Arithmetic Logic Unit(ALU) Mixed Level Modeling

Zachary Ian May

San Jose State University

Sunnyvale, CA

Zachary.May@sjsu.edu

***Abstract*—The goal of this project was to create a mixed behavioral and gate level model of a processor, without pipelining. All functionality except for the controller, of the CPU itself is done via gate level modeling.**

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

I. Introduction

The goal of this project is to create a functional model of a CPU using the ModelSim emulator, as has been the case with the past three projects. Through creating the functional model of this CPU, we will further understand the general approach to making a CPU, which lacks pipeline or cache implementation. This design for this CPU, much like the previous project, will implement an ALU using combinational logic, and as such does not support for division or floating-point values natively.

II. System Requirements

The DaVinci Processor's requirements are for the system to support the CS147sec05 instruction set, much as our previous project. This instruction set was presented at the start of the semester, and as such we have worked with it extensively. We use three patterns of instructions, those which deal with registers, R-type; those which deal with Immediate values, I-type; And finally those which deal with the stack and jumping to different addresses, J-type instructions. Each instruction is formatted differently, as can be seen in fig. 2-1. There are 5 different states which the processor cycles through. During these states, the processor retrieves the instruction to be executed, the instruction fetch phase; Deciphers the instruction, the instruction decode phase; These two phases are followed by the portion of the processor which still uses behavioral modeling. These next steps are the Execution phase, where operators are sent to the ALU, then the Memory phase, where any necessary memory access occurs. Finally, the Write back phase, where any information that needs to be written to registers is written.

In comparison to the previous model of the DaVinci CPU, the phases have fewer statements, however they have become more information dense. There is now a control signal, which sends any needed information to control the gate level model to the data path.

When compared to the previous models modules, this model is quite complex. We made minor changes to the overall structure of the primary modules. The CPU now consists of a Data Path and a Controller, which each use many smaller components. Additionally, the memory has been encapsulated by a wrapper module, which allows us to have separate read and write lines from the memory, instead of a combined inout line. We have also been required to create a gate level model of the register file within the CPU, we no longer use the variable like register available in Verilog, but instead have created several intermediate modules that lead up to a full 32-bit register. There are several other required sub-modules, however these will be discussed in later sections.

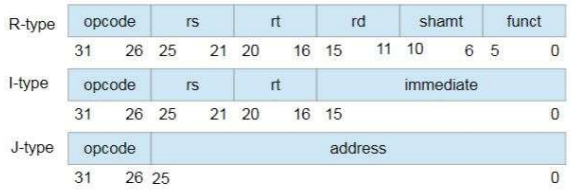

Fig 2-1. Instruction format according to instruction type [1]

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 Reused

As this project is an expansion on our model from the previous project, we have several structures that are reused. First, the Memory model is mostly reused, with the exception of the wrapper module provided to us for this project. As described in the requirements section, this memory model now has separate in and out lines, instead of the combined inout line of the previous project, the implementation of the wrapper module is much like what we were required to create for the previous project. As in the previous project the memory model follows a layout as presented in figure 3-1.

A similarly reused structure is the layout of the ALU, we have maintained a consistent function code arrangement for this project as we have done in previous projects. With the implementation done in this project, we can now see why some of the choices of fuction code for the ALU were made.

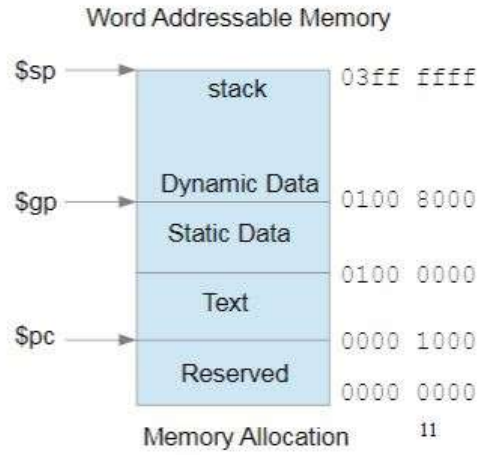

Fig 3-1. Memory Model [1]

IV. Design and Implemnation of the Processor and Submodules

The primary focus of this project was to translate what we have learned throughout the semester into a gate level version of what we already have made. The ALU and its sub-components were the first concern for this project, followed by the register model, and finally the data path.

The first steps for creating the ALU was to create our components. To begin, we had to create our ripple carry adder. This was done by first, creating a half adder, then from two half adders creating a full adder, which took two bits as input, and gave the sum of the two bits through a carry out bit, and two result bits. These full adders were then linked together to create a 32 bit ripple carry adder, in order to create a ripple carry adder/subtractor, we added an additional input of a subtraction bit, which was the carry in bit of the first full adder, this bit also was given to 32 xor gates, which allowed us to create a twos compliment of the number which was to be subtracted. As a result, we now have fully functional addition and subtraction for the ALU.

The next step in creating the ALU, was to create a multiplier. As we are implementing a combinational ALU, we had to use 31 32-bit ripple carry adders, and 32, 32 2x1 and gates. At this stage of the project, we created several useful submodules, most of which were 32-bit 2x1 versions of basic gates. This allowed us to create the multiplier relatively easily. For this stage of the project, I began to create scripts to speed up the repetitive task of instantiating over 60 modules. I created similar scripts for many of the other modules, which involved large quantities of repetitive instantiation.

The next component of the ALU, was the Barrel shifter. This section of the project is where I spent a large amount of time, before deciding to commit to writing a script efficiently. To create the barrel shifter, we first had to create multiplexers. After creating multiplexers, we were able to create left and right shifters. These would take the second input and shift the first input in stages according to which bits were 1. Each level shifting more than the previous. These left and right shifters, were then combined along with a multiplexer to determine which type of shift would be executed, and then with an additional multiplexer, to determine if the result is zero due to shifting out of range of the inputs.

From this stage we now had the modules needed to implement the ALU, we instantiated one of each of the following modules: ripple carry adder/subtractor, multiplier, barrel shifter, 32-bit 2x1 and, 32-bit 2x1 or, and 32-bit 2x1 nor. We also created a 32-bit 16x1 mux, which we used to implement the function codes for the ALU. We additionally created a 32input nor gate, which determined if the output of the ALU was zero, which we used to output for use in the SLT, BNE, and BEQ functions.

The next main component that we worked on was the Register file, this involved the creation of several submodules. The first of which being and SR-Latch, D-Latch, and D-Flipflop. These were created with both a reset and preset function, to better aid in creation of the register file. The D-Flipflop needed to be a positive edge triggered flip flop, so we inverted the clock signal from a negative edge triggered flip flop.

Through the creation of the D-Flipflop, we were then able to create a 1-bit register, using a multiplexer and a D-flipflop. From here, we were able to create a 32-bit register, by linking 32 1-bit registers. And expanding the 32-bit register to a 32x32 register file was only a matter of creating 32 32-bit registers and linking them to a 32x1 multiplexer in order to control which register was being written to. Additionally, special 32-bit registers were created for the Program counter register, and the Stack pointer register, which had to be instantiated to different values on reset. The general model of a 32x32 register file can be seen in figure 4.1.

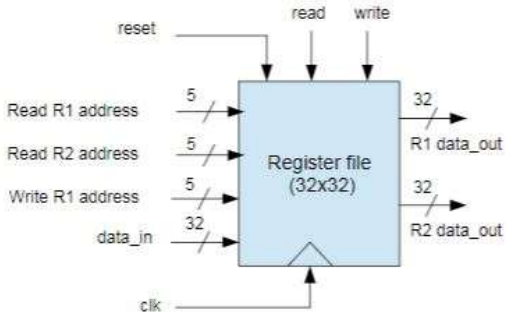

Fig 4-1. Register File Model [2]

The final components that we needed to create were the data path and the controller. These sections were where the

main difference between each student's project would appear, as this is where we determined which bits of the control signal, would control which multiplexer, or register input. This stage was a large amount of work, as it was not a repetitive task of instantiation like previous modules. This section involved creating each of the components we needed (ALU, PC, SP, register file, and bit replicators), as well as several multiplexers and wires to connect each of them. This section is where I began to fall behind and lose time approaching the deadline. As a result, this section is far less clean than other sections of this project.

The following is the description of my control signal as documented in the data path module.

```
/**
* CTRL description
* 0 - PC load
* 1 - IR load
* 2 - SP load
* 3 - reg write
* 4 - reg read
* (5-10) - ALU oprn
* 11 - PC s1
* 12 - PC s2
* 13 - PC s3
* 14 - Read addr s1
* 15 - Write addr s1
* 16 - Write addr s2
* 17 - Write addr s3
* 18 - Write data s1
* 19 - Write data s2
* 20 - Write data s3
* 21 - Op 1 s1
* 22 - Op 2 s1
* 23 - Op 2 s2
* 24 - Op 2 s3
* 25 - Op 2 s4
* 26 - Memory data s1
* 27 - Memory address s1
* 28 - Memory address s2
*/
```

The final section of the project was to translate the behavioral model of the controller into the mixed model required for this project. Overall, the change from registers to the control signal made the code look cleaner, however it was far more information dense, and as a result far more difficult to troubleshoot when problems became apparent. This section involved translating each instruction from register implementation, to determining which multiplexer needed to be changed. This section is also where the most improvement could be done. The control signal I have set up could be simplified by changing the input order of the multiplexers.

V. Test strategy and implemenation

Every subcomponent used in the project was tested separately in a testbench. Each test is available in the zip provided with this report.

The method of approach to testing for each component will be described here. For small components, who had a large impact on the system, every input combination was tested. Examples of these types of components were the half adder, a reduced version of a bit shifter, and multiplexers. Due to the significant impact that these would have on the project as a whole, it was worth the time to compare the expected results to the actual results.

For other components, who had a significant role, but who had a large number of potential inputs, all potential edge cases were used, with the addition of some exemplary normal tests, in order to determine that they resulted properly. For constructs of many subcomponents, such as the ALU or Register file, tests from previous projects were used in order to confirm their functionality. Finally, the functionality of the CPU as a whole was tested with the test files provided with the starter files.

Due to the amount of tests done, and the number of results compared, images of the waveforms of these components are not provided with the report, instead these may be obtained through running the files labeled with an additional "_tb". These files contain most of the tests used to confirm the functionality of the components. Some of the tests were lost in fatal crashes of my machine, and as such not all of the tests are provided. In future, I intend to create a backup of any testing materials prior to executing a test, in case of similar fatal errors.

Conclusion

This project has helped to provide deeper understanding of the workings of a basic processor. Though I was unable to complete the full functionality of the processor, I have a good idea of what I could do, given more time to finish the functionality of this project. The main mistakes made when conducting this project, were the poor time management. Should I have started off writing all repetitive modules with scripts, then I likely would have had enough time to do more thorough debugging of the data path and control unit, which I should have spent more time on, when compared to other modules used in this project.

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