Lab 3 Report

by Changjun Lim and Sungwoo Park

<Processor Architecture and Block Diagram>

We designed single-cycle CPU with the limited set of MIPS instruction (LW, SW, J, JR, JAL, BNE, XORI, ADDI, ADD, SUB, SLT), utilizing CPU components (ALU, adder, multiplexer, decoder, register) that we designed previously. We designed our CPU such that we direct the inputs and outputs into and out of appropriate components of the CPU for different operations using control signals and multiplexers.

For example, take a look at how the program counter behaves when *jal* instruction is given compared to when other instructions are given. Notice that in *jal* RTL, which is reproduced below, the program counter value is incremented by 8 and this value is saved into register 31.

R[31] = PC + 8
PC = JumpAddr

During all other instructions, the only value that is ever added to the program counter is 4. This difference in behaviors, which result in different inputs to the adder (more specifically to our implementation, ALU), is handled by the use of control signal *AdderValControl* and multiplexer.

Program counter incrementation for *jal* instruction is just one of many other diverging behaviors that our CPU has to take care of. Other control signals that take care of other instruction specific behaviors are shown below with their short descriptions.

RegWrEn: Enable writing to the register
MemWrEn: Enable writing to the data memory
PCSel: Determine how next program counter value will be determined
AdderValControl: Determine whether 4 or 8 will be added to current PC value.
RegDataWrSel: Determine which data will be written to the register
RegAddrWrSel: Determine which address the register will write the given data
BranchControl: Determine whether CPU will execute branch-specific behaviors
ALUImm: Determine whether operation is dealing with “immediate” value

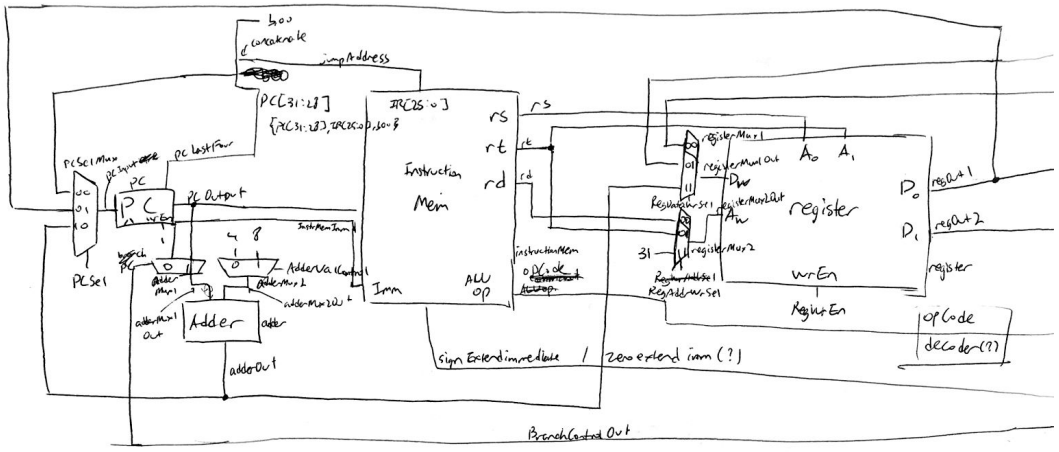
Those control signals are instruction-specific, thus determined by the instruction memory. In our implementation of the CPU, the instruction memory will output the appropriate control signal for the particular given instruction, along with necessary information for the instruction itself, such as the register addresses and immediate values.

Following table summarizes the control signals for each instructions that our CPU implements.

	RegWrEn	*MemWrEn*	*PCSel*	*RegDataWrSel*	*RegAddrWrSel*	*BranchControl*	*ALUImm*
LW	1	0	10	01	01	1	1
SW	0	1	10	01	01	1	1
J	0	0	00	01	01	1	1
JR	0	0	01	01	01	1	1
JAL	1	0	00	11	11	1	1
BNE	0	0	10	01	01	0	1
XORI	1	0	10	00	01	1	1
ADDI	1	0	10	00	01	1	1
ADD	1	0	10	00	00	1	0
SUB	1	0	10	00	00	1	0
SLT	1	0	10	00	00	1	0

The block diagram for our CPU design is included below to aid the understanding of our CPU implementation and architecture.

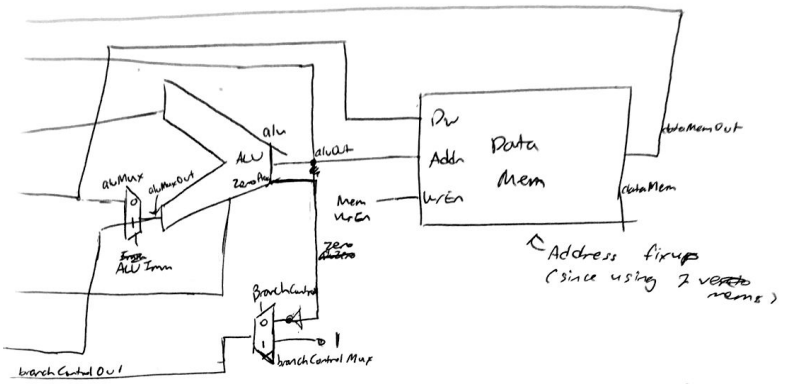
CPU Design (LW, LS, J, JR, JAL)



Control Signals

	RegWrEn	MemWrEn	PCSelect	AdderVal	RegDataWrSel	RegAddrWrSel	BranchCtrl	ALU Imm
LW	1	0	10	0	01	01	1	1
SW	0	1	10	0	01	01	1	1
J	0	0	00	0	01	01	1	1
JR	0	0	01	0	01	01	1	1
JAL	1	0	00	1	11	11	1	1

CPU Design (LW, LS, J, JR, JAL)

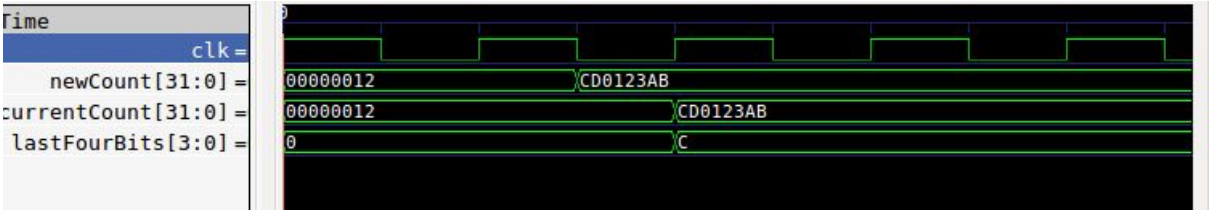


	RegWrEn	MemWrEn	PCSelect	AdderVal	RegDataWrSel	RegAddrWrSel	BranchCtrl	ALU Imm
bne	0	0	10	0	01	01	00	1
xori	1	0	10	0	00	01	1	1
addi	1	0	10	0	00	01	1	1
add	1	0	10	0	00	00	1	0
Sub	1	0	10	0	00	00	1	0
SLT	1	0	10	0	00	00	1	0

<Test Case Strategy & Result>

For each of the components of the CPU, we have individual unit test cases that test the functionality of each component.

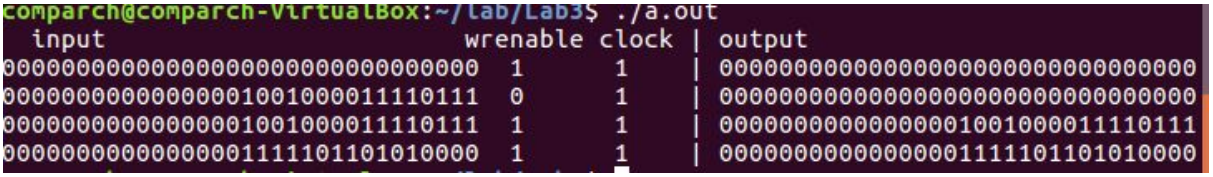
Program Counter



Program Counter test case waveform result

Program counter is essentially a flip-flop that takes in new program counter value to update its value with and outputs its current program counter value and 4 most significant bits. You can see in the above waveform that *currentCount* value and *lastFourBits* value is updated correctly after receiving new program counter value.

Register



1-bit register test case result

We can confirm that our single-bit register is working properly as it correctly saves the given value only when *wrenable* flag is set.

We have automated testing script that checks whether 32-bit register made out of single-bit registers are working properly. Running this file (*regfile.t.v)*, confirms that our 32-bit register used in the CPU is working properly.

ALU

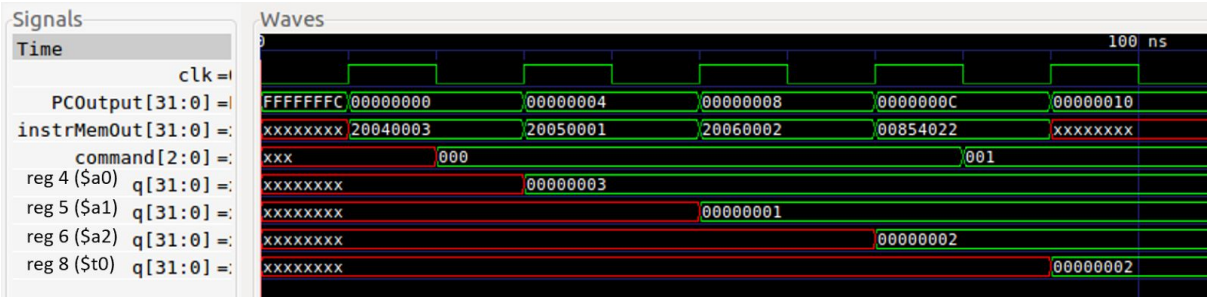
```
testing ADD
operandA operandB command |                result                carryout zero overflow| expected outputs
00000002 00000001 0       | 00000000000000000000000000000011    0     0   0       | 00000000000000000000000000000011  0  0  0
ffffffff ffffffff 0       | 11111111111111111111111111111110    1     0   0       | 11111111111111111111111111111110  1  0  0
00000000 00000000 0       | 00000000000000000000000000000000    0     1   0       | 00000000000000000000000000000000  0  1  0
7fffffff 00000001 0       | 10000000000000000000000000000000    0     0   1       | 10000000000000000000000000000000  0  0  1
testing SUB
00000003 00000001 1       | 00000000000000000000000000000010    1     0   0       | 00000000000000000000000000000010  1  0  0
80000000 00000001 1       | 01111111111111111111111111111111    1     0   1       | 01111111111111111111111111111111  1  0  1
00000000 00000000 1       | 00000000000000000000000000000000    1     1   0       | 00000000000000000000000000000000  1  1  0
ffffffff ffffffff 1       | 00000000000000000000000000000000    1     1   0       | 00000000000000000000000000000000  1  1  0
testing XOR
aa550055 aaff55aa 2       | 00000000101010100101010111111111    1     0   1       | 00000000101010100101010111111111  0  0  0
ffff0000 00ff00ff 2       | 11111111000000000000000011111111    1     0   0       | 11111111000000000000000011111111  0  0  0
testing SLT
555555aa 55aa55aa 3       | 00000000000000000000000000000001    0     0   0       | 00000000000000000000000000000001  0  0  0
555555aa 555555aa 3       | 00000000000000000000000000000000    1     1   0       | 00000000000000000000000000000000  0  0  0
00ff00ff ff00ff00 3       | 00000000000000000000000000000000    0     1   0       | 00000000000000000000000000000000  0  0  0
ffffff00 0000ffff 3       | 00000000000000000000000000000001    1     0   0       | 00000000000000000000000000000001  0  0  0
aaaa55aa aa5555aa 3       | 00000000000000000000000000000000    1     1   0       | 00000000000000000000000000000000  0  0  0
ff55ff00 ffff5500 3       | 00000000000000000000000000000001    0     0   0       | 00000000000000000000000000000001  0  0  0
testing AND
ffff0000 00ff00ff 4       | 00000000111111110000000000000000    1     0   0       | 00000000111111110000000000000000  0  0  0
ff00aa55 aaaa55aa 4       | 10101010000000000000000000000000    1     0   0       | 10101010000000000000000000000000  0  0  0
testing NAND
ffff0000 00ff00ff 5       | 11111111000000001111111111111111    1     0   0       | 11111111000000001111111111111111  0  0  0
ff00aa55 aaaa55aa 5       | 01010101111111111111111111111111    1     0   0       | 01010101111111111111111111111111  0  0  0
testing NOR
55550055 aaff55aa 6       | 00000000000000001010101000000000    1     0   0       | 00000000000000001010101000000000  0  0  0
ffff0000 00ff00ff 6       | 00000000000000001111111100000000    1     0   0       | 00000000000000001111111100000000  0  0  0
testing OR
55ffaa00 aaaa55aa 7       | 11111111111111111111111110101010    1     0   0       | 11111111111111111111111110101010  0  0  0
ffff0000 00ff00ff 7       | 11111111111111110000000011111111    1     0   0       | 11111111111111110000000011111111  0  0  0
```

Above test outputs show that the values that we get by running arithmetic and logic operations through ALU match the expected values.

For high-level overall CPU testing, our overarching test strategy was to write an assembly program with instructions that our CPU knows how to handle. After running this program, we will compare the saved value in the register with the expected value after the operations, and confirm that the CPU is working as expected.

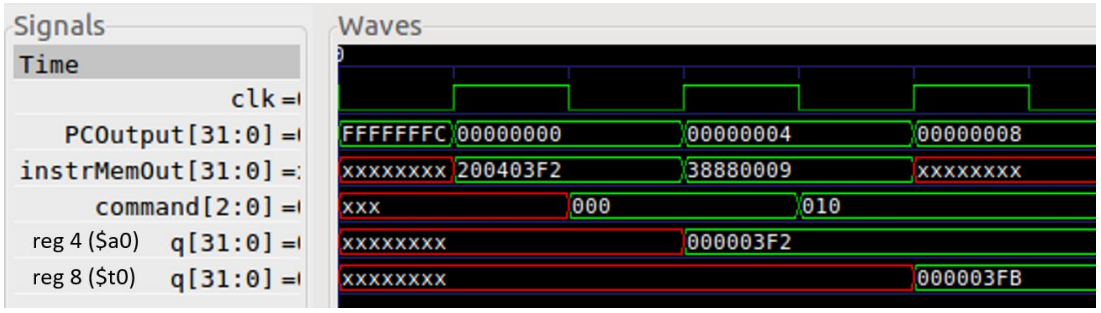
Following are the few examples of the assembly tests that we ran.

SUB Test



Initialize register $a0 with 3 and register $a1 with 1. Apply *SUB* operation with $a0 and $a1 and save the result to $t0. Confirm that the saved value of $t0 at the end of the program is 2. As you can see in the waveform result above, the value of register 8 ($t0) becomes 2 after 4 cycles (4 instructions in the program).

XOR Test

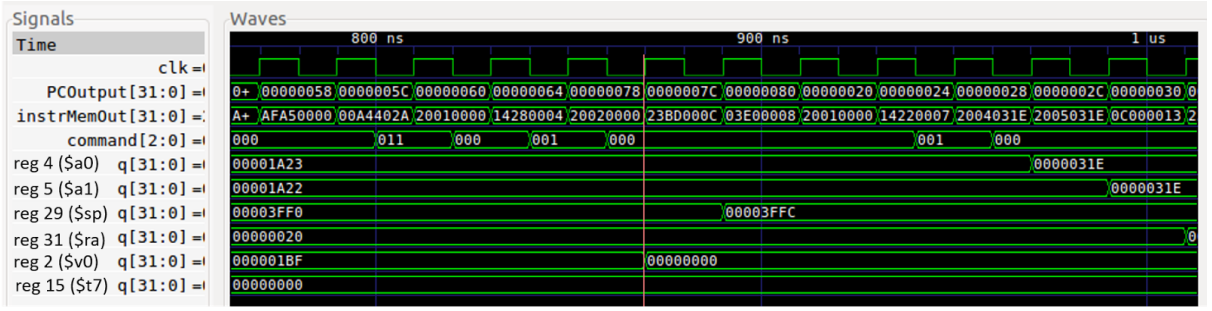


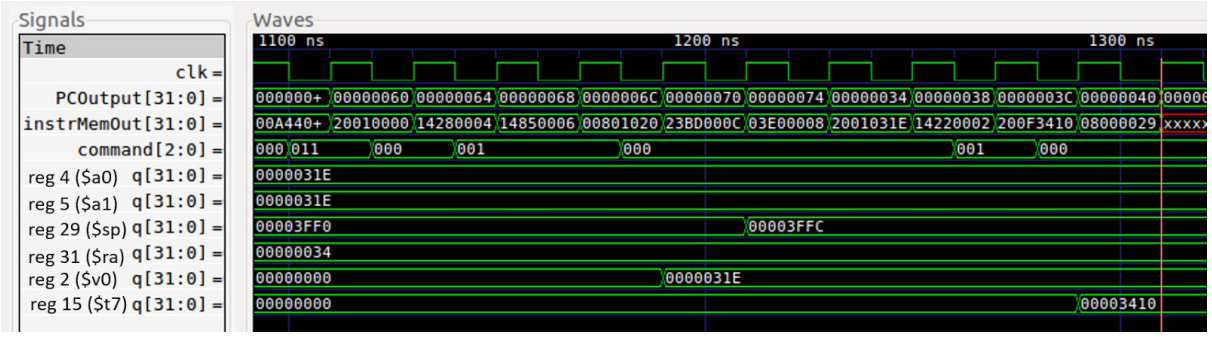
Initialize register $a0 with value d1010. Apply *XORI* operation with $a0 and d0011 and save the result in $t0. Confirm that the saved value of $t0 at the end of the program is h0003FB (which is the result of bitwise xor operation of d1010 and d0011 converted to binary).

Partial Sum Test

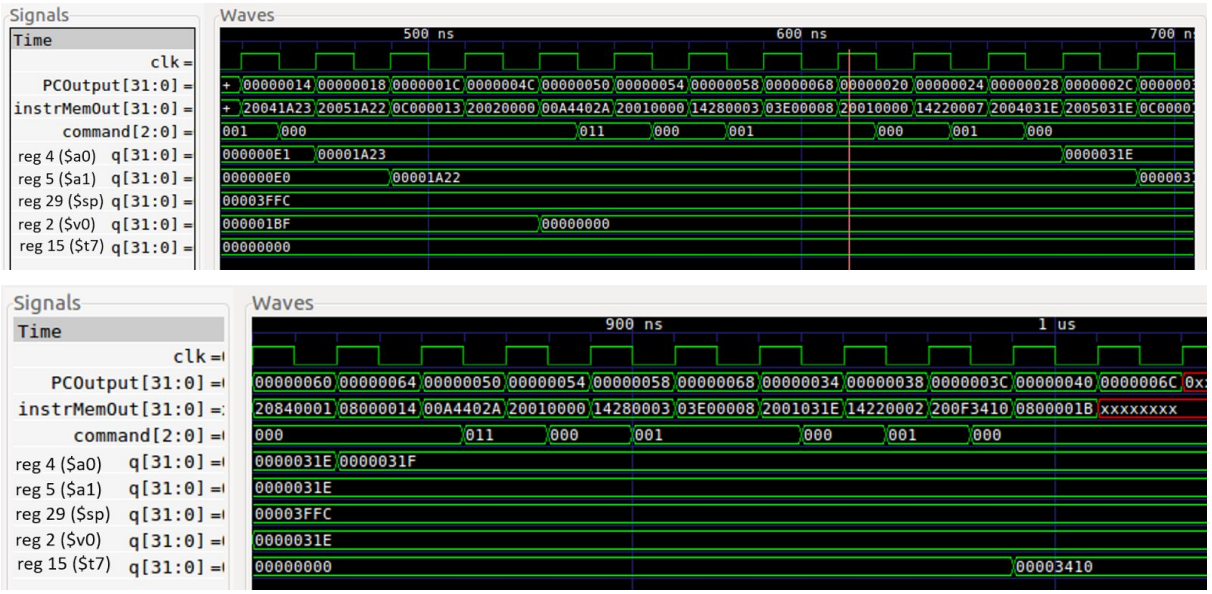
We made the 'partial sum' function which takes two integers and returns the sum of numbers between the two integers(including both). The first parameter should be equal or smaller than the second parameter. If the first parameter is bigger than the second, the function returns 0. We have tried 3 test cases: (i) psum(223,224) = 447 (ii) psum(6691, 6690) = 0 (iii) psum(798, 798) = 798. They are the cases that the first parameter is smaller(i), bigger(ii), or equal(ii) to the second parameter. Finally, if the 3 test cases pass, we set $t7 to have the value of '0x00003410'.

1. **Recursion**





2. **Loop**



In both results, if we see the value of $v0(the second line from the bottom), we can make sure that the function returns the proper values for the 3 test cases. (i) 0x00001BF(=447_{10}), (ii) 0x00000000 (iii) 0x0000031E(=798_{10}). And finally the register $t7 becomes 0x00003410.

<Performance/Area Analysis>

Performance: Our CPU implementation is single-cycle so each instruction will take 1 clock cycle. The propagation delay of the CPU will be the sum of all major components of the CPU (program counter, which is essentially just a register, instruction memory, register, ALU, and data memory). This is the case because all components are more or less connected in series, one after another.

Area: It seems like our single-cycle CPU design is slightly more efficient in terms of space occupying. There is no need for intermediate registers that hold intermediate values of different stages for multi-cycle or pipelined CPU. Also, we don't need extra hardwares that deal with hazards for pipelined CPU. Even though single cycle CPU might be slightly more space efficient, it is significantly less efficient in terms of performance compared to multi-cycle or pipelined CPU.

< Work Plan Reflection >

Scheduled work plan

1. Processor (7.5 hrs) ~11/12
    - Design a single-cycle processor (1 hrs) ~11/9
    - Write or reuse needed modules (0.5 hrs) ~11/9
    - Test & revise every module (1 hrs) ~11/10
    - Implement a single-cycle processor (1 hrs) ~11/11
    - Make a test bench for the processor (2 hrs) ~11/11
    - Test & revise the processor (2 hrs) ~11/12

2. Program (6 hrs) ~11/16
    - Test Fibonacci program & revise the processor (2 hrs) ~11/13
    - Write a test assembly program (1 hr) ~11/14
    - Test programs & revise the processor (3 hrs) ~11/16

Total scheduled time : 13.5 hrs

Actual time spent

1. Processor (5 hrs) ~11/12
    - Design a single-cycle processor (1 hr) ~11/7
    - Write or reuse needed modules (1 hr) ~11/9
    - Test & revise every module (1.5 hrs) ~11/9
    - Implement a single-cycle processor (1 hrs) ~11/12
    - Make a test bench for the processor (0.5 hrs) ~11/12
2. Program (9 hrs) ~11/16
    - Write a test assembly program (2 hrs) ~11/13
    - Test programs & revise the processor (7 hrs) ~11/17
    - Write Report (2 hrs) ~11/17

Total hrs spent: 16 hrs

We had a stretch goal of designing pipelined CPU if we get working single-cycle CPU in a reasonable amount of time, but we ended up spending significantly more time on designing and implementing single-cycle CPU than what we had anticipated. The major source of frustration was implementing the CPU design that we had on paper using Verilog. There were many minor mistakes that we overlooked while we designed on the paper that became apparent when we tried to implemented on Verilog. Even though those issues were minor in terms of complexity, they took quite a lot of time to debug and fix.