Lab 06
Jonathan Gusler
2/26/2019

Overview

This lab involved creating the hardware necessary to implement two new instructions for the MIPS hardware simulation used previously in Labs 3, 4 and 5. On top of creating new hardware, the new hardware had to be integrated with some of the current components. The new instructions were successfully added to the hardware.

Background

This lab used the same foundational hardware from Labs 3 [1], 4 [2]and 5 [3]. This lab requires familiarity with VHDL, MIPS assembly language and machine language, strong understanding of the basic processor simulated by the VHDL in the previous labs, particularly its data and control paths, and understanding of the function of the assembly language instructions included in MIPS, J (Jump) and BNE (Branch on not equal). Details on the assembly language instructions involved can be found at [4].

Discussion

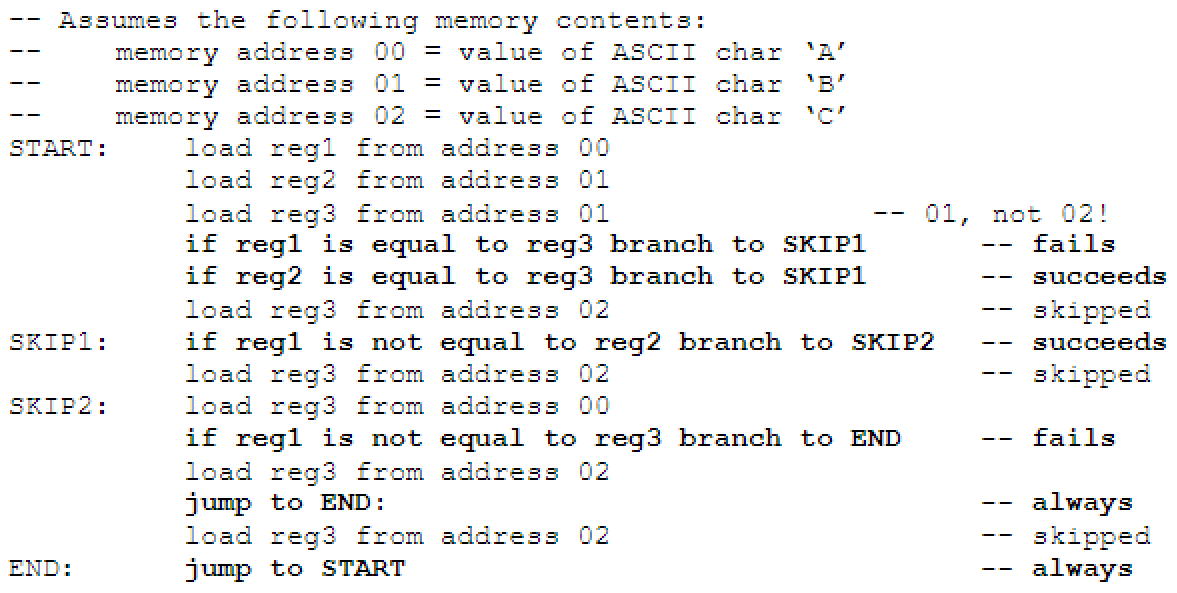
The hardware from Lab 05 only had one type of instruction that could result in the next instruction not being PC + 4, Branch on equal (Branch, BEQ). This lab added two new instructions to that list: Jump and Branch on not equal. Branch on not equal, as the name implies, branches to the specified PC value when the two values being compared are NOT equal. Jump goes to the specified PC value unconditionally. As a side note, Jump has a larger range than Branch, in terms of how far away from the current instruction the next instruction to be executed can be.

Adding Jump required adding a new adder to calculate PC + 4 + offset, offset being the relative location of the new PC value from PC + 4, and hardware to choose between PC + 4 and Jump. The offset value is embedded in the Jump instruction value and thus changes do not need to be made to the Decode hardware, only Fetch and Execute. No decision-making hardware was required for Jump because it is unconditional. If the Jump signal is high, choose the Jump value instead of PC + 4 or PC + 4 + branch offset.

BNE, on the other hand, requires modifications to the Decode, as well as Fetch and Execute, hardware because it needs to find out which two registers to compare for inequality. Alterations to the Fetch hardware comprised of changing the decision-making hardware for the next instruction location. Before BNE was included, all the hardware cared about when choosing between PC + 4 and PC + 4 + branch offset was is there a branch possibility and did the ALU raise the Zero flag. With the addition of BNE, the hardware needed to know what type of branch. The altered hardware chose to branch if branch and zero were high or BNE was high and zero was low.

A new program, pseudocode in Figure 1, and memory values, also shown in Figure 1, provided by Dr. Hutson in his lab prompt [5] were used to test the three instructions, Jump, BNE and

Branch. The new program tested both Branch types succeeding and failing to execute and the Jumps executing to show that the updated processor could do the instructions properly.



```
-- Assumes the following memory contents:
--      memory address 00 = value of ASCII char 'A'
--      memory address 01 = value of ASCII char 'B'
--      memory address 02 = value of ASCII char 'C'
START:     load reg1 from address 00
           load reg2 from address 01
           load reg3 from address 01               -- 01, not 02!
           if reg1 is equal to reg3 branch to SKIP1       -- fails
           if reg2 is equal to reg3 branch to SKIP1       -- succeeds
           load reg3 from address 02                      -- skipped
SKIP1:     if reg1 is not equal to reg2 branch to SKIP2   -- succeeds
           load reg3 from address 02                      -- skipped
SKIP2:     load reg3 from address 00
           if reg1 is not equal to reg3 branch to END     -- fails
           load reg3 from address 02
           jump to END:                                   -- always
           load reg3 from address 02                      -- skipped
END:       jump to START                                  -- always
```

Figure 1: Pseudocode for program (bottom) and data memory values (top).

An in-depth explanation of the program:

1) 'A' (0x00000041) loaded into register 1
2) 'B' (0x00000042) loaded into register 2
3) 'B' loaded into register 3
4) register 1 ('A') and register 3 ('B') tested for equality
5) register 1 and register 3 not equal so does not branch
6) register 2 ('B') and register 3 ('B') tested for equality
7) register 2 and register 3 are equal so does branch to SKIP1 and skips loading 'C' (0x00000043) into register 3
8) register 1 ('A') and register 2 ('B') tested for inequality
9) register 1 and register 2 are unequal so does branch to SKIP2 and skips loading 'C' (0x00000043) into register 3
10) 'A' loaded into register 3
11) register 1 ('A') and register 3 ('A') tested for inequality
12) register 1 and register 3 are equal so does not branch to END
13) 'C' loaded into register 3
14) Jump to END
15) 'C' loaded into register 3 skipped because of previous Jump
16) Jump to START

Analysis and Results

The annotated simulation waveforms starting on page 4 show in detail each of the instructions highlighted in bold in Figure 1. These instructions are the branches succeeding and failing and the jumps.

Table 1 describes the relevant signals involved in the simulation.

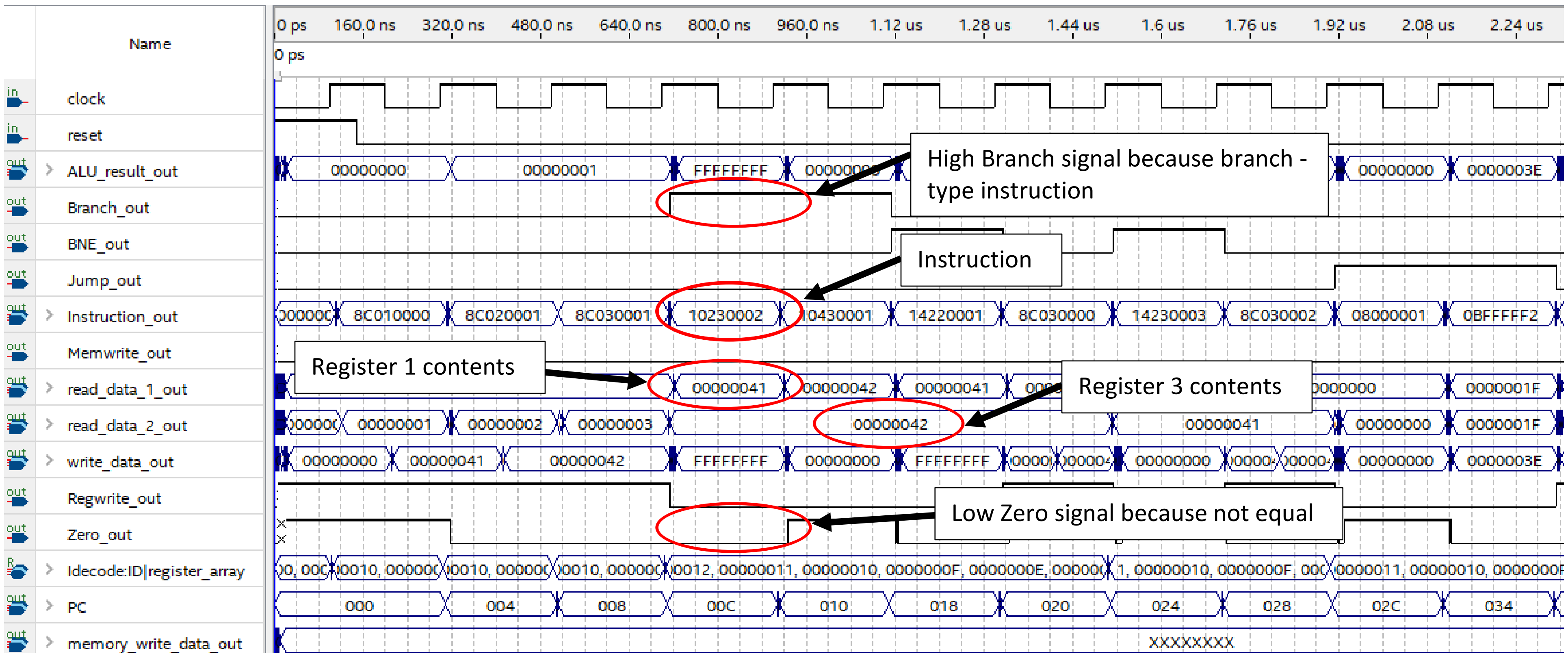
Signal Name	Description
clock	The clock for the system
reset	Resets system back to instruction on falling edge
ALU_result_out	Output of ALU
Branch_out	Branch control signal
Instruction_out	Instruction fetched from IM
Memwrite_out	Enable write to memory on high
read_data_1_out	Rs output from registers
read_data_2_out	Rd output from registers
write_data_out	Data to write to registers on high
Regwrite_out	Enable write to registers on high
Zero_out	Zero signal from ALU used in branch decisions
PC	Program counter

Table 1: Relevant signal names and descriptions for the following analyses.

The memory and program files used by the simulation are shown in Appendix A.

The first bolded instruction is a BEQ instruction comparing registers 1 and 3, with the result being not equal, thus failing.

10230002; -- beq $1,$3,2 ;fails
000100 | 00001 | 00011 | 0000000000000010
Opcode | $s = 1 | $t = 3 | offset = 2 instructions

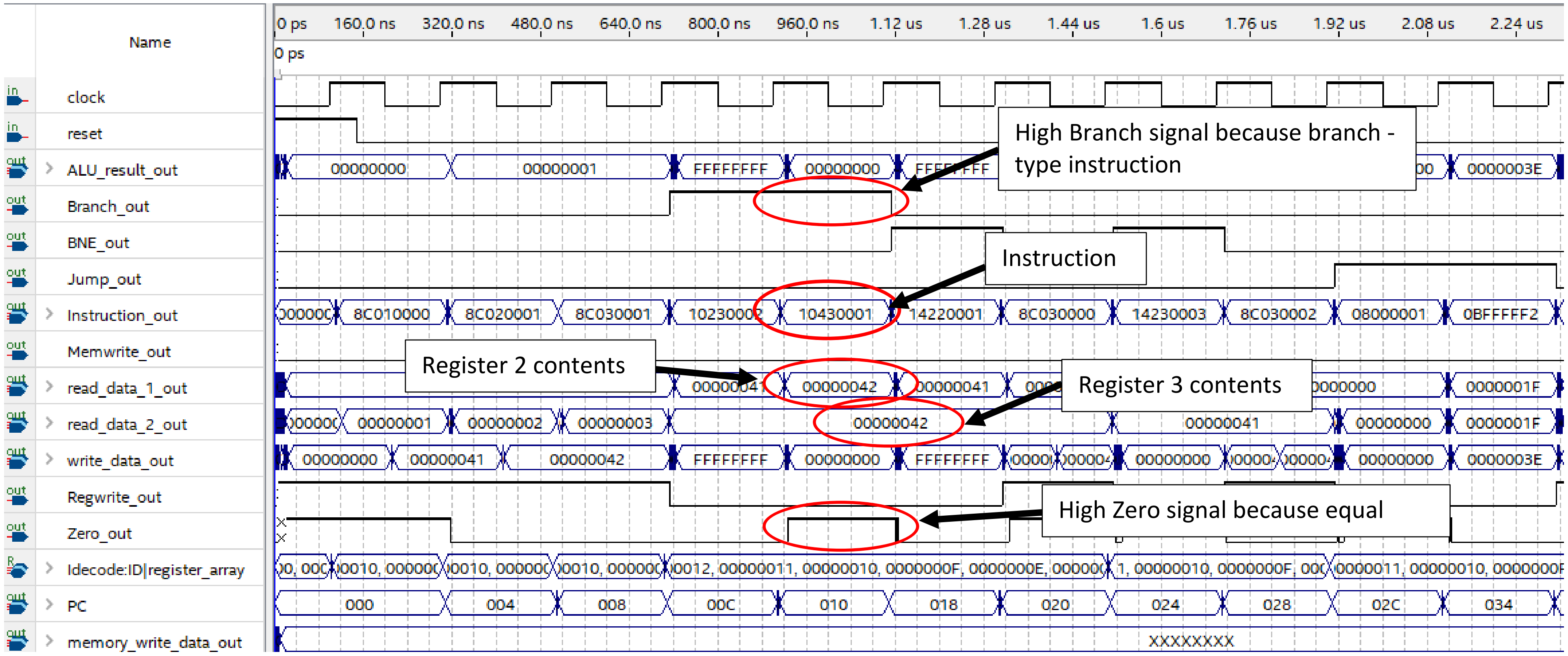


Control Signals Raised
Branch

Control and Data Flow
After the clock is raised for this cycle, the program counter is loaded into the PC register and the instruction at that address is pulled out of program memory and becomes available on the Instruction_out line. Bits 31-26 of Instruction_out are directed to the Control Unit. Bits 25-16 define which register's values to compare in the ALU. Bits 15-0 go to the sign extender to define the possible next instruction address.

The second bolded instruction is a BEQ instruction comparing registers 2 and 3, with the result being equal, thus branching.

10430001; -- beq $2,$3,1 ;succeeds
000100 | 00010 | 00011 | 0000000000000001
Opcode | $s = 2 | $t = 3 | offset = 1 instruction

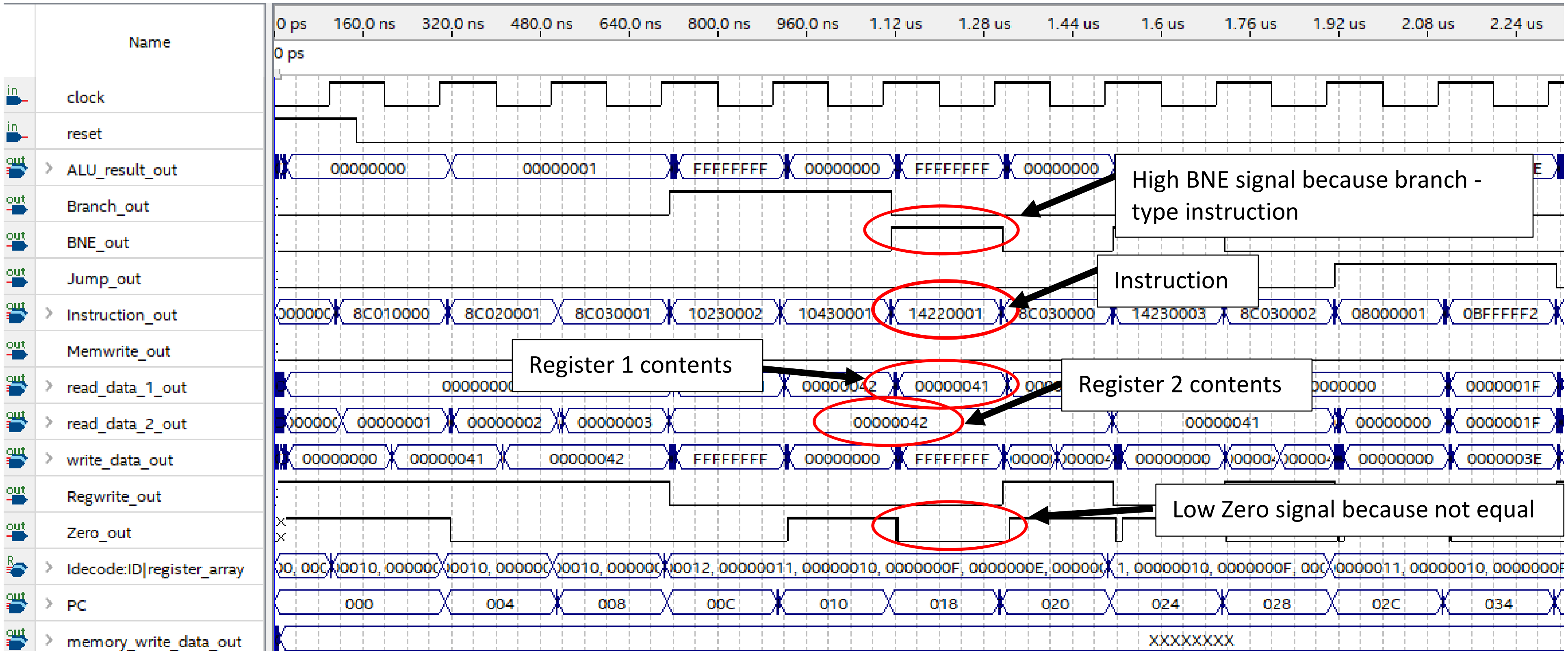


Control Signals Raised
Branch, Zero

Control and Data Flow
After the clock is raised for this cycle, the program counter is loaded into the PC register and the instruction at that address is pulled out of program memory and becomes available on the Instruction_out line. Bits 31-26 of Instruction_out are directed to the Control Unit. Bits 25-16 define which register's values to compare in the ALU. Bits 15-0 go to the sign extender to define the possible next instruction address.

The third bolded instruction is a BNEQ instruction comparing registers 1 and 2, with the result being not equal, thus branching.

14220001; -- bne $1,$2,1 ;succeeds
000101 | 00001 | 00010 | 0000000000000001
Opcode | $s = 1 | $t = 2 | offset = 1 instruction

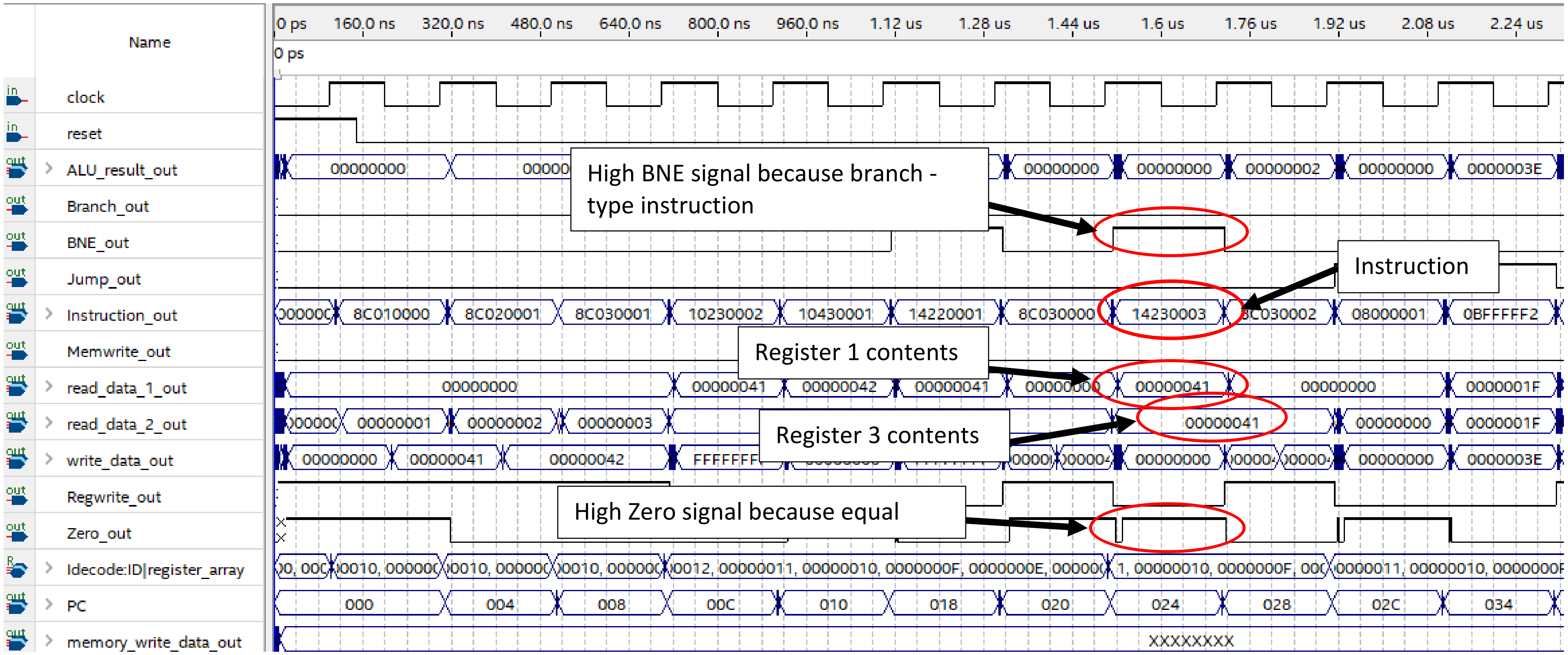


Control Signals Raised
BNE

Control and Data Flow
After the clock is raised for this cycle, the program counter is loaded into the PC register and the instruction at that address is pulled out of program memory and becomes available on the Instruction_out line. Bits 31-26 of Instruction_out are directed to the Control Unit. Bits 25-16 define which register's values to compare in the ALU. Bits 15-0 go to the sign extender to define the possible next instruction address.

The fourth bolded instruction is a BNEQ instruction comparing registers 1 and 3, with the result being equal, thus not branching.

14230003; -- bne $1,$3,3 ;fails
000101 | 00001 | 00011 | 0000000000000011
Opcode | $s = 1 | $t = 3 | offset = 3 instructions

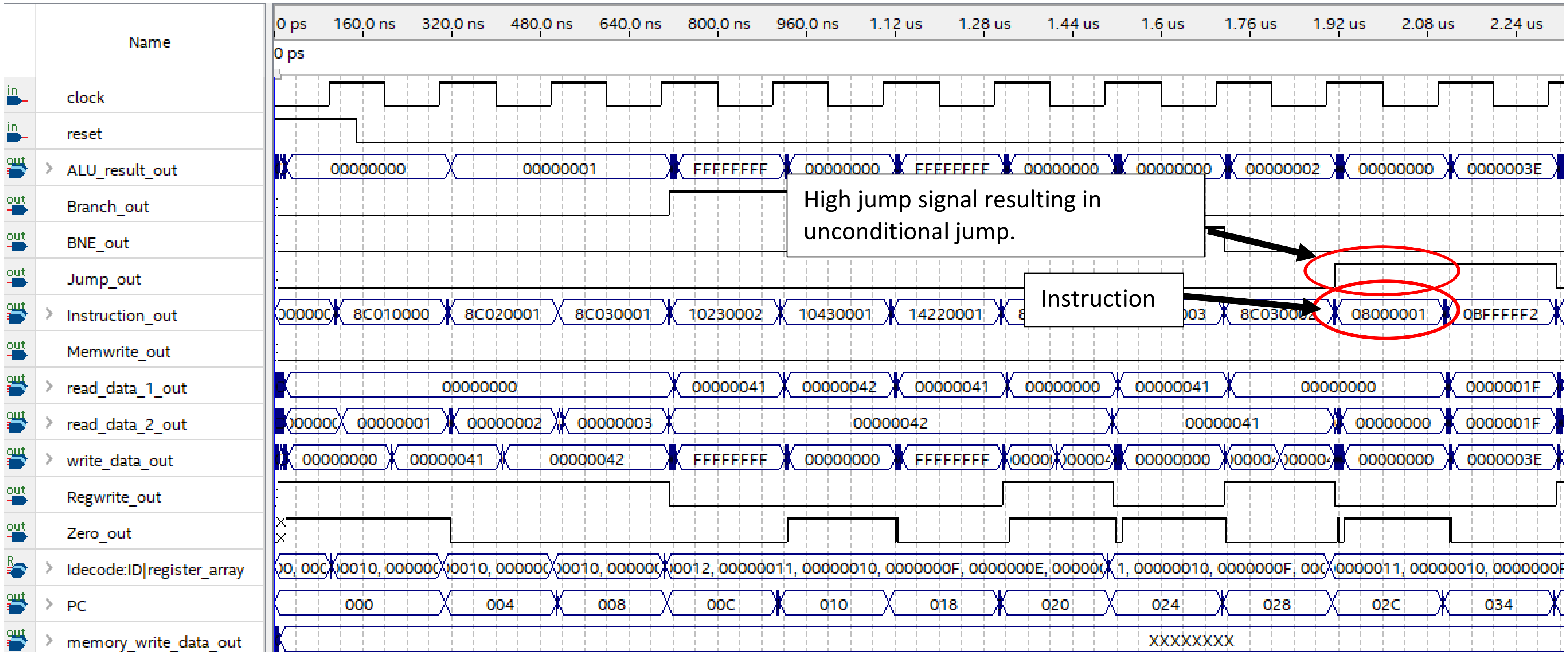


Control Signals Raised
BNE, Zero

Control and Data Flow
After the clock is raised for this cycle, the program counter is loaded into the PC register and the instruction at that address is pulled out of program memory and becomes available on the Instruction_out line. Bits 31-26 of Instruction_out are directed to the Control Unit. Bits 25-16 define which register's values to compare in the ALU. Bits 15-0 go to the sign extender to define the possible next instruction address.

The fifth bolded instruction is a Jump instruction to the last line of the program.

08000001; -- j 1 ;always
000010 | 00000000000000000000000001
Opcode | jump = 1 instruction

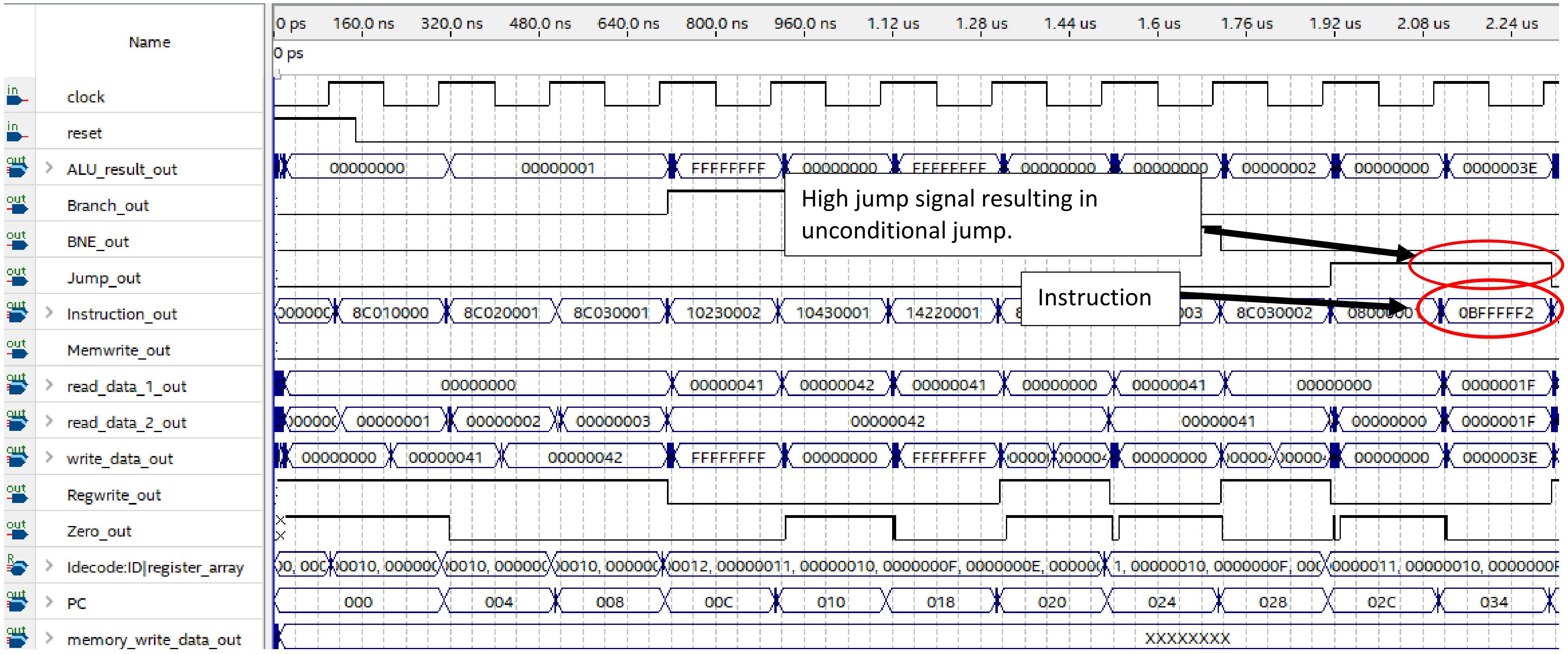


Control Signals Raised
Jump

Control and Data Flow
After the clock is raised for this cycle, the program counter is loaded into the PC register and the instruction at that address is pulled out of program memory and becomes available on the Instruction_out line. Bits 31-26 of Instruction_out are directed to the Control Unit. Bits 25-0 define which how many instructions to jump and in which relative direction.

The fifth bolded instruction is a Jump instruction to the beginning of the program.

0BFFFFF2; -- j -14 ;always
000010 | 11111111111111111111110010
Opcode | jump = back 14 instructions



Control Signals Raised
Jump

Control and Data Flow
After the clock is raised for this cycle, the program counter is loaded into the PC register and the instruction at that address is pulled out of program memory and becomes available on the Instruction_out line. Bits 31-26 of Instruction_out are directed to the Control Unit. Bits 25-0 define which how many instructions to jump and in which relative direction.

Conclusion
As shown in the analysis, the code executed the program proving the successful addition of the two instructions. Further expansion of the instruction set should be a simple task now that it has been done successfully twice.

Works Cited

[1] J. Gusler, "Lab 03," 2019.

[2] J. Gusler, "Lab 04," 2019.

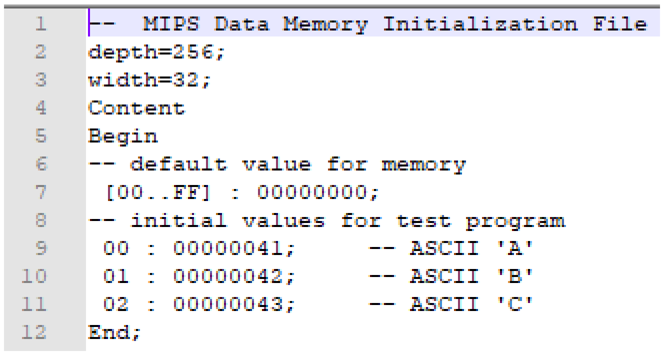
[3] J. Gusler, "Lab 05," 2019.

[4] "MIPS Instruction Reference," [Online]. Available: http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html.

[5] D. J. Hutson. [Online]. Available: https://lipscomb.instructure.com/courses/17375/files/folder/Lab%20Assignments?preview=648478.

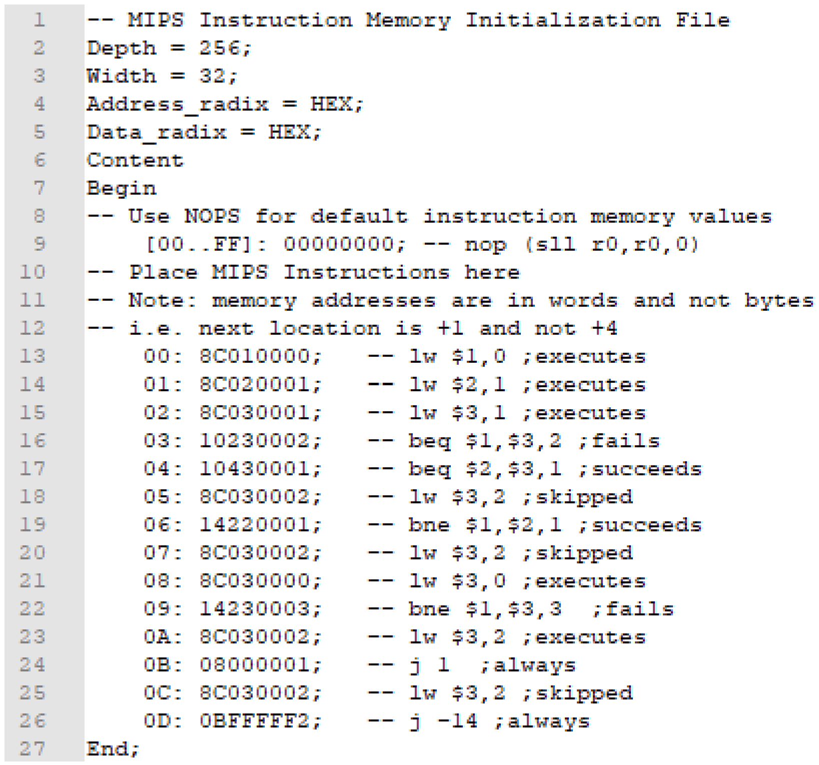
[6] H. F. Hamblen, in *Rapid Prototyping of Digital Systems, SOPC Edition*, p. Section 14.

Appendix A



```
--  MIPS Data Memory Initialization File
depth=256;
width=32;
Content
Begin
-- default value for memory
 [00..FF] : 00000000;
-- initial values for test program
 00 : 00000041;     -- ASCII 'A'
 01 : 00000042;     -- ASCII 'B'
 02 : 00000043;     -- ASCII 'C'
End;
```

Data memory file for simulation.



```
-- MIPS Instruction Memory Initialization File
Depth = 256;
Width = 32;
Address_radix = HEX;
Data_radix = HEX;
Content
Begin
-- Use NOPS for default instruction memory values
    [00..FF]: 00000000; -- nop (sll r0,r0,0)
-- Place MIPS Instructions here
-- Note: memory addresses are in words and not bytes
-- i.e. next location is +1 and not +4
    00: 8C010000;   -- lw $1,0 ;executes
    01: 8C020001;   -- lw $2,1 ;executes
    02: 8C030001;   -- lw $3,1 ;executes
    03: 10230002;   -- beq $1,$3,2 ;fails
    04: 10430001;   -- beq $2,$3,1 ;succeeds
    05: 8C030002;   -- lw $3,2 ;skipped
    06: 14220001;   -- bne $1,$2,1 ;succeeds
    07: 8C030002;   -- lw $3,2 ;skipped
    08: 8C030000;   -- lw $3,0 ;executes
    09: 14230003;   -- bne $1,$3,3  ;fails
    0A: 8C030002;   -- lw $3,2 ;executes
    0B: 08000001;   -- j 1  ;always
    0C: 8C030002;   -- lw $3,2 ;skipped
    0D: 0BFFFFF2;   -- j -14 ;always
End;
```

Program file for simulation.