The goal of this lab is two-fold. ① You should leave this section feeling comfortable that you can build, run, and debug all of the tools that we will be using in this course. ② You will reinforce some of the abstract concepts that we described in class with respect to the design of an assembly language.

1 Having a Stable Environment

In this course, there is no standardized IDE (e.g., VSCode, Eclipse, PyCharm, etc) in which you will do your development. However, the software for this course does assume that you have access to a Linux-based command line to build, run, and debug your environments. If you do not have access to a personal device with this available, the machines in the Lab (Edmunds 105) have all of the software available for you to use.

The projects that you will work from have various degrees of "finicky-ness" with respect to the system software installed and used. To help alleviate some of these configuration headaches, you are provided access to both a course VM and a course Docker container. Both environments have the appropriate software installed, and both certainly have their strengths and weaknesses.

If you have a Linux-based device and would like to install packages natively and locally outside of these environments, you can do so by installing the following (though this **is not required or necessarily recommended**.)

```
[sudo] apt install build-essential \
    # install dependencies for gem5
    scons python3-dev git pre-commit zlib1g zlib1g-dev \
    libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-dev \
    libboost-all-dev  libhdf5-serial-dev python3-pydot python3-venv python3-tk mypy \
    m4 libcapstone-dev libpng-dev libelf-dev pkg-config wget cmake doxygen \
    clang-format gdb lldb \
    # install dependencies for riscv gnu toolchain
    autoconf automake autotools-dev curl python3 python3-pip python3-tomli libmpc-dev \
    libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool \
    patchutils bc zlib1g-dev libexpat-dev ninja-build git cmake libglib2.0-dev libslirp
    -dev

# download and install gem5
git clone https://github.com/gem5/gem5.git /path/to/gem5
cd /path/to/gem5 && scons build/RISCV/gem5.debug -j4
cd ~

# download and install riscv toolchain
git clone https://github.com/riscv/riscv-gnu-toolchain /path/to/riscv-gnu-toolchain
cd /path/to/riscv-gnu-toolchain && ./configure --prefix=/opt/riscv && make
cd ~
```

If you intend to use Docker, please see the Docker resource on the course page for setup and usage. If you intend to use the Virtual Machine, please see the VM resource. Once you've gotten through this step, congratulations 🥳! You will use this environment for labs and assignments throughout the rest of the term.

2 Exploring Instruction Set Implications

In this lab, you will be getting used to writing various C++ programs to explore the implications of various instruction sets. In particular, you will dig into how each instruction set implements common high level programming language constructs and syntaxes. In doing so, you will use various tools to examine the implications of the binary states.

In this section, you may find it useful to use the Docker container. If you do not have space on your device to run the container, consider using the lab machines or working with someone who does.

Deliverable A goal of this lab is to get comfortable with C++ programming and assembly languages. However, this lab does not require that you submit anything, so you may use any tools (AI or otherwise) that you see fit to complete the tasks as long as it benefits your learning. You will be asked to complete an ungraded short reflection at the end of the lab to help me refine this exercise for future iterations – please describe your experiences there!

I would encourage you still follow the tasks and write your own code to build the mental-to-physical circuitry models that will refine your intuition.

2.1 Exploring Looping

In this section, you are asked to first write a C++ program that uses *loops*. You are encouraged to write and re-write the same program using various looping strategies to examine how your compiler translates these different implementations into assembly. For each section, you may find it useful to make note of how the compiler for the various ISAs interprets the various looping strategies.

In this section, you will implement a program that aggregates the outro lyrics to *Hey Jude* by the Beatles (see 3:09 to the end). In particular, the outro repeats the lyrics "Na-na-na-na-na-na-naa Na-na-na-naa, hey, Jude" twenty-four times before the song ends.

In doing so, please implement three programs: `jude-1.cc`, `jude-2.cc`, and `jude-3.cc` that use **for-loops**, **while-loops**, and **do-while loops** respectively. You may find the `string` library particularly useful in this exercise.

Building an Initial Target After you've written `jude-<n>.cc`, compile it to each of the target assemblies using the various compilers. The `arm64` compiler is the native compiler, since your container emulates an `arm64` host. First, use the command `g++ -S jude-<n>.cc -o jude-<n>.arm64` and examine the output. Then, add the `-g` flag. What changes between these files? What does `-g` do?

Comparing Outputs Given your previous answer, use the appropriate flags to build the appropriate targets for the `riscv64` and `x86_64` assemblies. How would you know which compiler to use? You may want to look at the output from `ls /usr/bin/*g++*`.

2.2 Conditionals

In the coda of *Hey Jude*, you may notice that Paul McCartney rhythmically adlibs all sorts of variations of the main body lyrics to the background of the coda. To add this behavior to your lyrical output, add an if-statement that, with some probability, adds a creative adlib to your lyrical output  . You may find the `<random>` library helpful!

Binary Disassembly Unlike in the previous step, compile your source to the full executable byte code. Test that your code works by running it! What error messages (if any) do you get for the non-native (i.e., `x86` and `riscv`) compilations? What might explain these error messages (or lack thereof)?

After you have built your executables, you will examine the output by *disassembling* it. This means transforming the binary format into readable assembly. To do so, you can use the `objdump` Linux command line tool. If you want to read it in the command line, you can use the `less` utility by running `objdump -d <binary file> | less`. If you want to open this up as a raw file, you can print the output to a file by running `objdump -d <binary file> > <output file name>`. This will allow you to open and search the file.

Comparing Outputs For simplicity, navigate to the header for the `main` function of your disassembly (note, this is different than the "main start" libc function). You will see several fields for each line here: the left-most column describes the program offset (i.e., what the program counter will refer to when reading that instruction), the next column are the raw bytes that the current instruction is represented as a hexadecimal number, after that is the actual instruction and the inputs to that instruction (typically registers, addresses, immediates, etc).

Once again, use the disassembled outputs to find your conditional calls and compare the instructions used to implement these high-level semantics in assembly across the languages. What is similar? What is different? You may find it helpful to think about expressiveness, variety of instructions used, instruction size (in terms of bytes), variation in instruction size, etc.

2.3 Vectors and Deques

As you will have noticed by this point, the `string` library in C++ makes managing strings much easier than in C. C++ has other libraries that help make managing containers of dynamically sized objects much easier. One of the most common is `std::vector`. A `vector` implements a *stack* (i.e., last in, first out), and supports `push_back` and `pop_back` modifiers. There are two main ways of iterating across a `vector`, iterating from 0 to `v->size()` and accessing elements as `v[i]` or using an *iterator*.

Iterators are across C++ objects are typically accessed using calls to `obj->begin()`. They tend to have very verbose syntax, so the standard convention to use iterators in a for-loop is as follows:

```
std::vector<int> obj;

for (auto it = obj.begin(); it != obj.end(); it++) {
    // execute code!
    int val = (int) *it;
}
```

There are a few things to notice here: ① when we declare the vector, we have to declare the type of values that the vector will maintain. At present, this loop will never enter the loop body because the vector is empty, so `obj.begin() == obj.end()`. ② The loop uses `auto` to instantiate the `it` variable which tells the compiler to infer the type of variable at compile time. *This can often lead to errors!* You can always spell out the full type of `std::iterator<std::vector>` to ensure the compiler catches any unintended declarations. ③ We can extract the value from an iterator by *dereferencing* it. Iterators act as a wrapper for elements in a container where the star (`*`) operator extracts the data. You'll notice the `++` operator is similar to calling `it = it->next` at the end of the loop.

Another helpful data structure in C++ is `std::deque`. A deque (or double ended queue) acts very similar to a vector, but has `push_back`, `push_front`, `pop_back`, and `pop_front` interfaces.

Using C++ Data Structures Modify your `jude-<n>.cc` files with the Paul McCartney adlibs so that, rather than merely appending to a string, first populates a `vector` or `deque` full of "Na-na" strings. Then, iterate through the elements in the data structure using the indices to randomly assign adlibs with random probability.

Afterwards, use an iterator-based loop back through the structure to extract elements and print out the data from each element of the container.

Analyzing the Output Take a minute to examine the produced binary (either by compiling with `-S` or using `objdump`) to understand how the data structure is being produced.

In addition, compare the resulting binary size once you've compiled your target for each of the ISAs. You can do so using the `du` (disk utilization) command line utility. For example, you may say `du -sh jude-1.x86` to see how many bytes that binary uses. What do you notice about the size of the outputs for the various instruction sets? What contributes to the binary size?

Notice, the `vector` and/or `deque` is created as a *heap* variable. How is this represented in the assembly? If we were to create an array, how might this be represented in the binary? Remember, the stack is maintained by the processor using the `sp` register, a special register referring to the bottom of the stack.

3 Summary

In this lab, you setup your development environment and reinforced concepts about assembly languages that we talked about in class. In addition, you got to practice developing toy programs in C++ and used common C++ libraries (which make it much easier to use than C!) to simplify the tasks. You were also exposed to several ways to compare program outputs across various compilers and explore different strategies to compare the trade-offs between

Before you leave, please fill out the following survey to help me refine this lab for future versions of this class. Thank you for your hard work!

`https://forms.cloud.microsoft/r/rb74cSs3Ld`