x86prime

A few tools for our x86 subset, x86prime, known as just "prime"

Before anything else

Clone this repo into a directory. The rest of this description assumes you've done so and are executing commands in said directory.

Prerequisites

Install ocaml and opam (ocaml package manager). This installs the default compiler version for the system.

On debian based linux this is

```
> sudo apt install ocaml opam
> opam init -a
```

The "opam init" command may ask if you want to update your path. If so, say yes. Read any messages carefully, just pressing return to questions asked, may not give you what you want.

On some systems, the default ocaml version is too old. Use "ocaml --version" to find which version you have. If you have something before 4.07.0, you need to upgrade to at least version 4.07.0. If you have 4.07.0 or later, you should not need to upgrade the ocaml toolchain. To upgrade do:

```
> opam switch 4.07.0
[at this point you may be asked to run "eval 'opam config env'" - do it]
```

On some systems, this is not enough, and we recommend that you exit your shell, then restart it before proceeding

You may have to use "opam switch create" instead of just "opam switch". This depends on the version of opam.

Next, install the tools "menhir" and "dune":

```
> opam install dune menhir
```

Building

We use 'dune' for building. Use the command "dune build" to build the four tools:

- primify, a tool which translates real x86 assembler into prime assembler
- prasm, the assembler, a tool which encodes prime assembler into a format known as "hex"
- prun, a tool which reads hex files and runs them
- prerf, like prun, but collect performance statistics

```
> dune build
```

The tools will be placed in the "_build/default/bin" subdirectory with a ".exe" suffix. You may want to create links to them, so that you don't need to type the full file name. For example

```
> ln -s _build/default/bin/primify.exe primify
```

and so on.

If by accident you have built part of the program using an old version, you may get an error during build indicating a version problem with part of the project. If this happens, remove the "_build" subdirectory.

And build again.

Generating x86 assembler

x86 assembler is in files with suffix ".s"

You can write one yourself. Or generate one from a program written in "C" using a C-compiler.

```
> gcc -S -Og -fno-optimize-sibling-calls -fcf-protection=none my_program.c
```

Translating x86 into prime (x86prime)

The "primify" program will translate an x86 assembler source file into correspondingly named ".prime" files.

```
> bin/primify my_program.s
```

This results in a new file, "my_program.prime"

Encoding into hex format

The simulators cannot directly read the symbolic prime assembler. You need to assemble it into hex-format. Use

```
> bin/prasm my_program.prime
```

This produces "my_program.hex", which can be inspected to learn how the prime program is encoded in numbers.

It also produces "my_program.sym", which is a list of symbols. This is used by the simulator to allow you to pick which part of the code to execute.

Running

Programs are simulated by "prun"

```
> bin/prun my_program.hex my_start_function
```

Here, the label "my_start_function" must have been defined by the original ".prime" program.

Without more options, the simulation is silent. To see what happens, add "-show" option

Sit back, relax and watch the blinkenlights.

Generating a tracefile

A tracefile records all changes to memory and register made by your program. You request a tracefile by the "-tracefile" option:

```
> bin/prun my_program.s my_start_function –tracefile prog.trc
```

The subdirectory "examples" includes an example C program (bubblesort.c) which can be used as introduction to the tools. It has 2 entry-points, "run" and "run2". If you use "run" it will silently input an integer from the keyboard. If running it seems to just hang, try providing a small number and press enter. If you use "run2", it will instead try to access any command line arguments. To pass command line arguments into the simulated program, just add them to the command line after all the other arguments. The bubblesort.c program illustrates how the program can access the additional arguments, see the "run2" function.

Limitations to cross-assembling

The translation from x86 to prime is not perfect.

- When gcc optimizes heavily ("-O2, -O3"), the code patterns generated will not be translated correctly. In most cases primify will stop with an exception. We believe "-Og" to be working reasonably well, so stick to that.
- Tail-call optimization will result in code, which cannot be correctly translated into "prime", worse: this currently goes undetected - to disable it use "-fno-optimize-sibling-calls"
- Intel CFE (control-flow enforcement technology) produces code that primify cannot handle.
- to disable it use "-fcf-protection=none"
- When gcc needs to use almost all registers in a function, translation will either fail or just be incorrect.
- Using combinations of signed and unsigned longs may not be handled correctly.
- Using constants which cannot be represented in 32-bit 2-complement form may not be handled correctly.
- Larger "switch" statements can not be translated

In short, we advise you to check the translation result for correctness instead of blindly trusting it.

Performance modelling

The "prerf" tool runs a program much like "prun", but includes a performance model.

The model include a selection of branch predictors, a return predictor and 2 configurable levels of cache.

It supports 3 different microarchitectures:

- A simle scalar pipeline. "-pipe simple"
- A 3-way in-order superscalar pipeline. "-pipe super"
- A 3-way out-of-order (superscalar) pipeline. "-pipe ooo"

There are separate primary caches for instruction and data:

- Access latency i 3 cycles, fully pipelined.
- Size 16K, 4-way associative, 32-byte blocks

The primary caches are backed by a secondary cache:

- Access latency 12 cycles (on top of the 3 cycles in L1)

- Size 128K, 4-way associative, 32-byte blocks

The main memory is an additional 100 cycles away.

The "prerf" tool support generation of execution graphs (da: "afviklingsplot").

Some options:

- "-show" display execution graph
- "-profile" display execution profile. Shows the execution count and average execution latency for each instruction.
- "-help" show list of options. Use this to see options for selecting performance model details.
- "-print_config" show configuration of microarchitecture and memory hierachy