

# MINIPS Fase II - Changelog

Eduardo Renesto

Arquitetura de Computadores 2021.1 - Prof. Dr. Emilio Franceschini

## O que é isso?

*LucasSilva: O relatório desse EP2, deve ser o outro + as implementações novas, ou só descrever o que foi mudado no código? franceschini: Complemente o anterior.*

Eu havia escrito esse documento como o relatório da fase II, antes de ler a mensagem. Como contém bastante coisa interessante, não quis apagar. Portanto, o presente documento fica como um *changelog* mais descontraído sobre o que mudou no emulador da fase I para a fase II.

## Introdução

Nesse texto apresento a segunda fase da implementação do minips-rs. Essa fase se resume em apenas duas grandes features: o *branch delay slot* e a implementação de partes do coprocessador 1 (responsável por operações de ponto flutuante).

Em geral, essa fase foi muito menos trabalhosa que a anterior – o emulador já executava todas as entradas depois de cerca de dois dias de trabalho.

## Desenvolvimento

### Infraestrutura

Nessa fase, todo o esforço feito no final da última fase para deixar a declaração de instruções automática foi pago. Como as instruções de ponto flutuante são encodadas de maneira diferente<sup>1</sup>, foi necessário fazer alterações na crate minips-macros para entendê-las do arquivo yml e então gerar toda a infraestrutura a partir delas. Essa parte do desenvolvimento envolveu basicamente só *copy-paste* de código que já estava pronto mais algumas poucas modificações.

Adicionalmente, comecei a implementar o *statistics counter* que apresenta estatísticas da simulação ao final da execução. Para destilar os tipos de instruções, adicionei a fase kind (minips-macros/src/kind.rs) que simplesmente implementa a enum Kind em cada instrução. Essa enum armazena o tipo da instrução – R, I, J, FR, FI.

### Dizia eu que a aritmética.....

Os registradores de aritmética são um novo atributo na struct Cpu: a tupla arith\_regs. O primeiro valor representa o registrador LO, e o segundo, o HI. As implementações das instruções em específico são triviais.

### Floating Point (cop1)

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**. Prefiro 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.

Para as computações com ponto flutuante, converto para os tipos nativos de ponto flutuante da linguagem (f32 e f64) e utilizo as operações nativas.

---

<sup>1</sup> *de jure* diferente, mas *de facto* igual às outras

## Branch Delay Slot

O branch delay slot foi implementado utilizando dois novos atributos na struct `Cpu`: o `branch_to`, que é setado para `Some(next_pc)` depois de toda instrução de branch que foi tomada, e o `in_delay_slot`, que é setado para `true` toda vez que um ciclo inicia e `branch_to` não é um `None`. Ao final da execução do ciclo, se `branch_to` não é `None` e `in_delay_slot` é `true`, o program counter recebe o valor anteriormente armazenado em `branch_to` e essas variáveis são resetadas para `None` e `false`. Caso esse não seja o caso, o program counter é apenas incrementado como sempre.

## Otimizações

As funções de conversão entre os tipos numéricos relevantes ao projeto, que estão no arquivo `src/cpu.rs`, são utilizadas amplamente durante a interpretação das instruções. Durante a fase I desse projeto, essas funções utilizavam um encadeamento de `from_le_bytes(to_le_bytes(src))`.

Agora, essas funções utilizam uma estratégia mais *C-like*: ao invés de passar por funções que possivelmente façam cópias, utilizo `unsafe` e simplesmente reinterpreto os bytes da fonte como se fossem bytes do tipo desejado. A ideia é transformar uma referência para o valor fonte em um *raw pointer* do mesmo tipo da fonte, depois fazer um `cast` para um *raw pointer* do tipo de destino, e por final fazer a dereferência desse ponteiro. Pelas regras de segurança de memória da linguagem Rust, é necessário usar `unsafe` para esse último passo.



Figura 1: Ilustração fiel do processo de otimização.

Como sempre tenho certeza que os dados são válidos no tipo de destino, não existem problemas com essa estratégia.

## Orgulhos!

Não tive muito tempo para brincar com essa implementação, se comparado com a fase anterior. No entanto, consegui extrair uma parte da BIOS do *PlayStation 1*, e o projeto conseguiu pelo menos fazer o disassembly. :)

## Experimentos Futuros e Melhorias

Na próxima fase, pretendo implementar o coprocessador 0 e memory mapping. Assim, consigo o meu tão querido *framebuffer* e posso sonhar com rodar GNU/Linux no emulador.

Pretendo fazer mais um overhaul na infraestrutura de instruções, e utilizar unions + a biblioteca `bitvec` para poder fazer `match` em todos os campos possíveis de uma instrução durante o parsing. Assim, evito as funções de parse específicas para cada tipo de instrução, e o parser vira mais versátil para possíveis novos tipos.

Também pretendo abstrair coprocessadores – talvez utilizar uma `trait` e resolver tudo em compile time, no lugar de ter todos os registradores de todos os coprocessadores direto na struct `Cpu`.