CompArch Lab 3: CPU

Kimberly Winter, Rocco DiVerdi, Kaitlyn Keil, and David Papp

November 17, 2017

1 Introduction

During this lab, we attempted to create a single-cycle CPU that would read MIPS format instructions and execute full programs, with store word, load word, three verisons of jump (straight jump, jump register, and jump-and-link), branch-not-equal, xor with an integer, add (both integer and a second value), subtract, and set-less-than all supported. However, while our pieces had been individually tested, we found that either store word or load word was not functioning properly, rendering the CPU ineffective. We spent far more time on this than our work plan indicated, for lower rewards, as we initially hoped to create a pipeline CPU. We were not able to fix the CPU within a reasonable amount of time.

2 Processor Architecture

We attempted to implement a 32-bit processor that handles SW, LW, J, JR, JAL, BNE, XORI, ADDI, ADD, SUB, SLT in MIPS format.

ADD, SUB, and SLT are all R-type instructions. These follow the format:

```
IF:
INST = IM[PC]
PC   = PC + 4

ID:
Rs = INST[25:21], Rt = INST[20:16], Rd = INST[15:11], Cmd = INST[5:0]
Da = Reg[Rs]
Db = Reg[Rt]

EX:
Reg[Rd] = ALU(Da, Db, Cmd)
```

where ALU(Rs, Rt, Cmd) is the output of the operation specified by Cmd on the operands Da and Db. R-type commands neither read nor write to memory. JR is also an R-type instruction, but follows a slightly different format:

```
IF:
INST = IM[PC]
PC     = PC + 4
ID:
Rs = INST[25:21]
Da = Reg[Rs]
EX:
PC = Da[25:0]
```

J and JAL are J-type instructions, which follow the pattern below:

```
IF:
INST = IM[PC]
PC     = PC + 4
ID:
IMM = INST[25:0]
EX:
Reg[31] = IMM // If JAL
PC = IMM
```

The rest are I-type, which vary more. Load Word and Store Word both have a memory stage. Load Word stores the value at DM[Reg[Rs] + IMM] in Reg[Rt]. Store Word puts the value at Reg[Rt] in DM[Reg[Rs] + IMM]. ADDI adds an immediate. BNE compares the values in Reg[Rs] and Reg[Rt], then branches to the value in IMM if the two are not equivalent. XORI exclusively ors the value at Reg[Rs] with IMM and stores the result in Reg[Rt].

At the moment, the PC updates on the positive edge of the clock and the Register File is written to on the positive edge of the clock. This currently is broken on the first cycle, where the register is never written to.

We broke our CPU into the different modules, where flags are turned on and off by a controller. The controller takes the opcode and function section of the instruction and sets values accordingly. Function only matters for R-type instructions. We also have a duplicated memory for the instruction memory and data memory, to replicate the idea of having a single memory to read from and write to.

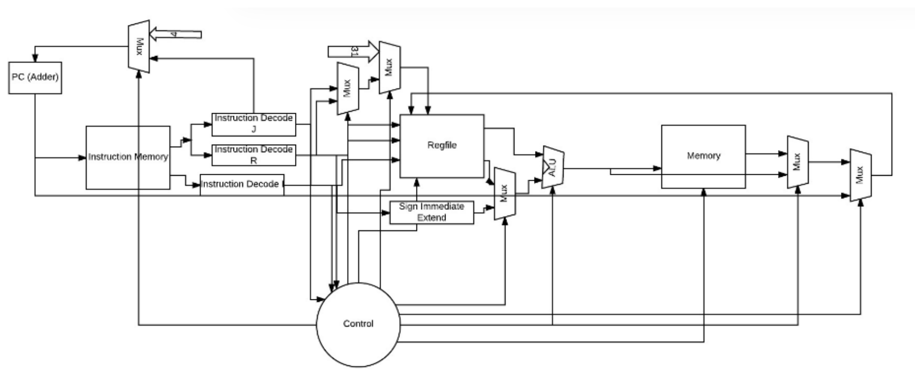

Figure 1: Circuit Diagram

3 Testing and Results

We tested all the components individually as we created (ifetch, control) or modified (ALU, register, memory) them. We also imported tests from other team members in order to evaluate the different options and choose the best one, and then make sure they were really running as intended. Once we had checked these, we connected the pieces of the CPU and began attempting to run assembly tests on it. We examined the GTKwave in order to track down issues in the results.

This worked reasonably well to track down major problems, but also meant there was a significant amount of time spent trying to decipher what caused an erroneous output in GTKwave because there were a lot more variables to keep track of in this lab than in the previous ones. It also made debugging specific modules difficult, because we could only see the inputs and outputs to potentially malfunctioning modules, rather than the variables that existed inside of the modules. So, even though we could use this tactic to locate where the malfunction might be occurring, it was not helpful to figure out what was exactly wrong within the module without isolating the malfunctioning part and running another test on that.

Despite our tests, the assembled CPU did not function as desired. Either Load Word or Store Word isn't working correctly. When we run the simulation of our CPU with assembly code ad compare it with the Mars simulation, we can step through and watch the registers, program counter, and instruction change correctly until we reach a store word, at which point the memory returns Xs. Because we can't see into memory, we can't tell whether the problem is with load word or store word, and we ran out of time to debug further. We also spent some time attempting to manipulate when on the clock different aspects of the program ran (positive edge or negative edge). In the end, we settled on everything updating on the positive edge (pc, register, etc.), which seemed to work better overall but did not fix the memory issue. In Figure 2, we can see that PC is clearly increasing and includes at least one jump. The instruction also changes for every line, and several registers are being written to. However, at the marker, register 31 is loaded with a value from memory, which results in all x's. Over the next three clock cycles, that value makes its way to the program counter (with a jump register instruction), at which point the program counter is set to all x's and the CPU ceaces to function altogether.

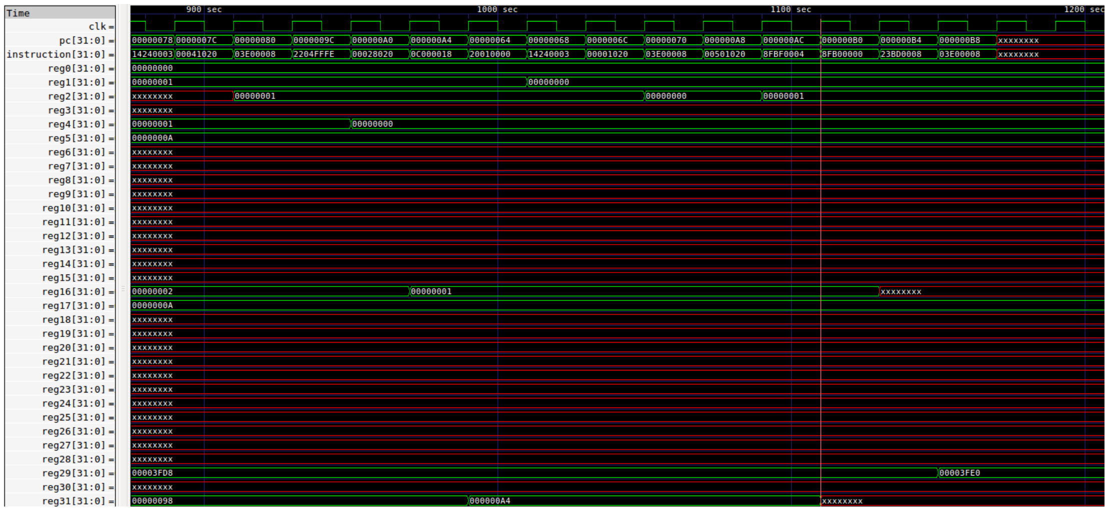

Figure 2: GTK wave results of running our CPU with the Fibonacci test program

4 How performance and area influenced our designs

Adder delay propagation: $2x32 + 4 = 68$
Adder size: $18x32 = 576$
ALU delay propagation: $2x32 + 5 = 69$
ALU size: $64x5 + 9x288 + 848 + 149 + 32 = 1,637$
Instead of implementing an ALU that increments the pc, we simply put in an adder. Firstly, we saved 1,061 space units and had an increased speed. The reason why we were able to optimize our design so much with this decision is because we are cutting out added, unnecessary features, such as set less than and nand operations.

5 Work Plan Reflection

We didn't stick to our work plan very well. We might have been able to if we had met with Ben or a NINJA at the beginning with a sketched-out design. However, we did not do this and thus spent much more time on this project than anticipated, with a lower reward (single-cycle that doesn't quite work rather than a pipeline). Our team also had some problems meeting due to various events and generally busy schedules. We also had communication issues, especially in regards to what was in the git at any given time. We could have saved more time by working on different features in parallel and then updating others on completed work. This would have saved people time when debugging other people's code or adding on to someone else's code. Overall, our schedule ended up being arbitrary as we stuck increasingly less to it as time went on.