universidade de aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Trabalho 1

Implementing a forwarding an stall unit in a pipelined architecture

Curso	[8240] MI em Engenharia de Computadores e Telemática
Disciplina	[47022] Arquitetura de Computadores Avançada
Ano letivo	2015/2016

Alunos	[68779] Rui Oliveira
	[68129] Tomás Rodrigues
Prática	P1
Docente	Professor José Luís Azevedo

Aveiro, 22 de Novembro de 2015

Conteúdo

1 Introdução

Pretende-se através deste relatório expor sob forma escrita, o nosso desempenho e objetivos alcançados no trabalho 1 desenvolvido no âmbito da unidade curricular de Arquitetura de Computadores Avançada.

Neste relatório pretenderemos explicar as técnicas por nós utilizadas para implementar as melhorias ao datapath inicialmente fornecido, tendo como base os conhecimentos adquiridos nas aulas práticas.

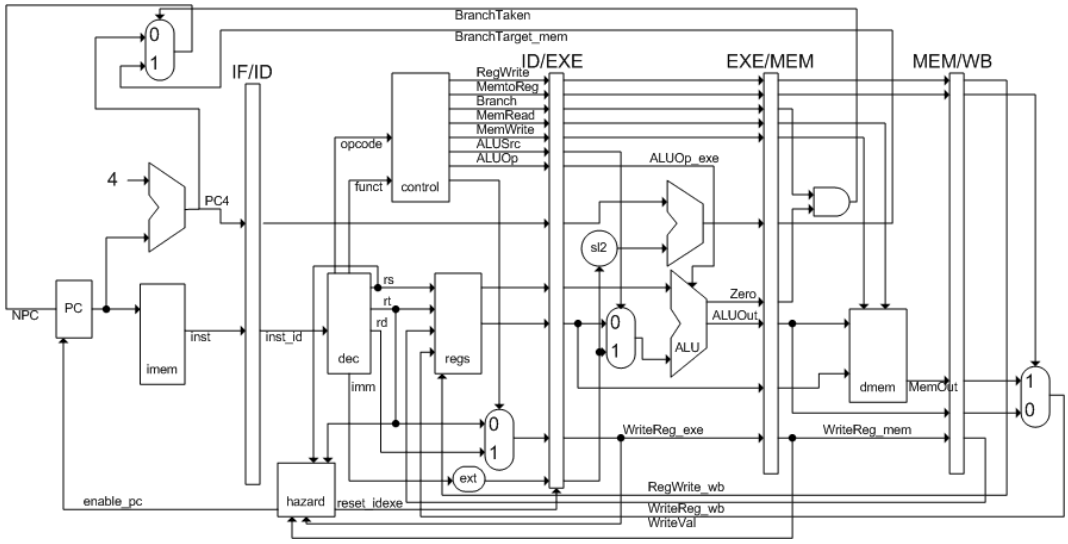

Figura 1: Datapath original

De seguida iremos enumerar as 5 fases em que se divide o nosso trabalho:

1. Divisão da fase *instruction decoder* (ID) em duas fases (ID1+ID2)
2. Permitir que os *branchs/jumps* sejam resolvidos em ID2 invés de MEM
3. Enumerar todos as situações de *forwarding*
4. Implementar uma unidade de *forwarding*
5. Implementar *forwarding* e *stalls*

2 Tarefa 1: divisão da fase ID

Com a possibilidade de aumentar a frequência no datapath, precisaremos de dividir a fase ID em duas: ID1 e ID2.

Para isso, criámos um novo registo (reg_id1_id2) de pipeline para manter os valores de controlo devido à transição de estado.

Na figura 2 encontra-se esquematizado o nosso datapath com as devidas alterações. O novo registo encontra-se representado de cor cinzenta.

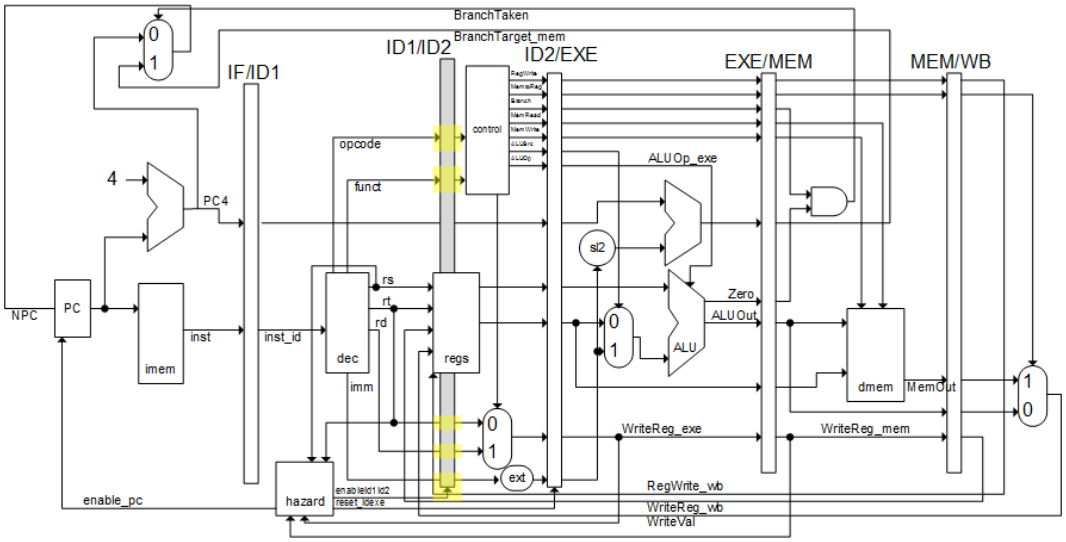

Figura 2: Primeira versão do Datapath: divisão da fase ID

Todos os sinais que são 'passados' pelo novo registo encontram-se assinalado a amarelo na figura acima. São eles:

- opcode
- funct
- rt
- rd
- imm
- PC4

O registo recebe também um novo sinal de enable da unidade de hazard.

3 Tarefa 2: instruções de Branch e Jump na fase ID2

3.1 Resolver instruções beq na fase ID2 usando *delayed branch*

No datapath fornecido as instruções de *branch* são resolvidos na fase MEM. Por outro lado, o *Branch Target Address* (BTA) é calculado na fase EXE, enquanto que o salto de decisão é feito na Unidade Lógica Aritmética (ALU), subtraindo-se ambos os registos.

Para que este tipo de instruções seja resolvidos na fase ID, alterámos o datapath do seguinte modo:

1. Deslocou-se o *Shift Left* (SL2) e o somador (ADDER) da fase EXE para ID2. Consequentemente, o sinal PC4 é movido para trás e removido do registo ID2/EXE e posteriormente ligado ao somador. Os sinais resultantes deste processo encontram-se representados a azul na figura da página seguinte.

2. O sinal immediate (IMM) e consequentemente o sinal resultante da extensão de sinal (EXT) e *Shift Left* (SL2) foram movidos para ID2 e posteriormente ligados ao ADDER. Agora, é possível calcular BTA em ID2. Os sinais resultantes deste processo encontram-se representados a azul na figura da página seguinte.

3. O sinal resultante da soma chama-se *BranchTarget* e é enviado para o *multiplexer* da fase IF. Ver sinal vermelho à esquerda.

4. O sinal de seleção do multiplexer - *Branch Taken* - é calculado através de um novo módulo - *Branch Unit Control*. Este módulo permite decidir o que sairá do MUX, se o valor PC+4 ou BTA.

5. A Branch Unit Control recebe como inputs o sinal de *branch* calculado na unidade de controlo (sinal a amarelo) e os sinais provenientes do regFile (sinal a verde).

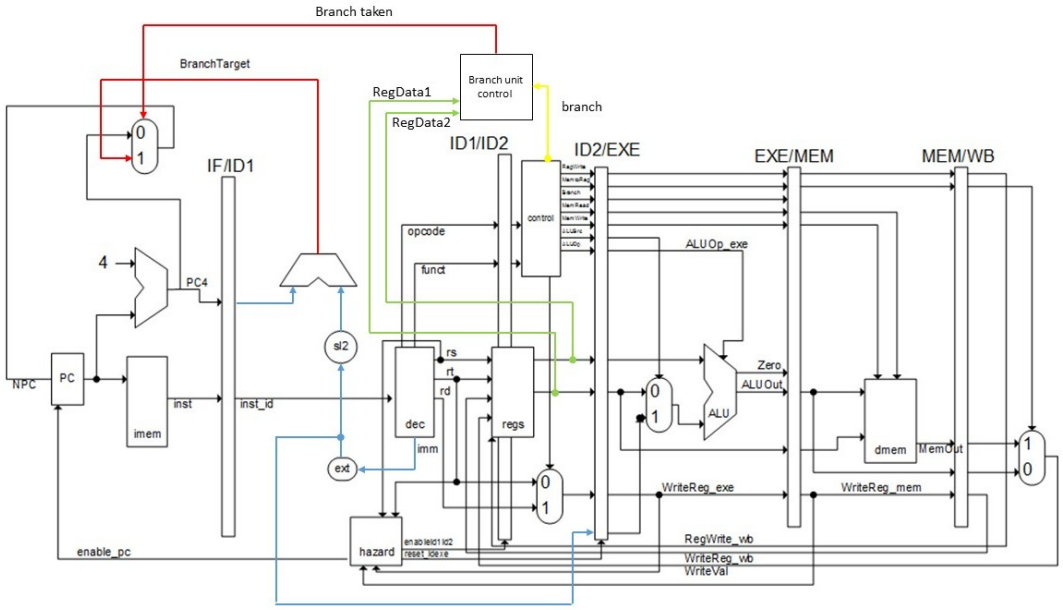

Figura 3: Segunda versão do Datapath: instruções beq

3.2 Resolver instruções Jump na fase ID2 e variações de instruções de salto condicional

As instruções de salto incondicionais não podiam ser executadas na versão anterior do datapath. De modo a implementar e resolver estas instruções na fase ID2, decidimos alterar a *Branch Unit Control* para poder lidar com as seguintes instruções:

- beq - Branch on equal
- bne - Branch on not equal
- j - Jump
- jr - Jump register
- bgtz - Branch on less than zero
- blez - Branch on less than or equal to zero

O módulo atualmente é chamado de *Jump Unit Control* e permite executar saltos condicionais e incondicionais. A *Jump Unit Control* permitenos remover a AND_gate da fase MEM e os sinais aí ligados. O sinal Branch foi mudado de 1 para 2 bits para distinguir os diferentes *branchs* (beq,bne,bgtz,blez).

Existe também uma nova unidade para calcular os destinos de salto: *Jump Target Calc*. Este módulo concatena os 4 bits mais significativos do endereço da instrução com os 28 bits menos significativos do *label* que provém do *decode* shiftados duas vezes.

A figura seguinte representa o esquema da interligação de todos os componentes necessários para para um bom funcionamento do descrito acima.

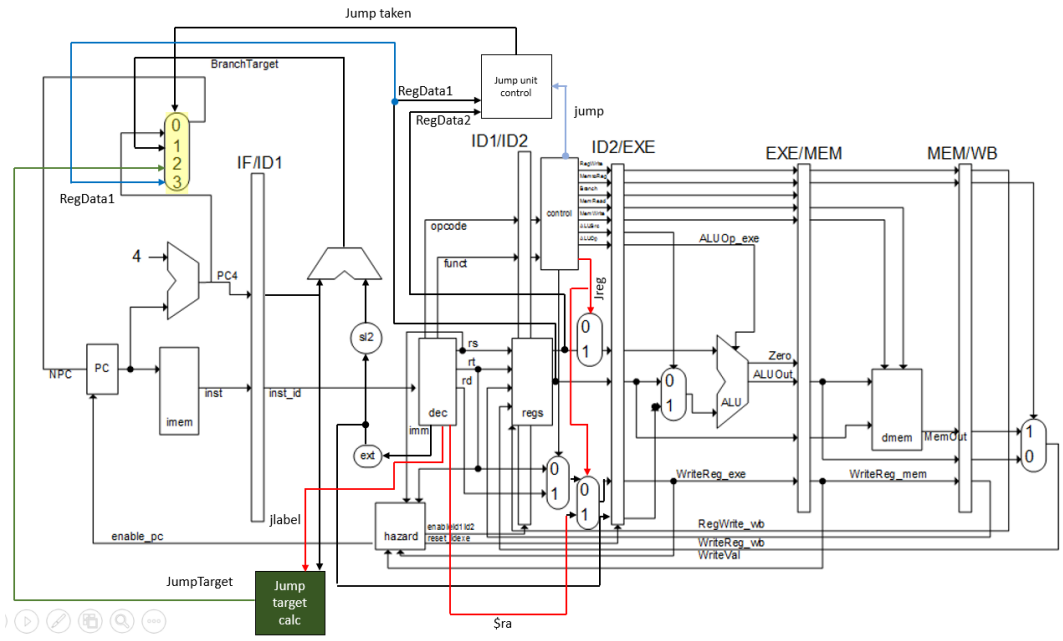

Figura 4: Terceira versão do Datapath: instruções jump

O excerto de código seguinte demonstra como é feita a verificação de cada tipo de instrução.

```
switch (sel.read()) {
    case 1:
        dout.write((asng == bsng) ? 1 : 0 ); break;     // beq
    case 2:
        dout.write((asng != bsng) ? 1 : 0) ; break;     // bne
    case 3:
```

```
    dout.write(2); break;    // jump
case 4:
    dout.write(3); break;    // jr
case 5:
    dout.write((asng > 0) ? 1 : 0); break;    // bgtz
case 6:
    dout.write((asng <= 0) ? 1 : 0); break;   // blez
default:
    dout.write(0); break;    // other
}
```

3.3 Descartar a instrução que entra erradamente no *pipeline* quando existe um salto que é *taken*

No caso do branch ser taken a instrução seguinte é sempre executada, sendo depois a outra descartada, assumindo portanto uma política de *delayed branch*. Assim quando o salto é *taken* e sabendo nós que um Reset e Enable a 1 num registo é uma bolha colocamos uma bolha em ID1 de forma a descartar a instrução na *pipeline* que já não tem de ser executada:

```
else if (JumpTaken.read()!=0)
{
    //printf("BranchTaken, discard if\n");

    enable_pc.write(true); //
    enable_ifid1.write(true); //
    enable_id1id2.write(true); //
    enable_regfile.write(true); //

    reset_ifid1.write(true); //
    reset_id2exe.write(false); //
}
```

Figura 5: Condição para Bubble em ID1

4 Tarefa 3

4.1 Identificar todos os tipos de forwarding existentes

Uma vez que os forwardings são resolvidos do registo para a fase, estes podem surgir dos seguintes registos:

- EXE/MEM
- MEM/WB
- EXE/MEM

E todos os tipos de forwarding são:

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

4.2 Exemplos de código para cada tipo de forwarding

4.2.1 Do registos para ID2

Os *forwards* para ID vão ocorrer quando houver uma instrução resolvida em ID (*branch* ou *jump register*) que precise do valor de um registo que está a ser calculado mais à frente na *pipeline*.

- **EXE/MEM - ID2**

```
add $1, $0, $0
nop
jr $1
```

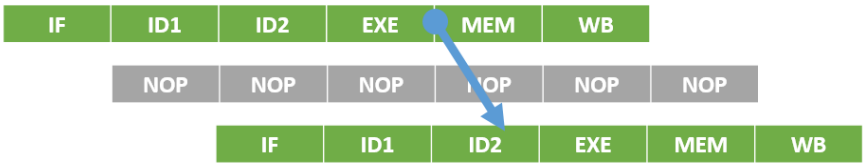

Figura 6: Caso tipo: exe/mem - id2

- **MEM/WB - ID2**

```
lw $1, 0($0)
nop
nop
jr $1
```

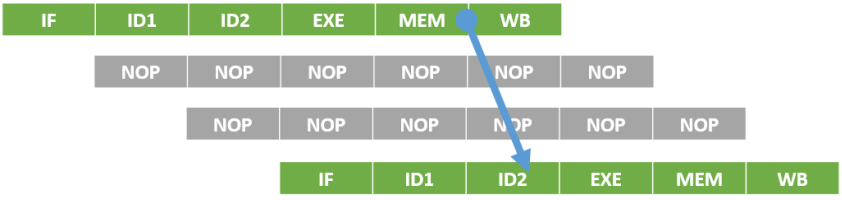

Figura 7: Caso tipo: mem/wb - id2

- **Exemplo de código para ID2**

```
                    # MEM/WB -> ID2
0x8c020008          # lw  r2,8(r0)  #r2=16
0x8c010004          # lab1:  lw  r1,4(r0)  #r1=56
0
0
0x1024fff9          # beq r1,r2,lab1
0                   # nop
0
0
                    #EXE/MEM -> ID2
0x8c020008          # lw  r2,8(r0)
0x8c010004          # lab1:  lw  r1,4(r0)
0
0x00411820          # add r3,r2,r1 #r3=r2+r1 # fw mem/wb para exe (r1)
0
0
0
0x00412020          # add r4,r2,r1 #r3=r2+r1
0
0x1064fff9          # beq r3,r4,lab1 # r3 = r2?
                    # fw exe/mem -> id2 (r4) fw mem/wb->id2 (r3)
0
0
                    # MEM/WB -> ID2
0x8c010004          # lw  r1,4(r0)  #r1=56
0x8c020008          # lab1: lw  r2,8(r0)  #r2=16
0
0
0
0x00612020          # add r4,r3,r1
0
0
0x1024fff9          # beq r1,r2,lab1
0                   # nop
0
0
```

4.2.2 Do registos para EXE

Os *forwards* para EXE vão ocorrer quando houver uma instrução resolvida em EXE (instruções aritméticas ou cálculos de *offset*) que precise do valor de um registo que está a ser calculado mais à frente na *pipeline*.

- **EXE/MEM - EXE**

```
add $1, $0, $0
add $2, $1, $0
```

Figura 8: Caso tipo: exe/mem - exe

- **MEM/WB - EXE**

```
lw $1, 0($0)
nop
add $2, $1, $0
```

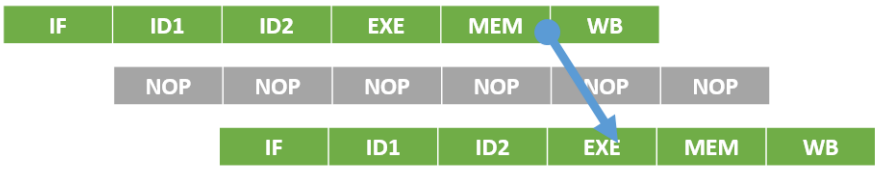

Figura 9: Caso tipo: mem/wb - exe

- **Exemplo de código para EXE**

```
                    # testa blez e bgtz
0x8c020008          # lw  r2,8(r0)  #r2=16
0x8c010004          # lab1:  lw  r1,4(r0)  #r1=56
0
0
```

```
0
0x00411820          # add r3,r2,r1
0
                    #0000.0000.0110.0001.0010.0000.0010.0000
0x00612020          # add r4,r3,r1
0xac040000          # sw r4,0(r0)
0                   # nop
0
0
```

4.2.3 Do registos para MEM

Os *forwards* para MEM vão ocorrer quando houver uma instrução que só precise do valor de um registo na fase MEM (*stores*), valor esse que está a ser calculado mais à frente na *pipeline*.

- **MEM/WB - MEM**

```
lw $1, 0($0)
sw $1, 0($0)
```

Figura 10: Caso tipo: mem/wb - mem

- **Exemplo de código para MEM**

```
0x8c020008          # lw  r2,8(r0)  #r2=16
0x8c010004          # lw  r1,4(r0)  #r1=56
                    #1010.1100.0000.0001.0000.0000.0000.0000
                    #1010 11ss ssst tttt iiii iiii iiii iiii
0xac010000          # sw r1,0(r0) # sw rt,offset(rs)
0
0
```

5 Tarefa 4

5.1 Especificar *multiplexers* necessários para implementar *forwarding*

Para implementar o *forwarding* no nosso *datapath* vamos precisar 5 novos *multiplexers* e de um novo módulo chamado de *unit forwarding*.

- dois *MUX's* em ID2 (representado pelo número 1 e 2 na figura 11)
- dois *MUX's* em EXE (representado pelo número 3 e 4 na figura 11)
- um MUX em MEM (representado pelo número 5 na figura 11)

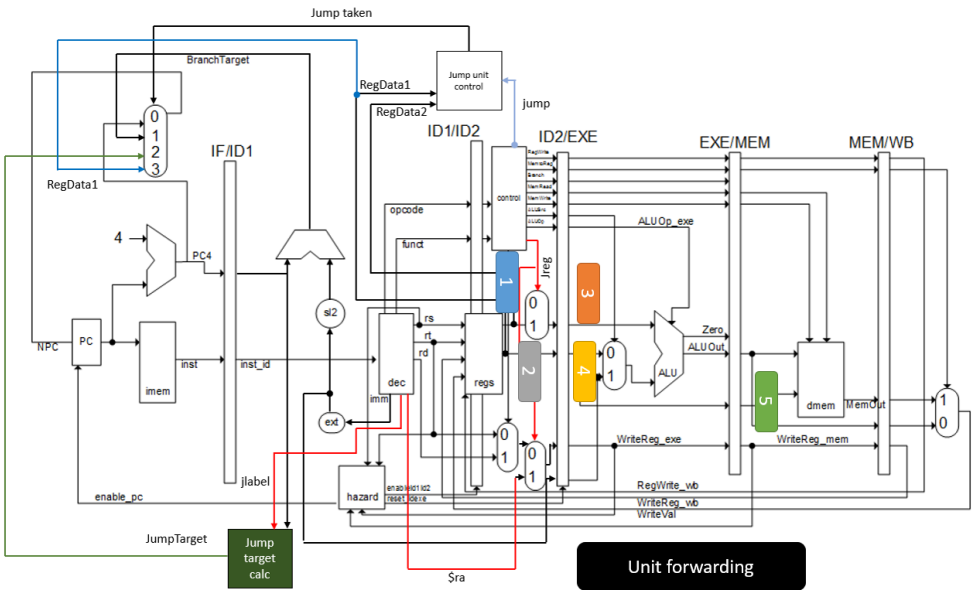

Figura 11: Datapath com os novos componentes necessários para implementar uma unidade de *forwarding*

Os *multiplexers* em ID2 e EXE podem ser vistos na figura seguinte, tal como os seus sinais de input/output/select.

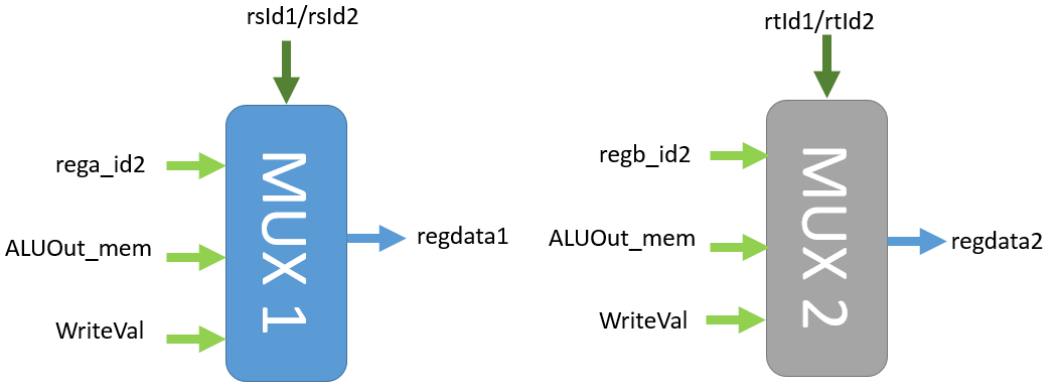

Figura 12: Input/Output/Select dos novos *multiplexers* em ID2

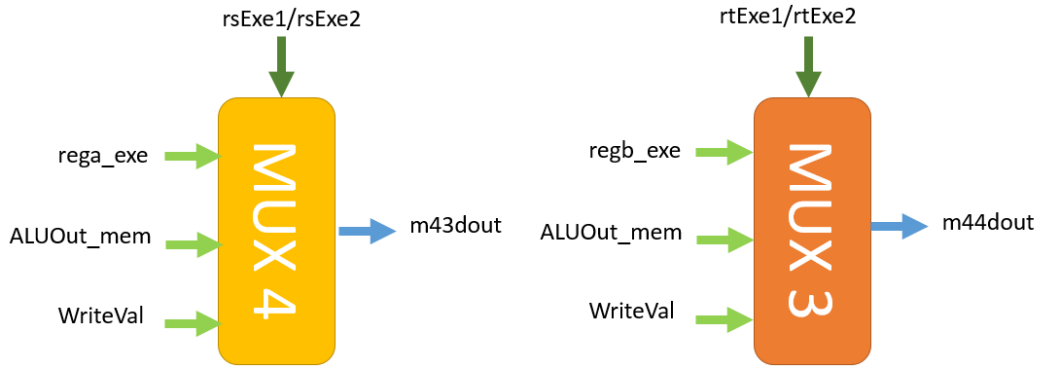

Figura 13: Input/Output/Select dos novos *multiplexers* em EXE

O *multiplexer* em MEM é utilizado no caso de haver *forwarding* para MEM. Os sinais necessários para este componentes encontram-se na figura 14.

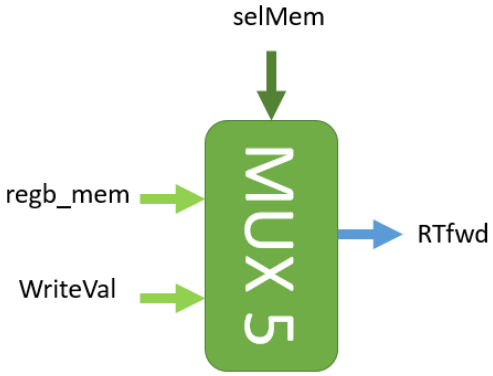

Figura 14: Input/Output/Select do novo multiplexer em MEM

5.2 Implementação de uma unidade de *forwarding*

Todas as situações de *forwarding* foram implementadas neste módulo. As saídas deste módulo são as entradas dos *multiplexes* apresentados acima.

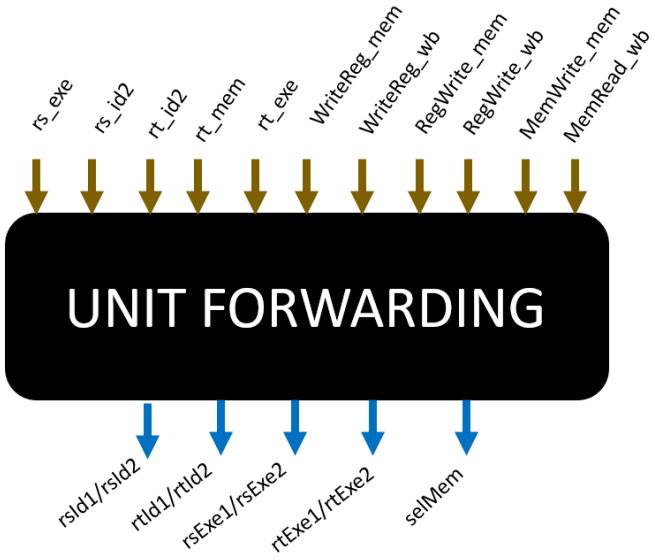

Figura 15: Input/Output da unidade de *Forwarding*

6 Tarefa 5

Na *hazard unit* depois do *forwarding* estar implementado, muitas situações que anteriormente tinham de 'esperar' agora já podem seguir. As situações que identificamos ainda precisar de *stall* são as seguintes:

EXE/MEM - ID2

```
add $1, $0, $0
jr $1
```

Figura 16: Situação de *bubble* em EXE/MEM - ID2

MEM/WB - ID2

```
lw $1, 0($0)
jr $1
```

Figura 17: Situação de *bubble* em MEM/WB - ID2

MEM/WB - EXE

```
lw $1, 0($0)
add $2, $1, $0
```

Figura 18: Situação de *bubble* em MEM/WB - EXE

Sendo assim as condições de *stall* necessárias diminuíram relativamente à tarefa 2. O excerto de código abaixo representa as nossas novas condições:

```
if( (rs.read() != 0 && rt.read() != 0) && (
    // add $1
    // jr  $1
    Jump.read()!=0 && rs.read() == WriteReg_exe.read()
    && RegWrite_exe.read() ||
    Jump.read()!=0 && rt.read() == WriteReg_exe.read()
    && RegWrite_exe.read()  && MemWrite.read()== false  ||

    // lw $1
    // jr $1
    Jump.read()!=0 && rs.read() == WriteReg_mem.read() &&
    MemRead_mem.read() == true || Jump.read()!=0 &&
    rt.read() == WriteReg_mem.read() &&
    MemRead_mem.read() == true && MemWrite.read()== false ||

    // lw  $1
    // add --, $1
    rs.read() == WriteReg_exe.read() &&
    MemRead_exe.read() == true ||
    rt.read() == WriteReg_exe.read() &&
    MemRead_exe.read() == true
    ))
```

7 Conclusões

Chegado ao final deste relatório, é nossa intenção efetuar uma retrospetiva da evolução do mesmo, tendo em conta os problemas com que nos deparámos, e principais conclusões retiradas.

Começando pelas tarefas 1 e 2 entregues anteriormente e após uma análise dos erros existentes, concluímos que os sinais de saída do módulo *jumpcontrol* estavam *unsigned*, tal não poderia acontecer pois na comparação dos valores um numero negativo seria interpretado como um positivo muito grande. Assim, fazendo um *cast* do sinal para *signed* efetuámos a correção do erro de modo a prosseguir as seguintes tarefas.

Relativamente às tarefas seguintes deparámos-nos com alguns problemas na implementação da unidade de forwarding mas conseguimos superá-los todos com sucesso e após testes intensivos pensamos que tudo o foi implementado encontra-se funcional.

De modo a finalizar o presente trabalho, tivemos que ter/obter conhecimentos avançados sobre o fluxo do *pipeline* no *datapath* e sobre o funcionamento de cada instrução numa determinada fase. Este conhecimento foi adquirido não apenas nas aulas Arquitetura de Computadores Avançada, mas também nas disciplinas antecedentes - Arquitetura de Computadores 1 e 2.