LAB B
Translating Code to Assembly

Perform the following groups of tasks:

LabB1.s

1. Create a directory to hold the files for this lab.

2. Create and run the Java app below. We seek to translate it to assembly.

```
public class LabB1
{
        public static void main(String[] args)
        {
                int t0 = 60;
                int t1 = 7;
                int t2 = t0 + t1;
                //-------------------
                int a0 = t2;
                System.out.print(a0);
        }
}
```

3. Launch your favourite editor and type the following program. Note that even though assembly language is free-form, it is a good idea to align the four fields of each line (label, instruction or directive, operands, comment) so as to enhance readability:

```
        .text
main:   #---------------------
        addi    $t0, $0, 60       # t0 = 60
        addi    $t1, $0, 7        # t1 = 7
        add     $t2, $t0, $t1     # t2 = t0+t1
        #---------------------
        addi    $v0, $0, 1        # service #1
        add     $a0, $0, $t2      # printInt
        syscall                   # do print
        #---------------------
        jr      $ra               # return
```

   Save the program under the name: LabB1.s.

4. Launch SPIM: If you are at a Unix/Linux station, run xspim and open the program created above using the load button. On a window station, run PCSpim and load the file using the File menu. This will not only read the program but also assemble it. Hence, any mistyped statement would be reported at this stage.

5. Run the program. In xspim, simply click the run button. Under Windows, select Go from the Simulator menu (as a shortcut, you can click the Go icon in the toolbar or simply press F5). In both versions, a dialog window will pop to prompt for the starting set-up. Accept the defaults by clicking OK.

6. Does the output make sense? Examine the program and note that there are two variants of the add instruction: one adds two registers and stores the sum in a third. The other adds a register and an immediate (a constant), and hence the **i** suffix.

7. We could have replaced the first statement, which uses add for assignment, with the following statement which uses the **li** pseudo-instruction:

```
li      $t0, 60          # t0 = 60
```

   The assembler removes all pseudo-instructions by replacing them with one or more real instruction that accomplishes the same thing (**add** with **$0** in the case of **li**). We will not use pseudo instructions in these labs except on one occasion.

8. One of the panes in the SPIM GUI shows the values of all registers. Verify that the value of **$t0** is 60 (decimal) or 3c (hex). Verify also the values of **$t1** and **$t2**.

9. The value of register $v0 seems odd: the program stored 1 in it yet the register pane reports 10 for it! It must be that SPIM ran additional instructions after executing our program. To confirm, let us single-step through the program: re-open the program in order to load a fresh copy and re-initialize SPIM. Instead of running, single-step by clicking the step button (xspim) or pressing F10 (PCSpim).

10. Each time you step, the statement highlighted in the text-segment pane is executed and the registers are updated. How many statements are executed before the first statement in our program?

11. Step through our program and watch the register pane. What is the value of **$v0** immediately after the execution of our program has ended?

LabB2.s

12. Modify the program by adding a directive before the first statement and a label for the last statement:

```
        .globl fini
        .text
main:   #---------------------
        ...
fini:   jr      $ra             # return
```

   Save the modified program as: LabB2.s.

13. Load a fresh copy of the program in SPIM and set a breakpoint at **main** and another one at **fini**. You do so using the *breakpt* button in xspim or the *Simulator* menu in PCSpim. This will force the program to pause when it reaches these locations.

14. Breakpoints are excellent debugging tools because when the program pauses, you get a chance to inspect the registers before resuming.

15. Breakpoints and single stepping can be used together to speed up program debugging: you set a breakpoint at the beginning of a segment of code and then single step through it.

16. You can set a breakpoint at a statement even if it is not labelled. You do so by adding its address instead. The address of an instruction appears in the text-segment pane as the leftmost field (bracketed and prefixed by `0x` to indicate it is in hexadecimal). You can copy it (with the `0x` prefix but without the brackets) and paste it in the breakpoint dialog. Use this technique to set a breakpoint at the statement that computes `$t2`.

LabB3.s

17. What will be the output of our Java app above if we replace its last statement with:

```
System.out.print((char)a0);
```

    It is recommended that you actually compile and run the app to verify your answer.

18. We seek to modify our assembly program to reflect the added type cast.

19. Our program printed the output using service #1, which prints integers. Modify the program so it prints using service #11, which interprets and prints the contents of `$a0` as a character. Save the modified program as: LabB3.s.

20. Run the program. Did you get the output you predicted?

LabB4.s

21. Revert back to LabB2.s and save it as LabB4.s. Modify it so that it prints the sum and the difference of `$t0` and `$t1` separated by a space. Use the `sub` instruction to subtract registers. In order to print a space delimiter, use service #11 with `$a0` being the space character:

```
addi     $a0, $0, ' '
```

    (Alternatively, you can store 32 in `$a0`.) Run and verify that the output becomes:

```
67 53
```

LabB5.s

22. Revert back to LabB2.s and save it as LabB5.s. Rather than hard-coding numbers in our program, let us read one of them from the user. The Java equivalent would be to replace "`int t0 = 60`" with:

```
int t0 = (new Scanner(System.in)).nextInt();
```

23. We do this using service #5, *readInt*. Replace the statement:

```
addi    $t0, $0, 60       # t0 = 60
```

with:

```
addi    $v0, $0, 5        # v0 = readInt
syscall
add     $t0, $0, $v0
```

Note that we set **$v0** to 5 prior to issuing syscall. Afterwards, the entered integer is returned to us in **$v0**. We copy the return to **$t0** so that the rest of the program is left unchanged. Run the program and enter 60. Do you obtain the expected output?

LabB6.s

24. Save LabB5.s as LabB6.s then modify it so that it processes the two numbers as follows:

```
if ($t0 == $t1)
{
    print($t0 + $t1);
}
else
{
    print($t0 - $t1);
}
```

We translate this control structure using a conditional transfer of control (a *branch*) and an unconditional one (a *jump*):

```
        if ($t0 == $t1) branch to XX;
        print($t0 - $t1);
        jump to YY;
XX:     print($t0 + $t1);
YY:     rest of program
```

The first instruction in the above pseudo-code can be realized using:

```
beq $t0, $t1, XX
```

The third instruction is realized using:

```
j YY
```

Run the program and verify that it works as expected. Note that:

beq (branch-on-equal) is used to translate ==
bne (branch-on-not-equal) is used to translate !=

We now seek a translation for the “<” and “>” relational operators.

LabB7.s

25. Save LabB6 as LabB7 and modify it so it processes the two numbers using this logic:

```
if ($t0 < $t1)
{
    print($t0 + $t1);
}
else
{
    print($t0 - $t1);
}
```

The MIPS assembly language has no branching instruction based on comparing two registers. We therefore reduce "less-than" to "not-equal" using a technique that relies on **slt**. The Java statement:

```
x = (y < z) ? 1 : 0;
```

translates to the following assembly instruction:

```
slt $x, $y, $z
```

It sets **$x** to 1 if **$y** < **$z** and sets it to zero otherwise. Using this "set-on-less-than" instruction, you can perform the above test using **bne**. Run the program and verify that it works as expected. Note that **slt** has an immediate variant (**slti**) that allows the third operand to be an immediate.

LabB8.s

26. Start fresh and create the program LabB8.s with the following body (between **main** and **fini**):

```
        addi    $v0, $0, 1
        add     $a0, $0, $0
loop:   slti    $t9, $a0, 5
        beq     $t9, $0, fini
        syscall
        addi    $a0, $a0, 1
        j       loop
```

It is important that you attempt to predict the output before you run the program.

27. Save the program and run it to confirm, or revise, your prediction.

28. Replace the statement before last in the above program with:

```
        addi    $a0, $0, 1
```

What will the output be in this case? Show that single stepping is ideally suited to debug such a (very common) logic error.

LabB9.s

29. Start fresh and create the program LabB9.s that translates this logic:

```
int s0 = 0;
int t0 = (new Scanner(System.in).nextInt();
for (int t5 = 0; t5 < t0; t5++)
{
    s0 = s0 + t5;
}
System.out.print(s0);
```

Note that there is no looping construct in assembly; you use branches and jumps. You can, for example, re-think the above loop as follows:

```
          t5 = 0;
loop:     if (! t5 < t0) branch to done;
          s0 = s0 + t5;
          t5++;
          jump to loop;
done:     System.out.print(s0);
```

Save the program then run it with 10 as input. Do you get 45 as output?

LabB10.s

30. Start fresh and create the following program:

```
          .text
main:     #---------------------
          addi      $t0, $0, 60
          addi      $t1, $0, 7
          div       $t0, $t1
          mflo      $a0
```

The **`div`** instruction is a translation of:

```
lo = $t0 / $t1;
hi = $t0 % $t1;
```

In other words, it sets `lo` to the quotient and `hi` to the remainder of dividing its two integer operands. Since these two registers are not programmable (i.e. cannot be referenced in a program), the two instructions: **`mflo`** (move-from-`lo`) and **`mfhi`** (move-from-`hi`) were added to the instruction set to facilitate copying the contents of `lo` and `hi` to programmable registers. Complete the above fragment by having it output `lo` and `hi`. Save the program under the name: LabB10.s. Run the program and verify that it works as expected.

31. Modify the program so that it also prints the product of the two integers. This can be done using the instruction:

```
mult      $t0, $t1
```

It multiplies its two operands and store the 64-bit product in `hi` and `lo`, i.e. its most-significant 32 bits in `hi` and its least-significant 32 bits in lo. Again, transfer these two pieces to general-purpose registers and print them.

LabB11.s

32. We now seek a translation of the following logic:

```
int t0, a0;
t0 = 60;
a0 = t0 >>> 1;
System.out.print(a0);
System.out.print(' ');
a0 = t0 << 1;
System.out.print(a0);
```

To that end, start fresh and create the program LabB11.s as follows:

```
          .text
main:     #---------------------
          addi      $t0, $0, 60
          srl       $a0, $t0, 1
          // print $a0
          // print ' '
          sll       $a0, $t0, 1
          // print $a0
          #---------------------
fini:     jr        $ra
```

Replace the comments by appropriate output statements. Run the program and examine its two outputs. Explain the output based on your understanding of how integers are represented in binary.

33. Set a breakpoint immediately after the **`srl`** instruction. When the program pauses, examine the values of **`$t0`** and **`$a0`** as shown in the register pane. Explain these values based on your understanding of how integers are represented in hex.

34. Change the shift amount from 1 to 2. Re-run the program and interpret its output.

35. We know that a left shift corresponds to a multiplication by a power of two. Does this remain true when the operand being shifted is negative? Change the **`$t0`** from 60 to -60 and observe.

36. We know that a right shift corresponds to a division by a power of two. Does this remain true when the operand being shifted is negative? Change the **`$t0`** from 60 to -60 and observe.

37. In order to address the shortcoming of right shifts for negative operands, a second form of the right shift instruction is available. Use **`sra`** (shift right arithmetically—Java operator **`>>`**) in place of **`srl`** (shift right logically—Java operator **`>>>`**). Does the new shift produce the correct result for negative as well as positive operands?

LabB12.s

38. Write the program LabB12.s that reads an integer *x* from the user and outputs *18x*, (*x* multiplied by 18) without using **`mult`**. Note that you can rewrite *18x* as a sum of two terms each of which multiplies x by a power of 2.

39. Replacing multiplications by shifts is often done in high-level languages by optimizing compilers. Why is shifting a register faster than multiplying?

LabB13.s

40. The program LabB13.s seeks to determine and output the value of bit #10 in **`$t0`** (recall that bits are numbered right-to-left starting with zero). It does so by isolating this bit using the following logic:

```
a0 = t0 << 21;
a0 = a0 >>> 31;
```

    Write this program so that it reads an integer into **`$t0`**; performs the above algorithm; and then outputs **`$a0`**.

41. Run the program and test it. For example, if the input is 5000, the output should be 0, but if the input were 6000 then output would be 1.

-1?

LabB14.s

42. Save LabB13.s as LabB14.s and modify it so it accomplishes the same goal using an appropriately chosen mask. In order to isolate bit #10, a mask of the form:

```
0000 0000 0000 0000 0000 0100 0000 0000
```

    is needed. If we *AND* this mask with **`$t0`** using the instruction:

```
andi     $a0, $t0, 1024
```

    then **`$a0`** will be zero if, and only if, bit #10 is zero. Note that if bit #10 were not zero then **`$a0`** would not be zero *but it would not be 1*. You must therefore check for zero or non-zero.

43. Complete the implementation and verify that the mask-based approach produces the same results as the shift based one.

44. Could we have written the mask in hex, i.e. 0x400 instead of 1024?

LabB15.s

45. We seek to write the program LabB15.s that clears bit #10 in **`$t0`**; i.e. sets it to zero. The idea is to read an integer into **`$t0`** and then *AND* it with the following mask:

```
1111 1111 1111 1111 1111 1011 1111 1111
```

Unfortunately this does not work because the immediate in the instruction:

```
andi    $t5, $t0, 0xffffbff
```

has a representation width of 16 bits. Indeed *all* MIPS instructions (with the exception of jumps) cannot have an immediate larger than 16 bits. *(Note that this stems from the 32-bit instruction width, not the CPU register width.)* We therefore must find ways around this. In this case, we can construct this mask in stages, as follows:

```
t5 = 0xffff;
t5 = t5 << 16;
t5 = t5 | 0xfbff;
```

Note that the immediate in the above instructions is 16-bit wide.

46. When you implement the program, make sure you use **ori** rather than **addi** to store a bit pattern in a register. This is because **ori** (and all the other logical instructions) zero-extend the 16 bit immediate whereas **addi** sign-extend it.

47. Verify that your program works as expected. For example, if the input were 5000 then the output should be 5000 whereas a 6000 input should lead to 4976 output.

LabB16.s

48. Save LabB15 as LabB16 and modify it so it generates the same mask in a different way. The idea is to observe that the mask we are after is the negation of a mask we generated earlier (in LabB14). Specifically, we seek to use the following algorithm:

```
t6 = 1024;
t6 = ~t6;
```

There is no *NOT* instruction in MIPS but there is **nor** (*NOT OR*).

49. Implement this algorithm and verify that it is equivalent to the previous one. You can simply inspect the register pane of spim and compare the contents of **$t5** and **$t6**.

LabB17.s

50. Save LabB15 as LabB17 and modify it so it generates the same mask in a way that is different from the previous two tasks. We will rely on **lui** (load-upper immediate). This instruction allows us to set the upper half of a register to a given 16-bit value:

```
lui     $t7, 0xffff
```

This instruction takes the specified 16-bit immediate as-is, with neither zero nor sign extension, and places it in the upper half of the specified register. The lower half of the register is zeroed out. Having constructed the upper half of the mask and stored it in the register, we store the lower half using:

```
ori     $t7, $t7, 0xfbff
```

51. Implement this algorithm and verify that it is equivalent to the previous two. Keep all three masks in your code and simply compare the contents of **$t5**, **$t6**, and **$t7**.

52. The **lui** instruction is used often not just for bit-level processing but also whenever we need to express a value that does not fit in 16 bits.

LabB18.s

53. It is often said that exclusive-or is useful for detecting differences in bit values. In the following Java computation, for example:

```
t0 = a0 ^ s0;
```

    The bits of **t0** reflect whether the corresponding bits in **a0** and **s0** are the same or not. In particular, a value of 1 implies that the corresponding bits are different, i.e. either 01 or 10. Justify this claim by examining the truth table of *XOR*.

54. The Java operator ^ translates in MIPS to the instruction **xor**.

```
xor     rd, rt, rs      # rd = rt ^ rs
```

55. MIPS also support an immediate version of **xor**:

```
xori    rt, rs, imm     # rt = rs ^ imm
```

56. Write the program LabB18.s that flips bit #10 in **$t0**. Prepare an appropriate mask and note that when you *XOR* a bit with 1, you get the negation of that bit, but when you *XOR* with 0, you get the original bit unchanged.

LAB B
Notes

- The immediate in all the arithmetic instructions, i.e. `addi`, `addiu`, `slti`, `sltiu` is treated as a 16-bit signed integer and is sign (*not* zero) extended to 32 bits. For example, if the immediate is +5 then it is treated as:

  ```
  0000 0000 0000 0101
  ```

  and then sign extended (by replicating its MSb):

  ```
  0000 0000 0000 0000 0000 0000 0000 0101
  ```

  Similarly, if the immediate is -5 then it is treated as:

  ```
  1111 1111 1111 1011
  ```

  and then sign extended:

  ```
  1111 1111 1111 1111 1111 1111 1111 1011
  ```

  Hence, the value of the immediate in arithmetic instructions is constrained to the 16-bit signed range, i.e. $[-2^{15}, +2^{15} - 1]$.

- The shift amount appears as an immediate in shift instructions but, in fact, it is represented in *five* bits as an unsigned integer. Hence, its range is [0, 31].

- The 16-bit immediate in non-shift logical instructions is treated as a 16-bit unsigned integer and is zero extended (*not* sign extended) to 32 bits. Hence, if you **`andi`** using 35000 as an immediate, you are *AND*ing with the 32-bit sequence:

  ```
  0000 0000 0000 0000 1000 1000 1011 1000
  ```

  The value of the immediate in non-shift logical instructions is thus constrained to the 16-bit unsigned range, i.e. $[0, 2^{16} - 1]$.