# MINIPS Fase I - 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:** https://www.youtube.com/watch?v=ObtdlKSBpvQ

## Introdução

Nesse texto apresento minha experiência desenvolvendo a primeira fase do `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

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.

## Orgulhos!

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.

[1] nome sujeito a mudanças (eventualmente)

Criei o newtype `Register` por uma razão: para facilitar o *pretty-printing* de instruções e registradores. Implementei a trait `Display` nesse novo tipo e nela fiz a conversão entre índices de registradores e seus aliases. Assim, o código

```
println!("{} {} {}", Register(0), Register(2), Register(10))
```

Gerará a saída:

```
$zero $v0 $t2
```

Semelhantemente, implementei o `Registers` para poder indexar o conjunto de registradores utilizando um `Register`. Isso tornou a implementação da execução das instruções bem amigável!

Esse jogo dos newtypes pode ser encontrado nos arquivos `src/emulator/{mod,cpu}.rs`.

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.

Em geral, me diverti bastante escrevendo esse projeto. Eu tenho bastante apreço por *baixo nível*, e me deixei levar um pouco – quis rodar programas escritos em C no meu emulador. Inicialmente, o código era compilado e *marretado* para que os endereços fossem calculados corretamente, e então era usado o `objcopy` para criar a dupla de arquivos `.{text,data}` que o emulador sabe carregar.

No entanto, essa workflow não era nada ideal e flexível. Portanto, utilizei a crate `goblin` e implementei o carregamento de arquivos ELF. Cada seção é carregada na memória do emulador no endereço correto, e o *program counter* inicial é definido como o *entry point* do arquivo ELF. Desse modo, o `minips-rs` consegue executar programas que foram gerados pelo linker sem nenhuma modificação. Essa feature pode ser achada no arquivo principal, `src/main.rs`.

## Experimentos Futuros

Com as próximas fases do projeto da disciplina, pretendo implementar exceções e mapeamento de memória. Tendo isso, consigo implementar um framebuffer simples (basta usar, por exemplo, a crate `minifb`, mapear um buffer na memória do emulador e fazer uma magia de sincronização para copiar esse buffer mapeado para o buffer do `minifb`).

Minha ambição final, então, é *rodar Linux e Doom* no emulador. Tenho alguma experiência com o kernel em embarcados, e até onde sei basta apenas criar uma device tree para o emulador (nela citando, por exemplo, o framebuffer simples) e um `tty` driver que utiliza as `syscalls`, e carregar esse pacotão utilizando o ELF loader já implementado.