# **[CprE 381] Computer Organization and Assembly-Level Programming, Fall 2018**

# **Project A Report**

Name(s) Diego Realpe & Shubham Sharma

## Section /Lab Time 5:10PM – 7:00PM

***Refer to the highlighted language in the Project A instruction for the context of the following questions****.*

1. [Part 1.a] Provide a description of how each of the control values in the architecture (i.e., the outputs of the main control unit) is used.

* For each output control signal, explain how it is used (i.e., what does it do?)
  + o\_reg\_dest

This line controls the multiplexer at the write address port, which chooses the correct bits that are going to be the destination of the data written. Since in R instruction type the bits are at 15-11 and in I type the destination is at bits 20-16. This line chooses the correct bits. Choosing line 0 when it is a type I instr or line 1 when it is a type R.

* + o\_jump

This controls the multiplexer on charge of selecting which kind of control instruction is performed. If the action is a jump that has a long offset, it needs to go through a channel to shift it by 2 and append PC+4. For this one the jump signal is high. However, if it is a normal branch instruction like BEQ with an offset that needs to be added (or subtracted) from PC+4 then it is low.

* + o\_branch

This is a signal specific to the branch instruction, it controls the information that is going to be written in the PC register. If the instruction is a normal ALU or MEM instruction, it will be 0 and only write PC+4 to the register and advance to the next instruction. However, if it is a branch instruction then it is going to be 1 and let the offset be placed in PC (if the branch instruction was true, determined by the zero ALU output that has the result of the comparison that gets && with this signal)

* + o\_mem\_to\_reg

This signal controls where the instruction wants the information from. If it is fetching from memory on something like a LW it will be 1 and open the port that brings data from the MEM. Else if the data it wants comes from the registers, it bypasses the MEM and gets the input it wants from the ALU result and sets it to 0.

* + o\_ALU\_op

It sends the OP Code information to the ALU control so it knows what function to execute in the ALU and if it is a type R and it should take into account the 5-0 function bits or to ignore them if it is a J or I instruction.

* + o\_mem\_write

It is the memory write enable. When the instruction is anything like a SW it will set this signal to 1 so the input in the write data port is enabled to be written where the address port is pointing to. If writing is not necessary it will be 0.

* + o\_ALU\_src

This signal controls if the instruction is one where the immediate offset is used and selects the source of data for the second port of the ALU is going to be information read from a register or info obtained in the immediate part. In other words, controls the difference of the ADD from the ADDI instruction.

* + o\_reg\_write

Similar to the memory write signal, it enables the registers on the file to be modified by the data on the data port at the address given. If writing to the registers is necessary, it will be 1 and if it is only going to read then it will be 0.

* For the 6 MIPS instructions to support, ADD, ADDI, LW, SW, BEQ and J, complete the following table.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ADD | ADDI | LW | SW | BEQ | J |
| o\_reg\_dest | 1 | 0 | 0 | 0 | 0 | 0 |
| o\_jump | 0 | 0 | 0 | 0 | 0 | 1 |
| o\_branch | 0 | 0 | 0 | 0 | 1 | 0 |
| o\_mem\_to\_reg | 0 | 0 | 1 | 0 | 0 | 0 |
| o\_ALU\_op | (0000) | (0000) | (0000) | (0000) | (0001) | (0000) |
| o\_mem\_write | 0 | 0 | 0 | 1 | 0 | 0 |
| o\_ALU\_src | 0 | 1 | 1 | 1 | 0 | 0 |
| o\_reg\_write | 1 | 1 | 1 | 0 | 0 | 0 |

* For each of the 6 MIPS instruction, explain the set values as you described in the table above.
  + ADD
* reg\_dest: R type instruction; RT is located in bits [15-11] => 1
* jump: Not a jump instruction => 0
* branch: Only moving by one instruction => 0
* mem\_to\_reg: Not reading information from memory => 0
* ALU\_op:
* mem\_write: Not writing data to memory => 0
* ALU\_src: ALU second port uses a target register, not imm => 0
* reg\_write: Instruction overwrites a register with add result => 1
  + ADDI
* reg\_dest: I type instruction; RT is selected from bits [20-16] => 0
* jump: Not a jump instruction => 0
* branch: Only moving by one instruction => 0
* mem\_to\_reg: Not reading information from memory => 0
* ALU\_op:
* mem\_write: Not writing data to memory => 0
* ALU\_src: ALU second port uses immediate value with number to be added => 1
* reg\_write: Instruction overwrites a register with add result => 1
  + LW
* reg\_dest: I type instruction; RT is selected from bits [20-16] => 0
* jump: Not a jump instruction => 0
* branch: Only moving by one instruction => 0
* mem\_to\_reg: Reading and loading information on a register => 1
* ALU\_op:
* mem\_write: Not writing data to memory => 0
* ALU\_src: ALU second port uses immediate data containing MEM address to be read from => 1
* reg\_write: Instruction overwrites a destination register with MEM data=> 1
  + SW
* reg\_dest: I type instruction; RT is selected from bits [20-16] => 0
* jump: Not a jump instruction => 0
* branch: Only moving by one instruction => 0
* mem\_to\_reg: Not reading information from memory => 0
* ALU\_op:
* mem\_write: Overwriting memory with stored register value => 1
* ALU\_src: ALU second port uses immediate data with MEM address to be written to => 1
* reg\_write: Only reading from registers and writing to MEM only => 0
  + BEQ
* reg\_dest: I type instruction; RT is selected from bits [20-16] => 0
* jump: Not a jump instruction => 0
* branch: Moving the PC by either 4 or a given offset => 1
* mem\_to\_reg: Not reading information from memory => 0
* ALU\_op:
* mem\_write: Not writing data to memory => 0
* ALU\_src: ALU second port uses immediate data containing offset to be added to PC if true => 1
* reg\_write: Operation only reading and comparing registers => 0
  + J
* reg\_dest: J type instruction, has no register destination => X (0)
* jump: Jump instruction, setting high to regardless move the PC counter to the given value => 1
* branch: Part of processor never used => X (0)
* mem\_to\_reg: Not reading information from memory => 0
* ALU\_op:
* mem\_write: Not writing data to memory => 0
* ALU\_src: Not using RT or immediate values => X (0)
* reg\_write: Only writing to PC => 0

1. [Part 1.a] How is the zero flag from the ALU used?

The zero flag is high when the output from the operation in the ALU resulted in 0. Used by BEQ, If the subtraction of RS and RT registers results on 0 it means they had the same value and were equal. This is AND-ed with the branch signal, and if this one is 1 (meaning the instruction is in fact a branch) it will allow the PC register to be incremented by the calculated offset instead of just 4. This by making the mux with the line containing offset+(PC+4) be sent to the PC register.

1. [Part 2] Simulate your processor in ModelSim and run the provided program in imem.mif with the data in dmem.mif.

* The [projectA > ASM Files > test\_with\_data\_seg.asm] file has the assembly code with data specified which is equivalent to the given imem/dmem.mif. Before simulating the final design, just by studying the code, please explain:

Code in Assembly:

LW $t0 0x0000 $zero # $t0 = 0xA (preloaded memory)

ADDI $t1 $zero 0x0004 # $t1 = 0x4

ADD $s0 $zero $zero # $s0 = 0x0

J 0x0000009 # jump

LW $t2 0x0000 $t1 # $t2 = 0x2 (0x0 offset 4)

ADD $s0 $t2 $s0 # $s0 = 0+2

SW $s0 0x0000 $t1 # $s0 = 0x4

ADDI $t1 $t1 0x0004 # $t1 = 0x8

ADDI $t0 $t0 0xFFFF # $t0 = 0xFFFF

BEQ $t0 $zero 0x0001 # $t0 == 0x0

J 0x0000004 # jump

Translated to C (Approx):

int i = arr[0]; // i = 10

int index = 1;

int totalSum = 0;

for(; i > 0 ; i-- ){

int val = arr[index];

totalSum += val;

arr[index] = totalSum;

++index;

}

* + What the code does

This code iterates through memory as many times as the vaule on address 0x0 tells it to. Along its path it gathers the sum of all the addresses it has gone through so far. Reads the next address, increases the total sum and overwrites said address with the total sum.

* + Which values will be written on DMEM at the locations of the first 11 words staring from the address 0.

The values of the memory addresses at those points will correspond to the addition of all the previous memory values. Address 0 is not written onto, it’s only read as the size of the memory to be iterated through. The detailed values can be seen at the end in the screenshot of the final memory values.

* + Which values will be written on the following registers: $8 (=$t0), $9 (=$t1), $10 (=$t2), and $16 (=$s0).

Referencing the translated code above the registers are used as following:

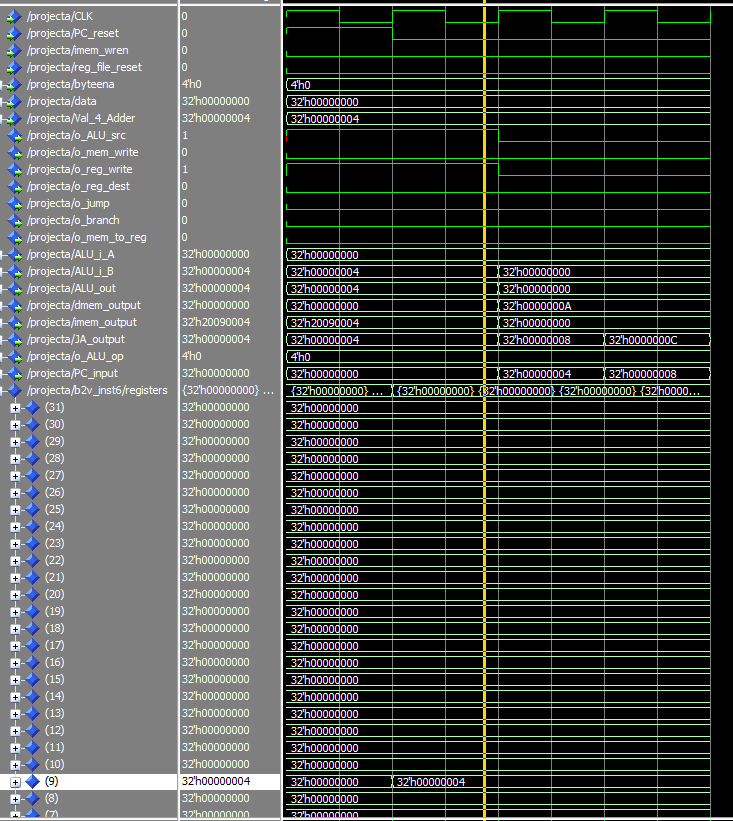
$t0 = i //The amount of iterations that the loop will do

$t1 = index //Where the memory is going to be read and wrote at

$t2 = val //Storing the read value from memory

$s0 = totalSum //Has the total sum of all the memory it has iterated through

* Provide a screenshot for each of the 6 MIPS instructions to support showing the correct functionality and explain how the instruction is working in accordance with the screenshot
  + **ADDI**

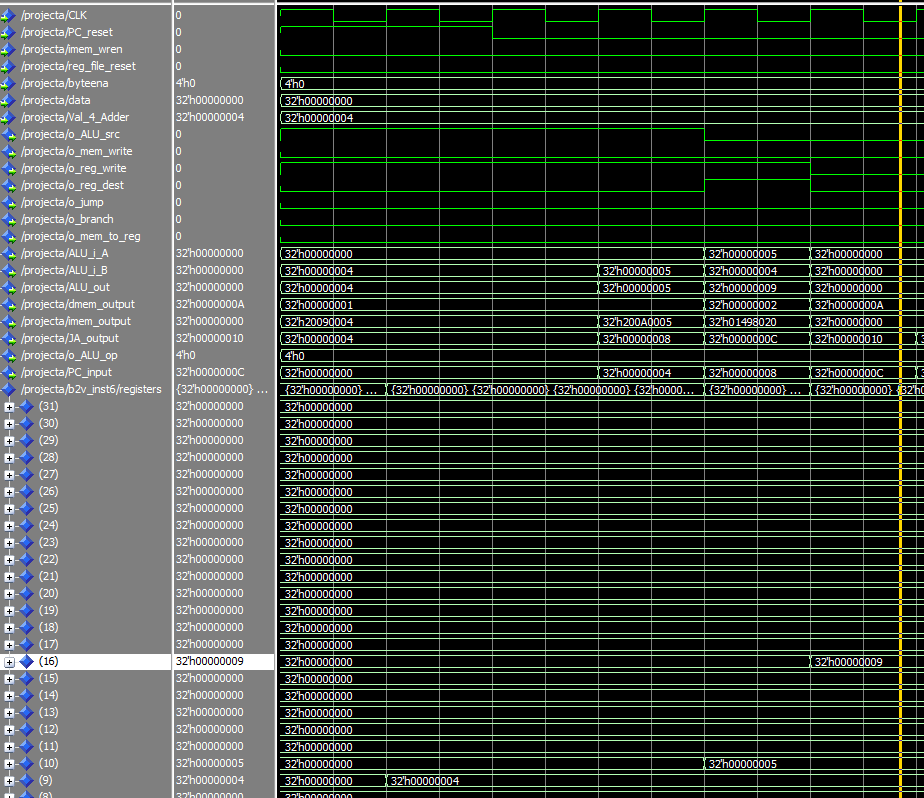


The screenshot above shows the simulation for the instructions:

addi $t1, $zero, 0x0004 #$t1=0x4

First we reset the PC, and then we execute the ADDI instruction. Here, we can see the value of the register **(9)** is equal to 0x4. This means the ADDI instruction has been successfully completed.

* + **ADD**

The screenshot above shows the simulation for the instructions: 

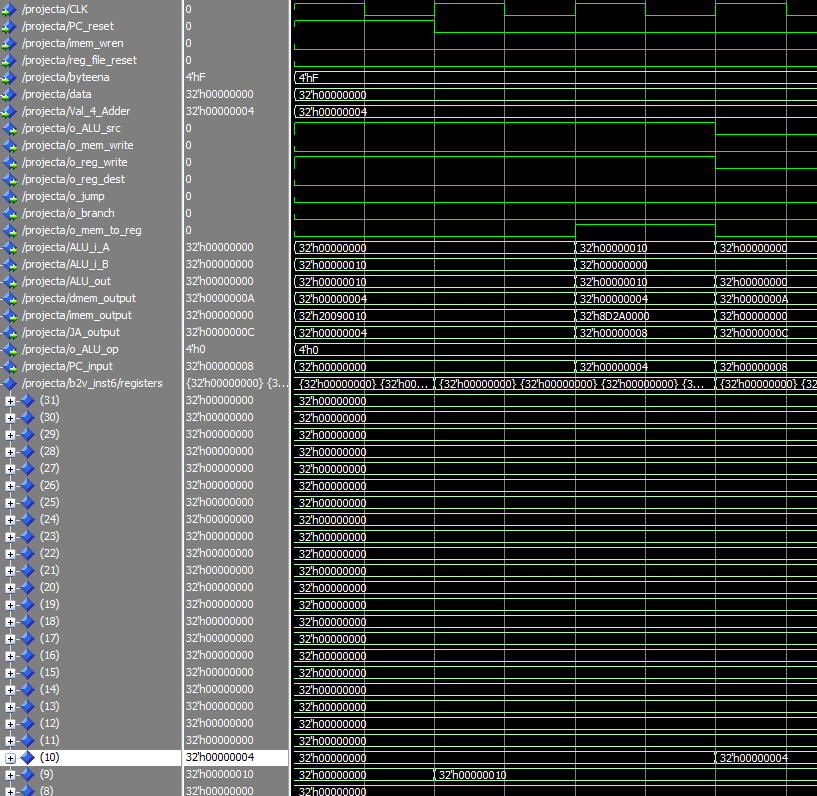
addi $t1, $zero, 0x0004 #$t1=0x4

addi $t2, $zero, 0x0005 #$t2=0x5

add $s0, $t2, $t1 #$s1=0x9

First we reset the PC, and then we execute the ADDI instructions. Which sets the value of reg 9&10 to 0x4 and 0x5 respectively. The result of the ADD instruction is stored in register **(16)** is equal to 0x9. This means the ADD instruction has been successfully completed as 4+5=9.

* + **LW**

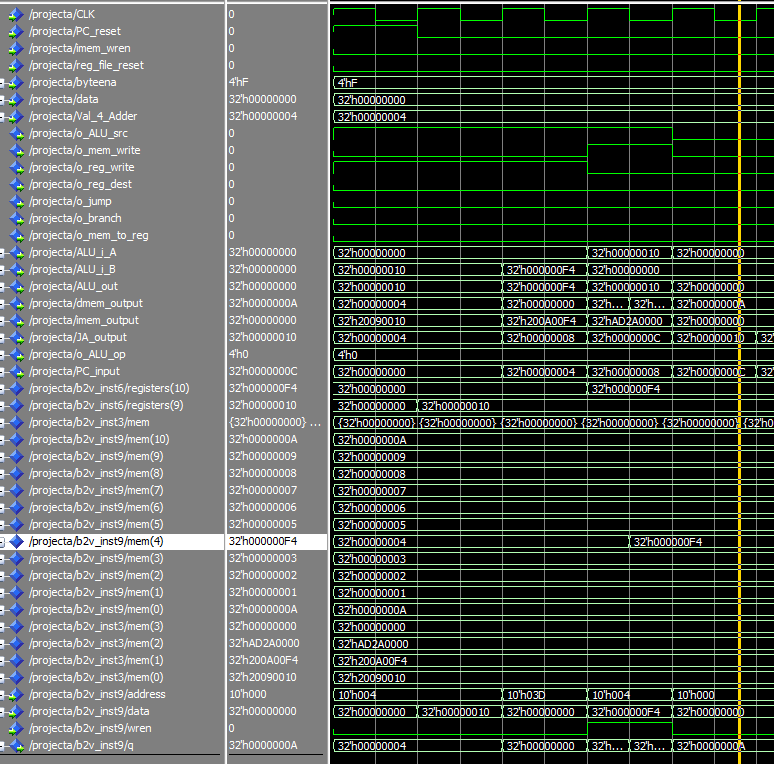
The screenshot above shows the simulation for the instructions: 

addi $t1, $zero, 0x10 #$t1=0x10

lw $t2, 0($t1) #$t2= value at 0x10

Register t1 (9) is loaded with a value of 0x10 which is 16 in decimal. This is because every word is 4 bytes which would mean to access the fourth element in the memory we would multiple the size of each word with the index position, i.e. 4x4=16. Hence, loading the value in the 0x10 address saves 4 to register t2 (10)

* + **SW**

The screenshot above shows the simulation for the instructions: 

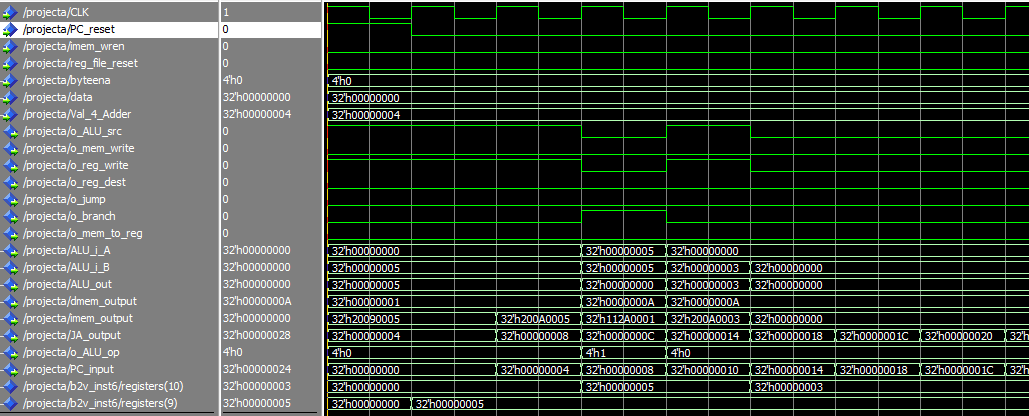
addi $t1, $zero, 0x10 #$t1=0x10

addi $t2, $zero, 0xF4 #$t2=0xF4

sw $t2, 0($t1) #Stores 0xF4 to 0x10

Register t1 (9) is loaded with 0x10 which is 16 in decimal. This is to write to the 4th element in the memory. Register t2 (10) is loaded with the value to store in the 4th element which is 0xF4. From the screenshot we can see a successful store word where the 4th element had the value of 0x04 and is updated to 0xF4

* + **BEQ**



The screenshot above shows the simulation for the instructions:

addi $t1, $zero, 0x5 #$t1=0x5

addi $t2, $zero, 0x5 #$t1=0x5

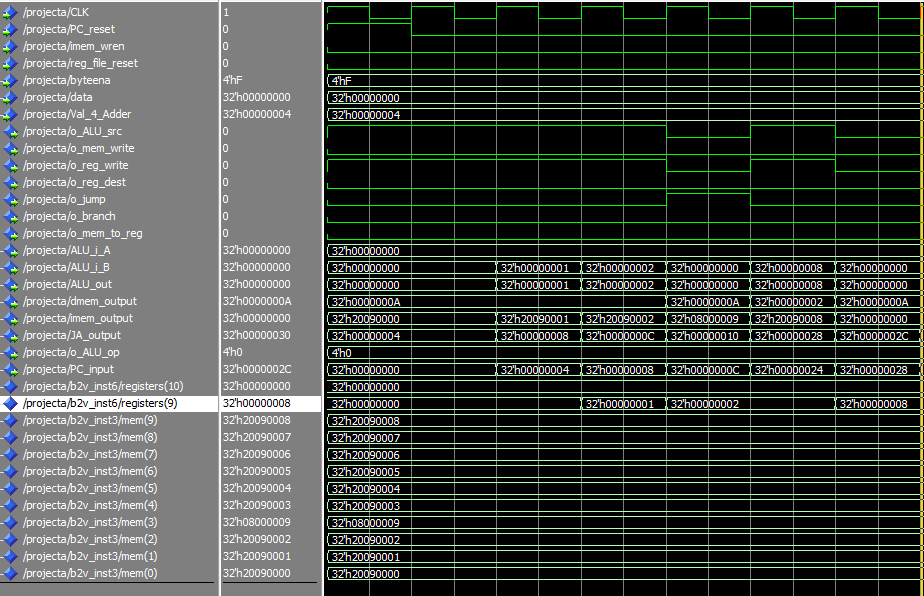
beq $t1, $t2, 0x1 # branch if t1==t2

addi $t1, $zero, 0x3 #$t1=0x3

addi $t2, $zero, 0x3 #$t1=0x3

Register t1 (9) is loaded with the value 0x5 and register t2 (10) is loaded with the value 0x5. Then the branch on equal instruction checks if the value of register t1 (9) is equal to the value of register t2 (10). Since 5=5, the statement after the offset (0x1) is executed, i.e. the statement that assigns t1(9) = 3 is skipped and the next statement is executed. These instructions can been seen in the simulation above.

* + **J**



The screenshot above shows the simulation for the instructions:

addi $t1, $zero, 0x0 #$t1=0x0

addi $t1, $zero, 0x1 #$t1=0x1

addi $t1, $zero, 0x2 #$t1=0x2

J 0x0000009 #Jumps to addr

addi $t1, $zero, 0x3 #$t1=0x3

addi $t1, $zero, 0x4 #$t1=0x4

addi $t1, $zero, 0x5 #$t1=0x5

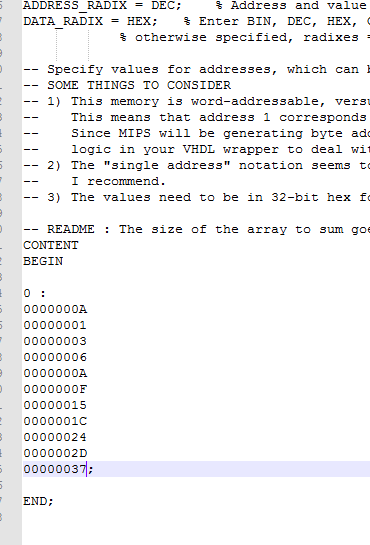
addi $t1, $zero, 0x6 #$t1=0x6

addi $t1, $zero, 0x7 #$t1=0x7

addi $t1, $zero, 0x8 #$t1=0x8

Here we see a code where register t1 (9) is assigned values from 0 to 8 individually on each step. We can see a jump instruction in between. The jump instruction jumps to address 0x9, which is the step where t1 is assigned the value of 8. So the value of t1 assigned are in the order as follows: 0x0, 0x01,0x02, (*jumps*), then 0x08. This behaviour can be seen in the screenshot.

* Provide screenshots showing the final values of the dmem and registers
  + DMEM (for the first 11 words starting from address 0)



* + REGISTERS

