Lab 3: Single Cycle CPU

Emma Westerhoff, March Saper

November 2, 2018

Intro

In this lab we implemented a 32-bit single cycle CPU following MIPS architecture. We tested our CPU using specially written assembly code.

Run Instructions

Each assembly file that has been tested on our CPU has its own testbench. In order to run our CPU with an assembly file, the following steps must be taken. If you wish to run another test on it, a test bench of similar format should be created.

- Change the settings file in the main folder to reflect the name of the assembly program you want to run.
- Navigate to the folder containing the assembly .asm file. Run the makefile from the command line.
- Navigate back into the main folder, and run the following lines of code:

```
iverilog -o [testName] [testName]_cpu_test.t.v
./[testName]
```

- The following should open the gtkwave wave model for viewing:

```
gtkwave [testName] waves.vcd
```

To run our unit tests, simply follow a similar schema to the above directions. There is no need to navigate to the .asm file.

Implementation

Block Diagram

Figure 2 on page 3 shows our block diagram. "Instruction decode and operation fetch" takes in the current instruction and outputs control signals to our datapath. The datapath performs the operations specified, and various signals along the datapath are sent to the PC logic. Normally, the PC will simply increment by four. If a special instruction is detected (jump or branch), the PC will increment accordingly.
PC+PC Special Case Logic: If a BEQ or BNE was detected, this block checks to see if the conditions were met (equal or not equal) and calculates the branch address. This logic is also in charge of selecting the PC that will be dealt with: the previous PC (for standard instructions) or the calculated addresses for branch and JR instructions.

PC Hold: A 32-bit chain of parallel DFFs that hold state for exactly one cycle (making this a single cycle CPU)

Add 4 (or Zero if JR): Adds four to the PC unless a JR operation was detected.

Jump Calc: Calculates the jump address, and selects if it a jump is detected. The RTL for calculating the address can be found below.

Instruction Memory: The .text section of the memory which stores instructions.

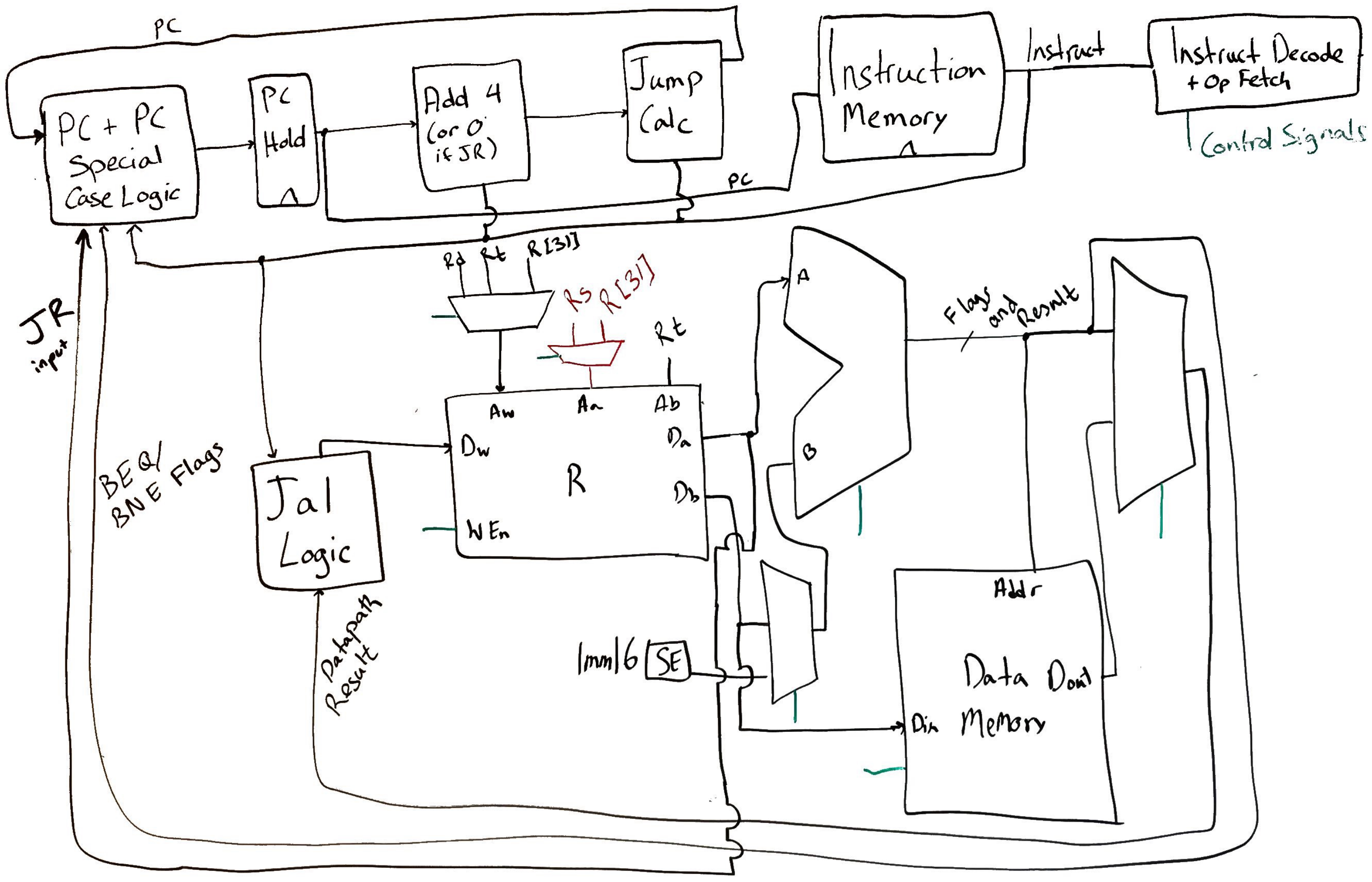


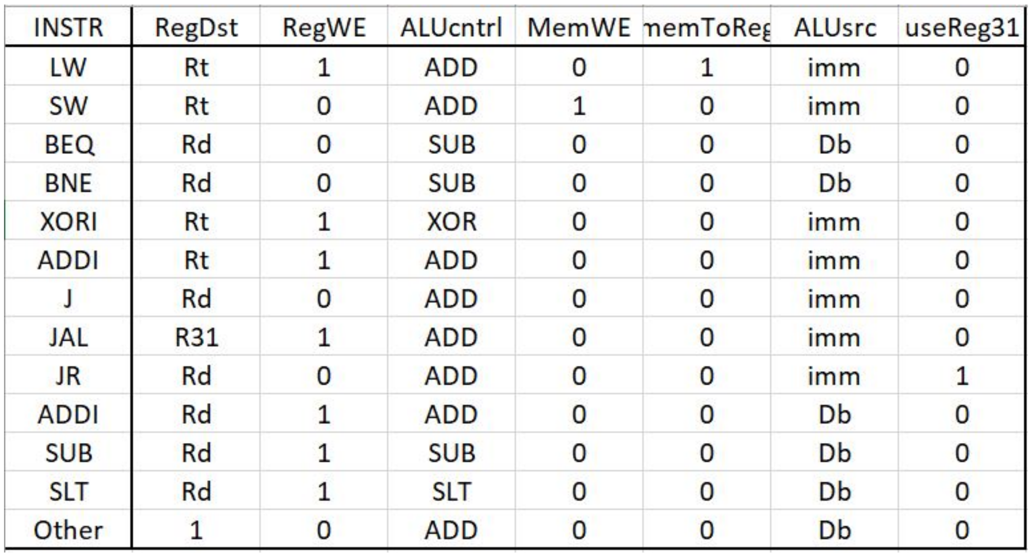
Figure 1: Block diagram

Instruction Decode and Operand Fetch: A LUT which looks at the OPCode of each instruction, and sets the register values, immediate value, and control signals for that cycle.

JAL Logic: Writes the linked jump address to R[31] if a JAL instruction was detected.

Data Memory: The .data section of the memory which stores data.

Control table



INSTR	RegDst	RegWE	ALUcntrl	MemWE	nemToReg	ALUsrc	useReg31
LW	Rt	1	ADD	0	1	imm	0
SW	Rt	0	ADD	1	0	imm	0
BEQ	Rd	0	SUB	0	0	Db	0
BNE	Rd	0	SUB	0	0	Db	0
XORI	Rt	1	XOR	0	0	imm	0
ADDI	Rt	1	ADD	0	0	imm	0
J	Rd	0	ADD	0	0	imm	0
JAL	R31	1	ADD	0	0	imm	0
JR	Rd	0	ADD	0	0	imm	1
ADDI	Rd	1	ADD	0	0	Db	0
SUB	Rd	1	SUB	0	0	Db	0
SLT	Rd	1	SLT	0	0	Db	0
Other	1	0	ADD	0	0	Db	0

Figure 2: Control Table

Instruction Set

Below is the listed RTL of each supported instruction.

- LW: R[rt]=M[R[rs]+SignExtImm]
- SW: M[R[rs]+SignExtImm] = R[rt]
- J: PC=JumpAddr
- JR: PC=R[rs]
- JAL: R[31]=PC+4; PC=JumpAddr
- JumpAddr= PC+4[31:28], address, 2'b0
- BEQ: if(R[rs]==R[rt]) PC=PC+4+BranchAddr
- BranchAddr = 14immediate[15], immediate, 2'b0
- BNE: if(R[rs]!=R[rt]) PC=PC+4+BranchAddr
- XORI: R[rt]=R[rs] XOR SignExtImm
- ADDI: R[rt]=R[rs]+SignExtImm
- ADD: R[rd]=R[rs]+R[rt]
- SUB: R[rd]=R[rs]-R[rt]
- SLT: R[rd]=(R[rs]¡R[rt])?1:0

Testing & Results

We borrowed many units from previous labs. These had already been unit tested and so we chose to trust them. As we built other behavioral modules, we performed quick unit tests on them as well. These were not fully automated, but were mostly for sanity checking purposes.
We did the majority of our CPU testing through running assembly tests. Going through each test line by line and comparing the instructions and behaviors was a significant aid in debugging. What follows is a demonstration of the functionalities of our CPU.

Load and Store Word

Figure 3 shows our CPU storing the word $0x0000059F$ into data memory and loading it into R11 on the following load word instruction.

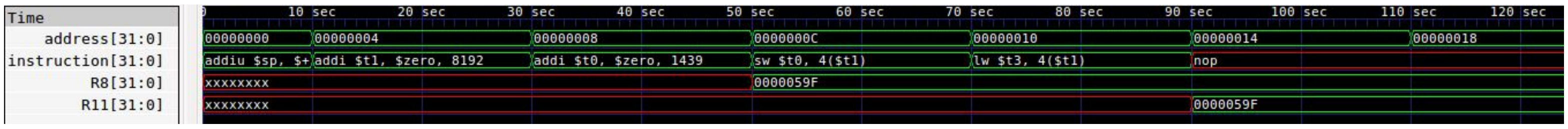

Time
address[31:0]
instruction[31:0]
R8[31:0]
R11[31:0]
10 sec 20 sec 30 sec 40 sec 50 sec 60 sec 70 sec 80 sec 90 sec 100 sec 110 sec 120 sec
00000000 00000004 00000008 0000000C 00000010 00000014 00000018
addiu $sp, $+ addi $t1, $zero, 8192 addi $t0, $zero, 1439 sw $t0, 4($t1) lw $t3, 4($t1) nop
xxxxxxxx 0000059F
xxxxxxxx 0000059F

Figure 3: Load and store word test

Jump Register and Jump and Link

Figure 4 shows address $0x00000018$ being stored into $R[31]$ following a JAL instruction. When the program counter hits a JR instruction, $0x00000018$ is returned as the new address.

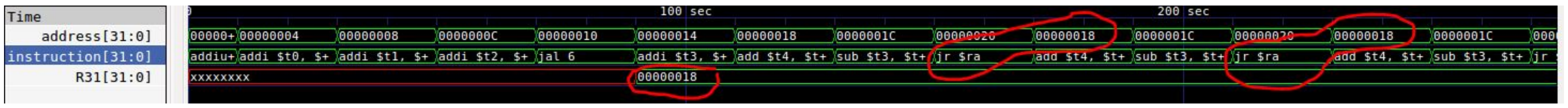

Time
address[31:0]
instruction[31:0]
R31[31:0]
100 sec 200 sec
00000+ 00000004 00000008 0000000C 00000010 00000014 00000018 0000001C 00000020 00000018 0000001C 00000020 00000018 0000001C
addiu+ addi $t0, $+ addi $t1, $+ addi $t2, $+ jal 6 addi $t3, $+ add $t4, $t+ sub $t3, $t+ jr $ra add $t4, $t+ sub $t3, $t+ jr $ra add $t4, $t+ sub $t3, $t+ jr
xxxxxxxx 00000018

Figure 4: Jump and link and jump register test

Add Immediate, Jump and Branching

In the following test, add immediate calls load the decimal values 2 and 3 into $R[8]$ and $R[9]$ respectively. A BEQ instruction compares these two values to decide if the program should jump to LOOPEND. Since these values are not equal $R[[8]$ is updated. Jump returns us to the BEQ instruction. On this second comparison $R[rs] == R[rt]$ so the program counter is set to LOOPEND. This is shown in Figure 5.

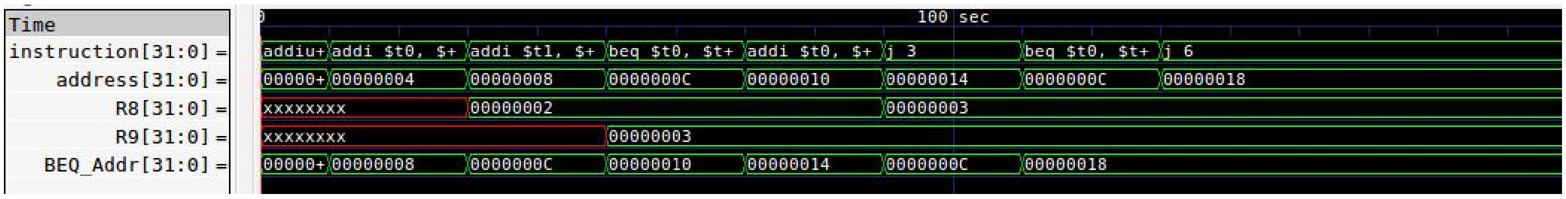

Time
instruction[31:0] =
address[31:0] =
R8[31:0] =
R9[31:0] =
BEQ_Addr[31:0] =
100 sec
addiu+ addi $t0, $+ addi $t1, $+ beq $t0, $t+ addi $t0, $+ j 3 beq $t0, $t+ j 6
00000+ 00000004 00000008 0000000C 00000010 00000014 0000000C 00000018
xxxxxxxx 00000002 00000003
xxxxxxxx 00000003
00000+ 00000008 0000000C 00000010 00000014 0000000C 00000018

Figure 5: Add immediate and branch if equal register test

Add, Sub, XORI, SLT

The first operation of this assembly program is not shown, but it loads decimal value 21 into $R[8]$. The second and thid operations show correct addition and subtraction functionality.

Performing a XORI operation with the values 010101 (found in $R[8]$) and 101010 (the immediate) produces $0xFFFFFFF7$. This is due to the way that we implemented our sign extend. Perhaps it was necessary to implement a XORI specific sign extend that would merely zero pad the immediate before xoring. Performing the same operation with 010101 (the immediate) produces $0x00000000$, as expected.
A SLT operation between $R[8]$ and $R[9]$ produces a result of $0x00000001$, as expected. The final SLT which would display $R[16]$ as having a value of $0x00000000$ is not shown due to cropping. However, the correct behavior of this module can be seen in our Waveforms folder on Github.

Figure 6 shows these behaviors.

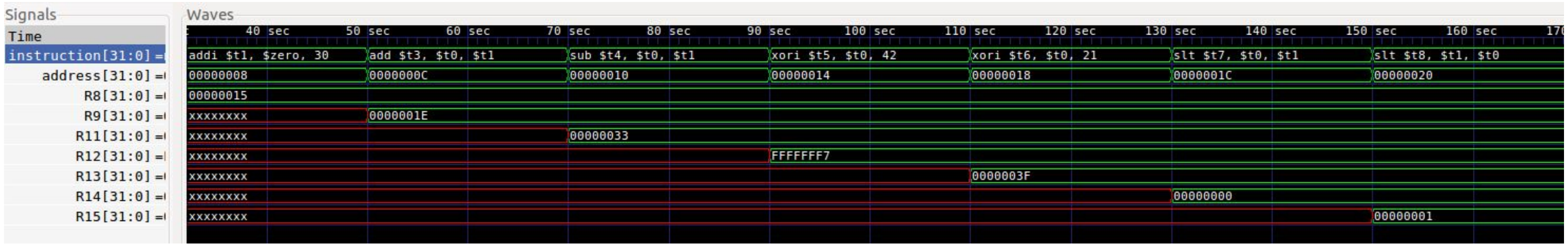

Signals
Waves
Time
instruction[31:0] =
address[31:0] =
R8[31:0] =
R9[31:0] =
R11[31:0] =
R12[31:0] =
R13[31:0] =
R14[31:0] =
R15[31:0] =
40 sec 50 sec 60 sec 70 sec 80 sec 90 sec 100 sec 110 sec 120 sec 130 sec 140 sec 150 sec 160 sec 170
addi $t1, $zero, 30 add $t3, $t0, $t1 sub $t4, $t0, $t1 xori $t5, $t0, 42 xori $t6, $t0, 21 slt $t7, $t0, $t1 slt $t8, $t1, $t0
00000008 0000000C 00000010 00000014 00000018 0000001C 00000020
00000015
xxxxxxxx 0000001E
xxxxxxxx 00000033
xxxxxxxx FFFFFFF7
xxxxxxxx 0000003F
xxxxxxxx 00000000
xxxxxxxx 00000001

Figure 6: Test of final functionalities

Performance Analysis

We are looking into the slowest path of our CPU, as this determined what the fastest clock cycle can be. Due to the way that we implemented our system, every addition and subtraction was done using an ALU (for realism). This means that the slowest path will likely be the one that passes through the most of these computations. After tracing through the paths, we found that the branch instructions (BEQ and BNE) took the longest. Following the path of the signal from the decoder, two registers are subtracted using an ALU. Their result is sent to the PC Case Logic. The branch address is calculated (using a simple concatenation), and then passed through an extra ALU to calculate the next PC. This extra ALU makes this the slowest path.

Overall, there are several ways this design could be optimized. There is really only a need for one ALU, and several adders. Aside from the ALU in the datapath, we create 4 more which are only used for add operations.

In addition, the LUT that we create are not the most efficient. In the PC call module, there is an embedded LUT system. This could be more efficient by expanding the two LUT into one larger one.

Workplan Reflection

Block Diagram: Planned 2 hours
Actual: 2 hours

Verilog submodules: Planned 4 hours
Actual: 2 hours. Because these components were so well planned out, their implementation went fairly quickly. Some unit testing was done, but a lot of the modules couldn't be tested until they were hooked up. A lot of the time that we scheduled into this got pushed to later, when we debugged these modules.

Integrating components: Planned 30 minutes
Actual: 2 hours. Took slightly longer than planned due to mixed syntax.

Test bench and debugging: Planned 3 hours
Actual: 22? Really unclear. As the sessions got longer they grew more unproductive, so that accounts for some of hours. The main problem here was not unit testing the submodules. We definitely could have been more creative and proactive in their testing. We aren't sure how much this actually would have reduced the number of hours spent, just moved them to a different phase of the project.

Assembly test script: Planned 2 hours
Actual: 2 + 1. The "due" assembly script took us about two hours, but we spent another writing miscellaneous tests.

Report: Planned 3 hours
Actual: 2 hours