a86 to arm64 Project Summary

As origninally proposed, my final project (which I call `a86->arm64`) is a translator from the course's a86 language to arm64 assembly (sometimes known as aarch64). In short, this project consisted of two tasks: writing a drop-in replacement for the `a86/printer` module, writing a script to patch the `Makefile` and `compile-stdin.rkt` in the course languages, and adapting the Loot tests for this new system. The most major one of those steps, the replacement printer, is described below. Additionally, `src/printer.rkt` contains numerous detailed comments about how everything works.

The Replacement Printer

The core of this project is the replacement for `a86/printer` found in `src/printer.rkt`. At a high level, this new printer has been modified to emit functionally equivalent arm64 assembly which when assembled using the `as` command that is a part of the Xcode Command Line Tools.

The first thing that I wrote for `a86->arm64` was the register map – essentially a map from x86 registers used in a86 to arm64 registers that serve similar purposes. For example, the equivalent of `rax` in arm64 is `x0` because both are used to hold the return value of a procedure call.

Next was the fun part, translating every instruction used by the course languages (through Loot) from x86 to their arm64 equivalents. Some instructions like `add` and `sub` were trival and only required minor adjustments to work correctly. Other instructions took searching the aarch64 ISA manual, scouring StackOverflow, and long LLDB sessions. The more interesting (and only moderately painful) translations I wrote are described below.

- **Every instruction used to work with the stack:** An interesting feature of arm64 are the `stp` (store pair) and `ldp` (load pair) instructions, which load and store a pair of registers from and to memory, respectively. As a result, clang uses these instructions liberally. However, both instructions require the stack pointer to be 16-byte aligned. This means that any time I want to use anything that C generates or use either instruction myself, the stack has to be in 16-byte alignment. This is incompatible with how our languages use the stack, so I just doubled the size of every stack unit and keep the value on the right side. This means that even the so-called trivial instructions `add` and `sub` needed to have a special case for if the register being added to was `rsp`.
- **`mov`:** On arm64, the `mov` instruction is far more limited, it only allows moving from one register to another or moving an immediate between 0 and 4095 (inclusive) into a register. As such, mov had to be rewritten to conditionally translate to instructions like `ldr`, `str`, `ldp` and `stp`.
- **In-language function calls:** Function calls from the course languages

to the runtime system were actually quite trivial. Surprisingly, it was the function calls within the course languages that were the most difficult to translate. The reason being is that arm64's `ret` does not pop the stack, instead it reads the "link register" `x30` to find out where to jump. In impletementing this, I didscovered that I was making an assumption about C function calls that wasn't necessary and realized that I should just save the stack frame info at the begining of the program. This meant that I could translate `ret` into popping the link register and then doing a `ret`.

Running `a86->arm64` yourself

To run this project locally, you will need:

- An Apple Silicon Mac (preferably), or an ARM Linux PC (such as a Raspberry Pi)
- The LLVM toolchain (with aliases for `gcc` and `as`)
  - On a Mac, this requires the XCode Command Line Tools
  - On Linux, I don't know how one would go about aquiring this setup
- Racket (the arm64 version that one can get on a Mac using `brew install racket`)
- The `langs` package
  - As a direct result of `langs` being meant for x86, the automated tests that raco runs will fail. This is fine. Regardless of failure, the important parts of the langs package will be installed and that is all that matters.

Once you have all of that, you can run `scripts/init.sh` in your terminal to check that you have everything set up and also to bootstrap some parts of the project.

Since I have completely reworked how building works from x86, I could not actually use the provided tests verbatim. As such, I adapted the I/O-free tests from Loot to test my translator. To run these tests, you can run `scripts/run_tests.sh`, it should just output a chain of text that says `PASS` repeatedly.

Of course testing is important – even in a unique final project like mine, but the place where this project really shines (in my opinion at least) is in being able to play around with any course language through Loot! To do this, run `scripts/copy_course_lang.sh <language name>`, replacing `<language name>` with the name of the language you want to try (in all lowercase). Once that finishes, you can `cd` to `courselangs/<language name>` to play around with the language's compiler. To compile a source file written in one of the course languages use `make -f arm.mk` in lieu of just running `make` (otherwise you would be running the origninal Makefile, which would fail).