Arquitetura de Computadores - Noturno

Relatório EP3

Universidade Federal do ABC, 2021

Gabriel de Mello Fiali - 11201721286

Github: https://github.com/fiali1

Vídeo: https://drive.google.com/file/d/1FU-vywfGyDcP4o3KGmAfL31-YL7NGh9i/view?usp=sharing

Introdução

Este relatório envolve a fase 3 do desenvolvimento do simulador de processador MIPS simplificado, a partir dos conhecimentos apreendidos nas aulas de Arquitetura de Computadores ministradas pelo professor Emílio Camargo Francesquini e pela documentação fornecida.

O Simulador - Revisão e Adições

A terceira versão do simulador conta como principal adição a simulação de uso de memórias cache em diferentes configurações, incluindo 4 das 6 configurações estipuladas pelo Exercício Programático; novos modos de operação, **trace** e **debug**; e a apresentação de valores de tempos de execução estipulados para a execução de instruções em com um processador modo **monociclo** ou **pipelined**. As primeiras funcionam ao considerar a ausência ou presença de memória cache L1, unificada ou dividida entre um conjunto dedicado a instruções e outro aos dados, através de mapeamento direto e políticas de substituição de linhas. Além disso, o simulador conta com a adição de novas instruções para os testes fornecidos, com destaque para instruções do tipo FI, como **bc1f** e **bc1t**.

Quanto à configuração de memória 1, o processo de leitura e gravação da memória na configuração anteriormente elaborada se manteve semelhante, com as mudanças ocorrendo na criação de novos métodos para contabilizar os acessos e ciclos, incluindo a simulação de latência acumulada. A grande mudança da etapa 2 para esta etapa se deu na refatoração da função **syscall** para impressão de strings. Como a implementação original do simulador se baseou em uma memória dividida em bytes, ao invés de separada em palavras, devido à problemas na decodificação de caracteres, foi necessário remanejar o processo da leitura do endereço-alvo para que começasse a ser feito a partir do início da palavra do caractere inicial, e não a partir da posição do mesmo. Caso contrário, ocorreria a contabilização incorreta do número de acessos à memória, por exemplo, com uma string de quatro bytes com o caractere inicial no meio de uma palavra. Seria contabilizado apenas um acesso, apesar da mesma estar dividida entre duas palavras distintas. Essa refatoração foi feita através do método **syscallString()**.

Ainda quanto às mudanças da etapa anterior, o método de conversão de floats e doubles armazenados nos registradores foi modificado para se apoiar menos no uso de strings. **convertFloatDouble()** agora realiza o mesmo processo que o anterior através de operações bitwise. A dependência de strings permanece na medida em que as mesmas ainda passam por um processo de conversão em binário na forma de string para serem armazenadas e representação por 32 bits nos registradores, na forma de valores int. Isso se deve à particularidades da linguagem python, que não consegui contornar, o que contribuiu para tempos de execução longos nos testes de multiplicação de matrizes*. Outra dessas particularidades se dá na operação de resto (%), na qual o valor resultante para essa linguagem tem o sinal associado ao segundo valor da operação realizada. Para o caso a % b = r, por exemplo, o resultado “r” será apresentado de forma a se associar ao sinal de “b”. Apesar da operação estar correta, isso fazia com que os valores apresentados diferissem dos estipulados pelo programa de referência. Isso foi alterado na etapa 3, através do método **fmod()**, da biblioteca **math**, produzindo resultados iguais à referência.

Quanto às configurações de memória novas, elas se destrincham numa classe dedicada, **cache**, que conta tanto com a representação das memórias cache *L1*, *L1i* e *L1d*,

quanto uma representação de *RAM* própria, para maior organização do código; e são selecionadas por meio da variável auxiliar **mode**, assumindo um valor de 1 a 4 para referenciar cada uma. Como a implementação das mesmas foi feita através de iterações, esses valores direcionam o programa para o uso de métodos próprios, que foram expandindo conforme a necessidade. A partir do valor 2 para a variável **mode**, a classe **cache** começa a ser utilizada pelo programa e passa, sendo chamada na etapa de leitura de instruções e na execução daquelas que operam com leituras e gravações na memória, especificamente as instruções, como **lw** e **sw**.

Para a memória *L1* unificada, em **mode** 2, utiliza-se uma estrutura de dicionário no qual cada índice conta com a representação de uma linha na forma de uma tupla contendo um booleano de validade, um inteiro como *tag*, um booleano de identificação de linha modificada e um inteiro, para os dados. O processo se dá com os valores-alvo sendo extraídos do endereço fornecido, e comparando os mesmos aos valores presentes na cache para direcionar o programa, seja identificando um miss ou necessidade de write back na memória *RAM*, ao mesmo tempo que contabiliza o número de hits, misses e custo de latência para cada etapa**.

As configurações **mode** 3 e 4, com a utilização de memória *L1* split em *L1i* e *L1d* seguem um raciocínio semelhante, mas contam com algumas alterações e métodos de substituição de linhas diferentes. Como estrutura, ambas também se apresentam como dicionários, onde cada índice contém uma tupla. A mudança agora é que as mesmas contam com um inteiro como representação de estado da linha, um inteiro para a *tag* e outro, para os dados. Esse primeiro valor é utilizado para manter a coerência de cache durante a execução da simulação, partindo das ideias do protocolo MESI e sistema snooping. Assim, pode assumir um valor de 0 a 3, onde 0 corresponde a uma linha modificada; 1, a uma linha exclusiva; 2, a uma linha compartilhada; e 3, a uma linha inválida. A cada etapa de operação estes valores são verificados para direcionar o programa, em conjunto com os parâmetros e condições estipulados pela configuração anterior.

Quanto aos modos **trace** e **debug**, os mesmos são executados conforme a especificação do usuário, sendo chamados dentro do método de leitura de instruções, para **mode** 1, e dentro dos métodos dedicados às simulações de memória, a partir de **mode** 2; assim como nas instruções que recorrem à leitura e gravação na memória, como especificado anteriormente. São passados os parâmetros que apresentam os acessos ao endereço de memória e linha correspondente, seguindo o padrão definido no programa de referência.

* A diferença nos tempos de execução das variantes dos algoritmos se dá pela distribuição dos dados na RAM e sua alocação na cache. Dependendo da variante adotada e do layout de memória, os dados podem ser dispostos em linhas ou separados colunas, fazendo com seu acesso possa levar a um número menor ou maior de hits entre os diferentes níveis. Logo, é necessário uma configuração que leve a uma boa distribuição espacial dos mesmos, seja com a alteração da ordem de iterações ou subdividindo as matrizes, para que haja uma tendência de dados consecutivos que sejam transferidos juntos ao nível superior em menos chamadas possíveis por já estarem disponíveis.

** O custo de latência implica na diferença entre o número de instruções e ciclos, uma vez que um acesso a um nível inferior na ocorrência de *miss* ou *write back* leva a um stall na execução do processador. Na configuração 1, por exemplo, cada instrução está sendo lida diretamente da memória e, portanto, é preciso que essa latência de acesso seja contabilizada. Ainda, nesse tópico, misses em um nível superior não necessariamente representam o total (hits mais misses) no nível inferior, uma vez que não refletem a ocorrência de *write backs*, que requisitam um acesso extra à memória, que por sua vez implica na contabilização do stall no processador.