MINIPS Fase I, II & III - 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
  - *Fase III:* https://www.youtube.com/watch?v=LMXMN6oRTAw

Introdução

Nesse texto, apresento uma visão geral da arquitetura e do desenvolvimento do `minips-rs`, minha implementação do projeto do quadrimestre da disciplina.

O `minips-rs` é um emulador que implementa um subconjunto da arquitetura MIPS. Além disso, com a parte III, também é um emulador de memórias cache e apresenta estatísticas *cycle-perfect* da emulação.

Sobre a arquitetura do projeto, destacam-se três partes importantes: as instruções, a CPU e a memória.

Instruções

O tratamento das instruções é gerado programaticamente por meio da macro `instructions_from_yaml`, que está implementada na subcrate `minips-macros`. Essa macro consome o arquivo `instructions.yml` e gera a declaração, *parsing* e *pretty-printing* de cada instrução lá descrita.

A declaração se dá por uma `enum Instructions`. Cada instrução é uma variante desse enum, que encapsula instâncias das structs `{R,I,J,FR,FI}Args` contendo os índices de registradores para cada operação. Essa enum pode ser encontrada na documentação pelo nome `minips_rs::emulator::autogen::Instruction`.

O *decode* das instruções está implementado na função `Instruction::decode`, que recebe uma palavra de 32 bits e retorna a instrução decodificada caso ocorra sucesso. Essa função também é gerada pela macro.

O *pretty-printing* das instruções é feito pela implementação da trait `Display` na enum `Instruction`. Naturalmente, essa implementação também é gerada pela macro.

CPU

A CPU está implementada no arquivo `src/emulator/cpu.rs`. O *register file* é modelado usando o `newtype Registers`, que encapsula uma array de 32 inteiros sem sinal de 32 bits. Os registradores de ponto flutuante, do coprocessador 1, são implementados exatamente da mesma maneira, no entanto com o `newtype FloatRegisters`. Os dois conjuntos de registradores são indexados pelos `newtype Register` e `FloatRegister`, que encapsulam inteiros de 32 bits e são instanceados durante a decodificação das instruções.

A CPU também faz o papel de controlador de memória – ela é parametrizada sobre dois tipos que implementam a trait `Memory` (vide abaixo): `TI` e `TD`, que são o tipo da memória de instruções e de dados, respectivamente. É armazenada uma referência a um objeto de cada tipo por meio do construto `UnsafeCell` – sua necessidade será explicada na próxima seção.

A função `cycle` em `CPU` faz o *fetch* da instrução apontada pelo *program counter* atual partir da referência da memória de instruções, e então a executa. Eventuais acessos à memória são feitos por meio da referência da memória de dados,

como esperado. Ao final do ciclo, é atualizada a lógica para tratar do *branch delay slot*, que é implementado utilizando os campos `branch_to` e `in_delay_slot`.

Memória

A memória no `minips-rs` é totalmente abstraída, e as interações da CPU com as memórias são feitas exclusivamente a partir da trait `Memory`. Em suma, todos os tipos que implementem essa trait devem implementar os métodos `peek`, `peek_into_slice` para leitura e `poke`, `poke_from_slice` para escrita. Como a CPU é parametrizada sobre tipos de memória, a mesma estrutura pode ser utilizada para diferentes configurações de memória. Como isso é feito a nível de tipos, a configuração total é monomorfizada e otimizada em tempo de compilação, não havendo custo de performance atrelado a essa abstração.

Todos os métodos de acesso a memória recebem uma referência mutável do objeto – isso é necessário inclusive nas operações de leitura, como por exemplo na cache. No entanto, pode ser necessário o acesso desses métodos a partir de vários lugares diferentes, ou seja, podem ser necessárias várias referências mutáveis para o objeto sendo acessado. Isso é um problema: as regras de segurança de memória da linguagem Rust explicitamente não permitem múltiplas referências mutáveis para um mesmo objeto. Isso é resolvido utilizando o pattern de *interior mutability* – geralmente feito com a construção `Rc<RefCell<T>>` em contextos single-threaded, como o nosso. No entanto, essa composição gera overhead, portanto foi escolhido utilizar (com cuidado) referências simples a `UnsafeCell<T>`, que dá acesso ao ponteiro cru por baixo dos panos.

Nesse sentido, podemos falar da cache, implementada em `src/emulator/memory/cache.rs`. Ela utiliza deliberadamente *const generics*, um construto recentemente estabilizado na linguagem[1]. A struct `Cache` é parametrizada em quatro variáveis: `T`, o tipo de memória do próximo nível, `L`, o tamanho em palavras de cada linha da cache, `N`, o número de linhas da cache, e `A`, a associatividade da cache. Cada configuração é essencialmente uma combinação dessas variáveis, e também é monomorfizada e otimizada em tempo de compilação. A implementação é a mesma para todas as configurações – o código leva em consideração cada variável e reage de acordo.

A cache armazena uma referência para o próximo nível na hierarquia de memória e uma referência opcional para a cache irmã. Ambas utilizam, novamente, `UnsafeCell`.

A função central na implementação da cache é a `find_line`: ela recebe um endereço "físico", calcula o número de linha e tag para o endereço, e verifica se há alguma linha no *set* relevante que possui a mesma tag. No caso em que há, retorna um `Hit(index)`; no caso em que não há, utiliza a política de substituição de linhas e retorna um `Miss(index)`. A função que chama esse `find_line`, então, trata de executar o procedimento adequado dependendo desse resultado. As funções `flush_line`, `load_into_line`, `invalidate_line` e `try_copy_from_sister` agem sobre uma linha e fazem, respectivamente, o write-back, o carregamento a partir do próximo nível, a invalidação e a tentativa de cópia a partir da cache irmã, se presente.

Orgulhos e Dificuldades

Utilizar o sistema de tipos para representar as configurações se mostrou extremamente eficaz: no modo `release`, a implementação consegue executar a entrada `18.naive_dgemm` na configuração 6 em aproximadamente 1.8 segundos, em comparação com 30 segundos da implementação do professor.

A maior dificuldade da fase final foi fazer o debug do comportamento das caches. Para isso, a função `debug` da implementação referência e a utilização de logs foram essenciais.

Outra dificuldade se deu na modelagem do problema: cada *corner case* exigia uma mudança na maneira com que a abstração foi feita. Isso acabou fazendo com que o código (da cache, em especial) ficasse relativamente bagunçado. Infelizmente não há prazo ~~(nem energia)~~ para uma refatoração significativa do código.

Conclusão e Perspectivas

Na primeira fase, prometi que implementaria um framebuffer e rodaria Doom no me emulador. Isso não ocorreu, porque não tive tempo de implementar mapeamento de memória. Nas próximas semanas, pretendo tomar o código aqui escrito como um projeto pessoal, separar a parte emulação em uma biblioteca independente, e me aventurar escrevendo um emulador de PlayStation.

Em conclusão, esse projeto foi um dos mais trabalhosos e satisfatórios que já entreguei, mas ao mesmo tempo é o do que mais me orgulho.

[1] Está presente na branch *stable* do toolchain a partir da versão 1.51. Atualize seu toolchain!

Bônus: Explicação DGEMM

Disclaimer: minha intenção era explicar isso no vídeo, mas não consegui fazer isso sem extrapolar os 5 minutos. Mesmo assim, explicando isso no relatório me faz extrapolar as 2 páginas, mas julgo isso "menos grave".

O ponto da diferença de performance entre os três DGEMM é sobre os padrões de acesso do código às caches. As matrizes são armazenadas linearmente na memória, e dependendo de seu tamanho e da configuração da cache, linhas e colunas das matrizes caem em mais de uma linha da cache diferentes.

Note que o `naive_dgemm` percorre as matrizes por colunas: então, o padrão de acesso de memória dele é em saltos. Isso vai contra o princípio da Localidade Espacial da cache; além disso, podem ser que, fixada uma linha na matriz, a o set na cache onde essa linha pode estar presente pode ser o mesmo para todas as colunas. Isso, dependendo da associatividade da cache, pode causar mais *replaces* a cada iteração.

O `regular_dgemm` itera por linhas. Deste modo, segue a Localidade Espacial, mas ainda faz *replaces* talvez desnecessários a cada coluna.

O `blocking_dgemm` corrige esse problema final, repartindo a matriz em pedaços pequenos que cabem nas mesmas linhas ou em sets diferentes. Assim, possíveis replaces obrigatórios só vão ocorrer em transições entre blocos, não mais entre cada linha e cada coluna. Por isso é o mais rápido, mesmo tendo mais instruções.