ENCM 369 Winter 2023 Lab 5 for the Week of February 13

Steve Norman
Department of Electrical & Software Engineering
University of Calgary

February 2023

Administrative details

Administrative details

You may work with a partner on this assignment

If you choose to work with a partner, please make sure that both partners fully understand all parts of your assignment submission, and please follow the instructions regarding submission of your PDF document.

Partners must be in the same lab section. The reason for this rule to keep teaching assistant workloads balanced and to make it as easy as possible for teaching assistants to record marks.

Special Due Date

This lab should not be difficult to complete during the week of February 13–17. However, because of the upcoming Winter Break week, the Due Date has been extended to the day classes resume after the break.

The Due Date for this assignment is 6:00pm Monday, February 27.
There is no Late Due Date—please submit your assignment by the 28th.

How to package and hand in your assignments

You must submit your work as a *single PDF file* to the D2L dropbox that will be set up for this assignment. The dropbox will be configured to accept only file per student, but you may replace that file as many times as you like before the due date.

See the Lab 1 instructions for more information about preparing and uploading your PDF file.

Important update for those working with a partner: Please submit only one PDF file for both students. On the cover page, put lab section, name and ICID information in this format:

> Group Submission for [lab section]
>
> Submitted by: [submitter's name]
> UCID: [submitter's UCID]
>
> Partner: [partner's name]
> UCID: [partner's UCID]

Marking scheme

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

Getting files for this lab

The files you need for this week's exercises are in a folder called `encm369w23lab05`, which has been uploaded to D2L in `.zip` format. Before starting work, you must download the `.zip` file, and extract its contents in a suitable place in the file system.

Exercise A: Practice with logical instructions

Read This First

All of the instructions used here are described in Section 6.3.2 of the course textbook.

What to Do

Suppose that before the following instructions are run, `t5` contains `0x6a00_9005` and `t6` contains `0x0fff_a000`.

```
lui     t0, 0x0b670
srli    t1, t0, 5
slli    t2, t0, 3
or      t3, t5, t6
andi    t4, t5, 0x3ff
xor     s0, t5, t6
xori    s1, t5, -16
```

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 Include in Your PDF Submission

Include well-organized 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 RARS and RISC-V assembly language, and will generate appropriate machine instructions to copy an address (with `la`) or a 32-bit constant (with `li`) into a GPR.

RISC-V assembly language offers 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` and bar are labels for a words allocated in with `.word` in a `.data` section in an assembly language file. It's possible to write this in a `.text` section:

```
sw      t0, foo, t1
```

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 pseudoinstruction must of course name the data source register, but it must also name another register (in this example, `t1`) to use in building the store address. Here is another example pseudoinstruction:

```
lw      s0, bar
```

In both examples it turns out that an assembler can generate code that is more efficient than using `la` followed by a store or a load.

RARS processes both of the above operations using an instruction called `auipc`—add upper immediate to PC. The assembly language syntax is

`auipc` *dest*, *imm*

where *dest* is a GPR and *imm* is a integer constant that can be encoded in 20 bits. The instruction shifts *imm* 12 bits to the left, then adds the address of the instruction itself to generate a result. For example, if

```
auipc   t6, 0x3
```

were located at address `0x0040_00bc`, `t6` would get a value of

`0x0040_00bc` + `0x0000_3000` = `0x0040_30bc`

What to Do

Open the file `encm369w23lab05/exB/lab5exB.asm` in RARS, and note that the assembly language code uses `lw` and `sw` pseudointructions as described above. Assemble the code so that you can see the real instructions generated from the pseudoinstructions. Answer the following questions:

1. Show that the first `auipc` instruction will give `t1` a value of `0x1001_0000`.
2. Why is 16 the correct offset for the `lw` instruction?
3. Show that the second `auipc` instruction will give `t3` a value of `0x1001_000c`.
4. Why is 8 the correct offset for the `sw` instruction?

What to Include in Your PDF Submission

Include your answers to the questions in "What to Do".

Exercise C: Machine code for RISC-V branches and jumps

What to Do

Consider the following sketch of some RARS assembly language code:

```
L1:     lbu     t6, (s1)
        beq     t6, zero, L2

        many more instructions

        addi    s1, s1, 1
        j       L1
L2:     or      s3, s1, zero
```

Suppose that the address of the `lbu` instruction ends up being `0x0040_1034` and the address of the `or` instruction ends up being `0x0040_10ac`. 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 Include in Your PDF Submission

Include 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 in C 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;
for (p = a; p != past_end; p++) {
  do something with *p;
}
```

You now have some experience translating both kinds of loops into RISC-V 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]

Notes about timing code

In 2020 and earlier, students did this exercise by building executables and running them on computers in a specified lab to collect timing data. That way everybody would be collecting data from machines with identical hardware and software.

The room used for ENCM 369 labs Winter 2023does not have any university-maintained computers, and it would be pretty challenging to guide students through setting up timing experiments with all their varied hardware and software. So for this exercise you will use data I generated using a Linux system I have.

Here is an important thing to know: You might think that a loop to do a simple task, like finding the maximum `int` in an array of 100,000 elements, would always take the same amount of time to run on a given laptop or desktop computer, but that is *very much not true!* Here are some (not all) of the reasons:

[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.

- Memory systems of current laptops and desktops are very complex. The average amount of time needed to load an array element into a GPR might vary significantly from one run of the loop to the next.
- Processor chips can speed themselves up when they detect that a program is making intensive use of one or more processor cores, but then slow themselves down if the high-speed work is making the chip too hot to run reliably. (Intel's current marketing term for this is "Turbo Boost".)
- On operating systems such as Windows, macOS and Linux, many programs will be running concurrently along with the operating system kernel. Memory and processor use by some running programs can have a negative effect on the performance of other programs.

What to Do, Part I

Have a look at the files in the directory `encm369w23lab05/exD`
Read the `.c` files and the `.h` file to see what the program does. Pay close attention to the comment describing the use of the `clock_gettime` function.

Next, have a look at the file `no-opt.txt`, which is the output of the program when the executable is built with this command:

```
gcc main.c func.c
```

Note that this command does not request any compiler optimization.

You will notice that the CPU time measurements are not identical from one run to the next. Use the following method to estimate CPU time over ten runs of `use_pointers` and CPU time over ten runs of `use_indexes`:

> Take the average of the fastest three times for each function. Ignore the slowest seven 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:

- *What is your estimated CPU time is used by a call to* ***`use_pointers`****? What is your estimated CPU time is used by a call to* ***`use_indexes`****?*
- *If* ***`use_indexes`*** *is considered to be the "old" version of a function and* ***`use_pointers`*** *is considered to be a "new" version, what is the speedup?.* Use the definition of *speedup* given earlier in these instructions.

What to Do, Part II

Now have a look at the file `O2.txt`. This is output from an executable built with

```
gcc -O2 main.c func.c
```

This is what the GCC manual (https://gcc.gnu.org/onlinedocs/gcc/) says about the `-O2` level of optimization, compared the more basic `-O` level:

> Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to `-O`, this option increases both compilation time and the performance of the generated code.

Use the same best 3-of-10 method to estimate CPU time for each of `use_pointers` and `use_indexes`. Then answer the following questions.

- *What is the speedup of* `use_pointers` *with* `-O2` *optimization relative to* `use_pointers` *with no optimization?*
- *What is the speedup of* `use_indexes` *with* `-O2` *optimization relative to* `use_indexes` *with no optimization?*
- *What appears to be a more important factor in getting speed in simple loops—preferring pointer arithmetic to use of array indexes, or asking for compiler optimization?* Give a brief reason for your answer.

Read This for Part III

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

Consider the following code to add up some array elements:

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

for (p = a; p != q; p += 4) {
  sum += *p;
  sum += *(p + 1);
  sum += *(p + 2);
  sum += *(p + 3);
}
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` optimization does not do loop unrolling unless you ask for it explicitly.

What to Do, Part III

Now have a look at the file `unroll.txt`. This is output from an executable built with

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

This command asks for `-O2` optimization and also for loop unrolling.

Use the same best 3-of-10 method to estimate CPU time for each of `use_pointers` and `use_indexes`. Then answer the following questions.

- *What is the speedup of `use_pointers` with `-O2` optimization and loop unrolling relative to `use_pointers` with `-O2` optimization only?*
- *What is the speedup of `use_indexes` with `-O2` optimization and loop unrolling relative to `use_indexes` with `-O2` optimization only?*

What to Include in Your PDF Submission

Include answers to all the questions in Parts I, II, and III. Carefully show your work for all calculations.

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 6.5 of your textbook, and recent lecture notes 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 laptop and desktop computers support what known as the *x86-64* ISA. (Macs that are more than two years old have Intel x86-64 processors, but Apple now makes Macs with M1 and M2 processors that use a completely different 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.

You'll be looking at x86-64 instructions as you work through this exercise.

What to Do

As with Exercise D, it would be great to be able to ask to do this exercise in a specified computer lab, so that all students would see the same results from commands to run the different steps in a C development toolchain.

That's not possible in Winter 2023, so instead you will look at some files I prepared using my Linux system using source code from Exercise D.

In a terminal window, change directories to `encm369w23lab05/exE`
Now follow this sequence of steps.

1. The file `funcs.i` was generated using the command

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

   The `-E` option says "preprocess only"; the `-o` option is used to specify the name of the output file. Use the command `less funcs.i` to have a look at the file `funcs.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 `funcs.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 "funcs.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. The file `funcs.s` was generated using the command

   ```
   gcc -S funcs.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 `funcs.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 RISC-V assembly language syntax and (b) the x86-64 instruction set is very different from the RISC-V 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 `for` loop in `use_pointers` will appear as 12 instructions starting with `jmp  .L2`—note that the test `p != end` to the botton of the loop. *Answer this question:* What are these 12 instructions? Just list them—you don't have to explain what they do. (By the way, the last three of the 12 instructions say "if `p != end`, goto `.L4`".)

3. The file `funcs-disassembly.txt` was generated with these two commands:

   ```
   gcc -c funcs.s
   objdump -d funcs.o > funcs-disassembly.txt
   ```

   The `-c` option says "generate an object file but do not go on to run the linker". The output will be a file called `funcs.o`. This file is *not* a text file, so it's not helpful to try to view it with `less`. Instead it can be inspected using the `objdump` program, which can get information from object files and display it in human-readable form. The `-d` option is for "disassemble"; what you'll see is hexadecimal representations of 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 the version of RISC-V we use in ENCM 369, where all instructions are four bytes long.

   Answer this question: What is the machine code for the instruction `addl $0x1,-0x8(%rbp)` ? Write your answer as sequence of bytes written in hexadecimal. (By the way, this instruction is the translation of `i++` in `use_indexes`.)

4. I repeated 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>` and `<time.h>`.
   - The assembly language file `main.s` is *not* very large—none of the type declarations and function prototypes in `<stdio.h>` and `<time.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_gettime` and `printf`.

   Answer these questions:

   - The line

     ```
     for (i = 0; i < N_ELEMENT; i++)
     ```

     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>` and `<time.h>`.)
   - What is the machine code for the instruction `cmpl $0x1869f,-0x3c(%rbp)` ? (This appears about 15–20 instructions from the beginning of `main`—it's an instruction that compares the immediate value `0x1869f`, which is 99,999, with the value of `i`, located in memory at address `-0x3c(%rbp)`. The instruction is part the loop that assigns values to elements of the array `arr`.) Write your answer as sequence of bytes written in hexadecimal.

5. Note that the command

   ```
   gcc funcs.o main.o -o exE
   ```

   could now be used to build an executable called `exF` from the two object files and pieces of library files.

What to Include in Your PDF Submission

Include well-organized answers to all of the questions asked in What to Do, Steps 2, 3, and 4.