Computer Architecture Lab 3: Single-Cycle CPU

Chris Aring, Robbie Siegel, Wilson Tang

November 2, 2017

1 Introduction

For this lab, we designed a single-cycle CPU and implemented it in Verilog. We began by designing a block diagram for a single-cycle CPU that included control signals. Once we had completed this design, we implemented individual blocks as modules within Verilog. We tested each module as we created it to ensure that it behaved as expected. We then created our top level CPU, which linked together all the high-level blocks and added control signals.

2 Processor Architecture

For the lab we implemented a single-cycle CPU. Our design is shown at a high level in the block diagram in Figure 1.

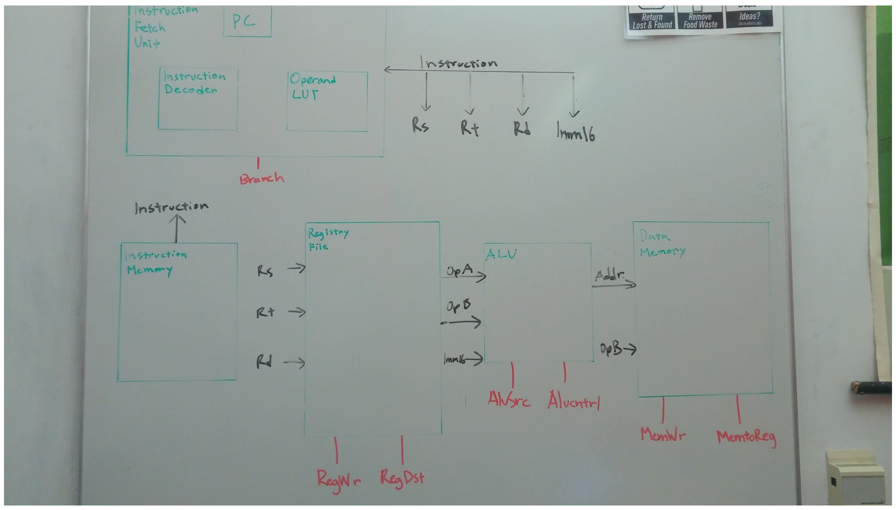


Figure 1: Single Cycle CPU Block Diagram

The CPU operates by retrieving a MIPS instruction from the Instruction

Memory and passing it into the Instruction Fetch Unit, which decodes the instruction into its core components (typically Rs, Rt, Rd, and Imm16). These components are passed to the Register to retrieve operandA and operandB, which are passed into the ALU and Data Memory. We can control the system by utilizing our control signals (red) to do a variety of operations.

We also implemented a branch control signal for branching. This branch signal controlled a mux that determined whether the next PC position was 4 away or a different position determined by the branch.

2.1 Program Counter

The program counter constantly assigns the final output of a MUX which decides if the program counter will increase by 4, or by another amount which allows it to jump.

3 Test Bench Design

3.1 Individual Components

As we implemented each individual block of our diagram, we wrote test benches in Verilog to ensure that each worked properly. This saved us many problems down the road when we tested our CPU.

When it came to testing our CPU we attempted to first test each of the individual operations to make sure that our system would correctly interpret each RTL operation. For example, we first tested a one line program that was simply an addi operation. From there, we tried other operations on our CPU.

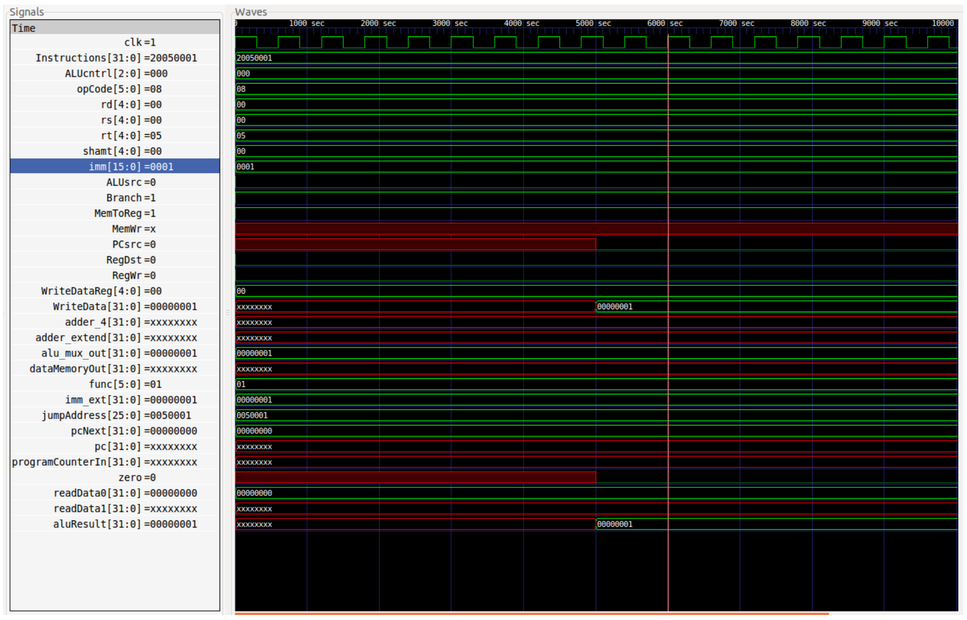
3.2 CPU Test Programs

We implemented two programs ourselves in assembly that we used to test our CPU. We first implemented the Fibonacci testing program that we began in class. For our second test, we decided to use assembly instructions that were not included within the Fibonacci test. These included bne, addi, and j. Based off of these instructions, we decided to implement a program that summed the first N integers (i.e. $1 + 2 + ... + N$).

4 Final Results and Debugging

We were unable to finish debugging our CPU as of turning in this report. One operation that our CPU was able to successfully complete was addi. The waveforms for running this operation, specifically adding the contents of the zero register to 1 and storing the result in $a0, are shown in Figure 2.

However, a significant amount of time (3-4 hours) was spent debugging and tracking our signal through the wires shown in our GTK wave. For example, we had an issue with our registers that we were able to identify when we were saw that the readData0 and readData1, which were the outputs of the register and input into the alu were not the value we expected despite the correct control signal.



```
Time
                 clk =1
Instructions[31:0] =20050001
     ALUcntrl[2:0] =000
       opCode[5:0] =08
           rd[4:0] =00
           rs[4:0] =00
           rt[4:0] =05
         shamt[4:0] =00
         imm[15:0] =0001
              ALUsrc =0
              Branch =1
            MemToReg =1
               MemWr =x
               PCsrc =0
              RegDst =0
               RegWr =0
 WriteDataReg[4:0] =00
  WriteData[31:0] =00000001
    adder_4[31:0] =xxxxxxxx
adder_extend[31:0] =xxxxxxxx
 alu_mux_out[31:0] =00000001
dataMemoryOut[31:0] =xxxxxxxx
         func[5:0] =01
    imm_ext[31:0] =00000001
jumpAddress[25:0] =0050001
     pcNext[31:0] =00000000
         pc[31:0] =xxxxxxxx
programCounterIn[31:0] =xxxxxxxx
                zero =0
  readData0[31:0] =00000000
  readData1[31:0] =xxxxxxxx
  aluResult[31:0] =00000001
```

Figure 2: The results of running an addi operation.

We know that our addi was correct because the result, shown in the bottom-most wave aluResult, was correctly calculated as 1 and stored in \$a0. However, there are still several issues with the resultant waveforms. Most notably, it takes nine clock cycles for the result to process. We were unable to figure out why this occurred. None of the waveforms at any stage of the process change until this ninth clock cycle.

The majority of our debugging was done by walking through each step of our CPU's processes very methodically. Once we found a result that was not expected, we worked backwards until we found the root of the problem.

The waveforms also show the issues we had with our program counter. We were in the process of debugging our program counter when we reached the timebox limit we had set for ourselves. Our faulty program counter prevented us from being able to run programs that were more than one line.

5 Work Plan Reflection

We were unable to stick to our initial work plan for a variety of reasons. When we first created our work plan, we decided that we would implement a multi-cycle design. However, as we began to design our CPU and look forward at our schedules, we realized that it would be much more feasible for us to implement a single-cycle. Especially since our team formed late, we felt we would have a more successful project and learn more if we attempted a single-cycle CPU.

We also structured our work plan very poorly in retrospect. We structured tasks by the operations we wanted to implement rather than by modules (i.e. registers or data memory). As we finished our system design and began implementation, we realized that we needed to complete modules and test those individually rather than try to get specific operations working.

We also did not account for how long debugging would actually take. We spent much more time debugging our CPU than we had accounted for, and we still did not get the CPU working.