Pipeline Hazards

Luke Skywalker

6 de outubro de 2015

pg 330 **Pipelining** é uma técnica de implementação em qual uma rajada de instruções são sobrepostas em execução. Postas em fila para serem executadas, para podermos extrair uma grande vazão de instruções aproveitando bastante os núcleos do processador.

pg 331 A analogia da lavanderia explica que se você tem varias pessoas revesando para cada um lavar suas roupas sujas (programas), então cada thread (pessoa) irá usar uma estrutura básica de cada vez, então quando ele passasse a usar outra estrutura para continuar seu processo de lavar roupa, outra pessoa poderia utilizar a estrutura que acabou de ser liberada. E assim sucessivamente ao final teríamos apenas o tempo da última pessoa usando a última estrutura para terminar o processo (por exemplo: 10 minutos) e após aproximadamente 70 minutos teríamos todas as peças de roupas de todos, lavadas e prontas no armário.

Ciclos/Tempo	2 min	2 min	2 min	2 min	2 min	2 min	2 min
i	A	B	C	D			
i+1		A	B	C	D		
i+2			A	B	C	D	
i+3				A	B	C	D

Tabela 1: Consideremos 2 minutos por estrutura para lavar a roupa, para facilitar o entendimento.

pg 332 Algoritmo simplificado do pipelining. Sabendo também que o MIPS o qual nós estamos estudando, tem 5 estágios (tática para simplificar o entendimento). Os 5 estágios estão listados abaixo, em forma algorítmica.

(a) **IF** Busca uma instrução da memória.

(b) **ID** Lê os registradores enquanto decodificando a instrução.

(c) **EX** Executa a operação ou calcula o endereço.

(d) **MEM** Acessa o operando na memória de dados.

(e) **WB** Escreve o resultado em registrador.

pg 333 Para deixa a discussão mais concreta, vamos criar um pipeline. Nesse exemplo, e no resto do capitulo vamos limitar nossa atenção para 8 instruções: load word (lw), store word (sw), adição (add), subtração (sub), e-lógico (and), ou-lógico (or), set less than (slt), e branch on equal (beq). Agora vamos comparar uma implementação de datapath monociclo e um enfileirado.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Tabela 2: Tempo total para cada instrução, calculado do tempo de cada componente. Esse calculo assume que os multiplexadores, unidade de controle, acessos ao PC, e extensão de sinal não tem atraso.

Nós podemos pegar a aceleração do pipeline e jogar em uma fórmula. Se os estágios estão perfeitamente balanceados, então o tempo entre instruções no processador enfileirado—assumindo condições ideais—é igual:

$$Tempo\ entre\ instrs_{pipe} = \frac{Tempo\ entre\ instrs_{nonpipe}}{Nro.\ de\ \textbf{estágios}}$$

Ciclos/Tempo	200 ps	100 ps	200 ps	200 ps	100 ps	200 ps	100 ps	200 ps	200 ps	100 ps	200 ps	100 ps	200 ps	200 ps	100 ps
lw $1, 100($0)	F	R	E	M	W										
lw $2, 200($0)						F	R	E	M	W					
lw $3, 300($0)											F	R	E	M	W

Tabela 3: Consideremos 2 minutos por estrutura para lavar a roupa, para facilitar o entendimento.——O problema que em sequencia assim como está, é muito desgastante para o computador, quando na verdade ele tem que processar "toneladas"de instruções.

Se nós implementarmos o datapath como sendo pipeline, teremos que adicionar muitas funcionalidades e controles para que ele execute de maneira consistente as instruções, tomando cuidado para não invadir o espaço de outras instruções (hazards estruturais), e também tomando cuidado ter o dado pronto para a próxima instrução sem penalizar o tempo dela de ser executada (hazards de dados).

Ciclos/Tempo	200 ps	100 ps	200 ps	200 ps	100 ps	200 ps	100 ps
lw $1, 100($0)	F	R	E	M	W		
lw $2, 200($0)		F	R	E	M	W	
lw $3, 300($0)			F	R	E	M	W

Tabela 4: Estilo enfileirando as instruções. Aparentemente é melhor, e utiliza a CPU de uma maneira mais elegante, mas não é sempre assim enfileirado, os programas tendem a grandes complexidades e falhas de programação impedem que a CPU projete rajadas de instruções enfileiradas para usufruir perto do melhor desempenho queimando instruções enfileiradamente.

No primeiro ensaio nós temos três loads seguidos, parece lúdico, mas isso acontece muito (vários loads seguidos), porém

em sequência mas esperando que cada um execute completamente, são $3 \times 800ps = 2400ps$ para executar esse programa completamente.

Agora o segundo ensaio nós temos um enfileiramento (pipelining), o que se faz é achar o mais tempo de atraso e acomodar todos os estágios nesse tempo. É uma tática boa. Veja que enfileiradamente nós temos "como se fosse uma única instrução de 7 estágios", na verdade são três instruções sendo executadas em paralelo (com uma diferença de 200 ps do IF) ou seja são $200ps \times 7$ estágios processados (o ponto de vista do pipeline mudo).

$$\frac{1000000 \times 800\ ps + 2400\ ps}{1000000 \times 200\ ps + 1400\ ps} = \frac{800002400\ ps}{200001400\ ps} = \frac{800\ ps}{200\ ps} = 4$$

Ou seja, tende a se aproximar dessa razão da duração média de uma instrução em um monociclo pelo mais tempo de atrasado no estágio $\frac{800\ ps}{200\ ps} = 4x$. Tende a 5 instruções em paralelo. Ou o pipeline tem 5 estágios. Precisaríamos adicionar mais instruções. Lembrar que isso é ficção, na verdade os hazards estruturais e hazards de controle, degeneram bastante essa razão.

pg 335 **Dependência Estrutural** é quando uma instrução planejada **não** pode executar no ciclo de clock atual (desejado) pois aconteceu uma exceção de combinação de duas estruturas digitais disputando pelo menos ciclo de clock. Isto é, ele não suporta certa combinação de instruções ao mesmo tempo, terá que ser tomado alguma medida drástica, normalmente atrasar um pouco a instrução seguinte para ela ser executada mais tarde.

Se alguma combinação de instrução não pode ser acomodada por causa de algum recurso conflitante, a máquina é dita como tendo uma dependência estrutural.

- **Exemplo 1**: A máquina pode ter apenas uma porta de escrita no banco de registradores, mas em alguns casos o pipeline poderá querer realizar duas escritas no mesmo ciclo de clock.
- **Exemplo 2**: A máquina pipeline compartilha uma única memória, para dados e instruções. Quando uma instrução contém uma referência à memória de dados (load word—`lw`), ele irá conflitar com a instrução mais adiante que entrar em paralelo mas estiver tentando acessar a área da memória de instruções.

lw $1, 100($0)	IF	ID	EX	**MEM**	WB			
instrução 2		IF	ID	EX	MEM	WB		
instrução 3			IF	ID	EX	MEM	WB	
instrução 4				**IF**	ID	EX	MEM	WB

Tabela 5: Para resolver isso, nós podemos inserir um mecanismo que prevê qual combinação vai conflitar e inserimos um atraso justo, para podermos acomodar as instruções e permitir que todas utilizem as estruturas digitais no seu ciclo de relógio adequado.

- Algumas unidades funcionais não são totalmente enfileiráveis. Então a sequencia de instruções usando não-pipeline não pode proceder na maior taxa possível de uma instrução por ciclo de relógio.
- Alguns recursos não foram duplicados o bastante para permitir todas as combinações possíveis de instruções para o pipeline executar.

lw $1, 100($0)	IF	ID	EX	MEM	WB				
instrução 2		IF	ID	EX	MEM	WB			
instrução 3			IF	ID	EX	MEM	WB		
instrução 4				X	IF	ID	EX	MEM	WB

Tabela 6: **Veja** que a instrução 1 não utilizará a memória de dados, portanto não vai acessar, e MEM será irrelevante para a instrução 1, o mesmo se aplica à instrução 2. Caso contrário, a instr. 3 só iniciaria no ciclo sétimo.

pg 336 **Dependência de Recursos** também chamada pipeline data harzard. É quando a instrução planejada não pode executar no ciclo de clock atual por causa que existe dados que essa instrução precisa e estão pendentes (estão sendo processados ainda por alguma(as) instrução(ções) a frente). O esquema a baixo exemplifica um `fowarding(add.EX, sub.EX)`.

```
add $s0, $t0, $t1  IF ID EX . MEM WB
                             \
                              \
                               \
sub $t1, $s0, $t3         IF ID . EX MEM WB
```

Forwarding/bypassing é um método de resolver hazard de dados, que consiste em fazer buffers internos que permitem passar os dados pendentes sem atrasar alguma instrução. Esses buffers vão guardando os dados das instruções que vão sendo processadas nos seus estágios e se necessário e possível (temporalmente falando), os dados serão distribuídos para as instruções que precisam deles. Para não atrasar o pipeline.

Sem intervenção, o hazard de dados, severamente, vai fazer *stalls* no pipeline, isto é, inserir *nops* para atrasar até que o dado fique pronto para a instrução logo atrás possa utilizá-lo.

Ou seja, o esquema acima seria assim, caso o pipeline não tivesse mecanismo para tratar dependência de dados.

```
add [$s0], $t0, $t1  IF ID EX MEM WB
sub $t1, [$s0], $t3      X  X  X  IF ID EX MEM WB
                         |  |  |
                         |  |  +-----> 1 cycle wasted
                         |  +--------> 2 cycle wasted
                         +-----------> 3 cycle wasted
```

pg 338 Desde que, até mesmo com forwarding, nós teremos alguns problemas de *stall* algum estágio para um possível load-use data hazard. O termo load-use data hazard é uma forma de data hazard específica, na qual o dado está sendo carregada por uma instrução load, ainda não está disponível para as próximas instruções que requerem esse dado (trazido da memória). O esquema abaixo mostra acontecendo o load-use data hazard, porém existe forwarding também caso contrário, teríamos que esperar a instrução load word terminar completamente para termos o dado no banco de registradores, disponível para o sub utilizá-lo.

```
lw $s0, 20($t1)    IF  ID  EX  MEM .   WB
```

```
sub $t2, $s0, $t3      X   X   X    \  X   X
                       |   IF  ID    . EX  MEM  WB
                       |
                       +-----> 1 cycle wasted
```

pg 339 Considere o seguinte segmento de código em C; e sua respectiva tradução para linguagem de montagem MIPS. Considerando que todas as variáveis estão em memória e são endereçáveis com deslocamentos a partir de $t0.

```
+----------------------+
| a = b + e;           |
| c = b + f;           |
+----------------------+
| lw  $t1,  0($t0)     |
| lw  $t2,  4($t0)     |
| add $t3,    $t1, $t2 |
| sw  $t3, 12($t0)     |
| lw  $t4,  8($t0)     |
| add $t5,    $t1, $t4 |
| sw  $t5, 16($t0)     |
+----------------------+
```

Achar os hazards para desviar a execução total de qualquer stall no pipeline.

```
lw  $t1,0($t0)   | IF ID EX MEM WB
lw  $t2,4($t0)   |    IF ID EX  MEM WB
add $t3,$t1,$t2  |       X  X   X   IF  ID  EX MEM WB
sw  $t3,12($t0)  |                      IF  ID EX  MEM WB
lw  $t4,8($t0)   |                          IF ID  EX  MEM WB
add $t5,$t1,$t4  |                             X   X   X   IF  ID  EX MEM WB
sw  $t5,16($t0)  |                                             IF  ID EX  MEM WB
```

Para arrumar o código MIPS basta mover `lw  $t4,8($t0)` para perto dos load words; assim desviará das dependências. Não acontece mais hazard estrutural pois a instrução add não acessa a memória. Mas enquanto o primeiro load esta acessando a memória de dados o primeiro add esta buscando a instrução na memória, aí temos dependência estrutural. Na página 339 do livro Computer Organization and Design (Patterson & Hennessy), os autores usam MIPS pipelining, que por sua vez se desvia facilmente de muitos hazards estruturais——pois usa duas memórias, uma memória para instruções e outra para dados, facilitando desviar-se de dependências estruturais.

```
lw  $t1,0($t0)   | IF ID EX MEM WB
lw  $t2,4($t0)   |    IF ID EX  MEM WB
```

```
lw  $t4,8($t0)   |       IF ID  EX  MEM WB
add $t3,$t1,$t2  |          IF  ID  EX  MEM WB
sw  $t3,12($t0)  |              IF  ID  EX  MEM WB
add $t5,$t1,$t4  |                  IF  ID  EX  MEM WB
sw  $t5,16($t0)  |                      IF  ID  EX  MEM WB
```

4.12.1a

4.12.2a

4.12.3a

4.13.2

4.13.3

4.13.4

4.13.5

4.13.6