Arquitetura de Computadores
Trabalho 2

Professora:
Simone de Lima Martins

Alunos:
Leonardo Brasil de Oliveira Cerne
Victor Teles Araujo Speorin

Dezembro 2024

Conteúdo

1 Capítulo 1 **3**
1.1 Introdução . . . 3
2 Capítulo 2 **3**
2.1 Funcionalidade . . . 3
3 Capítulo 3 **4**
3.1 Descrição do experimento . . . 4
3.2 Realização dos experimentos . . . 4
3.2.1 Primeira Execução . . . 4
3.2.2 Segunda Execução . . . 4
3.2.3 Comparação . . . 4
3.2.4 Sobre o Programa . . . 5
3.2.5 Executando o Programa e Comparando . . . 6
3.2.6 Reordenando o Programa . . . 7
3.2.7 Desenrolando o Programa . . . 8
3.3 Relatório Final . . . 9
3.3.1 Execução do Programa no Simulador EduMIPS64 . . . 9
3.3.2 Execução do Código Rearrumado . . . 10
3.3.3 Execução do Código Desenrolado . . . 10
3.3.4 Observações Finais . . . 11

Lista de Tabelas

1 Funcionalidades principais do EduMIPS64. . . . 3
2 Significado das estatísticas do EduMIPS64. . . . 4

3 Comparação da execução com e sem adiantamento do código reordenado 7
4 Comparação da execução com e sem adiantamento do código desenrolado 9
5 Comparação crítica das métricas entre as diferentes execuções com e sem adiantamento 11

1 Capítulo 1

1.1 Introdução

Neste trabalho, iremos analisar o comportamento de uma arquitetura com pipeline utilizando o simulador EduMIPS64, que implementa um subconjunto das instruções da arquitetura MIPS64. O objetivo principal é estudar os conceitos relacionados a paradas no pipeline (stalls) e o impacto do adiantamento (forwarding) de dados na execução de instruções. Além disso, será realizada uma comparação detalhada entre as execuções de programas com e sem técnicas de otimização, como reordenação de instruções e desenrolamento de laços.

Serão apresentados experimentos utilizando o programa fornecido no simulador para compreender métricas importantes, como Cycles, Instructions, CPI (Ciclos por Instrução) e Stalls, bem como os tipos de conflitos que afetam o desempenho do pipeline. Este relatório está estruturado da seguinte forma:

2 Capítulo 2

2.1 Funcionalidade

O EduMIPS64 é um simulador educacional que reproduz o comportamento de um processador baseado na arquitetura MIPS64. Ele foi desenvolvido para auxiliar no ensino de sistemas computacionais e arquiteturas de computadores, permitindo que usuários explorem conceitos relacionados ao funcionamento de pipelines, gerenciamento de registradores e memória, além de otimizações de performance.

O software apresenta uma interface gráfica intuitiva e fornece informações detalhadas sobre a execução das instruções em tempo real. As principais funcionalidades incluem:

Ferramenta	*Funcionalidade*
Visualização do Pipeline	Exibe o progresso de cada instrução pelas etapas do pipeline (fetch, decode, execute, etc.).
Estatísticas de Execução	Fornece métricas como Cycles, Instructions, CPI, e Stalls, que ajudam a compreender o impacto de diferentes configurações.
Configuração Personalizada	Permite ajustar parâmetros, como ativação ou desativação do adiantamento de dados, para explorar diferentes cenários de execução.
Modo de Simulação	Suporta execução em modo contínuo ou passo a passo (Single Cycle), facilitando a análise detalhada de conflitos e dependências entre instruções.

Tabela 1: Funcionalidades principais do EduMIPS64.

O EduMIPS64 pode ser instalado a partir do repositório oficial no GitHub (EduMIPS64 GitHub) e está disponível para sistemas Windows e Linux. A ferramenta suporta um subconjunto das instruções MIPS64, incluindo operações aritméticas, de carga e armazenamento, e de controle de fluxo, sendo suficiente para simular programas simples e explorar o funcionamento do pipeline de maneira prática e acessível.

3 Capítulo 3

3.1 Descrição do experimento

Nesse capítulo iremos exibir os resultados dos experimentos, propostos pela professora, realizados com a ferramenta Edumips64.

3.2 Realização dos experimentos

3.2.1 Primeira Execução

Ao Executar o programa mins2.s no simulador EduMIPS64 (sem adiantamento), obtemos 350 ciclos, 201 instruções, 1.741 de CPI (Ciclos por Instrução) e 105 *stalls*.

Estatística	*Significado*
Ciclos	Número de ciclos necessários para a execução total do programa.
Instruções	Número total de instruções executadas durante a execução do programa.
CPI	É o resultado obtido na divisão do número total de ciclos pelo número total de instruções.
Stalls	Número total de paradas causadas por qualquer tipo de conflito entre as intruções.

Tabela 2: Significado das estatísticas do EduMIPS64.

Executando o programa, percebe-se que ocorrem 105 paradas, todas causadas por dependência de dados, como mostra o simulador.

3.2.2 Segunda Execução

Ao executar o programa mins2.s no simulador EduMIPS64 (com adiantamento), obtemos 245 ciclos, 201 intruções, 1.218 de CPI e 0 *stalls*. Todas essas estatistícas estão melhor explicadas na Tabela 2. Percebe-se também que, por conta da ativação do adiantamento, não ocorrem paradas durante a execução do programa.

3.2.3 Comparação

Comparando a primeira execução com a segunda, percebemos que aquela que foi executada com adiantamento foi mais rápida, com um CPI menor, já que foram usados menos ciclos pelo mesmo número de instruções. Isso se dá pelo fato de que quando utilizamos adiantamento, a instrução atual não tem que esperar a execução da qual ela depende acabar, isso faz com que não ocorra paradas, resultando numa execução mais rápida.

3.2.4 Sobre o Programa

Utilizando como base o seguinte prgrama:

```
.data
.text
main:
DADDI R3,R0,8
DADDI R1,R0,1024
DADDI R2,R0,1024
Loop: L.D F0,0(R1)
MUL.D F0,F0,F2
L.D F4,0(R2)
ADD.D F0,F0,F4
S.D F0,0(R2)
DSUB R1,R1,R3
DSUB R2,R2,R3
BNEZ R1,Loop
HALT
```

Sem adiantamento:

Ao analisarmos a arquitetura do pipeline do simulador EduMips64, conseguimos verificar que as instruções de Load/Store/Operações com inteiros demoram 1 ciclo na fase de execução do pipeline (5 no total) , as operações de Mult e Div demoram 7 ciclos na fase de execução (11 no total) e as operações de Add e Sub demoram 4 ciclos nessa fase (8 no total). Além disso, nessa arquitetura, quando temos dependência de dados a operação que depende começa a fase de execução 1 ciclo depois do write back da instrução que fornece os dados.

Considerando que temos 128 loops nesse código, o primeiro loop executa em 29 ciclos, o último em 27 ciclos e os demais em 24. E com isso conseguimos descobrir que esse programa precisa de 3080 ciclos para finalizar sua execução.

Para calcularmos quantas instruções são executadas, para esse programa, podemos utilizar a seguinte fórmula:

(intruções fora do loop + ***(instruções dentro do loop*** x ***número de loops))*** +1

obs: a unidade somada no final da fórmula a por conta da instrução *HALT*, a qual é executada só no final da execução.

Usando a fórmula acima, chegamos em: $3 + (8\text{x}128) + 1$. O que resulta em 1028 instruções.

Calculando o CPI, chegamos em $\frac{3080}{1028}$. O que resulta em aproximadamente 3 Ciclos por Instrução.

Ao calcular as *stalls*, as instruções: MUL.D F0, F0, F2; ADD.D F0, F0, F4; S.D F0, 0(R2); BNEZ R1, Loop; possuem dependências de dados e causam paradas. A cada loop temos 15 paradas, e no começo da execução o L.D F0, 0(R1) também possui uma dependência, logo, são $(15 * 128) + 1 = 1921$ paradas*(stalls)*.

Com adiantamento:

Usando como base as explicações anteriores, mas dessa vez com adiantamento de dados, o conteúdo de L.D está disponível um ciclo depois da fase do pipeline de acesso à memória, e o conteúdo das instruções MUL.D, ADD.D e DSUB está disponível um ciclo depois do fim da fase de execução(ALU)

Levando isso em consideração, descobrimos que até o fim do primeiro loop temos 22 ciclos, no ultimo temos 23, e nos demais temos 19. Logo: $19 * 126 + 22 + 23 = 2439$ ciclos. Além disso, sabemos que o número de instruções será o mesmo, já que a configuração de adiantamento não altera esse dado.

Calculando o CPI, chegamos em $\frac{2439}{1028}$. O que resulta em 2,37 Ciclos por Instrução.

Por fim, calculando as *stalls*, agora com adiantamento, diminuímos as paradas nas instruções que tinham dependência de dados, passando para 10 paradas por loop, e não temos mais a parada da primeira instrução L.D. Com isso chegamos em $128 * 10 = 1280$ *stalls*.

3.2.5 Executando o Programa e Comparando

Ao executar o programa no simulador *EduMips64*, todos os resultados obtidos anteriormente são idênticos, tanto na execução com adiantamento, quanto na sem adiantamento. Isso comprova que os calculos utilizados são 100% eficazes. Mas ainda assim, ter essa ferramenta para verficar os resultados é fundamental para um bom projeto.

3.2.6 Reordenando o Programa

Para diminuirmos a quantidade de *stalls*, é possível rearrumar as instruções e, dessa forma, obtemos o seguinte programa:

```
.data
.text
main:
DADDI R3,R0,8
DADDI R1,R0,1024
DADDI R2,R0,1024
Loop:
L.D F0,0(R1)
L.D F4,0(R2)
MUL.D F0,F0,F2
DSUB R1,R1,R3
DSUB R2,R2,R3
ADD.D F0,F0,F4
S.D F0,8(R2)
BNEZ R1,Loop
HALT
```

Ele foi rearrumado assim, principalmente, para diminuirmos as paradas das instruções L.D F0, 0(R1) e MUL.D F0,F0,F2.

Ao executar o código rearrumado e, utilizando como base os métodos e explicações da subseção 3.2.4, é obtido os seguintes valores para as estatísticas do programa:

Estatística	Sem adiantamento	Com adiantamento
Ciclos	2696	2183
Instruções	1028	1028
CPI	2.62	2.12
Stalls	1537	1024

Tabela 3: Comparação da execução com e sem adiantamento do código reordenado

3.2.7 Desenrolando o Programa

Para diminuirmos o tempo de execução do programa, é possível desenrolá-lo 4 vezes, já que é divisor de 128 (número de loops ao executar o programa). Dessa forma, diminuiremos a quantidade de instruções do tipo BNEZ R1, Loop. Portanto, obtemos o seguinte programa:

```
.data
.text
main:
DADDI R3,R0,8
DADDI R1,R0,1024
DADDI R2,R0,1024
Loop:
L.D F0,0(R1)
L.D F4,0(R2)
MUL.D F0,F0,F2
DSUB R1,R1,R3
DSUB R2,R2,R3
ADD.D F0,F0,F4
S.D F0,8(R2)
L.D F6,8(R1)
L.D F8,8(R2)
MUL.D F6,F6,F2
DSUB R1,R1,R3
DSUB R2,R2,R3
ADD.D F6,F6,F8
S.D F6,16(R2)
L.D F10,16(R1)
L.D F12,16(R2)
MUL.D F10,F10,F2
DSUB R1,R1,R3
DSUB R2,R2,R3
ADD.D F10,F10,F12
S.D F6,24(R2)
L.D F14,24(R1)
L.D F16,24(R2)
MUL.D F14,F14,F2
DSUB R1,R1,R3
DSUB R2,R2,R3
ADD.D F14,F14,F16
S.D F14,32(R2)
BNEZ R1,Loop
HALT
```

Vale ressaltar que, por conta da diminuição do número de instruções BNEZ, o número de ciclos ao executar o programa diminui, o deixando mais otimizado. Por fim, as estatísticas obtidas são:

Estatística	Sem adiantamento	Com adiantamento
Ciclos	2344	1895
Instruções	932	932
CPI	2.51	2.03
Stalls	1409	960

Tabela 4: Comparação da execução com e sem adiantamento do código desenrolado

3.3 Relatório Final

Neste relatório final, iremos analisar criticamente os resultados obtidos nas execuções do programa com diferentes configurações de adiantamento e otimizações aplicadas. Vamos comparar as métricas de desempenho, como Ciclos, Instruções, CPI (Ciclos por Instrução) e *Stalls*, para entender o impacto de cada otimização no desempenho geral do programa. As comparações ocorrerão entre três cenários:

Primeiro: Execução do programa sem adiantamento e com adiantamento.
Segundo: Execução do programa rearrumado com e sem adiantamento.
Terceiro: Execução do programa desenrolado com e sem adiantamento.

3.3.1 Execução do Programa no Simulador EduMIPS64

Sem adiantamento:

Ciclos: 3080
Instruções: 1028
CPI: aproximadamente 3
Stalls: 1921

Com adiantamento:

Ciclos: 2439
Instruções: 1028
CPI: 2.37
Stalls: 1280

Análise: A execução sem adiantamento resultou em 3080 ciclos e 1921 *stalls*, o que significa que o pipeline foi frequentemente interrompido devido a dependências de dados entre as instruções. Esse comportamento é esperado, pois sem a técnica de adiantamento, as instruções que dependem de dados de instruções anteriores precisam esperar pela conclusão do processo de escrita dos dados no registrador. Já a execução com adiantamento diminuiu significativamente o número de ciclos (2439) e reduziu consideravelmente os *stalls*, demonstrando uma melhoria substancial no desempenho. A técnica de adiantamento permite que as instruções usem dados

de registros modificados em estágios anteriores do pipeline, evitando a necessidade de esperar pela conclusão do Write-back.

3.3.2 Execução do Código Rearrumado

O código foi rearrumado para diminuir as stalls nas instruções dependentes de dados, como as instruções L.D F0, 0(R1) e MUL.D F0, F0, F2, com o objetivo de otimizar o fluxo de execução.

Sem adiantamento:

Ciclos: 2696
Instruções: 1028
CPI: 2.62
Stalls: 1537

Com adiantamento:

Ciclos: 2183
Instruções: 1028
CPI: 2.12
Stalls: 1024

Análise: Após rearrumar o código, houve uma redução nas stalls em comparação com a execução original. As stalls diminuíram de 1537 (sem adiantamento) para 1024 (com adiantamento), graças à reorganização das instruções, o que permite maior paralelismo entre elas. A quantidade de Ciclos também foi reduzida significativamente, de 2696 para 2183, e o CPI diminuiu de 2.62 para 2.12. O adiantamento de dados, junto com o rearranjo do código, contribui para uma execução mais eficiente, com menos paradas e melhor utilização dos ciclos do processador.

3.3.3 Execução do Código Desenrolado

Neste caso, o programa foi desenrolado quatro vezes para reduzir o número de iterações do loop e diminuir a quantidade de instruções do tipo BNEZ.

Sem adiantamento:

Ciclos: 2344
Instruções: 932
CPI: 2.51
Stalls: 1409

Com adiantamento:

Ciclos: 1895
Instruções: 932
CPI: 2.03
Stalls: 960

Análise: O desenrolamento do laço resultou na redução do número de Ciclos e Stalls em comparação com o código não desenrolado. O total de Ciclos foi reduzido de 2344 para 1895, e as Stalls diminuíram de 1409 para 960. Isso ocorre porque o número de iterações do loop foi reduzido, resultando em menos dependências de dados entre as instruções e, consequentemente, menos bloqueios no pipeline. O CPI também caiu de 2.51 para 2.03, refletindo um melhor aproveitamento dos ciclos do processador, com menos interrupções e paradas no pipeline. O adiantamento de dados teve um impacto significativo, não só pela eliminação das paradas, mas também pela melhor distribuição dos dados nos estágios do pipeline, permitindo um fluxo mais eficiente de instruções.

Execução	Ciclos	Instruções	CPI	*Stalls*
Sem adiantamento	3080	1028	3	1921
Com adiantamento	2439	1028	2.37	1280
Sem adiantamento (Rearrumado)	2696	1028	2.62	1537
Com adiantamento (Rearrumado)	2183	1028	2.12	1024
Sem adiantamento (Desenrolado)	2344	932	2.51	1409
Com adiantamento (Desenrolado)	1895	932	2.03	960

Tabela 5: Comparação crítica das métricas entre as diferentes execuções com e sem adiantamento

3.3.4 Observações Finais

Adiantamento de Dados: O adiantamento de dados tem um impacto notável no desempenho, pois reduz os Ciclos e elimina as *Stalls*, resultando em uma execução mais rápida e eficiente. A diferença entre as execuções com e sem adiantamento é clara, como mostrado nas comparações da Tabela 5

Rearranjo do Código: A reorganização das instruções proporcionou uma redução nas *Stalls* e uma leve melhoria nos Ciclos e CPI, comprovando que a técnica de rearranjo pode ser útil para reduzir dependências entre instruções e melhorar o desempenho.

Desenrolamento de Laços: O desenrolamento de laços foi eficaz em reduzir o número de Ciclos e *Stalls*, além de melhorar o CPI. Isso ocorre porque o número de iterações do loop foi diminuído, o que reduziu as dependências de dados e, consequentemente, as paradas no pipeline.

Em resumo, tanto o adiantamento de dados quanto as otimizações de rearranjo e desenrolamento de laços têm um impacto positivo no desempenho do programa, resultando em uma execução mais rápida e eficiente. As métricas de Ciclos, CPI e *Stalls* mostram melhorias consistentes com cada técnica aplicada, demonstrando a importância da otimização para maximizar o desempenho de programas em arquiteturas com pipeline.