Computer Architecture Lab 3: Single Cycle CPU

Lauren Anfenson
Anusha Datar
Annie Kroo

November 13, 2018

1 Background

A single-cycle CPU is a CPU that completes one instruction in each clock cycle. Our CPU supports a 32-bit MIPS instruction set and specifically can execute the following instructions: Load Word, Store Word, Jump, Jump Register, Jump and Link, Branch on Equal, Branch Not Equal, XOR on an immediate, ADD an immediate, ADD, SUB, and SLT.

2 Our Architecture

Our CPU consists of five main components configured as shown in the block diagram below.

2.0.1 Program Counter

The program counter is a register that stores the address of the instruction that is currently being executed.

According to the MIPS architecture, our instruction memory is word addressed. So, by default our program counter increments by four on the clock's rising edge to read the next sequential instruction.
To support Jumping and Branching, we also had to include the functionality to pass a non-incremented address to the program counter. This was accomplished with a 4-input multiplexer with inputs of the next incremented counter, a jump address, a branch address, and the value of register Rs (for Jump Register). The select lines of this mux are set by the instruction decoder logic.

2.0.2 Memory

Our memory file includes both our instruction and data memory – instruction memory is stored from the bottom up, while data memory is stacked from the top down. The basic implementation of the memory is from the example code at the bottom of the Lab 3 assignment description, with the instruction memory changed to be word addressed.

2.0.3 Instruction Decoder Logic

To begin decoding the instructions from the instruction memory, we characterized each of the instructions using the MIPS reference sheet.

block.jpg

Figure 1: CPU Block Diagram.

Instruction	Type	RTL	rs	rt	rd	ALU Operation
LW	I	R(rt) = M[R[rs] + imm]	in	out	x	x
SW	I	M[R(rs) + imm] = R(rt)	in	out	x	ADD(R(rs) + imm)
J	J	PC=address	x	x	x	x
JR	R	PC=R(rs)	in	in	out	x
JAL	J	R(31)=PC+4; PC=address	x	x	x	x
BEQ	I	if (R(rs)==R(rt)): PC=PC+4+address	in	out	x	SUB(R(rs) - R(rt))
BNE	I	if (R(rs)!=R(rt)): PC=PC+4+address	in	out	x	SUB(R(rs) - R(rt))
XORI	I	R(rt) = R(rs) XOR imm	in	out	x	XOR(R(rs), imm)
ADDI	I	R(rt) = R(rs) + imm	in	out	x	ADD(R(rs) + imm)
ADD	R	R(rd) = R(rs) + R(rt)	in	in	out	ADD(R(rs) + R(rt))
SUB	R	R(rd) = R(rs) - R(rt)	in	in	out	SUB(R(rs) - R(rt))
STL	R	R(rd) = R(rs) ¡ R(rt)	in	in	out	SLT(R(rs), R(rt))

Then, we set a variety of flags within the decoder logic to dictate the behavior of the rest of the CPU. These flags are defined as the following:

Flag	Definition
reg_WE	Register Write Enable
op_imm	Operate on Immediate
dm_WE	Data Memory Write Enable
dest_add	Multiplexer Select Option for Address Destination
reg_in	Mutliplexer Select Option for Register Write Information
dm_add	Data Memory Address Input Enable
pc_op	Multiplexer Select Option for Program Counter

Then, we set these flags for each of the potential operations.

Instruction	reg_WE	op_imm	dm_WE	dest_add	reg_in	dm_add	pc_op
LW	1	1	0	1	1	1	0
SW	0	1	1	1	0	1	0
J	0	0	0	0	0	0	1
JR	0	0	0	0	0	0	3
JAL	1	0	0	2	2	0	1
BEQ	0	0	0	0	0	0	2
BNE	0	0	0	0	0	0	2
XORI	1	1	0	1	0	0	0
ADDI	1	1	0	1	0	0	0
ADD	0	0	0	0	0	0	0
SUB	0	0	0	0	0	0	0
SLT	0	0	0	0	0	0	0

2.0.4 Register File

The register file is a small, quick bit of memory that the CPU can use to store values for executing instructions. Our register file is able to store 32 32-bit registers and has the capability to read from two registers and write to one register at a time. According to the MIPS architecture, the first read address is always Rs, while the second read address is always Rt. The register being written to can be either Rd (ADD, XOR, SUB, and SLT), Rt (ADD an immediate, Load Word), or 31 (Jump and Link).
We used an existing register file from Homework 4 for our register file framework, adding a multiplexer to the write address to support the different values that our MIPS instruction set requires.

2.0.5 ALU

Our ALU supports the functions ADD, SUB, SLT, and XOR. Whenever we perform one of these operations we use the data stored in Rs, while the other operand is either the data in Rt (ADD, SUB, SLT, Branch on Equal, Branch on Not Equal) or a sign-extended immediate (ADD an immediate, Load Word, Store Word, XOR). These operands are selected using a 2-input multiplexer with select lines determined by the instruction decoder logic. The instruction decoder logic also sets the operation code that tells the ALU what to output.
We used an existing ALU from Lab 1 for our ALU framework.

3 Testing

The register file and ALU were written as a part of previous labs and so already had test benches that confirmed their functionality.

To test the program counter we wrote a test bench that incremented by 4 three times, then jumped to a random address, then used the branching input to jump to another random address, then incremented again. This validates that every input to our mux can be successfully passed to the program counter, and

that it defaults to incrementing by 4.

Our instruction decoder logic was exhaustively tested, with success defined by comparing the output of our decoder logic against the control logic table that we created.

Once we had tested the individual components of our CPU and integrated them according to our block diagram, we ran crowd-sourced assembly programs to test the overall functionality of the CPU. Our first test used the addition and subtraction functionality. Once we confirmed that this test ran successfully, we moved on to tests that used more involved functions like branching and jumping. Running the `yeet` assembly test, we were able to validate that our CPU could branch on equal, jump, jump and link, and jump register. Below is a picture of our CPU implementing the final step of the `yeet` program, storing the correct value of `118` into register t3.

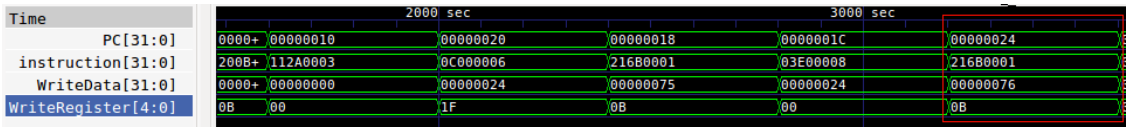

Figure 2: GTKWave screencap showing our CPU successfully completing the last step of yeet.asm.

To really confirm that our CPU could function as intended, we also ran two of the more complicated assembly tests: the Fibonacci sequence and tower of Hanoi. To confirm that these programs ran as intended we stepped through them in Mars, checking the program's behavior against the instructions implemented by our CPU. Each step that we checked was correct, but for brevity we're including screencaps of GTKWave showing our CPU sucessfully implementing the final instruction of both of these programs.

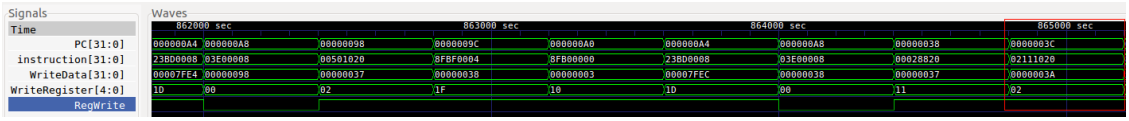

Figure 3: GTKWave screencap showing our CPU successfully completing the last step of $\text{fib}_{f}unc.asm.The value 58 (03 A in hexadecimal) is stored into register 2, which corresponds to v0.$

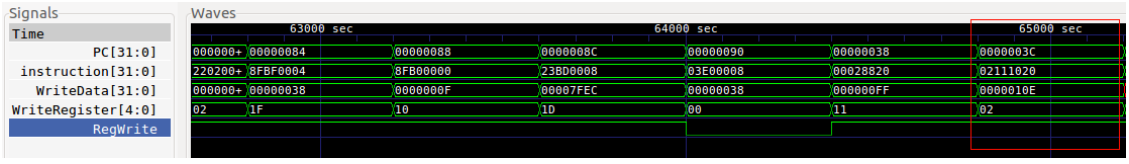

Figure 4: GTKWave screencap showing our CPU successfully completing the last step of hanoi.asm. The value 270 (10E in hexadecimal) is stored into register 2, which corresponds to v0.

4 Performance Analysis

We attempted to synthesize and implement our completed CPU in Vivado, but we were unable to complete implementation because of the large area of our program; it needed adjustments to be compatible with our FPGA. This implies that there are several ways that we can optimize our design to decrease the overall area cost.

Some of the area improvements we could implement include streamlining some of the decoder and program counter logic to decrease the total number of state variables and flags set. We also could have

decreased the total number of flags by passing the instruction more directly through the system rather than through a series of multiplexers, but this would have required a great deal of additional behavioral logic.

While unable to perform an exact timing analysis because we could not synthesize in Vivado, we recognize that our timing performance could also be improved by streamlining our decoding logic. However, because of the single-cycle nature of this CPU, we are still limited by the slowest instruction and so would predict that any optimization would have a lesser impact on timing performance than on area performance.

5 Reflection

Our work plan estimation and actual schedule for completing this lab is as stated below.

Task	**Estimated Time**	**Actual Time**
Generate block diagram and data flow.	2 hours	5 hours
Collect, evaluate, and re-test existing components without integration.	2 hours	3 hours
Integrate components into a complete module and create testing framework.	5 hours	10 hours
Write test program	30 minutes	30 minutes
Test module with class assembly programs and debug.	2 hours	20 hours
Report and documentation	1.5 hours	4 hours

Clearly, we spent much longer than we originally estimated completing this lab. The reason that we deviated so far from our original estimations was likely because we did not account for the difficulty associated with debugging the CPU; because the system is so large and has so many integrated dependencies, tracing the source(s) of problems is not trivial.