# Implementing a forwarding and stall unit in a pipelined architecture

Arquitectura de Computadores Avançada
Universidade de Aveiro

Diogo Silva 60337 Eduardo Sousa 68633

18 de Novembro de 2014

# Conteúdo

# Capítulo 1

# Implementação do Branch

## 1.1 Estratégia

No datapath original temos a leitura dos operandos na fase ID, a comparação entre eles é feita na fase EXE (BranchTarget também é calculado na fase EXE usando um somador) utilizando a ALU e só na fase MEM é que o branch é resolvido, tal como mostra a figura abaixo.

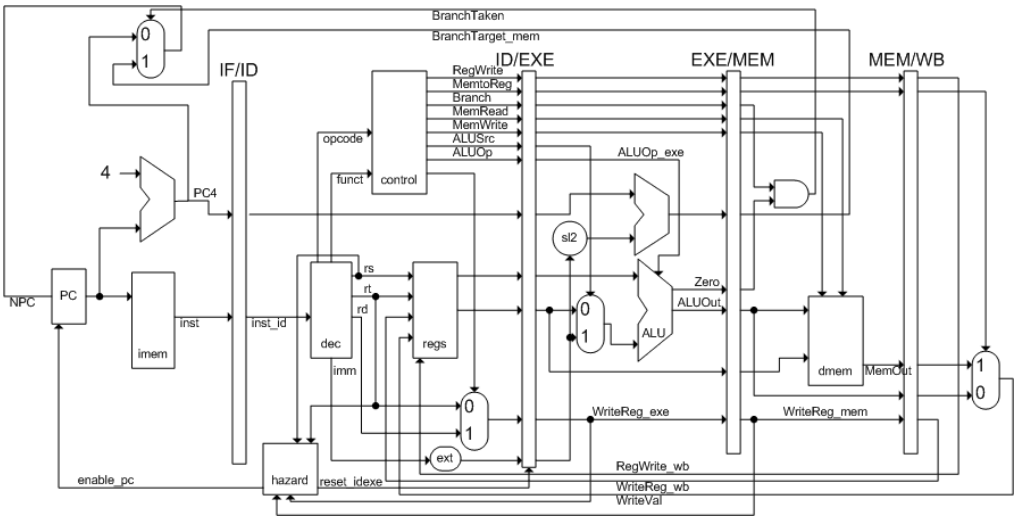


Figura 1.1: Versão original do datapath

Tendo em conta esta situação, a estratégia utilizada foi de passar a fazer a comparação dos operandos na própria fase ID, utilizando um comparador, deixando assim de precisar da ALU na fase EXE.

Como a comparação dos valores dos registos passou a ser feita na fase ID, basta passar a porta AND, que recebe o sinal do comparador e o sinal a indicar se a intrução é do tipo branch, da fase MEM para a fase ID.

Para que a instrução do tipo branch possa ser totalmente resolvida na fase ID, teve de se utilizar um somador para se obter o BranchTarget na fase ID, tal como ele já existia na fase EXE.

Como no caso do resultado da instrução Branch ser taken é necessário alterar o PC, foi utilizado um multiplexer com 2 entradas, onde a entrada 0 é o PC+4 e a entrada 1 é o BranchTarget, em que o sinal de controlo desse multiplexer é o sinal que sai da porta AND, que será chamado de BranchTaken. A saída desse multiplexer vai fornecer o próximo endereço ao PC.

O sinal BranchTaken terá tambem de ser enviado para a Hazard Unit, para caso o branch ser Taken ser introduzido um bubble, de forma a descartar a instrução que foi carregada na fase IF. Resolvendo se assim uma instrução do tipo beq na fase ID.

Para ser também possível resolver uma instrução do tipo bne, foi adicionado uma porta XOR onde recebe o sinal BranchNotEqual que sinaliza que a instrução do tipo bne e recebe o sinal Equal que sai do comparador dos valores dos registos.

O sinal que sai da porta XOR vai ligar a porta AND substituindo o sinal que vinha do comparador. A figura seguinte mostra o resultado final desta tarefa.

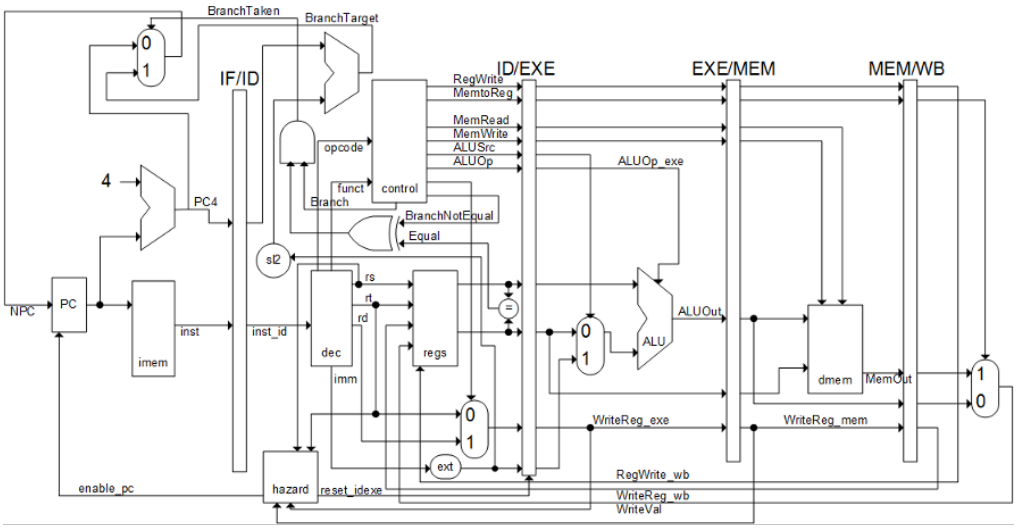


Figura 1.2: Datapath resultante da Task 1

Os testes efectuados sobre esta task foram um “beq” taken e outro not taken e um “bne” taken e outro not taken.

# Capítulo 2

# Implementação dos Jumps

Como se pode verificar na imagem seguinte foi criado dois sinais de controlo extra, JumpOnRegister e Jump para identificar o jump respectivo, para além disso também foi criada uma unidade de controlo que permite calcular o Jump Address.

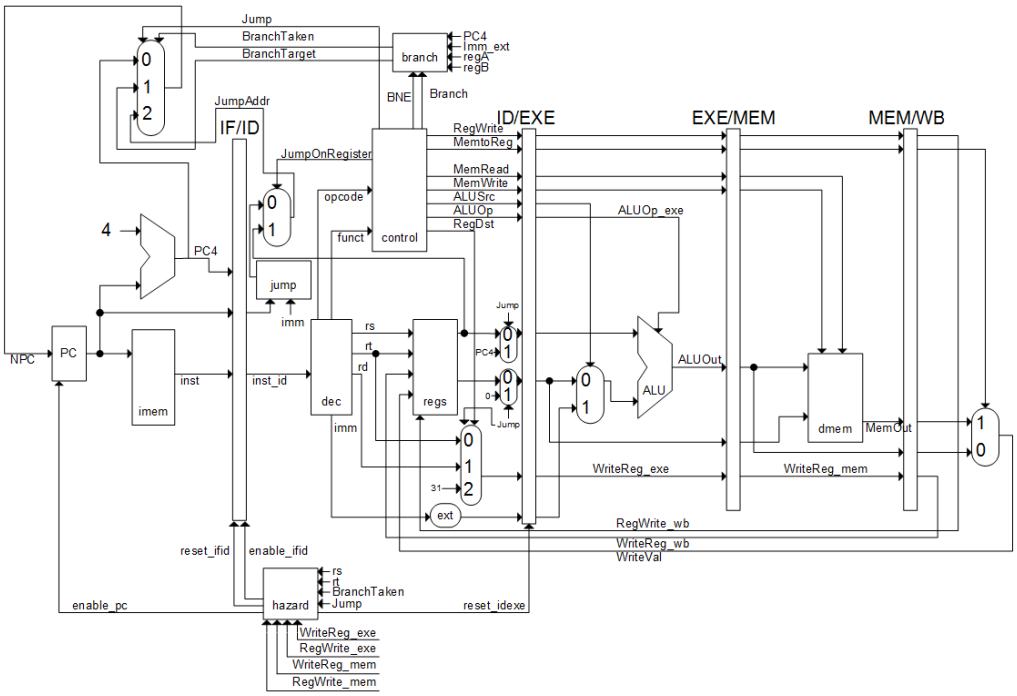


Figura 2.1: Datapath resultante da Task 2

### 2.0.1 Resolução do Endereço de Salto

A unidade de jump recebe o valor do PC desta instrução e o valor do immediate, tendo em conta isto, vai calcular o endereço que vai passar para o

PC, fazendo shift left de dois bits ao immediate e concatenando os quatro bits mais significativos do PC desta instrução ao valor do immediate já com a operação de shift efectuada.

Como o valor a ser passado ao PC depende da instrução do tipo jump, pode ser preciso escolher entre o valor calculado pela unidade de jump e o valor guardado no registo. O jalr e jr usam o valor resultante leitura do registo enquanto que o jal e o j usam o valor calculado na unidade de jump referida anteriormente, usando o sinal JumpOnRegister para destinguir ambos os sinais.

A entrada 0 desse multiplexer será o valor que sai da unidade de jump e a entrada 1 será o valor que vem do registo. O multiplexer de duas entradas já existente para selecionar se o valor a passar para PC era o PC+4 ou o BranchTarget vai ser trocado por um multiplexer de três entradas em que os seus sinais de controlo serão o Jump e o Branch Taken, sendo que o sinal Jump será o bit mais significativo da combinação.

A entrada 0 desse multiplexer será o PC+4, a entrada 1 será o BranchTarget e a entrada 2 será o valor que for selecionado no multiplexer que escolhe entre a saída da unidade de jump e o valor que vem do registo.

### 2.0.2 Escrita do endereço $ra

Como certas instruções do tipo jump tem-se a necessidade de escrever o valor do PC+4 dessa instrução no registo $ra (registo 31), sendo assim optou-se por colocar dois multiplexers de duas entradas nas saídas dos registos na fase ID, para selecionar que valor será passado para a fase EXE. Ambos os multiplexers têm o sinal de controlo Jump.

No multiplexer à saída do registo A, a entrada 0 é o valor que provem do registo e a entrada 1 é o PC+4. No multiplexer à saída do registo B, a entrada 0 é o valor que provem do registo B e a entrada 1 terá o valor 0.

Isto vai permitir que a operação de soma no próximo ciclo, faça PC+4 + 0, não alterando o valor do PC+4 que irá ser escrito na fase WB no registo $ra, mas para isso acontecer é preciso passar o número desse registo para as fases seguintes.

Para que isso fosse possível, foi alterado o multiplexer que seleciona o valor de WriteReg, passando a ser um multiplexer de três entradas, onde os sinais de controlo são o Jump e o RegDst, sendo que o Jump é o bit mais significativo da combinação.

A entrada 0 será o valor de rt, a entrada 1 será o valor de rd e a entrada 2 será 31, indicando o número do registo 31. Foi necessário alterar a unidade de hazard de forma a fazer bubble em IF/ID, da mesma forma que o branch faz.

### 2.0.3 Testes realizados

Foram realizados testes individuais para cada uma das situações com testes simples, por exemplo, jump para o endereço 0.

Para além dos testes individuais ainda foi criado um teste que usasse o $ra mais que uma vez consecutiva, de forma a verificar que tudo estava de acordo com o pretendido.

```
1  0x0c000007      #       jal atoi
2  0x00000000      #       nop
3  0x00000000      #       nop
4  0x00000000      #       nop
5  0x03e00008      #       jr $ra
6  0x00000000      #       nop
7  0x00000000      #       nop
8  0x00000000      # atoi: nop
9  0x00000000      #       nop
10 0x00000000      #       nop
11 0x00000000      #       nop
12 0x03e00009      #       jalr    $ra
13 0x00000000      #       nop
14 0x00000000      #       nop
15
```

Figura 2.2: Teste de jumps

O teste consiste em saltar para um label, usando o jal em que após 3 nops, vai saltar usando o jalr $ra, ou seja, vai voltar ao local onde deixou após entrar no primeiro jal, indo assim para a linha dois, não esquecendo que vai guardar o $ra, após isso vai voltar novamente para a linha 13 quando chegar ao jr $ra.

# Capítulo 3

# Divisão da fase MEM em duas fases, MEM1 e MEM2

Para separar a fase MEM em duas fases, foi necessário colocar um registo no meio da fase MEM (registo MEM1/MEM2), subdividindo-a em MEM1 e MEM2 e passar a memória de assíncrona para síncrona (da qual nos foi fornecido o módulo).

Antes na fase MEM, eram passados os valores para dentro da memória com os respectivos sinais de controlo para identificar se era uma escrita ou uma leitura saindo no mesmo ciclo de relógio o resultado, mas agora as entradas na memória aparecem na fase MEM1 e as saídas (que é apenas a MemOut) aparecem na MEM2, para além disso foi necessário propagar os sinais pelas duas fases para chegarem a fase WB, passando os de registo em registo, como por exemplo o WriteReg, que vai passar pelos registos EXE/MEM1, MEM1/MEM2 e MEM2/WB.

A figura seguinte mostra o resultado final.

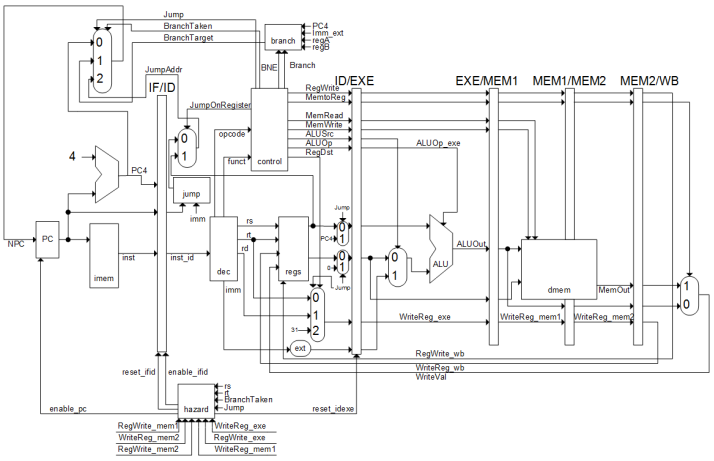


Figura 3.1: Datapath resultante da Task 3

# Capítulo 4

# Forwarding

## 4.1 Análise Prévia

### 4.1.1 Identificação e exemplos do forward

Antes de qualquer identificação de forward, o forward é feito da fase para o registo de cada fase, sendo que se decidiu resolver as depedências com Branch e Jumps para o registo IF/ID que é a única possibilidade, mas para as outras situações que só precisam do valor na fase EXE, optou-se por fazer o forward para o registo.

**EXE -> IF/ID**

Vai existir forwarding da fase EXE para o registo IF/ID, quando o valor a ser lido do registo pela instrução em ID só for calculado pela instrução em EXE, assim sendo o valor calculado pela ALU em EXE (ALUOut) vai ser passado para o registo IF/ID. Este tipo de forwarding vai ser utilizado quando o tipo de instrução em ID for branch ou jump e o valor WriteReg_exe for igual ou a rs ou a rt.

Exemplo de código:
add $1, $2, $3
nop
beq $4, $1, BRANCH

Esta situação também pode ocorrer com jr $1, ou jalr $1, porque também são resolvidos na fase ID.

**MEM1 -> IF/ID**

Vai existir forwarding da fase MEM1 para o registo IF/ID, quando o valor a ser lido do registo pela instrução em ID foi calculado por uma instrução que está em MEM1, assim sendo o valor de ALUOut_mem1 vai ser passado

para o registo IF/ID. Este tipo de forwarding vai ser utilizado quando o tipo de instrução em ID for branch ou jump e o valor WriteReg_mem1 for igual ou a rs ou a rt.

Exemplo de código:
add $1, $2, $3
nop
nop
beq $7, $1, BRANCH

Esta situação também pode ocorrer com jr $1, ou jalr $1, porque também são resolvidos na fase ID.

Os casos de MEM2 -> IF/ID não precisam de forward porque uma instrução vai avançar para WB e a outra para ID, e tendo em conta que WB vai escrever na transição de 1 para 0, não precisa de qualquer forward.

### EXE -> ID/EXE

Vai existir forwarding da fase EXE para o registo ID/EXE, quando o valor que foi calculado pela instrução que está em EXE(ALUOut) é necessário para realizar a instrução que vai passar para EXE. Este tipo de forwarding vai ser utilizado quando o tipo de instrução que vai passar para EXE, ou seja, se a depedência não for para um Branch ou um Jump e o valor WriteReg_exe da instrução que está em EXE for igual ou a rs ou a rt da instrução que irá passar para EXE.

Exemplo de código:
sub $1, $2, $3
add $4, $1, $2

### MEM1 -> ID/EXE

Vai existir forwarding da fase MEM1 para o registo ID/EXE, quando o valor que foi calculado pela instrução que está em MEM1(ALUOut_mem1) é necessário para realizar a instrução que vai passar para EXE. Este tipo de forwarding vai ser utilizado quando o tipo de instrução que vai passar para EXE, ou seja, se a depedência não for para um Branch ou um Jump e o valor WriteReg_mem1 da instrução que está em MEM1 for igual ou a rs ou a rt da instrução que irá passar para EXE.

Exemplo de código:
add $1, $2, $3
sub $4, $5, $6
add $7, $1, $8

Neste trecho de código vai haver forward do valor resultante do primeiro add para o RS do último add.

**MEM2 -> ID/EXE**

Vai existir forwarding da fase MEM2 para o registo ID/EXE, quando o valor que foi calculado pela instrução ou o valor que a instrução leu de memória que está em MEM2(ALUOut_mem2 ou MemOut) é necessário para realizar a instrução que vai passar para EXE. Este tipo de forwarding vai ser utilizado quando o tipo de instrução que vai passar para EXE não for branch ou jump e o valor WriteReg_mem2 da instrução que está em MEM1 for igual ou a rs ou a rt da instrução que irá passar para EXE.

Exemplo de código:
add $1, $2, $3
and $4, $5, $6
or $7, $8, $9
add $10, $1, $8

Neste trecho de código vai haver forward do valor resultante do primeiro add para o RS do último add, tal como acontecia no caso anterior. Também poderia ser um load word em vez de um add, tendo em conta que o valor também pode vir do MemOut.

**MEM2 -> EXE/MEM1**

Vai existir forwarding da fase MEM2 para o registo EXE/MEM1, quando um valor que é lido na memória em MEM2 (MemOut) vai ser utilizado pela instrução que está em EXE e vai passar para MEM1. Este tipo vai ser utilizado quando temos um lw (load word) em MEM2 e temos um sw (store word) em EXE, ou seja quando fazemos a leitura de um valor de memória e ele vai voltar a ser escrito pela instrução que se encontra em EXE.

Exemplo de código:
lw $1, 0($5)
add $2, $3, $4
sw $1, 0($4)

Este caso é uma situação muito específica em que o store word não precisa de calcular o endereço $1 para avançar para a fase MEM1, no entanto precisa do valor $1 no limite em MEM1, porque é o valor que vai escrever, se o load word escrevesse no $4, esta dependência já não podia ser resolvida assim, porque o endereço é calculado com o offset na fase EXE, logo, o store word teria de fazer um STALL na fase EXE, os stalls são analisados no próximo capítulo em detalhe.

## 4.2 Implementação

### 4.2.1 Forwarding para IF/ID

As fases que vão fazer forwarding para IF/ID são EXE e MEM1. Dessas fases podem ser passados os seguintes valores: ALUOut e ALUOut_mem1. Para que estes valores sejam guardados nos registos IF/ID foi necessário criar esses registos e passar os valores para a fase ID. A figura seguinte mostra o resultado.

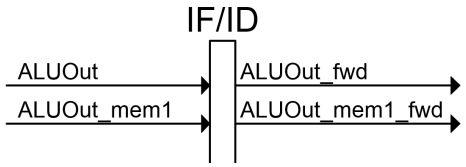


Figura 4.1: Passagem de valores pelo registo IF/ID

Com estas alterações, foi necessário introduzir lógica adicional em ID. Foram colocados multiplexers a saída do registo da fase IF/ID de forma a controlar os registos que vêm do registo da fase, podendo assim seleccionar o valor que a unidade de forwarding indica.

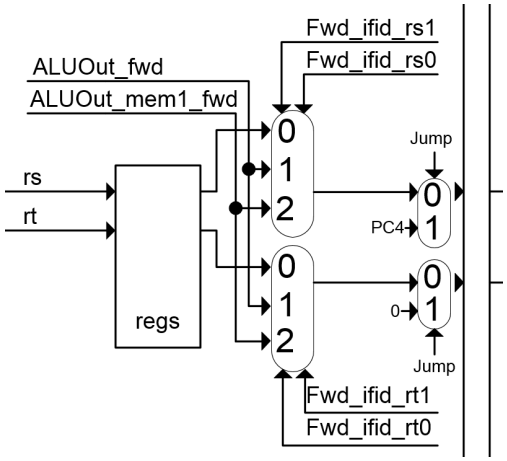


Figura 4.2: Multiplexer da fase ID

### 4.2.2 Forwarding para ID/EXE

As fases que vão fazer forwarding para ID/EXE são EXE, MEM1 e MEM2. Dessas fases podem ser passados os seguintes valores: ALUOut, ALUOut_mem1, ALUOut_mem2 e MemOut. Para que estes valores sejam guardados nos registos ID/EXE foi necessário criar esses registos e passar os valores para

a fase EXE. A figura seguinte mostra a passagem dos valores pelo registo ID/EXE.

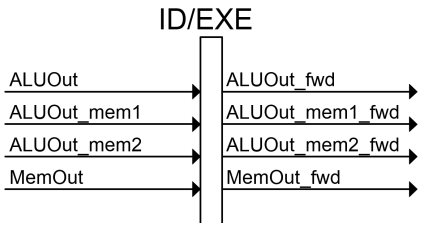


Figura 4.3: Passagem de valores pelo registo ID/EXE

Com estas alterações, foi necessário introduzir lógica adicional em EXE.

Foram colocados dois multiplexers (precisa-se de dois de 4 que é para seleccionar entre o RS e o RT) de 4 entradas para selecionar qual dos valores de que foi feito forwarding deve ser selecionado e dois multiplexers de duas entradas para selecionar entre o valor que foi selecionado pelo multiplexer de quatro entradas e o valor que vem da fase anterior.

Sendo que o segundo multiplexer serve para indicar se existe forward ou não. A figura seguinte mostra a estrutura da fase, apresentando ainda um exemplo do valor da AluOut a ir para o registo ID/EXE, saindo no próximo ciclo de relógio como AluOut_fwd. Os selectors dos multiplexers são gerados pela lógica da Unidade de Forwarding.

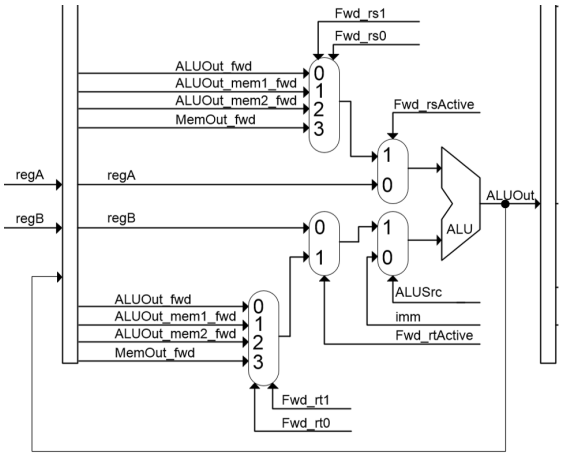


Figura 4.4: Multiplexers da fase EXE

### 4.2.3 Forwarding para EXE/MEM1

A fase que vai fazer forwarding para EXE/MEM1 é MEM2. Dessa fase podem ser passados os seguintes valores: WriteVal e MemOut.

Para que estes valores sejam guardados nos registos EXE/MEM1 foi necessário criar esses registos e passar os valores para a fase MEM1.

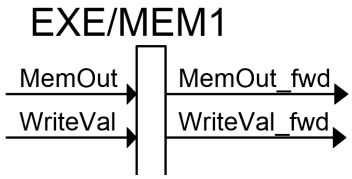


Figura 4.5: Passagem de valores pelo registo EXE/MEM1

Com estas alterações, foi necessário introduzir lógica adicional em MEM1.

Foi colocado um multiplexer de 3 entradas para selecionar entre o valor passado pela fase anterior e os valores que foram passados por forwarding para a fase, sendo os selectores gerados pela unidade de forwarding mais uma vez.

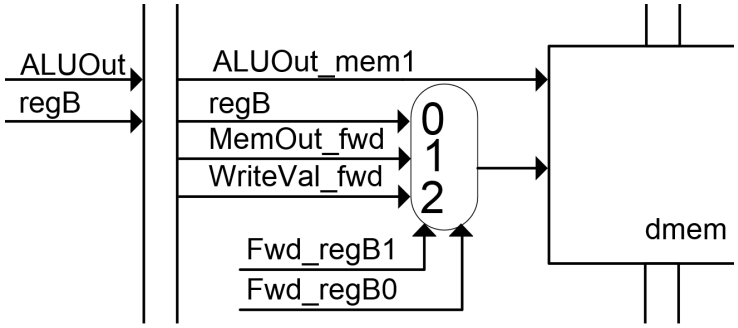


Figura 4.6: Multiplexer da fase MEM1

# Capítulo 5

# Stalling Unit

## 5.1 Identificação de casos possíveis

Para além da unidade de hazard descartar as instruções em IF caso a instrução em ID seja do tipo branch taken ou do tipo jump, a hazard unit vai fazer stall em mais 3 situações distintas, introduzir um stall num registo respectivo e fazer bubble no próximo.

### 5.1.1 Stall no registo IF/ID

Supondo o seguinte exemplo:

```
add $1, $2, $3
bne $1, $5, BRANCH
```

Nesta situação a instrução add só vai poder fazer forwarding do valor calculado em EXE (EXE -> IF/ID), assim sendo a instrução branch if not equal só vai poder passar para ID quando a instrução add estiver a terminar a fase EXE e fizer forwarding do valor calculado para IF/ID.

Esta situação é igual a mesma que ter um jr ou um jalr em vez do branch.

Neste caso a unidade de hazard, durante um ciclo, vai fazer stall do registo IF/ID e introduzir uma bolha no registo ID/EXE.

### 5.1.2 Stall no registo ID/EXE

Considerando o seguinte exemplo:

```
lw $3, 0($1)
sub $2, $3, $1
```

Nesta situação a instrução load word só vai poder fazer forwarding do valor lido de memória na fase MEM2(MEM2 -> ID/EXE), assim sendo a instrução sub tem de esperar que a instrução lw esteja a terminar a fase MEM2, para a mesma fazer forwarding do valor lido de memória.

Quando a instrução lw estiver a passar da fase MEM2 para a fase WB, a instrução sub estará a passar da fase ID para a fase EXE.

Neste caso a unidade de hazard, durante dois ciclos, vai fazer stall do registo ID/EXE e introduzir bolhas no registo EXE/MEM1.

Considere-se o seguinte exemplo:

lw $3, 0($1)

sw $2, 0($3)

Neste caso, o valor do $3 é preciso na fase EXE para calcular o seu endereço, logo ainda é abrangido por esta situação, é um caso limite deste forward.

### 5.1.3 Stall no registo EXE/MEM1

Este é um caso muito específico que é quando se tem um load word seguindo de um store word em que o load word escreve no registo em que o store word vai escrever o conteúdo desse registo num específico endereço de memória, ou seja, no campo RT do store word.

Exemplo:

lw $1, 0($2)

sw $1, 0($3)

Nesta situação a instrução load word só vai poder fazer forwarding do valor lido da memória quando a mesma se encontrar na fase MEM2(MEM2 -> EXE/MEM1), assim sendo a instrução store word só vai poder passar da fase EXE para a fase MEM1 quando a instrução load word estiver a terminar a fase MEM2.

Neste caso a unidade de hazard, durante um ciclo, vai fazer stall no registo EXE/MEM1 e introduzir uma bolha no registo MEM1/MEM2.

### 5.1.4 Descartar instrução Jump/Branch

Esta situação acontece quando temos um Jump ou um Branch que foi taken, sendo preciso descartar a instrução que foi carregada. Para descartar a instrução é colocado um bubble no registo da fase IF/ID.

# Lista de Figuras