MINIPS Fase I & II - Relatório

Eduardo Renesto

Arquitetura de Computadores 2021.1 - Prof. Dr. Emilio Francesquini

Dados

- **Nome:** Eduardo Renesto Estanquiere
- **RA:** 11201810086
- **Usuário do GitHub:** `EduRenesto`
- **Link do vídeo:**
  - *Fase I*: https://www.youtube.com/watch?v=ObtdlKSBpvQ
  - *Fase II:* https://www.youtube.com/watch?v=6ZF_8dQqiiI

Introdução

Nesse texto apresento minha experiência desenvolvendo o `minips-rs` [1], minha implementação do projeto do quadrimestre da disciplina.

Antes de tudo, afirmo que me diverti bastante no processo. Antes da disciplina, eu já havia começado a escrever um emulador para o *NES*, e tinha uma implementação do MOS 6502 razoavelmente funcional. Assim, já tinha em mente uma ideia de arquitetura para esse projeto, além de uns "reflexos" sobre o que fazer e não fazer.

Dificuldades

Fase I

Embora o desenvolvimento tenha sido relativamente tranquilo, as minhas maiores dificuldades se deram durante a implementação dos branches. Entender o cálculo do alvo, apesar de ser simples, olhando em retrospecto, demorou um pouco: tive alguns problemas ao levar em consideração que, na minha implementação, o *program counter* sempre é incrementado em 4 a cada instrução, independentemente de qual foi executada. Eventualmente, comparando a execução do meu código com o `minips` de referência, consegui fazê-los funcionar.

Também tive problemas com interpretação de inteiros – quando assumir que um valor é com ou sem sinal. Isso se deve ao fato de eu ter assumido que as instruções `ADD(I)U` apenas tratavam com inteiros `unsigned`. Teoricamente, como a soma de inteiros com e sem sinal, em complemento de 2, é igual bit-a-bit, não haveria problema. No entanto, como eu estava sempre fazendo extensão com zeros para os 32 bits, por achar que eram unsigned, o resultado não era o esperado. Depois de leitura do *greencard*, e escrevendo um assembly simples para teste, foi possível corrigir esse problema.

Faço uma observação: *debuggar assembly é difícil* `:P`, ainda mais sem um debugger no cliente. Imprimir a instrução atual, bem como os valores dos registradores, em alguns momentos foi útil em algumas situações.

Fase II

O desenvolvimento da fase II se deu sem problema nenhum – a implementação, graças ao sistema de geração de instruções prático, se deu ao longo de menos de uma semana.

No entanto, mais uma vez a maior dificuldade foi alinhar os branches, especialmente com o *branch delay slot*. Além disso, encontrar uma maneira *bonita* para implementar essa propriedade não foi tão trivial quanto eu inicialmente imaginava.

[1] nome sujeito a mudanças (eventualmente)

Orgulhos

Fase I

Depois de três disciplinas com o prof. Emilio, sinto que esse é o primeiro projeto que o entrego com orgulho e satisfeito. `:D`

Sobretudo, considero esse projeto como uma das bases de código Rust mais maduras que já escrevi. Além do modelo de *ownership* que funcionou *bem legal*, elenco duas principais faces: o uso de *idioms* e o de *macros*.

Em relação ao uso de *idioms*, o código inteiro faz tratamento de erros usando a monad `Result<T,E>` ("equivalente" ao `Either a b` no Haskell). Ainda, com a crate `color_eyre`, pude usar o operador `?` ao longo de todo o código e ter mensagens de erro bem bonitinhas e controladas. Experimente usar uma instrução não implementada, por exemplo!

Talvez o uso mais bonito de *idioms* aqui se deu no tratamento dos registradores. Usei dois `newtypes`: o `Registers`, que representa o conjunto dos 32 registradores MIPS e simplesmente encapsula um array de 32 inteiros de 32 bits, e o `Register`, que encapsula um "índice" de registrador, que é apenas um inteiro de 32 bits.

O segundo orgulho, talvez o maior, é a macro `instr_from_yaml` que pode ser encontrado no arquivo `minips-macros/src/lib.rs`. Meu workflow para implementar novas instruções até antes dessa macro era um pouco *janky*: declarar a instrução no `enum Instruction`, escrever a implementação do `Display` para o disassemble da instrução, adicionar mais um caso no decoder e finalmente implementar a execução da mesma. Bem repetitivo e verboso.

Como bom programador que adora gastar mais tempo automatizando a tarefa do que a fazendo manualmente, construí a (*procedural*) macro `instr_from_yaml`.

Ela recebe um arquivo `YML` contendo todas as instruções que o emulador reconhece, junto de propriedades das mesmas. A macro, então, gera a declaração, a implementação de `Display` e a implementação do parser para cada uma dessas instruções.

Dessa maneira, só adicionar a nova instrução no arquivo `instructions.yml` já é suficiente para que o emulador reconheça uma nova instrução. Daí, só falta implementar uma vez a instrução no arquivo `cpu.rs` e tudo funciona.

Fase II

A infraestrutura da macro de instruções foi atualizada para utilizar instruções dos tipos `FR` e `FI`. Isso tornou possível a rápida implementação das novas instruções para essa segunda fase, o que entendo como um triunfo da ideia da macro.

Continuando com o tema de `newtypes`, os registradores de ponto flutuante foram implementados na própria estrutura da CPU (`src/cpu.rs`), exatamente da mesma maneira que os registradores de propósito geral. A saber, foi implementado um *newtype* `FloatRegisters` que armazena 32 **inteiros sem sinal de 32 bits**. Preferi fazer desta maneira e fazer as conversões manualmente quando necessário, especialmente por `floats` e `doubles` compartilharem os mesmos registradores.

Esse *newtype* `FloatRegister` é indexado por um `FloatRegister`, que também implementa o pretty-printing, exatamente na mesma maneira que a dupla `Registers` e `Register` fazem para os GPRs.

Além disso, as funções de conversão entre os tipos numéricos foram otimizadas utilizando código `unsafe`.

Em geral, fiquei muito satisfeito com a performance do emulador ~~que é melhor que a da implementação do professor~~.

Experimentos Futuros

Desde a fase I, pretendo programar um framebuffer mapeado em memória, e eventualmente rodar o kernel Linux no emulador. Isso provavelmente será feito na próxima fase, quando implementarei paginação e caches, criando um controlador de memória que suportará mapeamento.

Ainda, durante essa fase fiz alguns experimentos tentando interpretar código escrito para o *PlayStation* no emulador. Como prova de conceito, consegui com sucesso fazer o disassemble de uma seção de código advindo da BIOS de tal console.