Universidade de Aveiro

departamento de electrónica, telecomunicações e informática

47022- Arquitectura de Computadores Avançada

Home group assignment 1

Implementing a forwarding and stall unit in a pipelined architecture

8240 - mestrado integrado em engenharia de computadores e telemática

António Rafael da Costa Ferreira
NMec: 67405

Rodrigo Lopes da Cunha
NMec: 67800

Docentes: Nuno Lau e José Luís Azevedo

Novembro de 2015
2015-2016

Conteúdos

1 Introdução

FALTAAAAAAAAAA

O trabalho proposto para o projeto da unidade curricular de Segurança é um IEDCS: Identity Enabled Distribution Control System. Para o efeito foi necessário implementar uma Ebook Webstore, um WebService e um Player de reprodução dos Ebooks em formato de texto.

O objetivo deste sistema é garantir a máxima e possível segurança do serviço, utilizando os conhecimentos adquiridos na unidade curricular de Segurança. Para isso são necessários vários processos como por exemplo, a utilização de certificados HTTPS, a cifragem de todo o material existente, derivação de chaves e registo de utilizadores.

O relatório reflete todos os passos e decisões tomadas na criação do sistema, assim como uma análise ao que foi mostrado na primeira apresentação e decisões que se tomaram depois desta, tecnologias utilizadas, descrição dos vários processos existentes e conclusão.

2 Exercício 1

2.1 Divisão da fase ID em duas fases ID1/ID2

Neste primeiro exercício, era pedido que se fizesse a divisão da fase ID, por duas fases ID1 e ID2.

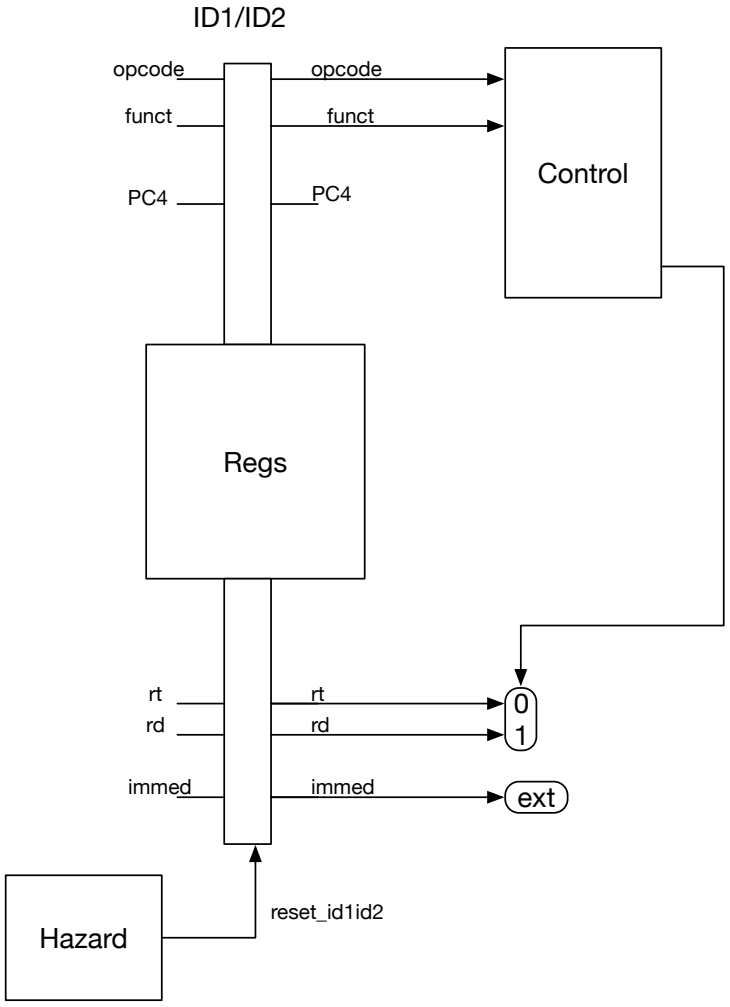


Figura 1:
Registo ID1/ID2

O inicio da leitura de registos tem inicio em ID1 e termina em ID2 (como podemos verificar na figura 1), sendo que na fase ID1 também é feito o decode da instrução para obter os sinais de rs, rt, rd, funct, opcode e immed. Tudo o que se resolvia em ID agora resolve-se em ID2, como por exemplo a unidade

de Controlo, o extend e o Mux entre o rt e o rd.

Para este primeiro exercício criou-se um novo barramento de registos ID1/ID2, que toma como entradas o *opcode*, o *funct*, o *immed*, o *rt*, o *rs* e o *rd* vindos do decoder de ID1, sendo as saídas as correspondentes a estas mesmas portas. Existe também uma entrada vinda da unidade de Hazard, *reset_id1id2*, para fazer reset a este registo aquando da necessidade de existirem stalls ou branchs.

Na unidade de hazard foram feitas algumas alterações para que estes fossem resolvidos em ID2, que anteriormente eram resolvidos em ID. A única diferença é que são os registos em ID2, rs e rt, que definem a existência de hazard tendo em conta as fases mais avançadas da pipeline, como EXE, MEM e WB.

As condições para a existência de hazards são:

```
rs_id2.read()!=0 && rs_id2.read()==WriteReg_exe.read()
&& RegWrite_exe.read()==true
|| rt_id2.read()!=0 && rt_id2.read()==WriteReg_exe.read()
&& RegWrite_exe.read()==true && MemRead.read()== false
```

add $3, $2, $1	IF	ID1	ID2	EXE	MEM	WB		
add $4, $3, $1		IF	ID1	ID2	ID2	ID2	EXE	MEM	WB

Nestas primeiras condições estudamos os casos em que o registo rs ou rt da instrução que está em ID2, é o registo onde vai ser escrito pela instrução que se encontra em EXE, pelo que é necessário a existência de stalls.

```
rs_id2.read()!=0 && rs_id2.read()==WriteReg_mem.read()
&& RegWrite_mem.read()==true
|| rt_id2.read()!=0 && rt_id2.read()==WriteReg_mem.read()
&& RegWrite_mem.read()==true && MemRead.read() == false
```

add 3,2, $1	IF	ID1	ID2	EXE	MEM	WB		
add 0,0, $0		IF	ID1	ID2	EXE	MEM	WB	
add 4,3, $1			IF	ID1	ID2	ID2	EXE	MEM	WB

Para os casos em que o registo rs ou rt da instrução que está em ID2, é o registo onde vai ser escrito pela instrução que se encontra em MEM, pelo que continua a existir a necessidade de stalls.

```
rs_id2.read()!=0 && rs_id2.read()==WriteReg_wb.read()
&& RegWrite_wb.read() == true
|| rt_id2.read()!=0 && rt_id2.read()==WriteReg_wb.read()
&& RegWrite_wb.read() == true && MemRead.read() == false) {
```

Por último existem ainda os casos onde o registo rs ou rt que está em ID2, é o registo que vai ser escrito pela instrução que se encontra em WB, pelo que só no ciclo a seguir o valor estará disponível no banco de registos, obrigando a introdução de um stall.

3 Exercício 2

Neste segundo exercício, foi-nos proposto para que todos os saltos condicionais e incondicionais, fossem resolvidos em ID2. Para isso foi criada uma unidade de branch, que através de entradas como *opcode*, *rsdata*, *rtdata*, *branch*, *target*, *imm_ext* e *PC4*, calcula o *branchTaken*, que nos diz se existe salto ou não, e o *branchTarget* (Figura 2).

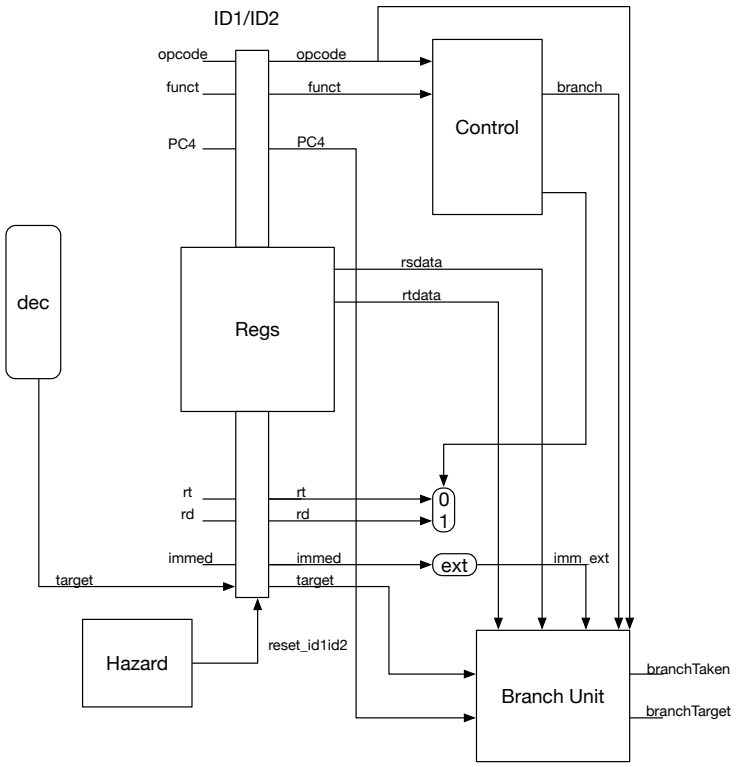


Figura 2:
Registo ID1ID2 com Branch Unit

3.1 Unidade de Branch

Para que os saltos fossem resolvidos em ID2, foi necessário na unidade de branch resolver os mesmos. Para isto dentro da unidade de controlo, alterou-se a informação que a saída de branch possuí. Disponibilizaram-se mais bits para que esta saída indique na unidade de branch o tipo de salto que estamos a resolver.

Posto isto na unidade de branch, efectuam-se 6 casos diferentes:

- BEQ: compara se o valor de *rsdata* é igual ao de *rtdata*

- BNE: compara se o valor de *rsdata* é diferente de *rtdata*
- BGTZ: compara se o valor de *rsdata* é maior que zero
- BLEZ: compara se o valor de *rsdata* é menor ou igual a zero
- JUMP: salta para o target da instrução
- JR: salta para o valor de *rsdata*

Sendo os saltos resolvidos em ID2, foram removidos de fases mais avançadas unidades que não seriam mais utilizadas, como por exemplo o *sl2* e a unidade de adição para branch em EXE e o *and* em MEM. Os portos *PC4* e *branch* também deixaram de ser propagados para as fases seguintes à ID2.

3.2 Unidade de Hazard adaptada para a nova resolução de saltos

Para que instruções que entrem de forma errada na pipeline quando um branch ocorre sejam descartadas excepto a seguinte (delayed branch slot), foi necessário alterar na unidade de Hazard as fases às quais é necessário fazer reset. Desta feita quando a nossa unidade de Hazard trata de descartar instruções que entrem de forma errada na pipeline e de colocar stalls aquando da ocorrência destes.

4 Forwarding

Na tarefa 3 foi-nos pedido para identificar quais são os tipos de forwarding que podem ocorrer na nossa arquitectura onde a origem dos valores são os valores dos registos da pipeline e o seu destino é a "stage".

Os tipos de forwarding são:

- EXE/MEM->EXE
- EXE/MEM->ID2
- MEM/WB->EXE
- MEM/WB->ID2
- MEM/WB->MEM

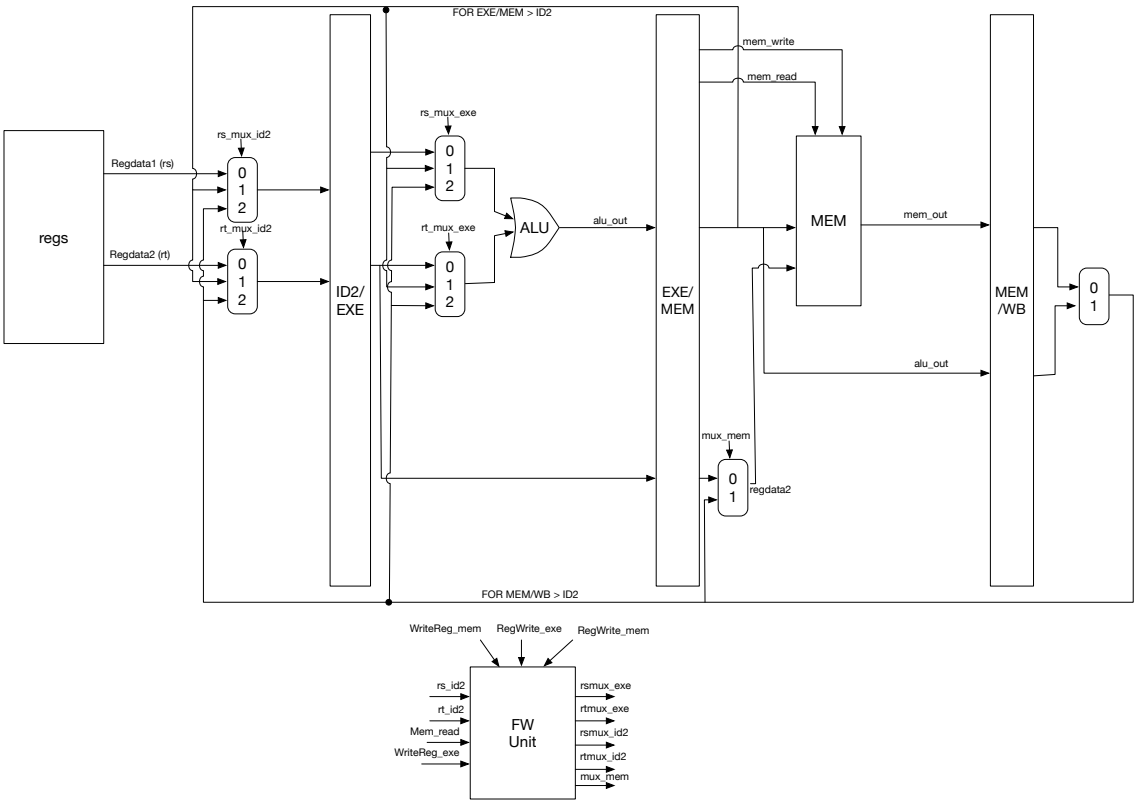


Figura 3:
Forwarding path

4.1 EXE/MEM->EXE

add $1, $2, $3	IF	ID1	ID2	EXE	MEM	WB	
lw $2, 0($1)		IF	ID1	ID2	EXE	MEM	WB

add $1, $2, $3	IF	ID1	ID2	EXE	MEM	WB	
sub $2, $1, $3		IF	ID1	ID2	EXE	MEM	WB

Este tipo de forwarding é causado por uma instrução que escreva num registo e de seguida uma outra instrução precise desse registo para realizar uma operação, ou seja, precisa de ler esse registo. Por exemplo, podemos ter um add $1, $2, $3 e depois um sub $2, $1, $3, a primeira instrução escreve em $1 e a segunda instrução precisa do $1 para realizar a operação de sub com $3.

Tanto para RS e RT em ID2 comparamos com o WriteReg, se o registo RS/RT em ID2 é diferente de 0 e se vai escrever no registo ambos em EXE. Caso esta condição seja verdadeira, vamos então mudar o sinal de seleção do multiplexer para 1, sendo este sinal o que vai selecionar no mux o valor da entrada da ALU como "rs/rt".

Teve-se em atenção que o rt quando MemRead é verdadeiro é porque se vai escrever em rt, e o que nós queremos é ler em rt e não escrever em rt por isso adicionou-se a condição para que MemRead_exe seja igual a false.

```
if(rs_exe.read()!=0 && rs_exe.read()==WriteReg_mem.read()
        && RegWrite_mem.read()==true){
    rs_mux_exe.write(1);
}
if(rt_exe.read()!=0 && rt_exe.read()==WriteReg_mem.read()
        && RegWrite_mem.read()==true
        && MemRead_exe.read()==false){
    rt_mux_exe.write(1);
}
```

4.2 EXE/MEM->ID2

add $1, $2, $3	IF	ID1	ID2	EXE	MEM	WB		
nop		IF	ID1	ID2	EXE	MEM	WB	
beq $1, $0, label			IF	ID1	ID2	EXE	MEM	WB

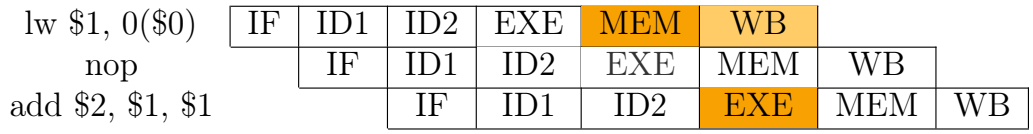
Este tipo de forwarding é causado por uma instrução que escreva num registo para o qual um branch condicional necessite de ler esse registo para tomar uma decisão. Por exemplo, neste caso, tem-se uma instrução add $1, $2, $3 que vai escrever em $1 e o beq vai necessitar de $1 para calcular a decisão, neste caso é o $rs mas também podia ser o $rt o registo necessário para calcular a decisão que estivesse a ser escrito por uma outra instrução. É feito então o forwarding da fase EXE/MEM para ID2.

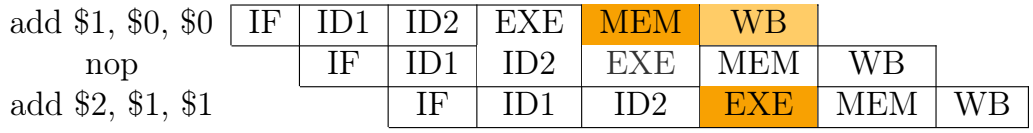
Nas condições que usamos para detetar estes casos de forwarding comparamos o rs/rt em id2 com o WriteReg em EXE, se vai escrever no registo em EXE e se é uma branch, ou seja, branch tem de ser diferente de 0, isto deve-se porque no exercício anterior colocou-se o sinal de branch a ter valores de 0 a 6. O multiplexer vai selecionar o forward de EXE/MEM para esta fase, fazendo com que os valores sejam selecionados os que está a ser feito o forward e não os que saem dos registos.

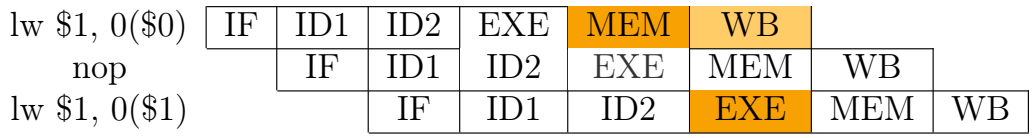
```
if(rs_id2.read()!=0 && rs_id2.read()==WriteReg_mem.read()
```

```
        && RegWrite_mem.read()==true && branch.read()!=0){
    rs_mux_id2.write(1);
}
if(rt_id2.read()!=0 && rt_id2.read()==WriteReg_mem.read()
        && RegWrite_mem.read()==true
        && branch.read()!=0
        && MemRead.read()==false){
    rt_mux_id2.write(1);
}
```

4.3 MEM/WB->EXE







Este tipo de forwarding é provocado por uma instrução do que escreva para um registo que uma terceira instrução na pipeline necessite de ler para efetuar a sua operação em EXE.

Por exemplo, no nosso caso temos uma operação do tipo LW que lê na memória para o registo $1, e depois, duas em EXE existe uma instrução que vai ler do registo que foi escrito pela instrução LW. Assim como podemos ter uma instrução do tipo R na fase MEM e uma instrução do tipo R que também necessite desse registo escrito pela primeira instrução na fase EXE.

```
if(rs_exe.read()!=0 && rs_exe.read()==WriteReg_wb.read()
  && RegWrite_wb.read()==true){
    rs_mux_exe.write(2);
}
if(rt_exe.read()!=0 && rt_exe.read()==WriteReg_wb.read()
  && RegWrite_wb.read()==true
```

```
   && MemRead_exe.read()==false){
      rt_mux_exe.write(2);
}
```

4.4 MEM/WB->ID2

lw $1, 0($0)	IF	ID1	ID2	EXE	MEM	WB		
nop		IF	ID1	ID2	EXE	MEM	WB	
nop			IF	ID1	ID2	EXE	MEM	WB
beq $1, $1, label				IF	ID1	ID2	EXE	MEM	WB

O caso de MEM/WB para ID2 apenas acontece quando existe uma instrução de leitura na memória e em que esse valor lido da memória apenas está disponível em WB e em ID2 está uma operação de beq ou outro tipo que leia um registo que está a para ser escrito pela instrução de lw. Será feito um forward desse valor de MEM/WB para ID2.

```
if(rs_id2.read()!=0 && rs_id2.read()==WriteReg_wb.read()
   && RegWrite_wb.read()==true){
      rs_mux_id2.write(2);
}
if(rt_id2.read()!=0 && rt_id2.read()==WriteReg_wb.read()
   && RegWrite_wb.read()==true
   && MemRead.read()==false){
      rt_mux_id2.write(2);
}
```

4.5 MEM/WB->MEM

lw $1, 0($0)	IF	ID1	ID2	EXE	MEM	WB	
sw $1, 0($0)		IF	ID1	ID2	EXE	MEM	WB

As condições para este caso acontecer são muito específicas, apenas dizem respeito a um lw seguido sw, não diz respeito a uma instrução do tipo R seguida de um sw porque nesse caso é feito o forward de EXE/MEM para EXE para resolver esse caso. Isto só acontece porque o lw só está disponível em MEM, ou seja, para a instrução de sw precisa na fase de MEM desse valor terá de ser feito o forward de MEM/WB para MEM. As condições são: em MemRead_Wb==true, MemWrite_Mem==true e WriteReg_wb==rt_mem!=0.

```
if(rt_mem.read()!=0 && rt_mem.read()==WriteReg_wb.read()
  && RegWrite_wb.read()==true
  && MemRead_wb.read()==true
  && MemWrite_mem.read()==true){
    mux_mem.write(1);
}
```

4.6 Forwarding para EXE

add $1, $2, $3	IF	ID1	ID2	EXE	MEM	WB		
add $2, $1, $1		IF	ID1	ID2	EXE	MEM	WB	
add $2, $2, $1			IF	ID1	ID2	EXE	MEM	WB

Neste caso o forwarding é feito desde MEM/WB para EXE e de EXE/MEM para EXE. A instrução que está em EXE necessita dos valores que estão disponíveis na fase de MEM e WB que foram calculados em fases anteriores, sendo assim, é feito o forward para EXE desde MEM/WB e EXE/MEM. Também poderia ocorrer o caso de estar a ser calculado um endereço em EXE de uma instrução do tipo lw ou sw e então ser feito forward para esta fase de um valor que esteja a ser calculado numa das outras fases.

4.7 Forwarding para ID2

lw $1, 0($0)	IF	ID1	ID2	EXE	MEM	WB		
add $2,$3, $3		IF	ID1	ID2	EXE	MEM	WB	
nop			IF	ID1	ID2	EXE	MEM	WB
beq $1, $2, label				IF	ID1	ID2	EXE	MEM	WB

lw $1, 0($0)	IF	ID1	ID2	EXE	MEM	WB		
add $2, $3, $3		IF	ID1	ID2	EXE	MEM	WB	
nop			IF	ID1	ID2	EXE	MEM	WB
add $2, $1, $2				IF	ID1	ID2	EXE	MEM	WB

lw $1, 0($0)	IF	ID1	ID2	EXE	MEM	WB		
nop		IF	ID1	ID2	EXE	MEM	WB	
nop			IF	ID1	ID2	EXE	MEM	WB
lw $2, 0($1)				IF	ID1	ID2	EXE	MEM	WB

Neste caso assumimos que só está disponível o valor de $1 quando o lw está na fase de MEM, e este é necessário por uma instrução que vai ler este

registo. O valor só vai ser escrito no final de WB, e se o valor for necessário na fase ID2 então teremos de fazer forward. No caso do add, o resultado de um add, este só vai ser gravado quando terminar a fase de WB, no entanto pode ser feito o forward de EXE/MEM para ID2, pois este valor é necessário na fase ID2 (devido ao salto condicional). No caso do lw é necessário fazer o forward de MEM/WB para ID2, porque o valor também ainda não está disponível no banco de registo.

4.8 Forwarding desde MEM/WB

lw $1, 0($0)	IF	ID1	ID2	EXE	MEM	WB			
sw $1, 8($0)		IF	ID1	ID2	EXE	MEM	WB		
add $2, $1, $1			IF	ID1	ID2	EXE	MEM	WB	
lw $2, 0($1)				IF	ID1	ID2	EXE	MEM	WB

Este caso é quando existe um valor que tem de ser lido da memória, ou alguma operação do tipo R que foi calculado na fase de EXE e é necessário para escrever na memória esse valor, entre a primeira e a segunda instrução existe uma dependência que tem de ser satisfeita através de um forward de MEM/WB para MEM. Outro caso é esse valor ser necessário para a operação que vai decorrer em EXE, fazendo assim um forward de MEM/WB para EXE. Outro caso é existir a necessidade de ler um registo que ainda não foi escrito e que a instrução que escreve nesse ainda não terminou, fazendo assim um forward de MEM/WB para ID2.

4.9 Testes

Foram pensados testes específicos para testar todos os forwards em conjunto, também é preciso salientar que todas os casos em cima foram testados individualmente.

Teste 1

Este teste pretende testar o forward de EXE/MEM para EXE e de MEM/WB para EXE.

add $3, $1, $2	IF	ID1	ID2	EXE	MEM	WB		
add $4, $3, $1		IF	ID1	ID2	EXE	MEM	WB	
add $5, $2, $3			IF	ID1	ID2	EXE	MEM	WB

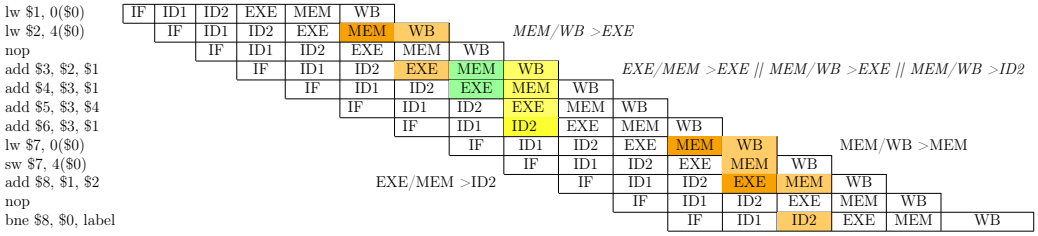
O segundo add vai necessitar de ler o $3 que está a ser calculado pelo primeiro add e está disponível nas fases MEM e WB. Sendo assim será feito forward de EXE/MEM para EXE para o segundo add e de MEM/WB para EXE por causa do terceiro add.

Teste 2

lw $1, 0($0)	IF	ID1	ID2	EXE	MEM	WB				
lw $2, 4($0)		IF	ID1	ID2	EXE	MEM	WB			
nop			IF	ID1	ID2	EXE	MEM	WB		
nop				IF	ID1	ID2	EXE	MEM	WB	
beq $2, $1, label					IF	ID1	ID2	EXE	MEM	WB

Este teste serve para testar um caso em que é necessário um valor no beq que está a ser calculado no segundo lw. Neste caso o forward é do tipo MEM/WB para ID2.

Teste 3



Neste teste pretendeu-se testar todos os tipos de forward, excluindo alguns que foram testados anteriormente. Os forwards presentes são: MEM/WB para EXE, EXE/MEM para EXE, MEM/WB para EXE, MEM/WB para ID2, MEM/WB para MEM e EXE/MEM para ID2.

5 Forward e stalls

No exercício 5, o último mas não menos importante, foi pedido que fosse identificado todas as condições que provocariam stall e posteriormente se mudasse a unidade de hazard para inserir stalls sempre que fosse identificado uma situação de stall.

5.1 lw e lw

lw $1, 0($0)	IF	ID1	ID2	EXE	MEM	WB		
lw $2, 4($1)		IF	ID1	ID2	ID2	EXE	MEM	WB

Esta condição acontece porque o primeiro lw só tem o valor disponível em MEM/WB, após ter lido da memória, e como o segundo lw necessita desse valor em EXE para calcular o valor do endereço, então terá de ser feito um forward de MEM/WB para EXE. Para isso acontecer terá de ser feito um stall em ID2.

5.2 add e beq

add $1, $0, $0	IF	ID1	ID2	EXE	MEM	WB		
beq $1, $1, label		IF	ID1	ID2	ID2	EXE	MEM	WB

Este stall acontece quando um add tem um beq que necessite de ler o resultado do add. O resultado do add apenas está disponível em MEM (EXE/MEM) e WB(MEM/WB) para se ser feito o forward, mas, mesmo com o forward é preciso ser feito um stall em ID2 para o valor de EXE estar disponível e depois ser feito um forward de EXE/MEM para ID2 para que depois o beq conseguir calcular a condição de salto.

5.3 lw e add

lw $1, 0($0)	IF	ID1	ID2	EXE	MEM	WB		
add $2, $1, $1		IF	ID1	ID2	ID2	EXE	MEM	WB

Neste caso tal como no primeiro caso, o valor lido da memória apenas está disponível na fase de WB, ou seja, é possível fazer forward de MEM/WB para outra fase. No nosso caso é feito forward diretamente para EXE causando um ciclo de stall.

5.4 lw e beq

lw $1, 0($0)	IF	ID1	ID2	EXE	MEM	WB			
beq $1, $1, label		IF	ID1	ID2	ID2	ID2	EXE	MEM	WB

Este é o único caso que provoca dois stalls em ID2. As condições de salto são calculadas em ID2 o que faz com que os valores que são usados para calcular a condição de salto têm de estar disponíveis em ID2 ou então é necessário que seja feito forward de barramentos da pipeline mais avançados para esta fase caso completem as necessidades.

Neste caso o lw apenas tem disponível em MEM/WB o valor lido da memória para fazer forward para ID2 e assim a condição de salto pode ser calculada. Para que o lw leia da memória e depois faça o forward é necessário que o beq adicione dois stalls.

6 Conclusão

Este trabalho foi útil para assentarmos todos os conhecimentos que fomos obtendo ao longo destes anos, tanto em Arquitectura de Computadores Avançada como em Arquitetura de Computadores I e II. Ambas as entregas foram primeiro planeadas em papel antes de se avançar para a implementação o que resultou em bons resultados a nível de tempo despendido, para ambos as entregas fez-se uso de dois dias de trabalho.

Na primeira entrega houve uma falha na implementação, esqueceu-se de realizar o shift left 2 do program counter para calcular o endereço para onde é suposto saltar a instrução jump.

Já na segunda entrega teve-se os máximos cuidados para que tudo funcionasse perfeitamente.