Trabalho Prático

Hierarquia de Memória Cache

Victor Bennes Raabe\*, Virginia Soares Müller∗, Sophia Mendes da Silveira†

Faculdade Politénica— PUCRS

06 de dezembro de 2024

**Resumo**

Este relatório descreve a solução para o trabalho prático proposto na disciplina de Arquitetura e Organização de Computadores, no segundo semestre de 2024, que aborda a implementação de um sistema de gerenciamento de hierarquia de memória. Para isso, foi utilizado um processador baseado na arquitetura MIPS, no qual foi implementado um atraso para auxiliar na simulação do comportamento de um hardware real, além de uma cache L1 utilizando mapeamento direto para compor a estrutura de hierarquia de memória com dois níveis.

Também foram desenvolvidos cinco programas em assembly, os quais foram testados no processador em três etapas distintas ao longo do desenvolvimento do trabalho. Por fim, foi coletado o tempo de execução desses programas para análise.

# Introdução

Dentro do contexto da disciplina de Arquitetura e Organização de Computadores, este trabalho aborda a implementação de uma hierarquia de controle com dois níveis (L1 e memória principal - MP) e tem como objetivo explorar os princípios de localidade espacial e temporal através de uma simulação prática.

A hierarquia de memória proposta incorpora uma cache de nível 1 (L1) com mapeamento direto, composta por 8 linhas, onde cada linha armazena blocos de 8 palavras, totalizando 64 palavras. Essa cache opera na mesma frequência do processador e permite acesso em um ciclo de clock, enquanto a memória principal apresenta um tempo de acesso de 16 ciclos. Este comportamento simula o tempo de execução entre níveis de memória em sistemas reais.

Além disso, o trabalho implementa uma memória de instruções baseada em um arquivo contendo código executável para o processador. Esse código é gerado a partir de programas escritos em assembly MIPS e compilados usando ferramentas como o MARS. A estrutura de hierarquia de memória lida com casos de cache hit e cache miss, garantindo a interação entre os níveis e validando as estratégias de localidade espacial e temporal do programa.

# Solução

Para a solução deste problema necessitaram-se de duas etapas: Primeiro um *process* com o atraso de 16 ciclos de clock para o processador receber um sinal que permite a busca de uma nova instrução. E em segundo, o desenvolvimento de uma entidade cache\_l1, que recebe um endereço de instrução do processador, e em caso de *miss* (endereço não encontrado na cache), solicita o endereço para a memória principal.

1. Atraso de 16 ciclos:

O processador busca uma nova instrução olhando para o *clock,* e quando um sinal *hold* esta em ‘0’. Portanto, para realizar o atraso de 16 ciclos, fora adicionado um *process* que olha para mudanças nos sinais de clock e reset, além de instanciar duas variáveis próprias quando reset em 1

Explicar como foi feita a Cache

Depois de considerar o problema, podemos perceber que o valor inicial *a* = 1 serve apenas para fornecer um número inicial sobre o qual podem ser aplicadas as regras para gerar outros números, e portanto seu valor não tem importância especial. Com esta constatação, e sabendo que em um inteiro de 32 bits podemos armazenar números até 231 ou até 320 (obtivemos estes dados através de testes), podemos propor uma solução bastante simples: bastaria criar frações 2*i/*3 *j* com todas as possibilidades de expoentes para 2*i* (ou seja, 0 *i* 31) e para 3 *j* (ou seja, 0 *j* 20), verificando quais números são gerados. O total de números a serem testados é de apenas 32 21 = 672, portanto o algoritmo deve ser executado com grande velocidade. Um algoritmo implementando esta idéia seria parecido com este:

∗

≤ ≤ ≤ ≤

1. Cache\_l1

1 *registro tab*[100] {

2 *int achei* , *int exp2* , *int exp3*

3 }

4

5 **procedimento** ALGORITMO1

6 **para** *i* 1 *até* 100 *faça*

←

7 *tab*[*i*] *FALSE*

←

8 **fim**

9 **para** *i* 1 *até* 100 *faça*

←

10 **para** *j* 0 *até* 20 *faça*

←

11 *num* 2*i/*3 *j*

←

12 // *Se está na faixa desejada e ainda não encontramos*

13 **se** *num* ≥ 100 *e tab*[*num*] = *FALSE então*

14 *tab*[*num*].*achei* ← *TRUE*

15 *tab*[*num*].*exp2 i*

←

16 *tab*[*num*].*exp3 j*

←

17 **fim**

18 **fim**

19 **fim**

20 **fim**

Foi criada uma tabela para manter os resultados encontrados, e a cada resultado encontrado seus expoentes são armazenados. Ao final basta imprimir a tabela, colocando-a no formato adequado de saída (este algoritmo é simples e não será mostrado aqui).

Também é interessante perceber que a solução apresentada até o momento acha os menores ex- poentes possíveis, mas precisa manter o controle de que números já foram encontrados, para não escrever expoentes maiores sobre valores já armazenados. Uma maneira muito simples de descartar este tipo de controle é fazendo o loop externo **decrescente**, garantindo assim que os menores expo- entes serão encontrados por último, e permitindo a escrita sobre valores já encontrados. Esta solução seria:

1 *registro tab*[100] {

2 *int exp2* , *int exp3*

3 }

4

5 **procedimento** ALGORITMO2

6 **para** *i* 31 *até* 0 *faça*

←

7 **para** *j* 0 *até* 20 *faça*

←

8 *num* 2*i/*3 *j*

←

9 **se** *num* ≤ 100 *então*

10 *tab*[*num*].*exp2 i*

←

11 *tab*[*num*].*exp3 j*

←

12 **fim**

13 **fim**

14 **fim**

15 **fim**

Esta idéia é bastante simples e eficiente, porém um teste mostra que ela acha apenas 53 dos 99 números desejados. Com isto podemos concluir que os outros 46 valores devem ser obtidos com expoentes maiores, e os números envolvidos não podem ser representados em inteiros de 32 bits.

*Note que esta seção dá uma idéia inicial, analisa-a de forma rápida e diz porque ela não serve, também de forma rápida, sem perder o tempo do leitor com papo furado. Essa precisão e detalhe são sinais de um trabalho onde vem coisas mais interessantes mais tarde. Veja que não foi usada uma linguagem de programação com todos os detalhezinhos! Os algoritmos estão em uma linguagem simplificada, que permite que se entenda rapidamente o que deve ser feito, sem excesso de detalhes inúteis.*

*Perceba também que o código não é cheio de explicações óbvias e inúteis (esta linha incrementa o contador, esta linha calcula o número), mas é clara o bastante para que outra pessoa entenda o que é feito e reproduza o algoritmo em sua linguagem preferida. Uma falha: terminamos dizendo que achamos 53 dos 99 números, mas sem dizer quais foram nem falar do tempo necessário para encontrá-los.* ***Moral****: quando terminar de falar de uma solução, apresente os resultados, comente o tempo e as (possíveis) falhas!!*

# Resultados

|  |  |  |  |
| --- | --- | --- | --- |
| Valor truncado | Fórmula | Valor exato | Excesso |
| 2 | 21 ÷ 30 | 2 | 0 |
| 3 | 25 ÷ 32 | 3.5555. . . | 0.5555. . . |
| 6 | 26 ÷ 32 | 7.1111. . . | 1.1111. . . |

Depois de implementar o algoritmo acima em linguagem C e executá-lo num tempo de aproximada- me

Após a obtenção destes valores, estes foram testados com ferramentas capazes de lidar com os grandes números envolvidos, e todos os resultados foram confirmados corretos.

*Por favor, não esqueça de apresentar os resultados, afinal deve ser isso que seu leitor quer ver!*

# Conclusões

As abordagens iniciais, mesmo oferecendo apenas resultados parciais, contribuíram bastante para o entendimento do problema e abriram caminho para a solução definitiva. Esta se mostrou bastante simples e eficiente, embora tenha exigido um estudo de como o erro se propaga quando as regras são aplicadas a novos números. A solução implementada não se preocupou em achar sempre os menores expoentes possíveis, mas mesmo assim ela foi bastante eficiente e os expoentes encontrados não são excessivamente grandes (foram feitas comparações com o primeiro algoritmo apresentado, e as diferenças foram poucas). Além disso, a solução recursiva adotada foi razoavelmente elegante e clara, não precisando de uma implementação complicada.

Embora não tenham sido mostrados neste artigo, foram feitos testes com outros valores iniciais (*a* = 1) e para números indo até bem mais do que 100, o algoritmo também funcionou bem.

/

Acreditamos ter desenvolvido uma solução interessante e barata a um problema relativamente complexo, já que em uma linguagem de programação usual não existe suporte direto para tratar com os números envolvidos, e conseguimos determinar resultados corretos que envolvem grandes núme- ros.

*Suas conclusões não precisam ser geniais nem mudar o mundo. Veja que a parte inteligente do artigo já foi feita, e veio antes! Agora é a hora de fechar e dizer o que você acha que poderia vir depois, etc etc. E se você está pensando em dizer coisas como “Provavelmente existem soluções*

*melhores”, então diga ao menos o que você pensa que pode melhorar (por exemplo, neste artigo eu gostaria de evitar o uso de números em ponto flutuante e preferiria trocar por algum tipo de controle com inteiros). Se você não der opções e só disser que dá pra fazer melhor, está se diminundo de graça. Pra que isso?*

*A bibliografia abaixo não foi usada neste artigo, e serve apenas de exemplo para a formatação.*

*Outros exemplos estão na página da biblioteca da PUCRS.*