DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 211 Computer Systems I, Fall 2024

Lab 7 Bonus: Supporting Branches in the "Simple RISC Machine"

This bonus is autograded only, the submission deadline is 11:59 PM Dec 6 (all lab sections).

1 Introduction

This bonus 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 7 or whether you pipelined your design. To make this lab a bit more fun, a part of your mark will be based upon how fast your final Simple RISC Machine is compared to your classmates.

When you submit your code it will be ranked versus other submissions that have already been submitted. Your code will not be ranked if our correctness checks fail (and moreover will not give you any feedback on what fails). This is to encourage you to test your design; you'll have to wait until after the submission deadline to get detailed feedback on any failing tests (you can still earn part marks even if some checks fail).

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 `lab7bonus_fig2.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).

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

```c
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

"`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

```
1  Address |  Content of memory  |  Assembly code (lab7bonus_fig2.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    |  1011101011011101   |      .word 0xBADD
```

Figure 2: Example (`lab7bonus_fig2.s`) with branch instruction (use with `lab7bonus_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)

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`" 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.

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

```
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

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								`R[7]=PC+1; 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=R[Rd]`
Indirect Call	*opcode*			*op*		*Rn*			*Rd*			*unused*					
`BLX Rd`	0	1	0	1	0	1	1	1	Rd			0	0	0	0	0	`R[7]=PC+1; PC=R[Rd]`

Table 2: Instructions for function calls and returns.

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 architectur2 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. 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.

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".

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. Modify the provided testbench `lab7bonus_stage2_tb.v` to work with your state machine as indicated by the comments in the file. Your `lab7bonus_top` should use the input signal `CLOCK_50`. If you look in `DE1_SoC.qsf` you will see `CLOCK_50` is connected to pin `PIN_AF14` on the Cyclone V on your DE1-SoC. On the printed circuit board of your DE1-SoC this pin is connected to another chip that generates a 50 MHz clock signal. In other words, `CLOCK_50` is a 50 MHz clock that is input to your `lab7bonus_top`. By the end this lab you will have built a real computer so it is finally time to hook it up to an external clock instead of pressing a key to advance each clock cycle. This will allows you to run longer programs quickly.

All student submissions that pass the Lab 8 auto grader will be ranked on performance. Performance on a given workload is given by 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

```
1  Address |  Content of memory  |  Assembly code (lab7bonus_fig4.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 (`lab7bonus_fig4.s`) with branch and link (use with `lab7bonus_stage2_tb.v`).

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. After submitting with handin (or HandinUI) the autograder will be run automatically and if you pass all checks your ranking will appear at a ranking webpage which is updated automatically each time you submit your code using handin or HandinUI. Note it may take some time for your ranking to appear as each submission takes about 10 minutes to run and other students may have submitted before you.

The autograder requires your design set `LEDR[8]` to one when executing HALT and zero otherwise. Thus, the autograder requires that after HALT executes, when reset is asserted and there is a rising edge of `clk`, `LEDR[8]` returns back to 0 and remains 0 until HALT is executed again. Also, ensure your program counter register output is called PC inside a module with instance name CPU. See `lab7bonus_autograder_check.v` for more details. You will note that the checker checks that PC is equal to 0x0F while the HALT instruction is stored in memory at address 0x0E. The reason for this can be seen by referring back to Figure 3 in the Lab 7 handout. In that figure you will see there is an "Update PC" state that sets `PC = PC + 1` and that this state is encountered before the "Decode" state where you would detect the instruction in the instruction register contains a HALT instruction. You don't have to use these same states in your controller, but to be compatible with the autograder your HALT instruction should set PC to the address of the next instruction. Note that your top-level module should be renamed to `lab7bonus_top`.

3 Marking Scheme and Competition Instructions

Ensure that when you simulate the module `lab7bonus_check_tb` in `lab7bonus_autograder_check.v` it prints out "`INTERFACE OK`" and that your code is synthesizable by Quartus (e.g. works when downloaded to your DE1-SoC). **Inferred latches and/or synthesis errors in Quartus will result in both disqualification from the performance competition and an additional 3.0 marks deducted.** Ensure you include Quartus and ModelSim Project files.

Table 1 instructions [6 marks autograder] Your autograder mark will be five marks for passing correctness checks for all required instructions.

Table 2 instructions [4 marks autograder] Your autograder mark will be 3 marks for passing all correctness checks for all three instructions in Table 2.

Competition [up to 10 marks] In addition to the above, after the submission deadline (and all legitimate autograder concerns have been addressed) the top 10 submissions that pass all checks will earn a 5 mark bonus (i.e., 15/10). The top ten is based upon your ranking in the ranking webpage (updated automatically each time you submit your code to github). Moreover, the #1 top ranked submission will earn 15 marks extra on top of the above (i.e., 30/10, equivalent to an extra 6% to their final grade), the #2 ranked will earn 10 marks extra (i.e., 25/10), and the #3 will earn 5 marks extra (i.e., 20/10). Submissions are ranked by geometric mean speedup across a set of "benchmarks", versus a low performance reference design.

Submitting early has the virtue of letting you know if your code passes our autograder checks. However, to encourage you to design your own testbenches you will not get any feedback on what has failed if anything until after the submission deadline.

4 Lab Submission

Submit your code using the github invite link for Lab-7 Bonus (only) from https://cpen211.ece.ubc.ca/cwl/github_info.php.