Computer Architecture Lab 03 : Single Cycle CPU

Yoonyoung Cho, Haozheng Du, Shruti Iyer

November 16 2016

1 Introduction

In this lab, we implemented a single-cycle CPU that supports the fundamental subset of MIPS operations, ranging *LW, SW, J, JR, JAL, BNE, XORI, ADDI, ADD, SUB,* and *SLT*.

2 Processor Architecture

The CPU contains the following blocks: program counter, instruction fetch, instruction decode, arithmetic logic unit, data memory for instructions and values.

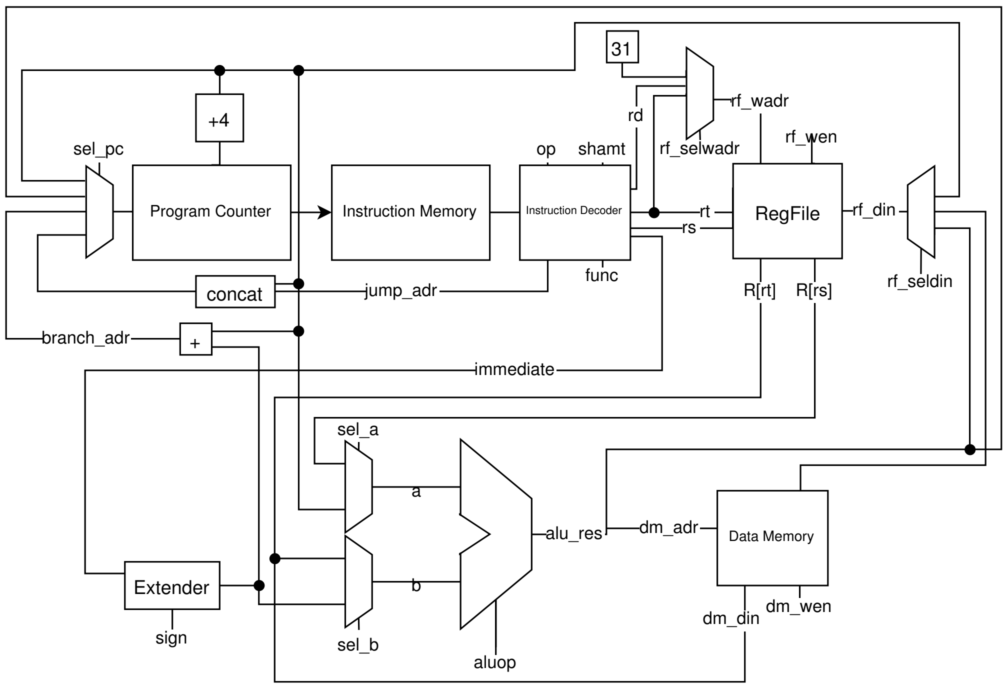

Figure 1: Cap

2.1 Control Signals

For each of the ten instructions, we determined the values of each control signal by hand. In the CPU program, these control signals are set depending on the value of the the opcode or, in the case of R-type instruction, funct.

Table 1: Control Signals for the CPU

Instruction	***sel_pc***	***sgn***	***sel_a***	***sel_b***	***aluop***	***dm_wen***	***rf_wen***	***rf_selwadr***	***rf_seldin***
LW	00	1	0	1	ADD	0	1	00	01
SW	00	1	0	1	ADD	1	0	x	x
J	10	x	x	x	x	0	0	x	x
JR	01	x	x	x	x	0	0	x	x
JAL	10	x	x	x	x	0	1	01	00
BNE	11	1	1	0	ADD	0	0	x	x
ADDI	00	1	0	1	ADD	0	1	00	10
XORI	00	0	0	1	XOR	0	1	00	10
ADD	00	x	0	0	ADD	0	1	10	10
SUB	00	x	0	0	SUB	0	1	10	10
SLT	00	x	0	0	SLT	0	1	10	10

Here are the control signals and their description:

- sel_pc: Selects between the regular PC increment (00), address stored in register (01), address calculated from the immediate (10), jump address (11)
- sgn: Flag to the extender that determines whether or not to extend the input with consideration for its MSB.
- sel_a: This signal controls the multiplexer into the first operand of the alu; this is only high when the instruction is BNE.
- sel_b: Selects between Reg[Rt] and sign-extended immediate.
- sel_aluop: This signal determines the alu's operation.
- dm_wen: The value is high whenever the data is being written to the data memory
- rf_wen: The The value is high whenever the data is being written to the register
- rf_selwadr: The value of this signal dictates which write-address should be chosen for the register, should it be enabled.
- rf_seldin: Selects the data being written into the register. The possible options are branch address from BNE (00), value from the data memory (01) and ALU result (10).

3 Test Plan and Results

In order to test the full capacity of the cpu, we developed a code that would establish dependencies to a prior operation such that the program would require every component of the cpu to be functional in order to output a correct result.

```
start:
    xori $sp $zero 0x00ff # initialize stack pointer
    j main

addsubslt:
    add $t2, $t1, $t0  # Register $t2 should hold 7
    sub $t3, $t1, $t0  # Register $t3 should hold 3
    slt $t4, $t3, $t2  # Register $t4 should hold 1
    jr $ra

store:
    sw $t0, 4($sp)     # push $t0
    sw $t1, 0($sp)     # push $t1
    jr $ra
```

```

load:
    lw $t5, 4($sp)      # Put 2 into register $t5
    lw $t6, 0($sp)      # Put 5 into register $t6
    jr $ra

main:
    xori $t0, $zero, 2  # Put 2 into register $t0
    xori $t1, $zero, 5  # Put 5 into register $t1
    bne $t0, $t1, skip
    addi $t1, $t1, 5 % Skip this code; otherwise $t1 == 10
skip:
    jal addsubslt

    addi $sp, $sp, -8  # Allocate space
    jal store
    jal load
    addi $sp, $sp, 8  # Delete space

    j end

end:
     j end

# Instructions used: ADD, SUB, SLT, XORI, J, JAL, JR, BNE, LW, SW
# No .data requirement
# At the end, the registers should have the following value
#   t0  2
#   t1  5
#   t2  7
#   t3  3
#   t4  1
#   t5  2
#   t6  5
```

Listing 1: Assembly Test Code for verifying CPU Operation

The above program jumps to the main function[1] at the start of the program, skipping over functions. Once the program enters the main code, it loads 2 to temporary register $t0 and 5 to $t1. Albeit a redundant step, it also conditionally branches to skip[2] the addition that would otherwise make $t1 hold 10. From there it invokes addsubslt to test the respective behaviors, and returns the execution to the main loop, where it stores data on the stack and retrieves it immediately to a different register. After this, it cleans up the stack and ends the program.

[1]it also initializes the stack pointer, but that is implicit.

[2]in this particular configuration, the program would always "skip" the first *addi*, hence the name of the label.

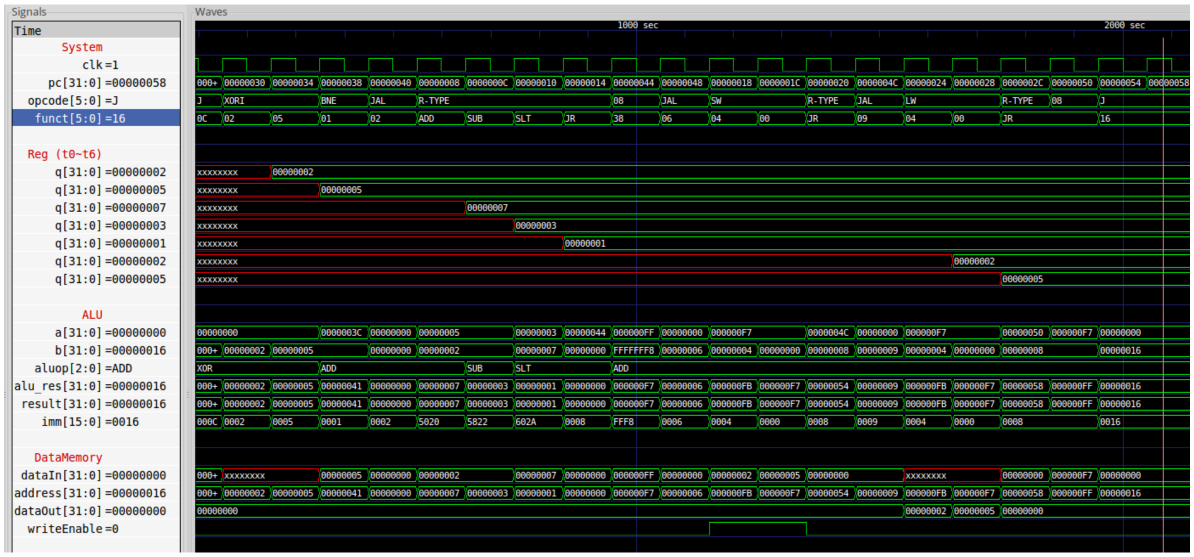

Figure 2: Waveform description that spans the program's execution; of particular note is the value contained in each of the registers, as the program propagates data through its execution.

Here, the full execution flow is described in waveforms. The execution path of the program simulated in MARS was observed to be congruent to that in verilog; the data flow matches as well, with the final output of:

Table 2: The final state of the seven registers, although they are temporary.

Register	*$t0*	*$t1*	*$t2*	*$t3*	*$t4*	*$t5*	$t6
Value	2	5	7	3	1	2	5

4 Area Analysis and Performance

4.1 Program Counter

Our program counter holds a 32-bit address, we estimated it cost as below:

Component	Unit Cost	Number Used	Total Cost
D-Flip-Flop	13	32	416

4.2 32-bit Adder

Component	Unit Cost	Number Used	Total Cost
1-Bit Adder	6	32	192

4.3 Instruction Memory

Our instruction memory holds a register array size at $32 * 2^1 0 = 32768$. The estimated cost shows below:

Component	Unit Cost	Number Used	Total Cost
DFF with Enable	20	32768	655360
1024-1 MUX	7161	32	229152

Thus, the total cost of instruction memory is 884512.

4.4 Instruction Decoder

Instruction decoder simply decodes the instruction through wires, requires no gate.

4.5 Registers

Component	Unit Cost	Number Used	Total Cost
DFF with Enable	20	1024	20480
MUX	217	32	6944

The total cost is 27424.

4.6 ALU

We implemented behavioral verilog for the ALU. our estimated cost is 3760.

4.7 Data Memory

Our data memory module is similar to the instruction memory module:

Component	Unit Cost	Number Used	Total Cost
DFF with Enable	20	32768	655360
1024-1 MUX	7161	32	229152

Total cost: 884512

4.8 Sign Extender

The sign extender module checks if the most significant bit is 1 or 0. This module implements an XOR gate, so the cost is 3.

4.9 Controller

The controller module is a look up table using 16 2-1 muxes, so the total cost is $16 * 7 = 112$.

4.10 Additions

We also implemented 2 2-1 Muxes and 3 3-1 Muxes. The total cost of these additional components is: $2 * 7 + 3 * 14 = 56$

4.11 Total Cost

From all the unit costs calculated above, our CPU total cost is: $416 + 192 + 884512 + 27424 + 3760 + 884512 + 3 + 112 + 56 = 1,800,987 GIE$

4.12 Performance

5 Work Plan Reflection

We spent around 12 hours working on the lab. We started by designing the CPU and drawing out different blocks. We also went through each instruction and determined values of each control signal. This process took two hours. We then built the data memory, sign extend , instruction decoder blocks all of which were done under an hour. We reused the ALU and Regfile from previous work in the class. Testing all the components took around 3 hours; we had to edit the previously written tests for Regfile and ALU to output "Done" or "xyz Test Failed" instead of printing actual and expected outputs. We spent another one hour building the parametrized mux and jump address blocks. Integrating all the sub-components together and debugging the CPU took 3 hours. Although we had a working single-cycle CPU by Nov 18, the team couldn't meet to finish the writeup. It was hard to communicate to all the team members because they weren't at Olin. We spent another 2 hours on Nov 20 trying to finish different parts of the writeup. Overall, although we had some challenges in finishing the writeup, the team worked smoothly and efficiently on designing and building the CPU.