Exercício Programa 1 — MINIPS

Matheus Campos Fernandes

09 de março de 2021

Dados e Links

- **Nome Completo:** Matheus Campos Fernandes
- **RA:** 23202110043
- **GitHub:** mcf1110
- **Link do Vídeo - E1:** `https://youtu.be/UAeIK1LLCng`
- **Link do Vídeo - E2:** `https://youtu.be/EL4T7GclFB0`

1 Implementação E1

A implementação do emulador foi feita em Haskell.

Utilizei uma abordagem de TDD, que trouxe como benefícios principais a facilidade de executar partes individuais do meu código sem precisar da aplicação inteira rodando; além de me dar a tranquilidade de que não "quebrei" nenhuma instrução anterior com a implementação de uma nova.

A princípio, achei que ia ser complicado modelar de forma imutável uma linguagem que, por definição, depende de mudanças de estado. Mas na verdade me surpreendi como não encontrei **nenhum** *bug* relacionado a isso, o que acredito que fatalmente aconteceria em uma linguagem imperativa.

Outro ponto positivo na escolha da linguagem foi como pude modelar as instruções por meio de *ADTs*, e resolver toda a parte da avaliação em um único *pattern-match*. Fico imaginando a pequena mostruosidade que surgiria caso quisesse usar uma Herança em *OOP*, ou mesmo usar *structs* simples e *Enums* para representar os tipos de instrução.

Em especial, acho que a parte do eval ficou bem bacana com o uso dos operadores. Por exemplo, poder escrever que

```
add rs rt rd = rd $<- rs $+$ rt
```

é algo raro em outras linguagens.

No geral, encontrei 3 *bugs* que demoraram mais que 5min para resolver, todos relacionados à conversão de dados. Um deles estava na forma de como

eu acessava as strings em memória, já que o acesso de uma *syscall* pode ser desalinhado. Foi um pouco complicado para fazer esse acesso funcionar e converter corretamente os bytes em *chars*, mas no final ficou uma solução melhor que a anterior. Os outros dois foram problemas de números com sinal negativo, um envolvendo a soma de dois números com um deles negativo, e outro envolvendo a comparação. Como o tipo que usei (`Word32`) sempre interpreta os números como *unsigned*, tive que tomar cuidados em alguns lugares para forçar a interpretação em complemento de 2, assim como simular o *overflow* quando este acontecia. Naturalmente, dois lugares passaram batidos e, portanto, dois *bugs* foram criados. Provavelmente vou encontrar mais alguns desses antes das próximas entregas, mas até o momento tudo leva a crer que está tudo funcionando bem.

Embora as correção tenham sido tranquilas de aplicar (depois que soube qual era o bug), o problema era que a única pista que eu tinha era "minha saída está diferente do MARS". Nesse processo de caçada de *bugs*, acabei desenvolvendo umas ferramentas que também achei que ficaram bem legais. Por exemplo, eu posso definir *breakpoints* para o meu programa, em que ele irá pausar a execução do programa e me mostrar o estado atual do computador. Esses *breakpoints* nada mais são do que endereços de memória que são comparados com o PC a cada iteração do estado.

2 Implementação E2

Ficou claro aqui que a abordagem TDD compensou. Durante a implementação, eu me via querendo reutilizar código antigo, e os testes me deram confiança pra refatorar sabendo que eu não ia quebrar o programa. A quantidade de instruções com também me levou a rearranjar os módulos do meu programa e dos testes, o que achei que ajudou bastante na organização do código.

Em especial, ocorreram dois bugs mais chatinhos, um deles envolvendo um $>$ em vez de $\geq$ na `blez`, e o outro em que eu tratava o *offset* como *unsigned* nas instruções de Load/Store (este último levando a algumas horas de frustração).