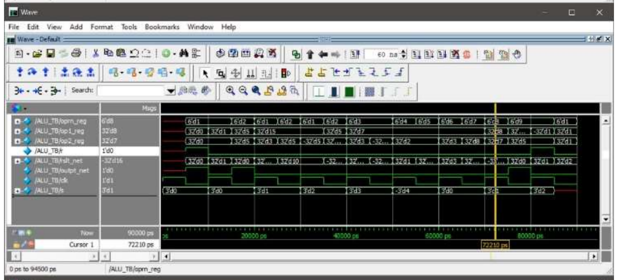
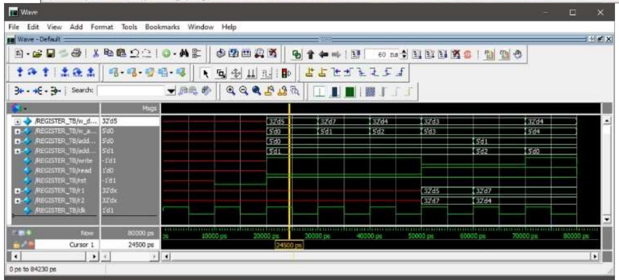
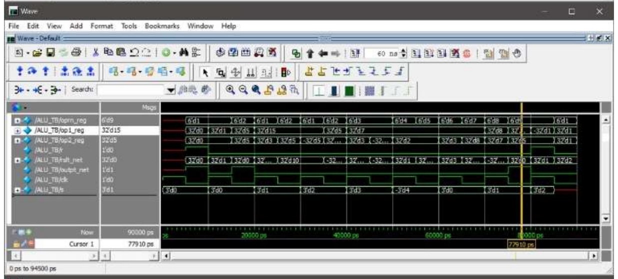
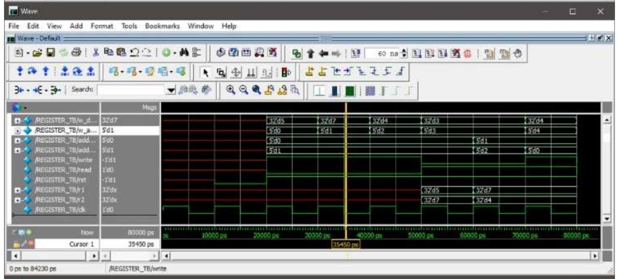
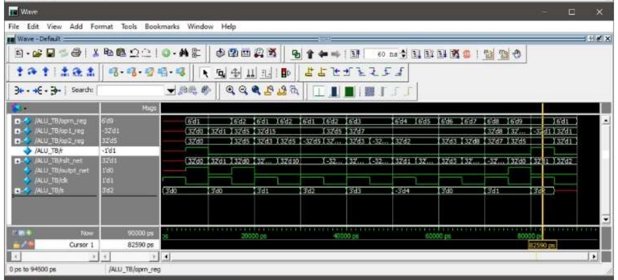
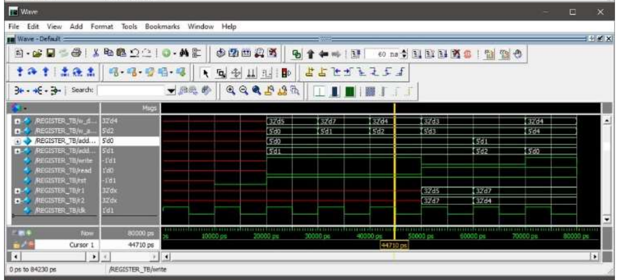
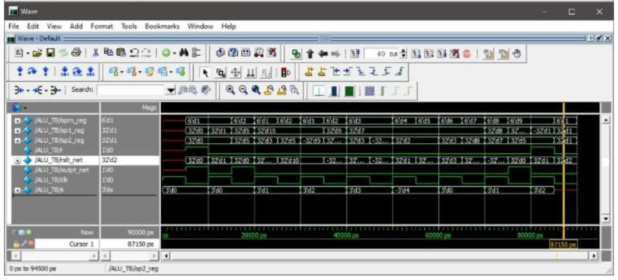
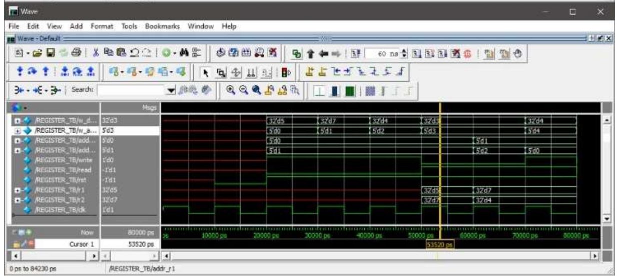
V. Test strategy and implemenation

The testing for each component of the DaVinci CPU happened in separate test benches. Each functional module was tested for several edge cases and on various normal cases. Through testing, some minor and major errors were discovered and corrected. Wave forms of tests are provided in this section. Testing of the Control Unit was done in a different method from most other modules. This module was tested by printing out the different received instructions, then comparing the Memory dump of the results of the test to the expected result of the test. For this only a small section of the executed instructions has been provided.

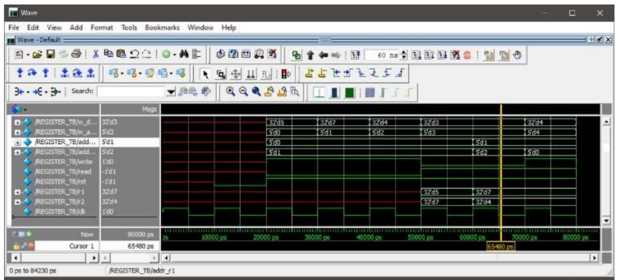
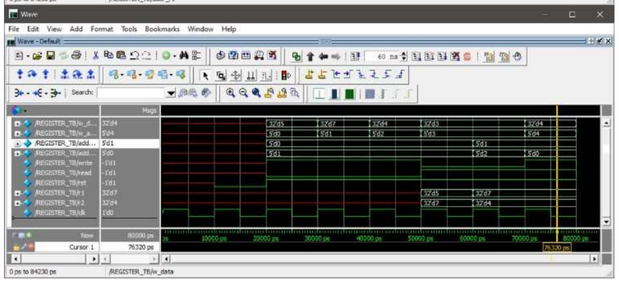
The following figures are a test of the ALU




Cursor 1 8870 ps

Cursor 1 37120 ps

Cursor 1 14190 ps

Cursor 1 41420 ps

Cursor 1 18750 ps

Cursor 1 47250 ps

Cursor 1 22420 ps

Cursor 1 51940 ps

Cursor 1 27360 ps

Cursor 1 57380 ps

Cursor 1 32680 ps

Cursor 1 62580 ps

The following figures are a test of the Register File

The following is the text output of the Control Unit test.

```
# @    20ns -> [0X20420001] addi r[02], r[02], 0X0001;
# @    70ns -> [0X3c000100] lui r[00], 0X0100;
# @   120ns -> [0Xac010000] sw r[00], r[01], 0X0000;
# @   170ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @   220ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @   270ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @   320ns -> [0X00411020] add r[02], r[01], r[02];
# @   370ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @   420ns -> [0X08001003] jmp 0X0001003;
# @   470ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @   520ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @   570ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @   620ns -> [0X00411020] add r[02], r[01], r[02];
# @   670ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @   720ns -> [0X08001003] jmp 0X0001003;
# @   770ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @   820ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @   870ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @   920ns -> [0X00411020] add r[02], r[01], r[02];
# @   970ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @  1020ns -> [0X08001003] jmp 0X0001003;
# @  1070ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @  1120ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @  1170ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @  1220ns -> [0X00411020] add r[02], r[01], r[02];
# @  1270ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @  1320ns -> [0X08001003] jmp 0X0001003;
# @  1370ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @  1420ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @  1470ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @  1520ns -> [0X00411020] add r[02], r[01], r[02];
# @  1570ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @  1620ns -> [0X08001003] jmp 0X0001003;
# @  1670ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @  1720ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @  1770ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @  1820ns -> [0X00411020] add r[02], r[01], r[02];
# @  1870ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @  1920ns -> [0X08001003] jmp 0X0001003;
# @  1970ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @  2020ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @  2070ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @  2120ns -> [0X00411020] add r[02], r[01], r[02];
# @  2170ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @  2220ns -> [0X08001003] jmp 0X0001003;
# @  2270ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @  2320ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @  2370ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @  2420ns -> [0X00411020] add r[02], r[01], r[02];
# @  2470ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @  2520ns -> [0X08001003] jmp 0X0001003;
# @  2570ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @  2620ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @  2670ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @  2720ns -> [0X00411020] add r[02], r[01], r[02];
# @  2770ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @  2820ns -> [0X08001003] jmp 0X0001003;
# @  2870ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @  2920ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @  2970ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @  3020ns -> [0X00411020] add r[02], r[01], r[02];
# @  3070ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @  3120ns -> [0X08001003] jmp 0X0001003;
# @  3170ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @  3220ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @  3270ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @  3320ns -> [0X00411020] add r[02], r[01], r[02];
# @  3370ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @  3420ns -> [0X08001003] jmp 0X0001003;
# @   3470ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @   3520ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @   3570ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @   3620ns -> [0X00411020] add r[02], r[01], r[02];
# @   3670ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @   3720ns -> [0X08001003] jmp 0X0001003;
# @   3770ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @   3820ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @   3870ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @   3920ns -> [0X00411020] add r[02], r[01], r[02];
# @   3970ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @   4020ns -> [0X08001003] jmp 0X0001003;
# @   4070ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @   4120ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @   4170ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @   4220ns -> [0X00411020] add r[02], r[01], r[02];
# @   4270ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @   4320ns -> [0X08001003] jmp 0X0001003;
# @   4370ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @   4420ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @   4470ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @   4520ns -> [0X00411020] add r[02], r[01], r[02];
# @   4570ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @   4620ns -> [0X08001003] jmp 0X0001003;
# @   4670ns -> [0X20000001] addi r[00], r[00], 0X0001;
# @   4720ns -> [0Xac020000] sw r[00], r[02], 0X0000;
# @   4770ns -> [0X20430000] addi r[02], r[03], 0X0000;
# @   4820ns -> [0X00411020] add r[02], r[01], r[02];
# @   4870ns -> [0X20610000] addi r[03], r[01], 0X0000;
# @   4920ns -> [0X08001003] jmp 0X0001003;
# @   4970ns -> [0X20000001] addi r[00], r[00], 0X0001;
```

Conclusion

The result of this project is a functioning model of a processor that implements as RISC style instruction set. As a result of this project, I have deepened my understanding of the functionality of a CPU, and the design of an instruction set. Though the behavioral model differs from a normal implementation in a few significant ways, it serves a good model of what goes into the creation of a processor, and gives enough insight into the process to allow the procedure to be transferred to a gate level modeling of a similar processor.

References

[1] K. Patra. CS147. Class Lecture, Topic: “Computer System, Instruction Set, ALU” San Jose State University, San Jose, CA, 2020

[2] K. Patra. CS147. Class Lecture, Topic: “Clock, Memory, Controller, Von Neumann Architecture, System SW” San Jose State University, San Jose, CA, 2020