DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 211 Introduction to Microcomputers, Fall 2019

Lab 8: Supporting Branches in the "Simple RISC Machine"

handin deadline is 9:59 PM the evening before your lab section the week of Nov 4 to 8

REMINDER: *As outlined in the CPEN 211 Lab Academic Integrity Policy, until all students involved have a grade for Lab 8 you must* **NOT** *share or describe any code you write for this assignment with anyone except your* **authorized** *lab partner for Lab 8,* **NOT** *ask for or use any code offered to you by anyone (other than your authorized lab partner) and* **NOT** *look at or use solution code from anywhere. If you are repeating CPEN 211 you may not reuse any of your code submitted in a prior year. Promptly report cases of misconduct you have first-hand knowledge of to the instructor. Your partner is "authorized" to work with you for Lab 8 if https://cpen211.ece.ubc.ca/cwl/lab_partners.php says they are your "current lab partner" at the time you start working together on Lab 8 up until you demo your code. The deadline to sign up or change lab partners using the above URL is 96 hours before your lab section. Your code* **will** *be checked for plagiarism using very effective plagiarism detection tools. As per UBC policy, all suspected cases of academic misconduct must be reported to the APSC Dean's office. Examples of outcomes for misconduct cases at UBC can be found online. See e.g.: https://universitycounsel.ubc.ca/files/2016/07/SD-2014-2015.pdf.*

1 Introduction

This lab completes your Simple RISC Machine by adding two types of branch instructions. When compiling high-level programming languages such as C, C++ or Java, or interpreting a dynamic language such as Python, *conditional* branch instructions are used to implement "for" and "while" loops and "if" and "switch" statements. Function calls are also implemented using a form of branch instruction. If you completed all of Lab 5 to 7, the changes in this lab will make your Simple RISC Machine "Turing complete". This means a program can be written to implement *any* algorithm that can run within the 256 word memory you added in Lab 7. How fast your computer runs depends on your detailed implementation such as how many states you used in your FSM from Lab 6. To make this lab a bit more fun, a small part of your mark for Lab 8 will be based upon how fast your final Simple RISC Machine is compared to your classmates.

If you did not complete Lab 7 you can use someone else's Lab 7 solution as a starting point for this lab, but you should first test their code to make sure it works. You **must** also mention who provided the starting code in your CONTRIBUTIONS file which is mandatory for Lab 8 even if you are working alone. If you are starting early be sure to first watch the online portion of Flipped Lecture #4 on edge.edx.org.

1.1 Branch Instructions

In Lab 7 the program counter (PC) was incremented after each instruction. To support loop and "if" constructs at the assembly level we introduce conditional branch instructions. In the Simple RISC Machine conditional branch instructions either update the PC to *PC+1* or *PC+1+sx(im8)*, where *sx(im8)* is the lower 8-bits of the instruction sign extended to 9-bits. When executing the conditional branch instruction the choice between updating the PC to *PC+1* or *PC+1+sx(im8)* is made by considering the values of Z, N, and V status flags. The `CMP` (compare) instruction from Lab 6 is used to set these flags. Thus, a conditional branch instruction determines which instruction to execute next based on the result of the computation so far. The ability of conditional branches to select which instruction to execute based upon the results of a past computation transforms your Simple RISC Machine into a general-purpose computer.

An example of a simple C program that includes a loop is shown in Figure 1. The corresponding Simple RISC Machine program is shown in Figure 2. The right side of Figure 2 shows the assembly code that defines the program. Rather than typing this out you can simply download it from Piazza (see `lab8fig2.s`). The left side of the figure shows the corresponding memory addresses where each instruction and data element

is placed in memory. Finally, the middle portion shows the contents of memory, in binary. Notice that any given memory location can contain either instructions or data and there is nothing about the memory that distinguishes data from instructions. The program in Figure 2 adds up the values in array "amount" identified by the label `amount:` on line 24 and stores the total in the memory location with address 0x14 identified by the label "result" on line 29. The initial value of "result" (memory address 0x14) is zero, but after the program reaches the `HALT` instruction the contents of memory location 0x14 should contain 850 (`0000001101010010` in binary).

```
int N = 4;
int amount[] = {50,200,100,500};
int result = 0;
int main(void) {
  int i = 0;
  int sum = 0;
  for(i=0; i<N; i++) {
    sum = sum + amount[i];
  }
  result = sum;
}
```

Figure 1: C code corresponding to assembly code in Figure 2

```
1  Address |  Content of memory  |  Assembly code (lab8fig2.s on Piazza)
2  --------+---------------------+-------------------------------------------------------
3  0x00    |  1101000000001111   |      MOV R0,N         // R0 = address of variable N
4  0x01    |  0110000000000000   |      LDR R0,[R0]      // R0 = 4
5  0x02    |  1101000100000000   |      MOV R1,#0        // R1 = 0; R1 is "i"
6  0x03    |  1101001000000000   |      MOV R2,#0        // R2 = 0; R2 is "sum"
7  0x04    |  1101001100010000   |      MOV R3,amount    // R3 = base address of "amount"
8  0x05    |  1101010000000001   |      MOV R4,#1        // R4 = 1
9          |                     |
10         |                     |  LOOP:
11 0x06    |  1010001110100001   |      ADD R5,R3,R1     // R5 = address of amount[i]
12 0x07    |  0110010110100000   |      LDR R5,[R5]      // R5 = amount[i]
13 0x08    |  1010001001000101   |      ADD R2,R2,R5     // sum = sum + amount[i]
14 0x09    |  1010000100100100   |      ADD R1,R1,R4     // i++
15 0x0A    |  1010100100000000   |      CMP R1,R0
16 0x0B    |  0010001111111010   |      BLT LOOP         // if i < N goto LOOP
17         |                     |
18 0x0C    |  1101001100010100   |      MOV R3,result
19 0x0D    |  1000001101000000   |      STR R2,[R3]      // result = sum
20 0x0E    |  1110000000000000   |      HALT
21         |                     |
22         |                     |  N:
23 0x0F    |  0000000000000100   |      .word 4
24         |                     |  amount:
25 0x10    |  0000000000110010   |      .word 50
26 0x11    |  0000000011001000   |      .word 200
27 0x12    |  0000000001100100   |      .word 100
28 0x13    |  0000000111110100   |      .word 500
29         |                     |  result:
30 0x14    |  0000000000000000   |      .word 0
```

Figure 2: Example (`lab8fig2.s`) with branch instruction (use with `lab8_autograder_check.v`)

Assembly Syntax (see text)	"Simple RISC Machine" 16-bit encoding																Operation (see text)
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Branch	*opcode*			*op*		*cond*			*8b*								
`B <label>`	0	0	1	0	0	0	0	0	`im8`								`PC = PC+1+sx(im8)`
`BEQ <label>`	0	0	1	0	0	0	0	1	`im8`								`if Z = 1 then` `  PC = PC+1+sx(im8)` `else` `  PC = PC+1`
`BNE <label>`	0	0	1	0	0	0	1	0	`im8`								`if Z = 0 then` `  PC = PC+1+sx(im8)` `else` `  PC = PC+1`
`BLT <label>`	0	0	1	0	0	0	1	1	`im8`								`if N != V then` `  PC = PC+1+sx(im8)` `else` `  PC = PC+1`
`BLE <label>`	0	0	1	0	0	1	0	0	`im8`								`if N!=V or Z=1 then` `  PC = PC+1+sx(im8)` `else` `  PC = PC+1`

Table 1: Assembly instructions introduced in Stage 1 (sx & im8 defined in Lab 6 handout)

The only instruction in Figure 2 that is new versus Lab 7 is the "`BLT`" instruction on line 16. As in ARM (see Flipped Lecture #4), `BLT` stands for "branch if less than". The `BLT` instruction determines the next program counter (PC) value by comparing the negative (N) and overflow (V) flags. Recall from Lab 5 and 6 that the status flags are determined by the ALU while performing a subtraction of *Bin* input from the *Ain* input during a `CMP` instruction. Thus, the `BLT` instruction on line 16 uses the values of N and V set by "`CMP R1,R0`" on line 15.

Ignoring the possibility of overflow, if the result of `R1-R0` is negative, then it must be true that `R1` is less than `R0`. However, if `R1` is negative and `R0` is positive, then it is possible the subtraction operation performed by `CMP R1,R0` overflows and results in a positive value at the 16-bit output of the ALU that feeds into register C in Figure 1 in the Lab 5 handout. To take account of the possibility of overflow `BLT` checks whether the negative flag (N) is not equal to the overflow flag (V). If they are not equal, then `R1` must have been less than `R0` when "`CMP R1,R0`" was executed and `BLT` updates *PC* to *PC + 1 + sx(im8)*. Here `im8` is the lower 8-bits of the BLT instruction and PC is initially the address of the BLT instruction in memory, which is 0x0B for the BLT instruction on line 16 in Figure 2.

The encoding for the `BLT` instruction is shown in Table 1. From the middle portion of line 16 in Figure 2 we can see that `sas` encoded "`BLT LOOP`" as "`0010001111111010`". The lower 8-bits corresponding to `im8` are "`11111010`" which is -6 in decimal. Thus, if `R1` is less than `R0` we update the PC to be equal to `0x0B + 1 + (-6) = 11 + 1 - 6 = 0x06` which is the address of the first instruction after the label "`LOOP:`" (line 10 in Figure 2).

The rest of the instructions in Table 1 operate as follows: The "`B`" instruction branches unconditinally to the label provided. This instruction is useful for implementing "if-then-else" constructs and "while" loops. The "`BEQ`" instruction updates the PC to point to the instruction after the label if the status flags indicate source operands of the last CMP instruction were equal. Similarly, the "`BNE`" instruction branches to the instruction at the label if the source operands of the last "`CMP`" instruction were not equal. Finally, the "`BLE`"

```
1  extern int leaf_example(int,int,int,int);
2
3  int result = 0xCCCC;
4
5  void main() {
6    result = leaf_example(1,5,9,20);
7  }
8
9  int leaf_example(int g, int h, int i, int j)
10 {
11   int f;
12   f = (g + h) - (i + j);
13   return f;
14 }
```

Figure 3

instruction updates the PC to point to the instruction after the label if the first operand was less than or equal to the second operand.

1.2 Supporting function calls

The final addition to the Simple RISC Machine is adding support for function calls and returns. If you are starting early, you should review Slide Set 9 which discuss a similar example to Figures 3 and 4.

To start, consider the C code in Figure 3. Recall that in C you must declare a function before calling it. The first line helps accomplish this by declaring that function `leaf_example` takes four arguments of type `int` and returns a value of type `int`. The function `main` calls `leaf_example` on line 5. The function `leaf_example` computes the value of "`(g + h) - (i + j)`" and returns it to main. How do we implement the function call to "`leaf_example`" at line 5 in Figure 3? You might think you could use the unconditional branch "B" from Table 1 to jump from "`main`" to the start of the function `leaf_example` and another unconditional branch "B" at the end of `leaf_example` to jump back to `main`. However, that would permit us to call `leaf_example` from only one place. Instead, you will implement the "branch and link", `BL` instruction in Table 2. The use of the BL instruction is shown in Figure 4 (described below), which is the Simple RISC Machine assemble equivalent to the C code shown in Figure 3.

The "branch and link" instruction (BL) performs two operations. First, it saves PC+1, which corresponds to the instruction stored at the next address after the BL instruction, to the *link register* R7. The choice of R7 is an architecture design decision that must be agreed upon between hardware and software designers. The Simple RISC Machine use of R7 for branch and link is similar to the use of R14 in ARM (see Slide Set 9). For the BL instruction on line 12 in Figure 4 PC+1 is 0x0a, which corresponds to the address of the next instruction *after* the function call. By saving this address in a known location, namely R7, we enable software to return after the function call is finished.

Second, the BL instruction updates the PC to the address associated with the start of the function being called. For the BL instruction on line 12 in Figure 4 this address is 0x0C, which is the address of the memory location containing the first instruction in the function `leaf_example`.

The use of STR and LDR instructions in `leaf_example` on lines 16, 17, 25 and 26 is for saving and restoring register values to the stack. The stack is described in Slide Set 9.

To return from `leaf_example` to `main` we use the instruction `BX R7` on line 27 in Figure 4. This BX instruction simply copies the named register into the PC.

The last instruction in Table 2 is BLX. This function is useful for supporting object-oriented programming because it essentially enables calling a function with a "pointer". Adding support for this instruction is optional (see Bonus #1).

2 Lab Procedure

Starting from your code from Lab 7, modify your design in two stages. In the first stage add support for the conditional branch instructions in Table 1 and in the second stage add support for the call and return instructions in Table 2. To do this think through all necessary changes to your existing design based on the instruction definitions in these tables.

All student submissions that pass the Lab 8 auto grader will be ranked on performance and the top designs will be announced. Performance will be computed as one over execution time, where execution time will can be computed using the following formula:

$$\text{Execution Time} = \text{Total Cycles} \times \text{Cycle Time} \tag{1}$$

Here Total Cycles is the number of clock cycles to execute a given program as measured using ModelSim using a counter reset to zero by the reset (KEY1) and which otherwise increments by one each cycle until the HALT instruction is executed and thus LEDR[8] is set to 1. For this competition, Cycle Time will be measured using TimeQuest Timing Analyzer in the Linux version of Quartus 15.0 after compiling your Verilog for the Cyclone V FPGA in the DE1-SoC. In Quartus look under "TimeQuest Timing Analyzer" -> "Slow 1100mV 85C Model" -> "Fmax Summary" -> "Fmax". The value of Cycle Time measured in seconds is 1 divided by Fmax.

For the autograder set `LEDR[8]` to one when executing HALT and zero otherwise. Also, ensure your program counter register output is called PC inside a module with instance name CPU. See `lab8_autograder_check.v`

```
1  Address |  Content of memory  |  Assembly code (lab8fig4.s on Piazza)
2  --------+---------------------+-----------------------------------------------------
3  0x00    |  1101011000011000   |      MOV R6,stack_begin
4  0x01    |  0110011011000000   |      LDR R6,[R6]      // initialize stack pointer
5  0x02    |  1101010000011001   |      MOV R4, result   // R4 contains address of result
6  0x03    |  1101001100000000   |      MOV R3,#0
7  0x04    |  1000010001100000   |      STR R3,[R4]      // result = 0;
8  0x05    |  1101000000000001   |      MOV R0,#1        // R0 contains first parameter
9  0x06    |  1101000100000101   |      MOV R1,#5        // R1 contains second parameter
10 0x07    |  1101001000001001   |      MOV R2,#9        // R2 contains third parameter
11 0x08    |  1101001100010100   |      MOV R3,#20       // R3 contains fourth parameter
12 0x09    |  0101111100000010   |      BL  leaf_example // call leaf_example(1,5,9,20)
13 0x0a    |  1000010000000000   |      STR R0,[R4]      // result=leaf_example(1,5,9,20)
14 0x0b    |  1110000000000000   |      HALT
15         |                     |  leaf_example:
16 0x0c    |  1000011010000000   |      STR R4,[R6]      // save R4 for use afterwards
17 0x0d    |  1000011010111111   |      STR R5,[R6,#-1]  // save R5 for use afterwards
18 0x0e    |  1010000010000001   |      ADD R4,R0,R1     // R4 = g + h
19 0x0f    |  1010001010100011   |      ADD R5,R2,R3     // R5 = i + j
20 0x10    |  1011100010100101   |      MVN R5,R5        // R5 = ~(i + j)
21 0x11    |  1010010010000101   |      ADD R4,R4,R5     // R4 = (g + h) + ~(i + j)
22 0x12    |  1101010100000001   |      MOV R5,#1
23 0x13    |  1010010010000101   |      ADD R4,R4,R5     // R4 = (g + h) - (i + j)
24 0x14    |  1100000000000100   |      MOV R0,R4        // R0 = return value (g+h)-(i+j)
25 0x15    |  0110011010111111   |      LDR R5,[R6,#-1]  // restore saved contents of R5
26 0x16    |  0110011010000000   |      LDR R4,[R6]      // restore saved contents of R4
27 0x17    |  0100000011100000   |      BX  R7           // return control to caller
28         |                     |  stack_begin:
29 0x18    |  0000000011111111   |      .word 0xFF
30         |                     |  result:
31 0x19    |  1100110011001100   |      .word 0xCCCC
```

Figure 4: Example (`lab8fig4.s`) with branch and link (use with `lab8_stage2_tb.v`).

Assembly Syntax (see text)	"Simple RISC Machine" 16-bit encoding																Operation (see text)
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Direct Call	*opcode*			*op*		*Rn*			*8b*								
`BL <label>`	0	1	0	1	1	1	1	1	`im8`								`R7=PC; PC=PC+1+sx(im8)`
Return	*opcode*			*op*		*unused*			*Rd*			*unused*					
`BX Rd`	0	1	0	0	0	0	0	0	`Rd`			0	0	0	0	0	`PC=Rd`
Indirect Call	*opcode*			*op*		*Rn*			*Rd*			*unused*					
`BLX Rd`	0	1	0	1	0	1	1	1	`Rd`			0	0	0	0	0	`R7=PC; PC=Rd`

Table 2: Instructions for function calls and returns.

for more details.

3 Marking Scheme and Lab 8 Competition Instructions

Both partners *must* be preset at the demo to receive a demo mark. Please remember to check your submission folder carefully as you will lose marks if your handin submission does not contain a quartus project file, modelsim project file, or programming (.sof) file. Ensure that when you simulate the module `lab8_check_tb` in `lab8_autograder_check.v` it prints out "`INTERFACE OK`" and that your code is synthesizable by Quartus 15.0 (e.g. works when downloaded to your DE1-SoC).

IMPORTANT: Note that for the autograder to be able to evaluate your design ALL code instantiated in your `lab8_top` module **MUST** be synthesizable by Quartus 15.0 AND you MUST include a Quartus Settings File (.qsf), called lab8_top.qsf, that indicates which Verilog files are part of your project and which can be used to synthesize your design in Quartus (this file should be automatically generated by Quartus when you created a project with lab8_top as the top level entity) and a ModelSim (.mpf) project file called `lab8.mpf` which includes all files used to simulate your design.

You **must** include at least one line of comments per always block, assign statement or module instantiation (e.g., in the datapath portion) and in test benches you must include one line of comments per test case saying what the test is for and what the expected outcome is. For your state machine include one comment per state summarizing the datapath operations and one comment per state transition explaining when the transition occurs. You can complete any combination of Bonus 1 and 2 for up to 50% extra marks on Lab 8. Your mark will be computed as the sum of the following:

Table 1 instructions [3 marks autograder; 3 marks TA assigned] Your autograder mark will be broken down as two marks for passing correctness checks for all required instructions and 1 mark for your performance relative to your peers (fastest design gets 1/1 and slowest design gets 0/1). The TAs will ask you to describe your design and ask both partners questions to assess their relative contributions (3 marks).

Table 2 instructions [2 marks autograder; 2 marks TA assigned] Your autograder mark will be 1 marks for passing all correctness checks and 1 mark for relative performance (fastest design gets 1/1 and slowest design gets 0/1). The TA assigned portion will be for a demo working on your DE1-SoC. To receive full marks the demo *must* use LEDR[7:0] and SW[7:0] using memory mapped I/O (from Lab 7) while driving the clock with the built-in 50MHz clock on your DE1-SoC (`CLOCK_50`). If your demo does not include both input and output memory mapped I/O the maximum mark for the demo will be 1/2.

3.1 Hint

The fastest processor designs generally employ a design technique called pipelining which is described in the second textbook, Computer Organization and Design.

3.2 Bonus #1: Virtual Functions [1 mark]

For this bonus, implement the BLX instruction AND write an assembly test case showing how you can implement a virtual function (see https://en.wikipedia.org/wiki/Virtual_function).

3.3 Bonus #2: Interrupts [4 marks]

For this bonus, implement interrupts (see Slide Set 10) in your Simple RISC Machine and demo them working on your DE1-SoC. To get 4/4 your demo must include a working interrupt service routine interfacing with KEY3 matching the following specification: When KEY3 is pushed whatever program is running should be interrupted and one should values on SW[7:0] should be added to a counter stored at a fixed memory location (you choose the address) The main routine (which will be interrupted when KEY3 is pressed) should continuously read both the value in that memory location and the values of SW[3:0]. It should display the lower 4-bits of the counter on LEDR[7:4] and the value of SW[3:0] on LEDR[3:0]. You will get 3/4 if you implemented changes to your CPU design and wrote the required assembly code and can explain how they should work and you can show the design almost working in simulation but it has some design errors and is not work on your DE1-SoC. You will get 2/4 if you implemented relatively complete changes (more than 50% of the required changes as judged by your TA) to your CPU design and coded up the required assembly code and everything compiles but you have design errors so that your does not work in simulation. You will get 1/4 if you tried to code up something but it is less than 50% completed as judged by your TA.

4 Lab Submission

Submit all files by the deadline on Page 1 using handin. If you are working with a partner, but each partner happened to create their own independent solution code, then you must submit both solutions through Partner #1's account following the instructions in the Peer Help Policy to use directories named "not-for-demo" and "for-demo".

IMPORTANT: All students including those working alone MUST include a CONTRIBUTIONS file for Lab 6 through 8. Your CONTRIBUTIONS file MUST accurately and truthfully report the contributions of each student. While one partner may draft the CONTRIBUTIONS file, by submitting a CONTRIBUTIONS file Partner 1 is officially stating for the record that BOTH partners have affirmed the contents of the submitted CONTRIBUTIONS file are accurate and truthful. If you are Partner 1 you should obtain an email from your partner confirming their approval of the version of the CONTRIBUTIONS file you are submitting before you submit it and you should retain that email for your records. We may ask to see this email at a later date. Follow the guidelines in the Lab 3 handout for the types of things to comment upon in this file. **If no CONTRIBUTIONS file is submitted via handin your grade for the lab will be at most 5 out of 10 (even if you are working alone).**

Use the same procedure outlined at the end of the Lab 3 handout except that now because you are submitting Lab 8, you would use:

```
handin cpen211 Lab8-<section>
```

where `<section>` should be replaced by your lab section. Remember you can overwrite previous or trial submissions by adding `-o`. Ensure no files are missing or you may lose significant marks.

5 Lab Demonstration Procedure

As with Lab 3 to 7, your TA will have your submitted code with them and have setup a "TA marking station" where you will go when it is your turn to be marked. Be sure to bring your DE1-SoC in case the TA does not have theirs and/or they need to mark multiple groups in parallel.