CSE3038 Computer Organization - Spring 2021
Assignment 2 Report

Sameeh Kunbargi 150119693

Alper Dokay 150119746

Ali Reza Ibrahimzada 150119870

Introduction:

In this report, we discuss the implementation of newly added instructions in the single cycle MIPS datapath. In the beginning, we start with giving a big picture of the datapath, and most importantly, we discuss the major changes in the control units.

Datapath & Control Design Changes

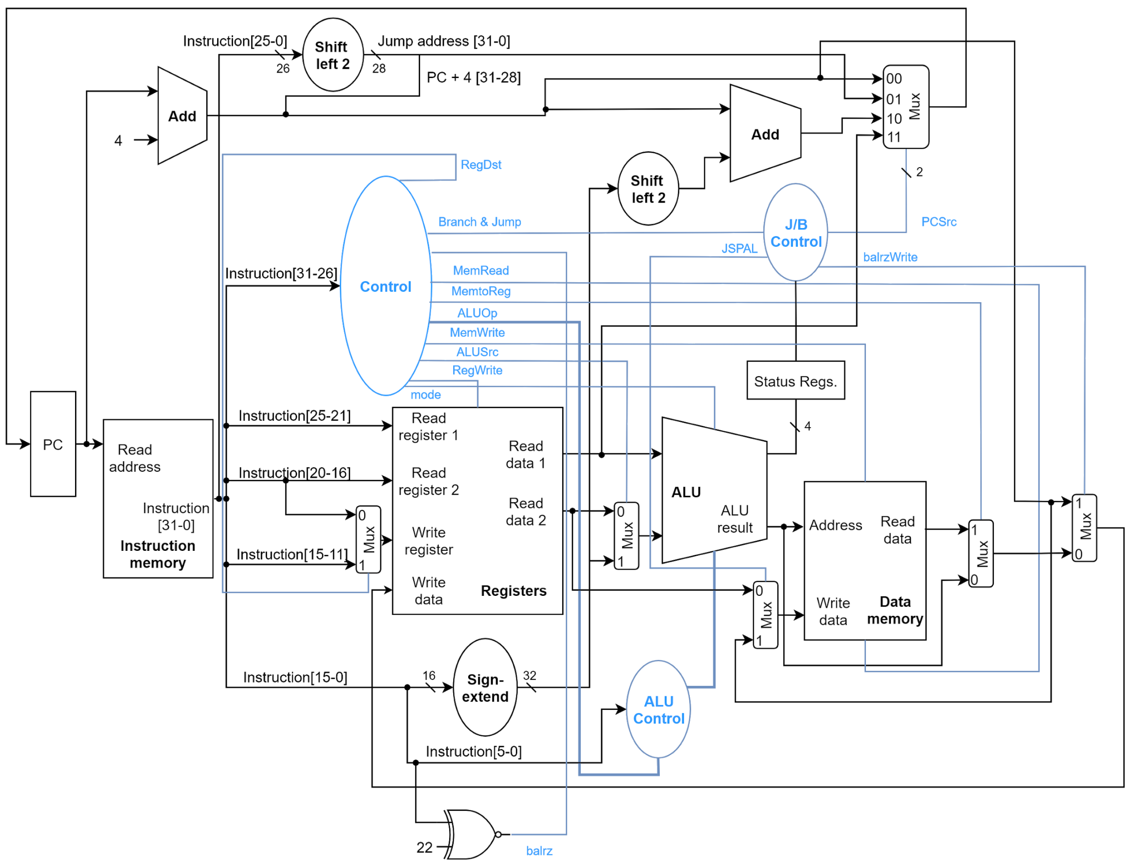

Figure 1: Revised datapath

Changes to Main Control Unit

We updated the Main Control Unit (MCU) to support new control lines like, i.e., branch & jump (3-bits), and mode (1-bit) , while keeping the existing control lines unchanged. We have used the Main Control Unit with some additions. There were some needs to handle the operations assigned to our group. On the other hand, the Main Control Unit now has another input called balrz, which decides whether a given R-format instruction is an existing one or it is balrz. We believe that the current operations will not be affected with the changes/additions we made. Existing control lines will be implemented as they are so as to keep the current operations working fine. The mode control line decides whether an instruction is balrz/bz. The purpose of this control line is to let ALU know when to update the status registers.

Changes to ALU Control Unit

We have extended the ALU Control Unit in order to satisfy some operations like NOR and Right Shift. These operations are added along with the current operations add, and, or, etc. We have integrated ALUOP0 and ALUOP1 for these operations and made the ALU control input 5 bits to easily differentiate the requested operation. The following schema describes the new design of our ALU Control Unit.

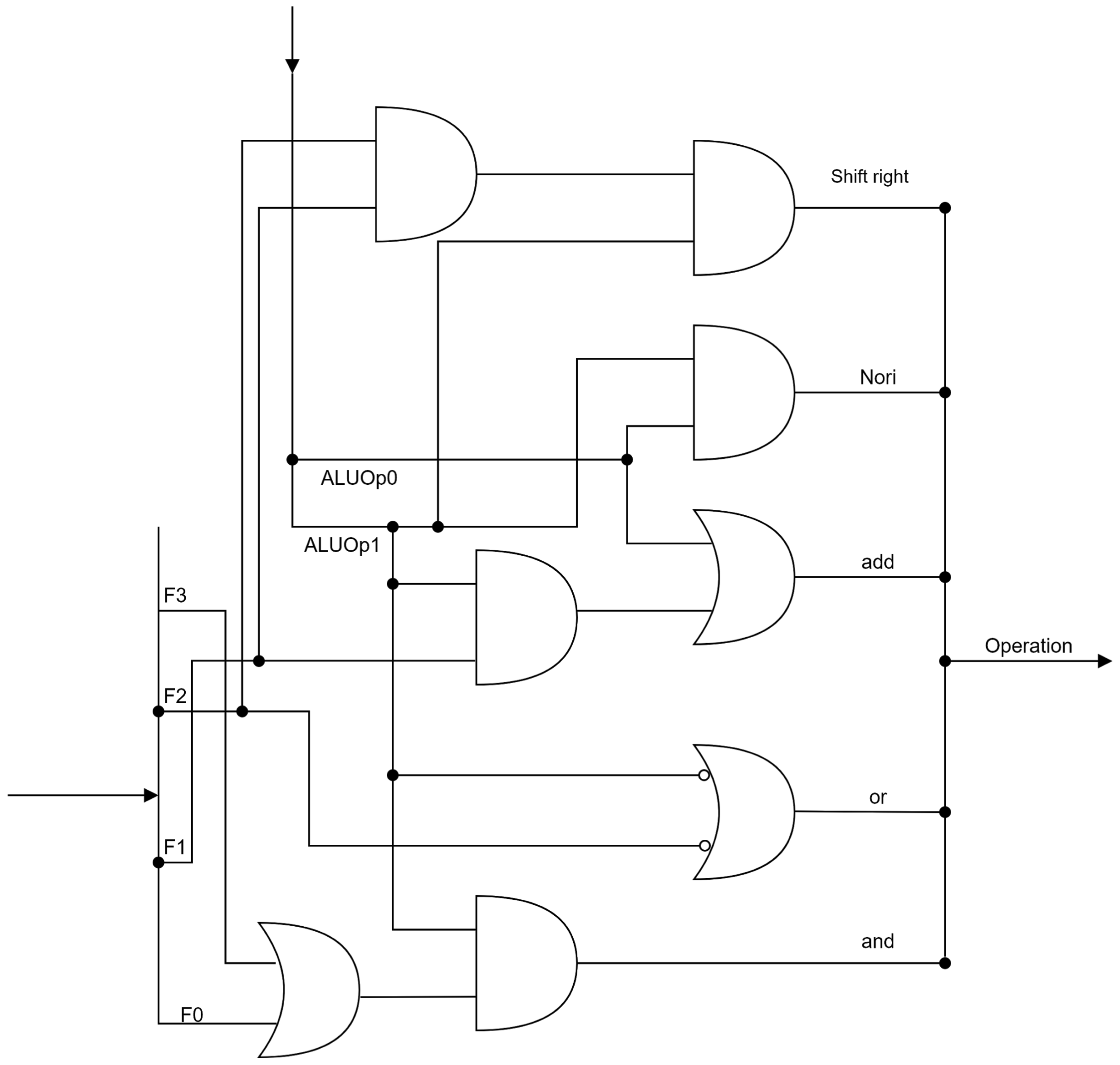

Figure 2: Updated ALU Control Unit

The J/B Control Unit

We have added a new Control Unit and named it as Jump & Branch Control Unit. The purpose of this unit is to manage the decisions of branching and jumping related operations. It takes inputs from the Status register (Z-bit and N-bit) and Branch & Jump from the Main Control Unit. With the given inputs, it provides some outputs to control the operation right after the execution of ALU operation. These outputs are used for the (i) PCSrc in order to control the PC Multiplexer, (ii) BalrzWrite for the Balrz instruction operation and lastly (iii) the JSPAL instruction operation for the target addressing control.

Additional Changes

We have updated some existing multiplexers and added some new ones. We have extended the multiplexer after the branch target addressing to 2 bits so that we can cover all branch & jump operations on this multiplexer only. We have doubled the multiplexer right after the memory block to get back to the Jump & Branch Control Unit to execute the requested operation. This multiplexer is added on behalf of the operation named balrz. If a write operation is needed, the additional multiplexer will take care of the requested execution in the process.

New Implemented instructions:

1. bltz , I-type opcode=1, bltz $rs, Target if R[rs] < 0, branches to PC-relative address (formed as beq & bne do)

   For implementing bltz, we are interested in the value of the negative bit in the status register. In order to determine whether $rs is negative or not, we simply add $rs with $zero and then check the value of the negative bit, and provide it as input to the J/B Control Unit along with Jump & Branch control lines. The J/B Control unit will then decide the value of PCSrc.

2. nori, I-type opcode=13, nori $rt, $rs, imm16, Put the logical NOR of register $rs and the zero- extended immediate into register $rt.

   For implementing nor, we added an extra bit to ALU control input in order to invert the values of both operands at the same time. Moreover, we assigned the ALUOp = 11 to this instruction, so that ALU could differentiate it from other ones.

3. balrz , R-type funct=22, balrz $rs, $rd if Status [Z] = 1, branches to address found in register $rs link address is stored in $rd (which defaults to 31)

   We implemented this instruction by checking the z-bit of the status register. There will be no arithmetic operation for this instruction, but instead we sit the z-bit to either 0/1 from a previous instruction.

4. bz , J-type opcode=24, bz Target, if Status [Z] = 1, branches to pseudo-direct address (formed as j does)

   This instruction's condition is similar to balrz, but the instruction type is totally different. That is, similar to balrz, we sit the value of z-bit from the previous instruction.

5. jspal , I-type opcode=19, jspal, jumps to address found in memory where the memory address is written in register 29 ($sp) and link address is stored in memory (DataMemory[Register[29]]).

   For implementing jspal, we add $sp with $zero and use it as the address to write the value of PC + 4. Since jspal is an unconditional jump instruction, we do not check any status register bits, and instead produce the required control lines from control units.

6. srlv , R-type func=6, srlv $rd, $rt, $rs, shift register $rt to right by the value in register $rs, and store the result in register $rd.

   For implementing shift right operation, we added an extra bit to the operation section of ALU. The reason being shift would be the 5th primitive operation an ALU would do alongside AND, OR, ADD, and SLT.

Control Lines

In this section, we present the control line table which includes the control lines of newly added instructions.

Instructions	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	ALUOp	Jump & Branch	Mode	Jspal	PCSrc	balrzWrite
R-format	1	0	0	1	0	0	10	000	0	0	00	0
lw	0	1	1	1	1	0	00	000	0	0	00	0
sw	x	1	x	0	0	1	00	000	0	0	00	0
Branch equal	x	0	x	0	0	0	01	110	0	0	10	0
Jump	x	x	x	0	0	0	xx	010	0	0	01	0
bz	x	x	x	0	0	0	xx	001	1	0	01	0

bltz	x	0	x	0	0	0	00	101	0	0	10	0
balrz	1	x	x	1	0	0	xx	100	1	0	11	1
jspal	x	0	x	0	0	1	00	011	0	1	11	0
nori	0	1	0	1	0	0	11	000	0	0	00	0
srlv	1	0	0	1	0	0	10	000	0	0	00	0

Simulation Guidelines:

In this section, we shall describe a method of simulating the project. Unfortunately, the provided .dat files were not properly fetched by modelsim. Hence, we decided to create our test cases manually, and then export them for future use. After the simulation starts, the end-user needs to open the memory list section and manually import the provided .mem files into the simulation.

Test Cases:

The simulation can only be done with a set of predetermined instruction memory, register file, and data memory. Please check below for detailed test cases of each instruction.

1) **bltz and jspal:** For test cases, we first start by testing the instructions **bltz $rs, Target** and **jspal**. More specifically, we test bltz instruction considering 3 different situations, namely (i) R[$rs] < 0 (successful branch), (ii) R[$rs] = 0 (unsuccessful branch), and (iii) R[$rs] > 0 (unsuccessful branch). Moreover, since jspal is an unconditional jump, we only show its functionality using a single test case.

bltz_jspal_mem.mem (this file corresponds to instruction memory of bltz and jspal prepared and submitted by our group, and a detailed sequence of executed instructions have been explained below)

0x00000000	**0x00000000**	
0x00000004	**0x05000003**	**# bltz $t0, 0x0003 (failure because $t0=0)**
0x00000008	**0x05200003**	**# bltz $t1, 0x0003 (failure because $t1>0)**
0x0000000c	**0x05400003**	**# bltz $t2, 0x0003 (success because $t2<0)**
0x00000010	xxxxxxxxxxx	
0x00000014	xxxxxxxxxxx	**# jspal makes this address as pc**
0x00000018	xxxxxxxxxxx	
0x0000001c	**0x4FA00000**	**# jspal (successful bltz makes this as pc)**

bltz_jspal_reg.mem (**this file corresponds to the register file of bltz and jspal prepared and submitted by our group. Please note that only registers which hold actual data have been shown below**)

0x00000000	**0x00000000**	
…		
0x00000008	**0x00000000**	**# represents register $t0**
0x00000009	**0x00000001**	**# represents register $t1**
0x0000000a	**0x80000001**	**# represents register $t2**
…		
0x0000001d	**0x00000014**	
0x0000001e	**xxxxxxxxxxx**	
0x0000001f	**xxxxxxxxxxx**	

bltz_datmem.mem (**this file has been excluded for bltz instruction because it does not read the data memory**)

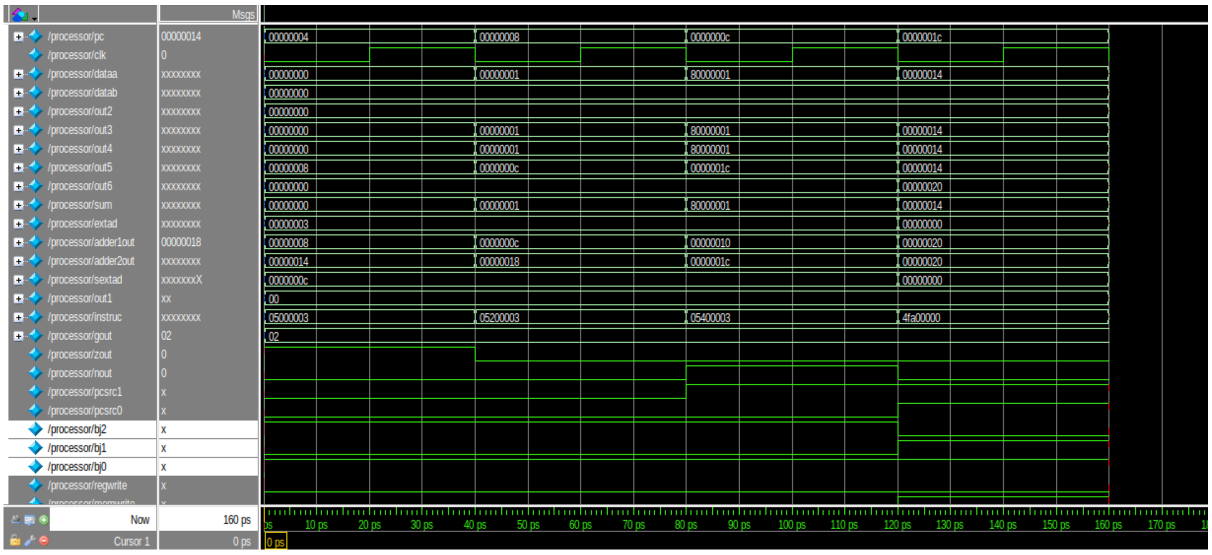

Figure 3: Sample simulation of bltz and jspal based on the above .mem files

2) **nori and srlv:** Next we test the **nori $rt, $rs, imm16** and **srlv $rd, $rt, $rs** instructions.For these instructions, we don't really have multiple cases to consider because it is not a branch one. Thus, we are only testing these instructions by giving some simple values to registers.

nori_srlv_mem.mem (**this file corresponds to instruction memory of nori and srlv prepared and submitted by our group, and a detailed sequence of executed instructions have been explained below**)

0x00000000	**0x00000000**	
0x00000004	**0x352869D1**	**# nori $t0, $t1, 0x69D1**
0x00000008	**0x01685006**	**# srlv $t2, $t0, $t3**
0x0000000c	xxxxxxxxxxx	
0x00000010	xxxxxxxxxxx	

0x00000014	xxxxxxxxxxx
0x00000018	xxxxxxxxxxx
0x0000001c	xxxxxxxxxxx

nori_srlv_reg.mem (**this file corresponds to the register file of nori and srlv prepared and submitted by our group. Please note that only registers which hold actual data have been shown below**)

0x00000000	**0x00000000**	
…		
0x00000008	xxxxxxxxxxx	**# represents register $t0**
0x00000009	**0xFFFFF0F0**	**# represents register $t1**
0x0000000a	xxxxxxxxxxx	**# represents register $t2**
0x0000000b	**0x00000004**	**# represents register $t3**
…		
0x0000001f	**xxxxxxxxxxx**	

nori_srlv_datmem.mem (**this file has been excluded for nori and srlv instructions because they do not read the data memory**)

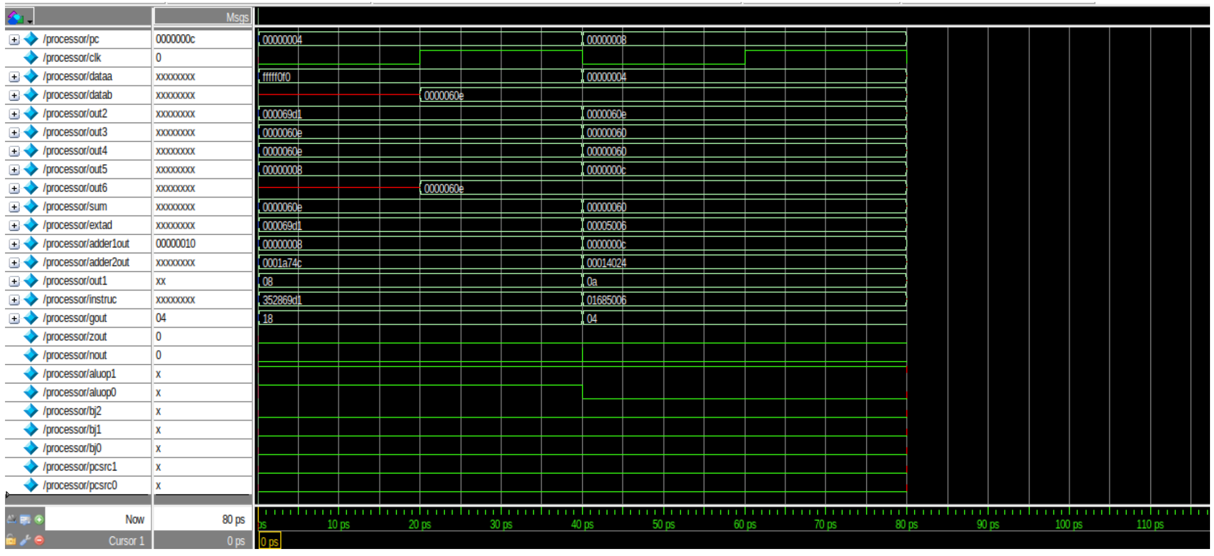

Figure 4: Sample simulation of nori and srlv based on the above .mem files

3) **bz and balrz:** Next we test the **bz Target** and **balrz $rs, $rd** instructions. For these instructions, we check the cases where the Z-bit from the status register has values of either 0 or 1.

bz_balrz_mem.mem (**this file corresponds to instruction memory of bz and balrz prepared and submitted by our group, and a detailed sequence of executed instructions have been explained below**)

0x00000000	**0x00000000**	
0x00000004	**0x012A4022**	**# sub $t0, $t1, $t2**
0x00000008	**0x60000005**	**# bz 0x000005**

0x0000000c	**0x014B4022**	**# sub $t0, $t2, $t3**
0x000000010	**0x60000006**	**# bz 0x000006**
0x00000014	xxxxxxxxxxx	
0x00000018	**0x0180F816**	**# balrz $t4, $ra**
0x0000001c	xxxxxxxxxxx	

bz_balrz_reg.mem (**this file corresponds to the register file of bz and balrz prepared and submitted by our group. Please note that only registers which hold actual data have been shown below**)

0x00000000	**0x00000000**	
…		
0x00000008	xxxxxxxxxxx	**# represents register $t0**
0x00000009	**0x00000001**	**# represents register $t1**
0x0000000a	**0x00000003**	**# represents register $t2**
0x0000000b	**0x00000003**	**# represents register $t3**
0x0000000c	**0x00000014**	**# represents register $t4**
…		
0x0000001f	xxxxxxxxxxx	

bz_balrz_datmem.mem (**this file has been excluded for bz and balrz instructions because they do not read the data memory**)

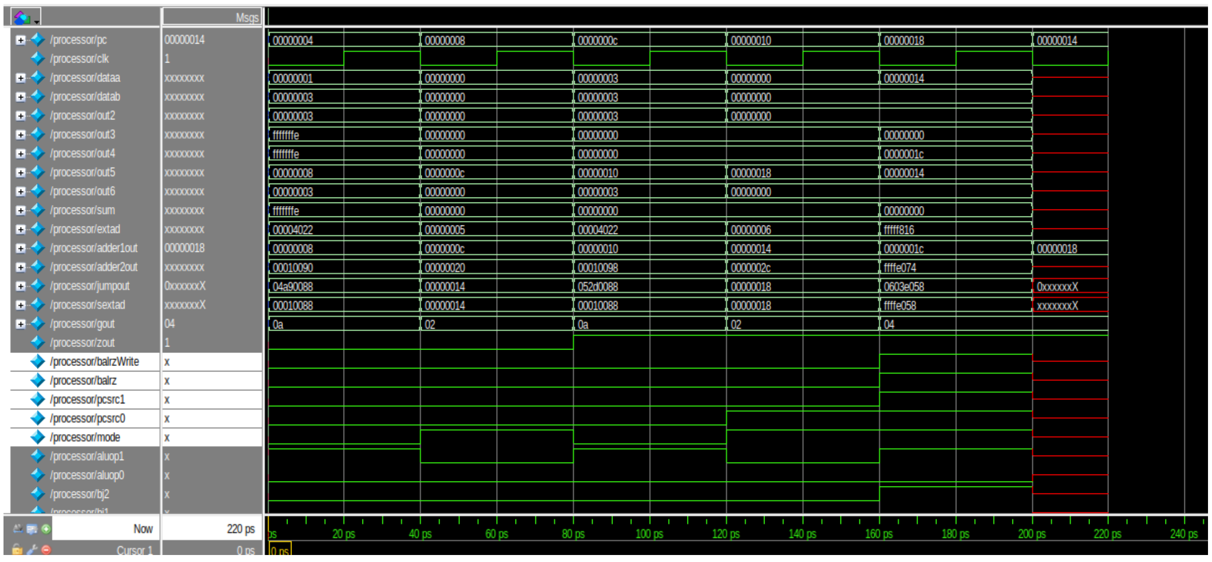

Figure 4: Sample simulation of bz and balrz based on the above .mem files

4) **Existing instructions:** In this section we show the functionality of existing MIPS instructions after changing the single cycle datapath. For simplicity, we test and (skipping other R-format instructions because they are implicitly used in beq, lw, sw, etc.), beq, lw, sw, etc.

existing_mem.mem (**this file corresponds to instruction memory of existing instructions on single cycle datapath prepared and submitted by our group, and a detailed sequence of executed instructions have been explained below**)

0x00000000	**0x00000000**	
0x00000004	**0x012A4024**	**# and $t0, $t1, $t2**
0x00000008	**0x8D6C0004**	**# lw $t4, 4($t3)**
0x0000000c	**0xAD680000**	**# sw $t0, 0($t3)**
0x00000010	**0x118D0002**	**# beq $t4, $t5, 0x0002**
0x00000014	xxxxxxxxxxx	
0x00000018	xxxxxxxxxxx	
0x0000001c	**0x11880003**	**# beq $t4, $t0, 0x0003**

existing_reg.mem (**this file corresponds to the register file of existing instructions on single cycle datapath prepared and submitted by our group. Please note that only registers which hold actual data have been shown below**)

0x00000000	**0x00000000**	
…		
0x00000008	xxxxxxxxxxx	**# represents register $t0**
0x00000009	**0x000015AB**	**# represents register $t1**
0x0000000a	**0x0006C891**	**# represents register $t2**
0x0000000b	**0x00000008**	**# represents register $t3**
0x0000000c	xxxxxxxxxxx	**# represents register $t4**
0x0000000d	**0xFFFFFFFF**	**# represents register $t5**
…		
0x0000001f	xxxxxxxxxxx	

existing_datmem.mem (**this file corresponds to data memory of existing instructions on single cycle datapath prepared and submitted by our group**)

0x00000000	**0x00000000**
0x00000004	xxxxxxxxxxx
0x00000008	xxxxxxxxxxx
0x0000000c	**0xFFFFFFFF**
0x00000010	xxxxxxxxxxx
0x00000014	xxxxxxxxxxx
0x00000018	xxxxxxxxxxx
0x0000001c	xxxxxxxxxxx

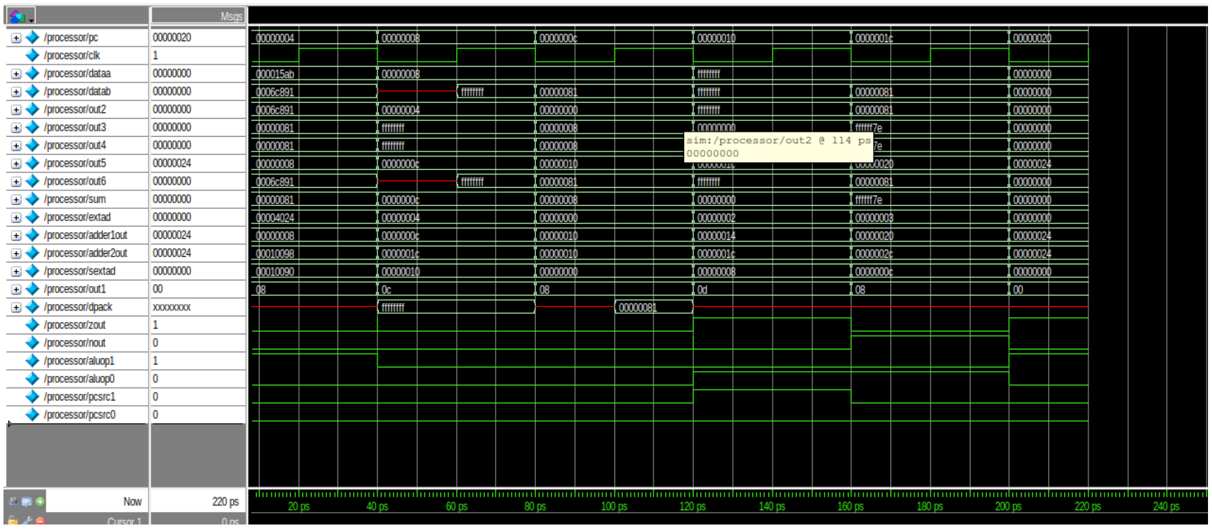

Figure 5: Sample simulation of existing instructions based on the above .mem files