Resolução da lista de Pipeline

**Links úteis:**

[Lista](https://cin.ufpe.br/~if674/arquivos/2017.2/Aulas/lista-pipe-superscalar-revisao-17-2.pdf)

[Aula sobre o Alg. de Tomasulo](https://www.youtube.com/watch?v=bOshOOpUHVU)

[Classificação dos Conflitos de dados em ILP](http://web.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/dataHazClass.html)

[Como Tomasulo resolve conflitos](https://prosuncsedu.wordpress.com/2011/12/06/how-speculative-tomasulo-prevent-wawwarrawrar-hazards/)

[AAAAARRAY](https://www.youtube.com/watch?v=l-FxY25lYzY)

**Resumo de conceitos importantes/recorrentes:**

~~Numa pipeline “ingênua”:~~

LW - 2 versao definitiva permanente ate o fim

SW - 2

R-TYPE - 2

BRANCH - 3

* ~~serão necessários 2 nops entre instruções com conflito de dados.~~
* ~~Com exceção dos LW, que são 3 nop~~
* ~~Serão necessários 3 nops após conflito de controle.~~

Numa pipeline com unidade de adiantamento (forwarding unit):

* Loads exigirão 1 nop entre instruções seguintes e conflitantes.
* Aritméticas conflitantes não necessitam de nenhum nop.
* Conflitos de controle ainda precisam ser tratados com 3 nops.

Numa pipeline com hazard detection:

* Será necessário apenas 1 nop para tratar conflito de controle.
* Loads ainda precisarão de 1 ciclo ocioso.

*1º)*

*a)*

start:

lw $t1, 4($t4)

lw $t3, 8($t4)

lw $t2, 12($t4)

lw $t0, 0($t4)

addi $t1, $t1,5

beq $t0, $t1, endX

sub $t0, $t0, $t1

sub $t3, $t3, $t2

addi $t0, $t0, 1

addi $t2, $t2, 1

sub $t2, $t2, $1

sub $t1, $t1, $3

j endY

endY:

sw $t0, 0($t4)

sw $t1, 4($t4)

sw $t2, 12($t4)

4 instruções LOAD

7 instruções aritméticas

2 instruções de desvio

3 instruções de STORE

5\*4 + 4\*7 + 3 + 4 + 4\*3 = 67 pulsos de clock.

CPI = Pulsos de Clock / Número de instruções

CPI = 67 / 16 = 4,1875

*2º) a)*

start:

lw $t1, 4($t4)

lw $t3, 8($t4)

lw $t2, 12($t4)

lw $t0, 0($t4)

addi $t1, $t1, 5

nop

nop

beq $t0, $t1, endX

nop

nop

nop

sub $t0, $t0, $t1

nop

sub $t3, $t3, $t2

addi $t0, $t0, 1

addi $t2, $t2, 1

nop

nop

sub $t2, $t2, $1

sub $t1, $t1, $t3

j endY

nop

nop

nop

endX:

sub $t3, $t3, $t2

nop

add $t0, $t0, $t1

addi $t3, $t3, -1

addi $t2, $t2, -1

nop

nop

add $t2, $t2, $t1

add $t1, $t1, $t3

endY:

nop

sw $t0, 0($t4)

sw $t1, 4($t4)

sw $t2, 12($t4)

Em Pipeline -> Pulsos de Clock = Número de instruções (Inclui-se NOP’s) + 4 = 28 + 4 = 32

CPI = Pulsos de Clock / Número de instruções (Sem NOP’s)

CPI = 32 / 16 = 2

Ocorreu uma redução de aproximadamente 52,24% no tempo de execução do código.

*2º) b)*

start:

lw $t1, 4($t4)

lw $t3, 8($t4)

lw $t2, 12($t4)

lw $t0, 0($t4)

addi $t1, $t1, 5

**~~nop~~ //removido por fwd unit**

**~~nop~~ //removido por fwd unit**

beq $t0, $t1, endX

nop *//com o uso de uma hazard detection unit, esse nop seria desnecessário via software, mas não haveria melhora no cpi.*

**~~nop~~ //removido por bypass para unidade de comparação (adiantamento de desvio)**

**~~nop~~ //removido por bypass para unidade de comparação (adiantamento de desvio)**

sub $t0, $t0, $t1

**~~nop~~ //removido por fwd unit**

sub $t3, $t3, $t2

addi $t0, $t0, 1

addi $t2, $t2, 1

**~~nop~~ //removido por fwd unit**

**~~nop~~ //removido por fwd unit**

sub $t2, $t2, $1

sub $t1, $t1, $t3

j endY

**~~nop~~ //removido por adiantamento de desvio no segundo estágio**

**~~nop~~ //removido por adiantamento de desvio no segundo estágio**

nop *//com o uso de uma hazard detection unit, esse nop seria desnecessário via software, mas não haveria melhora no cpi.*

endX:

sub $t3, $t3, $t2

add $t0, $t0, $t1

addi $t3, $t3, -1

addi $t2, $t2, -1

add $t2, $t2, $t1

add $t1, $t1, $t3

endY:

**nop //removido por fwd unit**

sw $t0, 0($t4)

sw $t1, 4($t4)

sw $t2, 12($t4)

Com a inserção de uma Forwarding Unit, para adiantar dados ao estágio de Execução e Decodificação, neste evitando conflitos de dados em desvios condicionais e, naquele, impedindo os mesmos conflitos nas demais instruções. Adicionando também uma Hazard Detection Unit para lidar com conflitos de memória e de controle.

Em Pipeline -> Pulsos de Clock = Número de instruções (Inclui-se NOP’s) + 4 = 18 + 4 = 22

CPI = Pulsos de Clock / Número de instruções (Sem NOP’s)

CPI = 22 / 16 = 1.375

*3. Um problema sério na implementação em pipeline é a ocorrência de conflitos de dados.*

*Descreva 1 técnica de software e 1 de hardware que resolvem este tipo de conflito detalhando se o desempenho é degradado ou não e se algum suporte adicional (arquitetura ou hardware) se faz necessário.*

Para evitar conflitos de controle, por meio de **software**, o compilador pode reordenar as instruções independentes assim como pode inserir nops entre instruções que geram conflitos. O desempenho utilizando essa última técnica é degradado em pelo menos 3 “bolhas” onde a busca de instruções deve ser interrompida ou nops devem ser executados, até que o valor seja escrito no registrador.

Independente do compilador, uma solução via **hardware** pode ser usada, a fim de evitar ciclos ociosos utilizando nops, através da inserção de uma unidade extra que leva dados já computados nos estágios MEM/WB ao estágio EX, quando, através de lógica interna, são detectadas dependências nos estágios anteriores do pipeline. Isso é feito curto-circuitando a saída do valor recém calculado na ALU (antes de ser escrito no estágio WB) com a entrada da ALU na próxima instrução. Vale notar que como instruções tipo LOAD só disponibilizam o valor um estágio depois, o forwarding ainda é possível, porém ainda é necessário stall de 1 estágio, com um nop via software ou bolha via hardware.

Assim, o desempenho não é degradado em instruções aritméticas, e tem apenas um atraso em instruções tipo load (assumindo que reorganização de código não foi possível/utilizada).

4.

adress valor simbólico

128 0 i

136 5 A(0)

140 2 A(1)

144 1 B(0)

148 3 B(1)

152 -3 C(0)

156 -2 C(1)

lui $3, 128

srl $3, $3, 16 ***//$3 =128***

lw $1, 0($3) ***//$1 = 0***

lui $2, 1

srl $2, $2, 16 ***//$2 =1***

beq $2, $1, fim  ***//0 == 1 ?***

loop:

lw $1, 8($3) ***//$1 =5 (1º Iteração)*** ***//$1 =2 (2º Iteração)***

lw $4, 24($3) ***//$4 =-3 (1º Iteração)*** ***//$4 =-2 (2º Iteração)***

beq $1, $4, oper\_b ***//5 == -3? (1º Iteração) //2==-2? (2º Iteração)***

j oper\_c

oper\_b:

lw $4, 16 ($3)

oper\_c:

jal swap ***//$1 = 5 ^ $4 = -3 (1ª iteração) //$1 = 2 ^ $4 = -2 (2ª iteração)***

add $5, $5, $2 ***//$5 =1 (1º Iteração) //$5 =2 (2º Iteração)***

addi $3, $3, 4 ***//$3 =132 (1º Iteração) //$3 =136 (2º Iteração)***

slti $6, $5, 2 ***//1<2 ? (1º Iteração) //2< 2 ? (2º Iteração)***

bne $6, $0, loop ***//1 == 0?(1ª iteração) //0 == 0? (2ª iteração)***

fim: break ***// FIM!***

swap: [*https://en.wikipedia.org/wiki/XOR\_swap\_algorithm*](https://en.wikipedia.org/wiki/XOR_swap_algorithm)

***// O algoritmo na lista é diferente do swap por XOR***

xor $1, $1, $4 ***//$1 =-8 (1º Iteração)*** ***//$1 = -4 (2º Iteração)***

xor $4, $1, $4 ***//$4 =5 (1º Iteração) //$4 = 2 (2º Iteração)***

xor $1, $1, $4 ***//$1 =-3 (1º Iteração) //$1 = -2 (2º Iteração)***

sw $1, 8($3) ***//A(0) = -3 (1º Iteração) //A(1) = -2 (2º Iteração)***

jr $31

Acima de loop: 6 instruções. (3+4+5+3+4+4 = 23)

em Loop: 4 instruções repetidas duas vezes. (5+5+4+3 = 17 x 2 = 34)

oper\_c: 5 instruções repetidas duas vezes. (4+4+4+4+4 = 20 x 2 = 40)

swap: 5 instruções repetidas duas vezes. (4+4+4+4+4 = 20 x 2 = 40)

Total de pulsos: 40+40+34+23 = 137.

Número de instruções: 34. ***(BREAK NAO CONSIDERADO NO CALCULO)***

CPI = Pulsos de Clock / Número de instruções

CPI = 137 / 34 = 4,03

4. b) **NOPS ~~TACHADOS~~ NÃO SERÃO UTILIZADOS NA LETRA C**

lui $3, 128

~~nop~~

~~nop~~

nop

srl $3, $3, 16

~~nop~~

~~nop~~

lw $1, 0($3)

lui $2, 1

~~nop~~

~~nop~~

nop

srl $2, $2, 16

~~nop~~

~~nop~~

beq $2, $1, fim

nop

~~nop~~

~~nop~~

loop:

lw $1, 8($3)

lw $4, 24($3)

nop

~~nop~~

~~nop~~

beq $1, $4, oper\_b

nop

~~nop~~

~~nop~~

j oper\_c

nop

~~nop~~

~~nop~~

oper\_b:

lw $4, 16 ($3)

oper\_c:

jal swap

nop

~~nop~~

~~nop~~

add $5, $5, $2

addi $3, $3, 4

~~nop~~

slti $6, $5, 2

~~nop~~

~~nop~~

bne $6, $0, loop

nop

~~nop~~

~~nop~~

fim: break

swap:

xor $1, $1, $4

~~nop~~

~~nop~~

xor $4, $1, $4

~~nop~~

~~nop~~

xor $1, $1, $4

~~nop~~

~~nop~~

sw $1, 8($3)

jr $31

nop

~~nop~~

~~nop~~

O número de instruções executadas continua sendo a mesma, 34

Porém muitos NOPs foram colocados, 13 no start, 9 no loop e no oper\_c assim como no swap, tendo um total de 40 ciclos ociosos. Note que o NOP do jal não é só executado após a leitura do jal, ele também será lido após o retorno. Por isso, no oper\_c, teremos um ciclo ocioso a mais.

Em Pipeline -> Pulsos de Clock = Número de instruções (Inclui-se NOP’s) + 4 = 76 + 4 = 80

CPI = Pulsos de Clock / Número de instruções (Sem NOP’s)

CPI = 80 / 34 = 2,35.

Ocorreu uma redução de aproximadamente 41,69% no tempo de execução do código.

*4 c) Pode, só ver a 2 questão.*

31 nops foram removidos, 10 no start, 6 no loop, 7 no oper\_c, 8 no swap

Com uma hazard detection unit, os nops se tornam desnecessários, estão presentes no código para abstração, com isso, o ciclo ocioso após o JAL passa a ser executado apenas uma vez.

Em Pipeline -> Pulsos de Clock = Número de instruções (Inclui-se NOP’s) + 4 = 43 + 4 = 47

CPI = Pulsos de Clock / Número de instruções (Sem NOP’s)

CPI = 47 / 34 = 1,38.

Ocorreu uma redução de aproximadamente 65,76% no tempo de execução do código.

*5. Considere o programa MIPS descrito abaixo o qual opera em um array de elementos*

*inteiros. O registrador R1 aponta para o início do array (da primeira posição) e o*

*registrador (R2?) aponta para a última posição do array. O array possui 1000 elementos.*

*O projeto básico é um pipeline de cinco estágios.*

ANDI R3, R3, 0 # R3 = 0

LOOP:

LW R4, 0(R1) # R4 = array[i]

SUB R5, R4, R4 # Ta errado, Edna querida ;3

ADD R5, R3, R5 # R5 += R3

SW R5, 0(R1) # new[i] = old[i - 1] + old[i]\*old[i]

ADDI R3, R4, 0 # R3 = R4

ADDI R1, R1, 8 # R1 += 8

BNE R1, R2, LOOP # do { … } while( i < array.size() );

*a. Quais as dependências que existem no programa?*

*Explique estas dependências e quais os problemas que podem causar?*

LW -> SUB: O sub não usará o valor lido pelo LW

SUB -> ADD: O add usará um valor datado

ADD -> SW: O sw não guardará um valor atualizado

ADDI -> BNE: O bne utilizará dados antigos

*b.)*

**Escalonamento Estático:**

|  |  |  |
| --- | --- | --- |
| **STATIC Dual Issue** | | |
| **Ciclo** | **ALU/BRANCH** | **LOAD/STORE** |
| 1 | ANDI R3, R3, 0 | NOP |
| **LOOP (500 iterações)** | | |
| 2 | NOP | LW R4 0(R1) |
| 3 | Delay do LW ao SUB em função do R4 | |
| 4 | SUB R5, R4, R4 | NOP |
| 5 | ADD R5, R3, R4 | NOP |
| 6 | ADDI R3, R4, 0 | SW R5 0 (R1) |
| 7 | ADDI R1, R1, 8 | NOP |
| 8 | BNE R1, R2, LOOP | NOP |

Temos uma instrução assim como um ciclo de clock antes do LOOP, após ele, teremos 500 repetições, já que temos um array de 1000 elementos e o índice, R1, é incrementado em 8 a cada iteração, note que estamos utilizando inteiros, que ocupam 4 bytes na memória. A cada iteração, executaremos 7 instruções em 7 ciclos de clocks, então o nosso IPC será dado por:

**Loop Unrolling:**

|  |  |
| --- | --- |
| *Loop Unrolling* | *Code Rearrange* |
| ANDI R3, R3, 0 | ANDI R3, R3, 0 |
| *Loop:* | *Loop:* |
| LW R4, 0(R1) | TLW R4, 0(R1) |
| SUB R5, R4, R4 | ADDI R1, R1, **32\*** |
| ADD R5, R3, R5 | LW R6, -24(R1) |
| SW R5, 0(R1) | LW R8, -16(R1) |
| ADDI R3, R4, 0 | LW R10, -8(R1) |
| LW R6, 8(R1) | SUB R5, R4, R4 |
| SUB R7, R6, R6 | SUB R7, R6, R6 |
| ADD R7, R5, R7 | SUB R9, R8, R8 |
| SW R7, 8(R1) | SUB R11, R10, R10 |
| ADDI R5, R7, 0 | ADD R5, R3, R5 |
| LW R8, 16(R1) | ADD R7, R5, R7 |
| SUB R9, R8, R8 | ADD R9, R7, R9 |
| ADD R9, R7, R9 | ADD R11, R9, R11 |
| SW R9, 16(R1) | SW R5, 0(R1) |
| ADDI R7, R9, 0 | SW R7, 8(R1) |
| LW R10, 24(R1) | SW R9, 16(R1) |
| SUB R11, R10, R10 | SW R11, 24(R1) |
| ADD R11, R9, R11 | ADDI R3, R4, 0 |
| SW R11, 24(R1) | ADDI R5, R7, 0 |
| ADDI R9, R11, 0 | ADDI R7, R9, 0 |
| ADDI R1, R1, **32\*** | ADDI R9, R11, 0 |
| BNE R1, R2, LOOP | BNE R1, R2, LOOP |

|  |  |  |
| --- | --- | --- |
| **Loop Unrolling + STATIC Dual Issue** | | |
| **Ciclo** | **ALU/BRANCH** | **LOAD/STORE** |
| 1 | ANDI R3, R3, 0 | nop |
| - | ***loop*** | |
| 2 | ADDI R1, R1, **32** | LW R4, 0(R1) |
| 3 | nop | LW R6, -24(R1) |
| 4 | SUB R5, R4, R4 | LW R8, -16(R1) |
| 5 | SUB R7, R6, R6 | LW R10, -8(R1) |
| 6 | SUB R9, R8, R8 | *nop* |
| 7 | SUB R11, R10, R10 | *nop* |
| 8 | ADD R5, R3, R5 | *nop* |
| 9 | ADD R7, R5, R7 | SW R5, 0(R1) |
| 10 | ADD R9, R7, R9 | SW R7, 8(R1) |
| 11 | ADD R11, R9, R11 | SW R9, 16(R1) |
| 12 | ADDI R3, R4, 0 | SW R11, 24(R1) |
| 13 | ADDI R5, R7, 0 | *nop* |
| 14 | ADDI R7, R9, 0 | *nop* |
| 15 | ADDI R9, R11, 0 | *nop* |
| 16 | BNE R1, R2, LOOP | *nop* |

Obtém-se um melhoramento no IPC ao custo de se utilizar mais registradores e mais código. Ao “desenrolar” o loop quatro vezes, podemos nos aproveitar melhor da técnica de escalonamento estático, fazendo com que o número médio de instruções executadas em cada ciclo de clock aumente consideravelmente.

*6) 7)*

ANDI R3, R3, 0 # R3 = 0

LOOP:

LW R4, 0(R1) # R4 = array[i]

SUB R5, R4, R4 # Ta errado, Edna querida ;3

ADD R5, R3, R5 # R5 += R3

SW R5, 0(R1) # new[i] = old[i - 1] + old[i]\*old[i]

ADDI R3, R4, 0 # R3 = R4

ADDI R1, R1, 8 # R1 += 8

BNE R1, R2, LOOP # do { … } while( i < array.size() );

1ª iteração:

Initial Branch PC = 00

executando BNE = errou, altera previsão para 01

**total de erros: 1**

2ª iteração:

Initial Branch PC = 01

executando BNE = errou, altera previsão para 11

**total de erros: 1**

3ª iteração:

Initial Branch PC = 11

executando BNE = acertou, continua em 11

**total de erros: 0**

4ª-499ª iteração:

Initial Branch PC = 11

executando BNE = acertou, mantém previsão em 11

**total de erros: 0**

500ª iteração:

Initial Branch PC = 11

executando BNE = errou, altera previsão para 10

**total de erros: 1**

Sabemos pelas questões anteriores que o loop desse programa irá se repetir por 500 iterações. Assim, o total de erros de predição do programa será de 3 em 500 desvios tomados.

Precisão da predição: 99.4%

00 - not taken

01 - not taken

10 - taken

11 - taken

Utilizando uma BHT, armazenar-se-ia o estado da predição dos desvios de acordo com seus bits menos significativos. Supondo que os desvios compartilhem os mesmos bits menos significativos, no próximo desvio, a predição apontaria que o desvio seria tomado. Uma desvantagem é possuir mais hardware no processador, além de lidar com predições erradas, as consertando e desfazendo operações indevidas. Em troca, algumas repetições costumam executar muitos desvios, como ler caracteres de uma string, a fim de diminuir o tempo de ciclos ociosos, a CPU irá executar instruções baseadas na predição melhorando o tempo de execução.

*8)*

[*https://prosuncsedu.wordpress.com/2011/12/06/how-speculative-tomasulo-prevent-wawwarrawrar-hazards/*](https://prosuncsedu.wordpress.com/2011/12/06/how-speculative-tomasulo-prevent-wawwarrawrar-hazards/)

Queremos tratar as seguintes dependências de dados através do algoritmo de Tomasulo:

**RAW** - Preciso que o registrador seja atualizado antes que eu leia

**WAR** - Preciso ler o registrador antes que seja atualizado

**WAW** - Preciso ter certeza que o valor escrito no registrador é realmente o valor final desejado.

O algoritmo de tomasulo funciona da seguinte maneira:

1. As instruções do programa são armazenadas em uma fila. Assim que possível, elas são encaminhadas para a *reservation station* correspondente, onde são cadastradas junto com os operandos que desejam utilizar
2. Execução: Aqui as instruções na *reservation station*  são executadas (não necessariamente na mesma ordem que foram cadastradas) após as dependências de dados serem verificadas (através da verificação dos outros operandos em uso em todas as *reservation stations* através da tabela de reservas*).* Se houver alguma dependência, a instrução é “segurada” até que o dado necessário esteja disponível. Quando isso acontece, a tabela de reservas é atualizada e a instrução pode utilizar o operando. **(“segurar” a instrução resolve o conflito RAW)*.***
3. Write result: Os resultados calculados nas *reservation stations* são enviados para todas as outras e para o *reorder buffer* através de barramento compartilhado, para que sejam utilizados pelas instruções que estão esperando. A *station* é marcada como disponível para receber outra instrução e o registrador correspondente ao resultado é escrito.

O **WAW** será evitado utilizando o REORDER BUFFER, descrito na questão seguinte, independente da ordem de execução, o buffer escreverá os resultados na ordem certa.

O **WAR** será resolvido lendo dados antes que sejam sobrescritos por instruções seguintes. O algoritmo de Tomasulo lida com isso, porque quando uma instrução é despachada em ordem, a *reservation station* armazena de onde os operandos vêm, assim, se uma instrução de leitura for enviada para execução antes de outra de escrita, terá seu operando antes dele ser atualizado por outra instrução de escrita.

*9)* O reorder buffer é uma excelente ferramenta para lidar com escalonamento dinâmico com especulação pois ele mantém numa memória temporária os valores das operações sendo realizadas nas unidades de reserva. Isso significa que ao precisar de um operando, se este não estiver no reorder buffer, significa que ainda será produzido pela sua unidade funcional. Uma analogia possível é a de que o reorder buffer funciona de maneira semelhante a uma forwarding unit num pipeline estático, adiantando os resultados das operações.

Num escalonamento dinâmico com especulação, um quarto estágio do algoritmo é adicionado, onde os valores no reorder buffer passam pelo processo de “commit”: São escritos definitivamente no registrador de destino. Como dito anteriormente, o reorder buffer faz com que essa escrita aconteça na ordem especificada no programa, evitando conflitos WAW.

Se tratando de especulação, o reorder buffer é especialmente útil quando uma previsão errada é feita: Podemos facilmente fazer um “flush” dos dados nele contidos, e descartar quaisquer resultados enganosos que possam ter vindo de uma previsão errada de pegar ou não um desvio condicional. Ou seja, o buffer contém os resultados das instruções que ainda **não terminaram**. Assim, se foram instruções que não deveriam ter sido executadas em primeiro lugar, o buffer é limpo, e retomamos a execução a partir das instruções certas. Caso tenha sido uma previsão correta, a instrução termina e as mudanças temporárias no buffer são escritas nos registradores.

Quais as principais modificações devem ser feitas em um processador para se introduzir suporte

Para o processador especular instruções, se faz necessário armazenar as instruções executadas em uma memória auxiliar(reorder buffer) já mencionada anteriormente. Além disso, é preciso de um comando de limpeza no buffer caso a especulação seja errada. Dessa forma, garante-se que o código seja executado de maneira segura e com eficiência. Também é necessário a inserção de uma *commit unit*, que terá o papel de controlar as atualizações nos registradores e também na memória através de um barramento compartilhado. Essencialmente, essa unidade decide quando é ou não seguro escrever o resultado de uma operação nos registradores.

