ENCM 369 Winter 2018 Lab 5 for the Week of February 12

Steve Norman
Department of Electrical & Computer Engineering
University of Calgary

February 2018

Lab instructions and other documents for ENCM 369 can be found at
http://people.ucalgary.ca/~norman/encm369winter2018/

Administrative details

Attention: Equipment for Exercises D and E

Exercises D and E require use of a computer in ICT 320. If you choose not to do those exercises during your scheduled lab period, please make some time while you're on campus to get the necessary work done in the lab—it's impossible to do them remotely.

You may work in pairs on this assignment

You may complete this assignment individually or with *one* partner.

Students working in pairs must make sure both partners understand *all* of the exercises being handed in. The point is to help each other learn *all* of the lab material, not to allow each partner to learn only half of it! Please keep in mind that you will not be able to rely on a partner to do work for you on midterm #2 or the final exam.

Two students working together should hand in a *single assignment* with names and lab section numbers for both students on the cover page. Names should be complete and spelled correctly. If you as an individual are making the cover page, please get the information you need from your partner. For partners who are not both in the same lab section, please hand in the assignment to the collection box for the student whose last name comes first in alphabetical order.

Due Dates

The Due Date for this assignment is 3:30pm Friday, February 16.
The Late Due Date is 3:30pm Monday, February 26.

The penalty for handing in an assignment after the Due Date but before the Late Due Date is, **as a special policy related to Reading Week, only 1 mark.** In other words, X/Y becomes (X–1)/Y if the assignment is late. There will be no credit for assignments turned in after the Late Due Date; they will be returned unmarked.

Marking scheme

A	4 marks
B	2 marks
C	2 marks
D	4 marks
E	4 marks
TOTAL	16 marks

How to package and hand in your assignments

Please see the Lab 1 instructions. In addition, if you work with a partner, be accurate about both partners' names and lab section.

Exercise A: Practice with logical instructions

Read This First

All of the instructions used here are described in Section 6.4.1 of the course textbook.

What to Do

Suppose that before the following instructions are run, `$t8` contains `0x0000_0c54` and `$t9` contains `0x0000_0a36`.

```
lui     $t0, 0x9a63
srl     $t1, $t0, 6
or      $t2, $t8, $t9
andi    $t3, $t8, 0x07fc
nor     $t4, $t8, $t8
nor     $t5, $t8, $zero
xor     $t6, $t8, $t9
xori    $t7, $t8, 0x07fe
```

Without the aid of a computer, determine what values will be in `$t0`–`$t7` after the above instructions are executed. Show intermediate steps in base two (except for the `lui` instruction—there's no real intermediate work to be done there) and then express your final answers in base sixteen.

What to Hand In

Hand in neatly hand-written solutions to this exercise.

Exercise B: Pseudoinstructions with real instruction mnemonics

Read This First

In lectures and labs you have already seen the `la` and `li` pseudoinstructions. These can be used in MARS and MIPS assembly language, and will generate appropriate machine instructions to copy an address (with `la`) or a 32-bit constant (with `li`) into a GPR.

Many MIPS assemblers offer a very wide selection of pseudoinstructions, including, somewhat confusingly, some that share mnemonics with real instructions.

Here is an example, using `sw`. Suppose that `foo` is a label for a word allocated in with `.word` in a `.data` section in an assembly language file, and suppose also that the address of this word will be `0x1001_000c`. It's possible to write this in a `.text` section:

```
sw      $t0, foo
```

That's not a real instruction—I certainly hope that by this point in the course you know that the address in a real `sw` instruction involves a GPR and a constant offset. The assembler will deal with the above pseudoinstruction by generating these two real instructions:

```
lui     $at, 0x1001
sw      $t0, 12($at)
```

Note the use of the GPR `$at` ("assembler temporary"), and note how the address of `foo` generates both the constant in the `lui` instruction and the offset in the `sw` instruction.

Here is another example. Many MIPS assemblers will accept this:

```
add      $s0, $s1, 0x123456
```

There are two reasons why that is not a real instruction. First, `add` needs two GPRs as sources, not one GPR and a constant. Second, it's not even acceptable as an alternate syntax for a real `addi` instruction, because the constant `0x123456` can't be packed into the 16-bit constant field in an `addi` instruction.

Read This Second

Here is some helpful information about machine code formats for `lui` and `ori` instructions:

- For `lui`, the opcode, in base two, is 001111. Bits 25–21 are 00000, the destination register number is given in bits 20–16, and the constant is given in bits 15–0.
- For `ori`, the opcode, in base two, is 001101. The source register number is given in bits 25–21, and the destination register number is given in bits 20–16. The constant is given in bits 15–0.

What to Do

This is a pencil-and-paper exercise. There is no need to use a computer.

(1) Suppose that `str99` is a label for a byte allocated in a `.data` section, and that the address of this byte is `0x1003_49a0`. Determine the *machine code* that would be generated as a result of this pseudoinstruction:

```
sb     $t8, str99
```

Use the above example with `sw` as a guide, and show the steps you used to arrive at an answer. Give the machine code in both base two and hexadecimal representations.

(2) Find machine code for

```
add     $s4, $s1, 0x5dc70
```

Show the steps you used to arrive at an answer, and give the machine code in both base two and hexadecimal representations. (*Hint:* you will need three instructions in total.)

What to Hand In

Hand in handwritten answers for items (1) and (2).

Exercise C: Machine code for MIPS branches and jumps

What to Do

Consider the following sketch of some MIPS assembly language code:

```
L1:     lbu     $t0, ($s2)
        beq     $t0, $zero, L2

        many more instructions

        addi    $s2, $s2, 1
        j       L1
L2:     or      $s5, $s2, $zero
```

Suppose that the address of the `lbu` instruction ends up being `0x0040_7320` and the address of the `or` instruction ends up being `0x0040_73a4`. What will be the machine code for the `beq` and `j` instructions?

Take time to write the steps taken to determine the answers, and express the answers as eight-digit hexadecimal numbers.

What to Hand In

Hand in your answers along with the work needed to find those answers.

Exercise D: Pointer/Index Speed Comparison

Read This First

You have seen that given an algorithm that needs to access each element of an array, you can implement it in two rather different-looking ways. The first way uses an integer index variable—in the following code, `i` is the index variable:

```
for (i = 0; i < n; i++) {
  do something with a[i];
}
```

The second way uses pointer arithmetic—in the following code, `p` and `past_end` are pointers:

```
past_end = p + n;
p = a;
while (p != past_end) {
  do something with *p;
  p++;
}
```

You now have some experience translating both kinds of loops into MIPS assembly language, and you should have noticed that the second of the two above code fragments results in a shorter sequence of instructions for the loop than does the first.

Most modern compilers have *optimization* options. If you don't ask for optimization, a C compiler will generate instructions by translating C expressions in a straightforward manner—for example, if `a` is of type `int*` and `i` is of type `int`, the address of `a[i]` would be computed by adding four times `i` to the address in `a`.

If you do ask for optimization, a compiler will try to generate sequences of instructions that have the effect specified by the source code and that achieve that effect as quickly as possible. Optimizing C compilers use a lot of different tricks to generate fast machine code; some of these tricks are very complex. Here are two important and relatively simple optimization tricks:

- When compiler optimization is *not* requested, local variables and function arguments are often all placed on the stack—that turns out to be helpful if a tool called a *debugger* is used to examine contents of variables and arguments belong to a running executable.

  When optimization *is* requested, frequently-accessed local variables and function arguments will be put in registers instead; the result can be a large reduction in the number of memory accesses.

- Source code that accesses array elements using square brackets can be translated into assembly language that achieves the same effect with instructions that work using pointer arithmetic.

In this exercise you will use `gcc` on Cygwin64 to compare the speed of index-based and pointer-based functions, with and without compiler optimization.

The definition of *speedup*

Speedup is a commonly used method for quantifying the relative performance of two computer programs that do the same job, or two parts of programs, where the parts do the same job.

There are lots of possibilities for what the two programs or two parts of programs might be. Here are some common examples:

- There are "old" and "new" versions of a program, where the source code of the "new" version has been rewritten in an attempt to improve performance.

- The "old" and "new" versions have exactly the same source code, but the executable file for the "new" version is built using options that are expected to generate faster machine code.

- The "old" and "new" executables are exactly the same, but are run on different hardware. In this case we're measuring the effect of changing hardware, while in the above two cases we're measuring the effect of changing software.

Let T_{old} be the "old" running time, and T_{new} be the "new" running time. Then *speedup* is defined as

$$\text{Speedup} = \frac{T_{\text{old}}}{T_{\text{new}}}$$

Note that a speedup that is > 1 indicates that the change from "old" to "new" really is an improvement, while a speedup that is < 1 indicates that the change actually made performance worse.[1]

[1] Also—somewhat but not totally joking—a speedup of ∞ means that the new program works but the old one didn't; a speedup of 0 means that the old program works but the new one is broken.

Notes about timing code with `clock`

In this exercise the `clock` function from the Standard C Library will be used to time how long a portion of a program takes to run. Here are a few brief remarks about using `clock`:

- To get reasonably accurate measurements of how long it takes part of a program to run, the total time measured should be on the order of one second or more. So it may be necessary to call a function thousands of times in between calls to `clock`, in order to get a long enough time interval.
- The `clock` function will measure only processor time used by the program that calls it, not by other programs that may get some processor "timeslices" while the program being measured is run. However, these other programs may cause the program being measured to use excessive processor time due to events called "cache misses". (We will cover caches later in the course.) Therefore when doing measurements such as the ones in this exercise, it is best to reduce activity of other programs as much as possible.

  When you are running the executables you generate in this exercise, avoid having other applications run at the same time, especially a web browser, which can get quite busy grabbing data from the Web and updating one or more of its windows or tabs.
- There are a few more things to read about `clock` in the comments in the file `main.c`.

What to Do, Part I

In order to ensure that all students use the same hardware and compiler version, please do this exercise on one of the computers in ICT 320.

Copy the directory `encm369w18lab05/exD` , then read the three files in the directory to see what the program does. Pay close attention to the comment describing the use of the `clock` function.

Build an executable with the command

```
gcc main.c functions.c
```

and run the executable. You will have to wait 15 seconds or so for it to finish. Run the executable a few more times.

You will probably notice that the CPU time measurements are not quite identical from one run to the next, indicating that there is some degree of error in these measurements. Use the following method to estimate CPU time over ten runs of the program:

> Take the average of the fastest five times. Ignore the slowest five times.

The rationale for this is that the operating system kernel or other programs will occasionally do things to hurt the performance of the program being tested, but normally can't do anything to give an unfair improvement to that performance.

Answer the following questions:

- *How much CPU time is used by 1,000,000 calls to* `index_version`*? How much CPU time is used by 1,000,000 calls to* `pointer_version`*?*
- If `index_version` is considered to be the "old" version of a function and `pointer_version` is considered to be a "new" version, *what is the speedup?*. Use the definition of *speedup* given on page 5.

Now build an optimized executable with the command

```
gcc -O2 main.c functions.c
```

(That's an uppercase letter O, not the digit 0.) The `-O2` option asks the compiler to optimize fairly aggressively. Run the new executable, then answer these questions:

- *What is the speedup for* ***`index_version`*** *compiled with* ***`-O2`*** *optimization relative to* ***`index_version`*** *compiled without optimization? What is the speedup for* ***`pointer_version`*** *compiled with* ***`-O2`*** *optimization relative to* ***`pointer_version`*** *compiled without optimization?* (Show the data and calculations you used to answer these two questions.)
- *With optimization, is* ***`pointer_version`*** *significantly faster than* ***`index_version`****?*
- *Which seems to be a more important factor in array-processing speed: using pointers instead of indexes, or asking the compiler for optimization?*

Read This, Part II

This part of the exercise asks you to try one more compiler optimization option, called *loop unrolling*.

Consider the following code taken from `functions.c`:

```
int sum = 0;
const int *p;
const int *past_last;
past_last = a + n;
p = a;
while (p != past_last) {
  sum += *p;
  p++;
}
```

Each pass through the loop executes the comparison `p != past_last` and the pointer update `p++`. The code could be sped up if those two operations were performed less often:

```
int sum = 0;
int *p;
int *q;

/* q = a + (n rounded down to multiple of 4) */
q = a + (n & 0xfffffffc);

p = a;
while (p != q) {
  sum += *p;
  sum += *(p + 1);
  sum += *(p + 2);
  sum += *(p + 3);
  p += 4;
}
switch (n % 4) {
case 3: sum += *(q + 2);
case 2: sum += *(q + 1);
case 1: sum += *q;
}
```

Note that the comparison and pointer update are now done only once for every four array element accesses. The `switch` statement is needed in case `n` is not a multiple of four. (Choosing to do four element accesses in the loop body is just an example. It might be even more efficient to do eight element accesses in the loop body.)

The new code is longer and much less readable than the original, so it is probably not a good idea to modify the original C code. However, `gcc` and many other optimizing compilers can translate C code like the original simple loop into assembly language that works like the faster, more complicated C code. This is called *loop unrolling*. `gcc` with `-O2` does not do loop unrolling unless you ask for it explicitly.

What to Do, Part II

Build a new executable with the command

```
gcc -O2 -funroll-loops main.c functions.c
```

What is the speedup for `pointer_version` with `-O2` and loop unrolling, relative to `pointer_version` compiled with `-O2` only? (Again, show your data and calculations.)

What to Hand In

Hand in neatly typed or hand-written answers to all of the questions asked *in italics* in What to Do, Parts I and II.

Some Final Remarks

This is just one experiment with one not-very-practical program. Don't assume that compiler optimization will always be as effective as it appears to be in this exercise.

The most important factor in program speed is use of *fast algorithms*. Compiler optimization might make a sort function run several times faster than the same function compiled without optimization, but switching from insertion sort to quicksort might speed up the function by a factor of *over a thousand* if the array being sorted is very large.

Modern operating systems provide alternatives to `clock` that may in some cases be more precise and helpful. If you are looking for those alternatives—not needed for ENCM 369, but possibly useful for projects in other courses or in "the real world"—here some names of functions you could do Web searches for:

PLATFORM	FUNCTION
Linux	`clock_gettime`
Windows	`QueryPerformanceCounter`
Mac OS X	`mach_absolute_time`

Whatever you find and choose, read the documentation carefully—it's easy to make big mistakes in performance measurement if you do not understand exactly what information is being given to you by time-measuring function calls!

Exercise E: Steps in building an executable

Read This First

The point of this exercise is to help you get a better idea of the individual steps used to create an executable from C source files. You may wish to re-read Section 6.6 of your textbook, and lecture notes from early February to be reminded of the meanings of terms such as *compiler*, *object file*, *linker*, and *executable file*.

Read This Second: x86 and x86-64

The term *x86* usually refers to an instruction set architecture (ISA) that first became available with the Intel 80386 processor in 1985. Section 6.8 in your textbook does a good job of summarizing the main features of the x86 architecture.

Most processor chips installed in current PC and Apple Mac laptop and desktop computers support what known as the *x86-64* ISA. The term x86-64 is essentially a generic term for the two nearly-identical architectures named AMD64 and EM64T by AMD and Intel, respectively. This is described very briefly in Section 6.8.6 of the textbook.

Something that *isn't* mentioned in the textbook is that x86-64 has 16 GPRs, instead of just the eight that x86 has.

x86-64 processors can run most programs that have been compiled, assembled, and linked for x86. This is important because there is a huge base of software developed for x86 that users continue to want to run.

The computers in ICT 320 have x86-64 processors. The version of `gcc` available on Cygwin64 these computers is configured to generate executables for the x86-64 architecture. So you'll be looking at x86-64 instructions as you work through this exercise.

What to Do

Since this exercise, unlike Exercise D, is not about program performance, you do not have to use a specific model of hardware for it. However, you do have to use a specific version of `gcc` for x86-64. By far the easiest way to be sure of using the correct version of `gcc` is to use a computer in ICT 320.

Get into the same directory you used for Exercise D. Now follow this sequence of steps.

1. Run the preprocessor on `functions.c` with the command

   ```
   gcc -E functions.c -o functions.i
   ```

   The `-E` option says "preprocess only"; the `-o` option is used to specify the name of the output file. Use the command `less functions.i` to have a look at the file `functions.i`, which is the *translation unit* produced by the C preprocessor. You should see that the contents are C code that include the function prototypes from `functions.h`. (Notes on `less`: use the space bar to page forward; use the `b` key to page backward; use the `q` key to quit; use the `h` key for help on other commands.) By the way, lines such as

   ```
   # 1 "functions.c"
   ```

   appear in the translation unit so that the compiler can know the locations of lines of C code in their original `.c` or `.h` files.

2. Run the compiler on `functions.i`:

   ```
   gcc -S functions.i
   ```

   The `-S` option says "translate to assembly language, but do not go on to run the assembler". The output will be a file called `functions.s`. Have a look at this file with `less`. You probably won't be able to understand all the details, because (a) the x86-64 assembly language syntax used by the GNU assembler is a bit different from the MIPS assembly language syntax and (b) the x86-64 instruction set is very different from the MIPS instruction set. Nevertheless,

you should be able to see the general idea—for each of the two C functions there is a label followed by sequence of instructions that do the work of the function.

The `while` loop in `pointer_version` will appear as 8 instructions starting with `jmp .L2`. (By the way, that first instruction is an unconditional jump to the place in the code where the comparison `p == past_last` is done.) *Answer this question:* What are these 8 instructions? Just list them—you don't have to explain what they do.

3. Run the assembler on `functions.s`:

```
gcc -c functions.s
```

The `-c` option says "generate an object file but do not go on to run the linker". The output will be a file called `functions.o`. This file is *not* a text file, so it's not helpful to try to view it with `less`. Instead you can use the `objdump` program, which can get information from object files and display it in human-readable form. Run the command

```
objdump -d functions.o | less
```

This will direct the output of `objdump` to the input of `less`, so you can view the output of `objdump` one page at a time. The `-d` option is for "disassemble"; what you'll see is bit patterns for x86-64 instructions along with translations of those bit patterns back into assembly language. Notice that different instructions are different numbers of bytes in length—that's very different from MIPS, where all instructions are four bytes long.

Answer this question: What is the machine code for the instruction `addl $0x1,-0x4(%rbp)` ? Write your answer as sequence of bytes written in hexadecimal. (By the way, this instruction is the translation of `i++` in `index_version`.)

4. Repeat the above three steps starting with `main.c`. Here are three things you should observe:

   - The translation unit `main.i` is a large file because a lot of C code is included from `<stdio.h>`.
   - The assembly language file `main.s` is *not* very large—none of the type declarations and function prototypes in `<stdio.h>` cause the compiler to generate any assembly language code.
   - Looking at `main.s`, you will *not* find assembly language translations of the library functions `clock` and `printf`.

   Answer these questions:

   - The line

     ```
     sum = index_version(x, ARRAY_SIZE);
     ```

     appears in `main.c`; what does this line get replaced with in the translation unit `main.i`? (Hint: If you press the `/` key in `less`, you will be allowed to enter a string that `less` will search for, which is more fun than paging through a thousand lines of stuff from `<stdio.h>`.)
   - What is the machine code for the instruction `movl $0xf4240,-0x8(%rbp)` ? (This appears about 30 instructions from the beginning of `main`—it's an instruction that gives the immediate value `0xf4240`, which is 1,000,000, to the variable `count`, located in memory at address `-0x8(%rbp)`.) Write your answer as sequence of bytes written in hexadecimal.

5. Run the linker with the command

```
gcc functions.o main.o -o exE.exe
```

   Here again the `-o` option is used to name the output file. To confirm that you have actually built an executable, run it with the command `./exE.exe`

What to Hand In

Hand in neatly typed or hand-written answers to all of the questions asked in What to Do.

Optional Exercise F: The `jr` instruction and C `switch` statements

Read This First

This exercise won't be marked and, *unlike many other unmarked lab exercises in this course*, is not related to possible midterm or final exam questions.

However, it does provide some insight into how C `switch` statements can be translated into highly efficient assembly language code, and shows a use of the `jr` instruction that has nothing to do with returning from a procedure.

Consider this example `switch` statement:

```
switch(i) {
 case 1: case 3: case 5:
   printf("odd, between 1 and 6\n");
   break;
 case 2: case 4: case 6:
   printf("even, between 1 and 6\n");
   break;
 default:
   printf("less than 1 or greater than 6\n");
}
```

How could it be translated into assembly language? One approach would be to write an equivalent `if` statement ...

```
if (i == 1 || i == 3 || i == 5)
  printf("odd, between 1 and 6\n");
else if (i == 2 || i == 4 || i == 6)
  printf("even, between 1 and 6\n");
else
  printf("less than 1 or greater than 6\n");
```

... which could then be used as a guide for generating an appropriate collection of branch instructions.

But what if there are a hundred cases in a frequently-executed `switch` statement? (Some real-world C `switch` statements do have that many cases!) Coding a hundred branch instructions would result in slow code—on average, when the `switch` statement is entered, fifty or so branch instructions would have to be executed before a branch would cause a "goto" to the appropriate statement. (The number fifty is based on an assumption that all cases are reached with equal frequency; even if the assumption is not true there is still a potential for a lot of branches to be executed.)

Is there a way to jump more quickly to the appropriate statement? The answer to this is yes, using something called a "jump table" and the MIPS `jr` instruction (similar instructions, when programming other processor architectures).

What to Do

Copy the directory `encm369w18lab04/exF`

Carefully study the files `switch1.c` and `switch1.asm`. Note the key role played by the array that is set up with `jump_table` as a label, and the use of the instruction `jr $t5`.

Consider, for example, the situation when `k` is 9. After checking for `k < 0` and `k > 10`, there is no checking for `k == 1`, `k == 2`, and so on, one case at a time, up to `k == 9`. Instead, a four-instruction sequence rapidly puts the right address to jump to into `$t5`, and then there is a jump straight to the instructions needed to handle `k == 9`.

Now study the file `switch2.c`. Using the code in `switch1.asm` as a guide, write a MARS translation of `switch2.c`, using a jump table to implement your translation of the `switch` statement.

What to Hand In

There is nothing to hand in. You may check your work against a solution that will be posted on the Web sometime before the end of Reading Week.

Optional Exercise G: Recursion in Assembly Language

Read This First

Like Exercise F, this exercise won't be marked and, again, *unlike* many other unmarked lab exercises in this course, is not related to possible midterm or final exam questions.

Recursion is a powerful tool in computer programming. One general form of recursive solution to a problem involving a collection of data items is as follows:

- If there is only one item—or sometimes, if the collection is empty—solve the problem in some very easy way. For example, it requires no work at all to sort a collection that has only one item.
- If there are two or more items:
  - Divide the collection into two parts, each roughly half the size of the original.
  - Make two recursive calls to solve the problem for each partial collection.
  - Combine the results produced by the recursive calls.

This general structure loosely describes many important algorithms, including the famously efficient sorting algorithms called *quicksort* and *mergesort*.

In this exercise, you will look at recursive code that performs a much easier task than sorting: summing the element values of an array. Obviously, a recursive solution is neither the easiest nor the most efficient way to perform this task using C or assembly language. However, the code is very short, and relatively easy to understand compared to complicated sort functions.

What to Do

Copy the directory `encm369w18lab05/exG`

Study the C code and the corresponding assembly language code. Bring the assembly language code into MARS, and assemble it.

Set a breakpoint on the first `sw` instruction in the prologue `array_sum`. Run the program to the breakpoint, and then single-step through the four `sw` instructions in the prologue, watching a return address and three incoming arguments get copied into the stack.

Click on the Run icon; you will find that the program stops at the same breakpoint. Single-step again through the prologue and observe that the saved arguments and return address are not exactly the same as in the previous stack frame.

Click on the Run icon again, and once again watch data get copied into the stack. Repeat this as many times as necessary to get an idea of how the sequence of recursive calls gradually accumulates the sum of all the elements of the array `x` in `main`.

What to Hand In

Nothing.