



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	2
2	Exercício 1	3
	2.1 Divisão da fase ID em duas fases ID1/ID2	3
3	Exercício 2	5
	3.1 Unidade de Branch	5
	3.2 Unidade de Hazard adaptada para a nova resolução de saltos	6
4	Forwarding	6
	4.1 EXE/MEM->EXE	7
	4.2 EXE/MEM->ID2	8
	4.3 MEM/WB->EXE	9
	4.4 MEM/WB->ID2	10
	4.5 MEM/WB->MEM	10
5	Conclusão	11

1 Introdução

FALTAAAAAAAAAAAA

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.

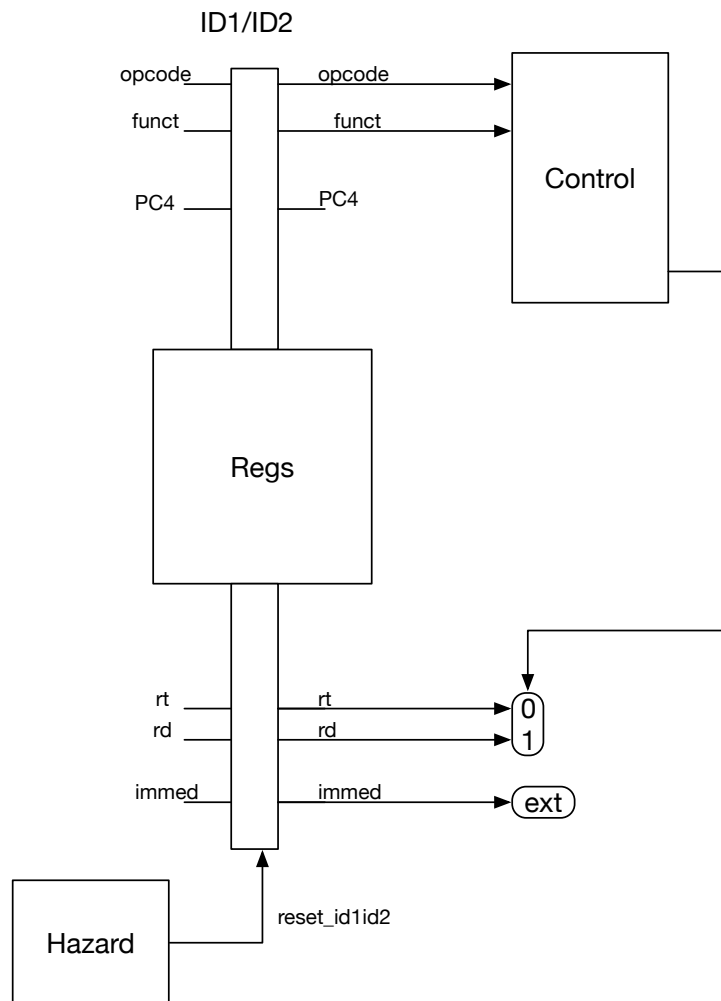


Figura 1:
Registo ID1/ID2

O início da leitura de registos tem início 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 registo novo 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 branches.

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.

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).

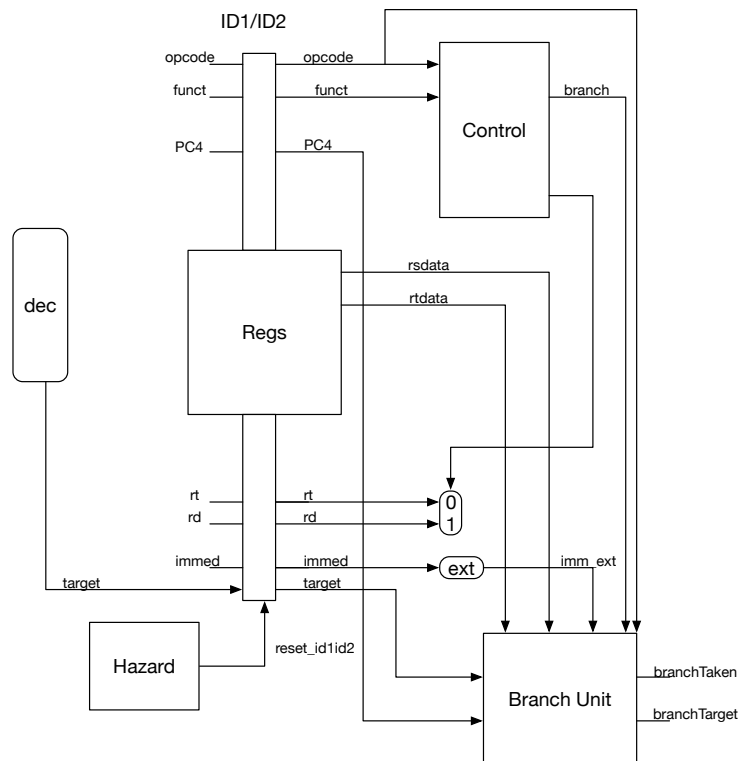


Figura 2:
Registo ID1ID2 com Branch Unit

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 possui. 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

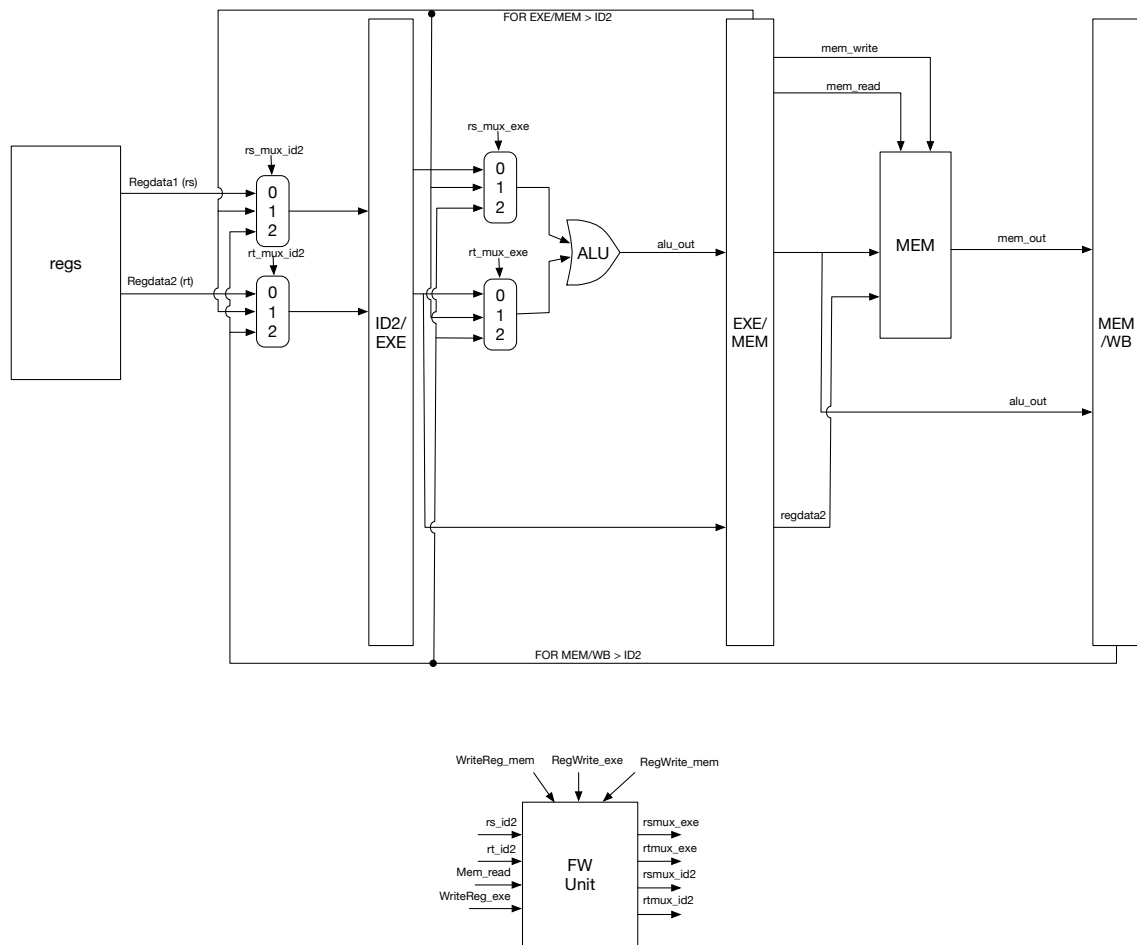


Figura 3:
Forwarding path

4.1 EXE/MEM->EXE

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 seja igual a false.

```
if(rs_id2.read()!=0 && rs_id2.read()==WriteReg_exe.read()
    && RegWrite_exe.read()==true){
    rsmux_exe.write(1);
}
if(rt_id2.read()!=0 && rt_id2.read()==WriteReg_exe.read()
    && RegWrite_exe.read()==true
    && MemRead.read()==false){
    rtmux_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_exe.read()
    && RegWrite_exe.read() == true && branch.read() != 0) {
    rsmux_id2.write(1);
}
if (rt_id2.read() != 0 && rt_id2.read() == WriteReg_exe.read()
    && RegWrite_exe.read() == true
    && branch.read() != 0
    && MemRead.read() == false) {
    rtmux_id2.write(1);
}

```

4.3 MEM/WB->EXE

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

add \$1, \$0, \$0	IF	ID1	ID2	EXE	MEM	WB		
nop		IF	ID1	ID2	EXE	MEM	WB	
add \$2, \$1, \$1			IF	ID1	ID2	EXE	MEM	WB

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

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_id2.read() != 0 && rs_id2.read() == WriteReg_mem.read()
    && RegWrite_mem.read() == true) {
    rsmux_exe.write(2);
}
if (rt_id2.read() != 0 && rt_id2.read() == WriteReg_mem.read()

```

```

&& RegWrite_mem.read()==true
&& MemRead.read()==false){
    rtmux_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

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

5 Conclusão