Assignment n^o 2

Breno de Castro Pimenta

RA: 2017114809

DCC006: Computer Organization I
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) Brasil

PROBLEM 1: 5-STAGE PIPELINE

O loop do programa descrito tarda nove ciclos de clock e é executado da seguinte forma: O loop se inicia com com o carregamento do próximo valor do vetor (lw x12 0(x9)). Em seguida ele realiza um Add com dependência do valor carregado na instrução anterior o que gera um Stall. Após o Stall é realizado mais duas adições. E no final para que o branch possa tomar a decisão de continuar no loop ou continuar com as instruções normais, é realizado outros dois Stalls, pois o Branch necessita do valor calculado na instrução anterior. No próximo ciclo ou a próxima instrução (fora do loop) que já realizou o instruction fetch será realizada ou será ignorado seu instruction fetch e voltará a executar outra iteração do loop. A imagem abaixo demonstra o esquema gerado pelo simulador Ripes, com os ciclos enumerados.

lw x12 0(x9)	F	D	E	M	W				
add x10 x10 x12		F	D	-	E	M	W		
addi x9 x9 4			F	-	D	E	M	W	
addi x11 x11 1					F	D	E	M	W
blt x11 x6 -16						F	D	-	-
addi x11 x10 0							F	-	-
	1	2	3	4	5	6	7	8	9

Figure 1. Diagrama de execução do loop

PROBLEM 2: STUDY QUESTIONS

Os cincos estágios padrões de um pipeline de um processador RISC:

1. Fetch instruction: Busca a próxima instrução na memória de instruções.
2. Instruction decode: Decodifica a instrução e lê do banco de registradores.
3. Execute: Executa operações na ALU, desde operações binárias, a aritméticas, até cálculo de endereços.
4. Memory Access: Acessa a memória principal para realizar escrita ou leitura (apenas parte das instruções realiza esse passo).
5. Write Back: Escreve resultados de operações ou buscas no banco de registradores (apenas parte das instruções realiza esse passo).

Hazards:

Katz (1996) descreve Hazards como limitações do pipeline que previne a próxima instrução de ser executada no tempo de clock designada a ela.

Existem três tipos de Hazards:

Hazard Estrutural: **Definição:** Quando uma instrução, devido ao conjunto de intruções que precedem ou sucedem essa instrução, a impedem de utilizar a estrutura de dados, ou seja de executar o procedimento que devia no ciclo de clock adequado. Sumariamente, são instruções tentando utilizar a mesma estrutura ao mesmo tempo.
Exemplo: Um exemplo para Hazard Estrutural é o levantado por Patterson e Hennessy (2018), onde é criado uma situação hipotética onde só haja uma estrutura de memória no processador, e tenhamos uma instrução que acessa a memória no quarto ciclo e esse ciclo é executado simultâneo a uma outra instrução que está executando seu primeiro ciclo, buscando na memória também, a tentativa de uso simultaneo da mesma estrutura no mesmo ciclo de clock gera então o Hazard Estrutural. E nesse caso os próprios Patterson e Hennessy (2018) explicam que o conjunto de instruções do pipeline foi projetado para evitar esse tipo de instruções, tornando simples para os designers evitarem esse tipo de hazard, como no caso a cima que a utilização de duas memórias evita esse erro.

Hazard de Dados: **Definição**: Como esclarecido por Katz (1996) hazard de dados são causados por instruções, que para serem executadas, dependem de valores gerados por outras instruções que ainda estão sendo processadas.
Exemplo: Um exemplo simples é a execução de duas instruções do Tipo-R consecutivas, onde o resultado gerado na primeira é utilizado no processamento para gerar o resultado da segunda. Uma opção é deixar o compilador ordenar a execução das instruções de forma a evitar a execução sequencial de intruções interdependentes, porém essas dependências são muito frequentes e uma solução simples para situações como do caso apresentado a cima no exemplo, é a criação de encaminhamentos. Encaminhamentos são a adição de hardware (barramentos) de forma que permitam, assim que calculado o valor da instrução anterior, ele já esteja disponível para o uso na próxima instrução antes mesmo que ele esteja disponível no banco de registradores ou na memória.

Hazard de Controles: **Definição:** Centoducatte (1998) explica que hazard de controle, são problemas ocasionados pela execução de instruções de desvio. Ou seja, quando uma instrução deve mudar o fluxo de execução para outro ponto do código, enquanto outras instruções sequenciais a ela já começaram a ser executadas. **Exemplo:** O exemplo padrão é uma instrução branch, que o tempo entre o processar o início da instrução, executar a comparação e decida realizar o jump, toma quatro ciclos de clock. Nesse processo a instrução seguida ao branch já terá executado também três ciclos, o que é um problema, pois caso o jump não for para a próxima instrução uma inconsitência terá sido gerada.Para solucionar esse tipo de problema é sempre necessário criar stalls após o branch. Em casos normais 2 Stalls serão necessários para garantir a estabilidade do sistema, no entanto esse valor pode ser otimizado com a implementação de um hardware auxiliar para tentar prever a ação do branch, antes mesmo de ser executado, o que em casos positivos economizaria 1 Stall ao processo.

PROBLEM 3: EXERCISE 4.22 REPRODUCED FROM THE TEXTBOOK

3.1) O diagrama abaixo demonstra onde irá acontecer os dois hazard estruturais caso o fragmento de código fosse executado sem a adição dos Stalls.

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11	CC12
sd x29 , 12 (x16)	IF Acesso Mem.	ID	Exec.	M Acesso Mem.	WB							
ld x29 , 8 (x16)		IF Acesso Mem.	ID	Exec.	M Acesso Mem.	WB						
sub x17 , x15 , x14			IF Acesso Mem.	ID	Exec.	M Acesso Mem.	WB					
beqz x17 , label				IF Acesso Mem.	ID	Exec.	M Acesso Mem.	WB				
add x15 , x11 , x14					IF Acesso Mem.	-	-	ID	Exec.	M Acesso Mem.	WB	
sub x15 , x30 , x14						-	-	IF Acesso Mem.	ID	Exec.	M Acesso Mem.	WB

Figure 2. Diagrama de localização dos ciclos onde devem ser realizados os Stall
s

O primeiro Stall é no quarto ciclo, quando o store tenta escrever na memória ao mesmo tempo que o branch tenta ler a instrução também na memória. O segundo Stall deve acontecer no quinto ciclo, onde o load tenta acessar a memória, enquanto ao mesmo tempo o comando add tenta buscar na mesma memória sua instrução. Para resolver isso deve ser executado um Stall em ambos os ciclos mencionados a cima resultando no seguinte diagrama.

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11	CC12	CC13	CC14
sd x29 , 12 (x16)	IF Acesso Mem.	ID	Exec.	M Acesso Mem.	WB									
ld x29 , 8 (x16)		IF Acesso Mem.	ID	Exec.	M Acesso Mem.	WB								
sub x17 , x15 , x14			IF Acesso Mem.	ID	Exec.	M Acesso Mem.	WB							
beqz x17 , label				-	-	IF Acesso Mem.	ID	Exec.	M Acesso Mem.	WB				
add x15 , x11 , x14							IF Acesso Mem.	-	-	ID	Exec.	M Acesso Mem.	WB	
sub x15 , x30 , x14								-	-	IF Acesso Mem.	ID	Exec.	M Acesso Mem.	WB

Figure 3. Diagrama com os Stall's

3.2) Há como reorganizar as instruções para ter um ganho com a diminuição de Stalls. Partindo do princípio que as instruções que precedem o branch não podem ser realocadas para depois do branch, pois o branch pode pular para outro lugar e as instruções que o precedem já devem ter sido executadas enquanto as que o sucedem devem. Logo o inverso também é verdadeiro, não se pode trocar as instruções que sucedem com as que precedem o branch. Com isso dito, temos dois blocos que podem ser reorganizados, no entanto as intruções que sucedem o branch, tanto não geram ganho em relação a Stalls, como não podem ser trocadas devido a depedências. Já detre as intruções que precedem o Branch não se pode trocar o load com o store, entre si, já que o valor que o strore armazena pode ser diferente do que o load está carregando. Finalmente resta apenas uma opção, movimentar a instrução **sub** que precede o branch, ou para o início das instruções load e store ou para entre elas. Em ambos os casos anteriores, ao mover-se a instrução **sub**, o fragmento de código terá o ganho de um Stall, pois o load será executado logo antes do Branch, o que acarretará ao acesso á memória do load, do seu quarto ciclo, seja executado juntamente com um Stall do branch, como pode ser visto no diagrama abaixo em verde.

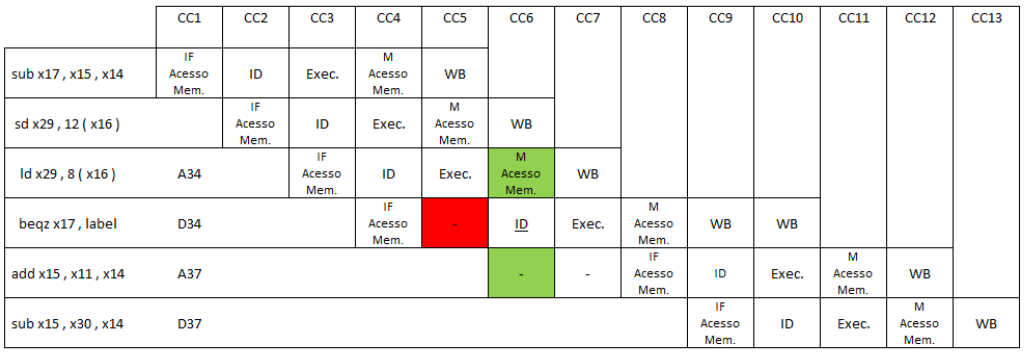

Figure 4. Diagrama com a reorganização do código, gerando a economia de um Stall

3.3) O Hazard de Dados enfrenta um problema de localização espacial de dados já processados, neste caso a adição de barramentos, também conhecido como encaminhamento, soluciona o problema, pois o simples acesso a um dado em uma posição diferente soluciona o problema. Diferentemente do anterior, o Hazard estrutural enfrenta um problema de processamento de dados, pois ele exige dois processamentos distintos de uma mesma estrutura de dados no mesmo ciclo de clock. A simples adição de barramento para melhorar a disposição espacial dos dados não faz diferença para a realização dessa tarefa. A única solução é adicionar outras estruturas de dados ao data path e apontar cada instrução para uma estrutura diferente, de forma a realizar ambos os processamentos em paralelo e ao final do ciclo ter processado ambos os dados. Outra opçao é trocar a estrutura de dados em questão por uma mais avançada que permita processamento em paralelo em um único ciclo (caso exista).

REFERENCES

1. CENTODUCATTE, P. C. **Pipeline**. Morgan Kaufmann Publishers. Unicamp. Campinas, São Paulo, Brasil, 1998. *http://www.ic.unicamp.br/ pannain/mc722/aulas/arq$_h$p6.pdf*
2. KATZ, R. H. **Lecture 7**: Introduction to Pipelining, Structural Hazards, and Forwarding. Berkeley, CA, Estados Unidos, 1996. *http://bnrg.cs.berkeley.edu/ randy/Courses/CS252.S96/Lecture07.pdf*
3. PATTERSO, D. A.; HENNESSY J. L. **Computer Organization and Design:** THE HARDWARE/SOFTWARE INTERFACE. RiscV Edition. Morgan Kaufmann. Cambridge, MA, Estados Unidos, 2018. *https://www.academia.edu/38301807/Computer$_o$rganization$_a$nd$_d$esign$_R$ISC$_V$*