Sebenta AC1

Duarte Ferreira Dias

30 de Janeiro de 2017

Conteúdo

Capítulo 1

Disclaimer

Esta sebenta tem como objetivo principal ajudar todos os deseperados em fazer AC1 com o mínimo de esforço possível,(ou pelo menos com a menor dispersão possível :-).

Os conteúdos são baseados nos slides disponibilizados no e-learning e no livro recomendado pela regência da cadeira(Henessey).

A leitura deste texto de apoio(Sebenta) não dispensa a ida às aulas (Excepto se forem as do Ferrari(aka:Fiat Panda) eheh, se a regência for outra, os professores são excelentes e esforçam-se por tornar a matéria acessível tal como esta sebenta pretende) nem adispensa a leitura quer do livro quer dos slides.

Para quem quiser experimentar o mundo do latex está livre de adicionar conteúdo e adicionar o se nome como co-autor da sebenta.

Boa Sorte e Votos de Bom Estudo

eetésolidário odetienosso

PS: tb já há uma sebenta de AC2 :-)

Capítulo 2

Estrutura básica de um Processador

Os cinco componentes básicos de um computador são o input, output, memória, *datapath* e controlo.Estando os últimos dois elementos, geralmente integrados no processador.

Segundo o modelo de von Neumann existem três tipos de BUS.

- **Data Bus**: barramento de transferência de informação;
- **Adress Bus** : identifica a origem/destino da informação;
- **Control Bus** : sinais de protocolo que especificam o modo como a transferência deve ser feita;

2.1 Memória/Registos

A memória do mips merece um capítulo por si só mas fica aqui uma introdução...

O endereço é um número que identifica um úncio registo de memória, estes são contados de 0 até ao limite definido pela arquitectura.

Por consequêrncia o espaço de endereçamento é a gama total de endereços que o CPU consegue referenciar, isto é, o numero de registos de memória endereçaveis pelo sistema. Por exemplo, um cpu com um barramento de enderços de 16 bits consegue referenciar endereços que se encontrem na gama : 0x0000 <-> 0xFFFF.

2.2 Dados

A secção de dados, masi conehcida por datapath, envolve todos os elementos operativos e ou funcionais para o encaminhamento, processamento e armazenamento de informação. Entre eles, destacam-se os muxs, ALU e registos internos.

2.3 Unidade de Controlo

A unidade de controlo, comforme o tipo de datapath implementado, pode ser puramente combinatória ou uma máquuina de estados finit, algo que vai ser explorado no capítulo de Datapaths. Esta tem como função de gerar sinais(sinais de controlo) que em função das instruções recebidas pelo datapath permitem o correto encaminhamento, processamento e armazenamento das instruções.

Independentemente da unidade de controlo o CPU é e será sempre uma máquina de estados síncrona.

Capítulo 3

Operações na Arquitectura MIPS

3.1 Operações Aritméticas

Na arquitectura MIPS existem várias instruções aritméticas das quais se destacam :

1. add rd,rs,rt: corresponde à adição de b a c cujo resultado é guradado em $rd;
2. addu rd,rs,rt adição de grandezas sem sinal;
3. sub rd,rs,rt : subtração de b a c armazenada em $rd;
4. subu rd,rs,rt;

3.1.1 Adição e Subtração

Na Arqutectura MIPS a adição é feita bit a bit da direita para a esquerda somando o carry se necessário. A subtração assenta na adição pelo que o um dos operandos é negado.

O overflow da operação ocorre quando o valor calculado excede o limite da representação.

Abaixo estão representadas as condições para as quais existem overflow na soma e subtração.

Operation	Operand A	Operand B	Result indicating overflow
$A+B$	≥ 0	≥ 0	< 0
$A+B$	< 0	< 0	≥ 0
$A-B$	≥ 0	< 0	< 0
$A-B$	< 0	≥ 0	≥ 0

Figura 3.1: Condições que comfirmam a existência de Overflow

As instruções add,addi e sub originam uma excepção no sistema quando é detetado overflow na operação.

Já as instruções addu,addiu e subu não lançam essas excepções.

Uma vez que a linguagem C ignora as situações de overflow, por isso os compiladores C do MIPS e geram sempre as versões sem sinal das operações aritemáticas(unsigned).

3.1.2 Exemplo 1

Vamos efectuar as operações add e sub.

```
f = (g+h)-(i+j); // Sintaxe em C
```

```
#Sintaxe em Assembly do MIPS
add $t0,$1,$2;   # $t0 contem $1 e $2 -> g+h
add $t1,$3,$4;   # $t1 contem $3 e $4 -> i+j
sub $s0,$t0,$t1;
```

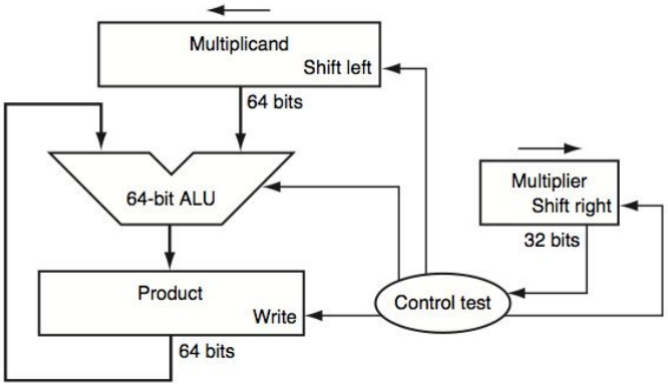

Figura 3.2: Primeira Implementação de um muiltiplicador

3.1.3 Mutlipicação

Usando as operações mult e mulu é possivel multiplicar o conteudo de dois registos

Numa primeira abordagem a execução desta operação consite em replicar em hardware as iterações que se fazem no papel.

Seguindo o método usual seria necessário shiftar 32 vezes o multiplicando para efetuar a operação. Essa operação de shift implica que o multiplicando se mova 32 bits para a esquerda, obrigndo a que o registo que o alberga tenha 64 bits de tamanho.Esse registo é shiftado um bit para a esquerda por cada iteração.

Os três passos de execução apresentados são repetidos 32 vezes para executar a multiplicação.Se cada operação fosse feita a cada ciclo de relógio demoraria 96 ciclos de relógio a se executada, o que torna a implementação bastante ineficiente.

Uma forma de otimizar este processo é paralelizar algumas das operações, reduzindo significativamente os ciclos de relógio, mais especificamente 1/3 dos ciclos necessários.Podemos com esta configuração efetuar a operação com uma iteração por ciclo de relógio.

Outra maneira utilizada pelos compiladores do MIPS é usar shifts para a esquerda ao invés da multiplicar, porém só é valido quando os registos são multiplicados por constantes, ou registos cujo valor equivalha a uma potência de dois;

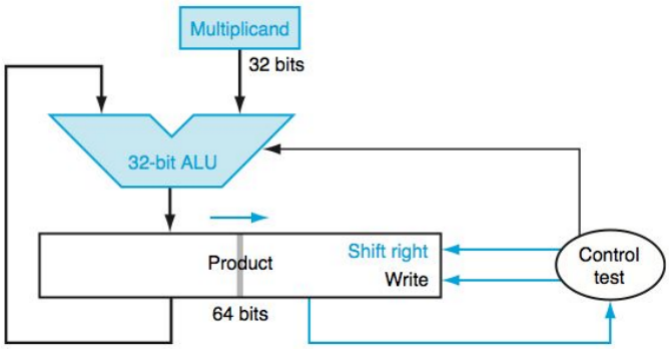

Figura 3.3: Implementação otimizada de um multiplicador

Resta abordar a forma como se armazena o resultado da operação uma vez que o tamanho do produto é de 64 bits.O MIPS disponibiliza dois registos de 32 bits para armazenar o produto um chama-se Hi e o outro Lo. Para buscar o valor na forma de inteiro de 32 bits é usada a pseudoinstrução mflo(*move from lo*).

3.2 Operaçõe s Load Store

Como as operações ariteméticas operam apenas sobre registos, torna-se necessária a criação de instruções que permitam carregar(*load*) e guardar(*store*) os valores dos registos da e para a memória. Surge, então, duas operações, a *load-word* com a sintaxe **lw**$rd,offset($registobase) e que permite a transferência de dados da memória para um regsto de destino, já a instrução *load-word* **sw**$rs,offset($registobase) que por oposição guarda na memória os valores de um registo.

3.2.1 Exemplo 2

Aplcação da instrução *load-word*

```
//sintaxe C
g = h + A[8];

#sintaxe Assembly do MIPS
lw $t0,32($s3);   #offset (4*8) da memoria
add %s1;$s2,$t0; #soma
```

Outro Exemplo agora com a aplicação de ambas as operações.

```
//sintaxe C
A[12] = h + A[8];

#sintaxe Assembly do MIPS
lw $t0,32($s3);   #leitura com offset (4*8) da memoria
add %s1;$s2,$t0;  #soma
sw $s2,48($s4);   #armazenamento com offset (12*4) da memoria
```

3.3 Operações com constantes e imediatos

Grande parte dos programas recorre a um numero de variáveis maior que o o número de registos do processador. Por isso o compilador tenta guardar apenas aquelas variáveis que que são mais utilizadas nos registos e move as outras para a menmória.

Um dos principios fundamentais do design de processadores surgere que a velocidade de acesso aos registos é maior que a de acesso à memória.

Para implementar operações com imediatos pode-se recorrer ás seguintes instruções:

```
lw $t0,const4addr($s1);#t0 = 4
add $s3,$s3,$t0; #s3 = s3 + t0(4)
```

A alternativa para implementar esta instrução é usar:

```
addi $s3,$s3,4;
```

O MIPS aceita nativamente esta instrução, permitiundo suprimir o uso da instrução lw juntamente com a add, optimizando assim a execução do programa.

3.4 Operações Lógicas

O mips suporta as seguintes operações lógicas:

- sll -> shift left logical;
- srl-> shift right logical;
- and,andi -> bitwise AND(pode ser feito com um imediato);
- or,ori -> bitwise OR (pode ser feito com um imediato);
- nor;

Uma aplicação bastante não tão óbvia do shift left logical permite multiplicar o valor do registo a shiftar pela potencia de 2 da quantidade de shift.

3.4.1 Exemplo 3

A operação de shift left logical :

```
sll $t2,$t0,4 # $t2 = $t0 <<4;
```

op	rs	rt	rd	shmnt	funct
0	0	16	10	4	0

3.5 Operações de Salto Condicional

As estruturas de decisão mais usadas em programação são os if's. As instruções que permitem implementar essas estruturas em MIPS são:

- beq $rd,$rt,label -> branch if equal -> consiste em comparar valores de dois registos e deteminar se são iguais;
- bne $rd,$rt,label-> branch not equal -> consiste em determinar se dois registos são diferentes slatando para uma label onde estará o conjunto de instruções que deve ser executado caso a instrução se confirme.

3.5.1 Nota sobre as operações virtuais

Todas as outras operações de salto condicional resultam da conjugação da operação (slt ou slti) que retorna um caso o valor lógico 1 caso o primeiro registo seja menor do que o o segundo e 0 caso o contrário.

A conjugação do set on less com as operações de salto condicional bne e beq permitem impelemtar as operações:

- bge;
- bgt;
- ble;
- bnez;
- beqz;
- blt

3.6 Operações de salto

Estas operações são necessárias para efetuar um salto de uma instrução para a outra.

Existem três operações de salto a JAL , JR e J.

A jal (mais conhecida por *jump and link*) é utilizada para saltar para o endereço da instrução que cooresponde ao inicio duma função, para elem disso guarda o valor do Program Counter e adiciona-he mais quatro para que quando a função acabar de executar e retornar o resultado ela vai saltar para o endereço da instrução seguinte à da execução da função;

O jr ou *Jump Register* consiste em saltar para um endereço especificado por um registo.

Capítulo 4

Memória do MIPS

Todas as linguagens de programação suportam vários tipos de dados cuja complexidade pode variar do muito simples(constantes, variáveis) para tipos de dados mais complicados(são exemplo os arrays e estruturas de dados). Uma vez que o processador não consegue armazenar grandes volumes de dados no seu interior(32 registos de 32 bits no caso do MIPS), este delega essa função para a memória que por sua vez é capaz de guardar grandes volumes de dados.

Por vezes torna-se útil endereçar individualmente cada um dos quatro bytes que constitui uma palavra. Pode-se dizer que a memória é*byte-adressable* É por isso que na arquitetura MIPS o endereçamento à memória deve ser feito em múltiplos de quatro. Para que este tipo de endereçamento seja possível é necessário inserir um offset para aceder aos vários bytes que constituem a palavra. Este método de endereçar os dados armazenados permite otimizar a performance geral do sistema pois otimiza as transferências na memória.

Capítulo 5

ISA-Instruction Set Architecture

De uma forma geral, o CPU segue a seguinte ordem de execução das instruções que lhe são fornecidas:

1. **Instruction Fetch** -> passo no qual é feita a leitura de código máquina da instrução.(Instrução reside em memória);

2. **Instruction Decode** -> nesta fase é feita a descodificação da instrução pela unidade de controlo presente no processador;

3. **Operand Fetch** -> leitura dos operandos;

4. **Execute** -> é feita a execução da operação especificada pela instrução;

5. **Store Result** -> o resultado das operações efetuadas no passo anterior são posteriormente transferidas para o o registo especificado pela operação.

O *Instruction Set* propriamente dito corresponde ao conjunto de instruções que o processador é capaz de executar. Por norma cada arquitetura de processadores ou micro-controladores tem o seu **Instruction Set** dentro das mais populares destacam-se os ISA's do MIPS, ARM, Intel x86, Power PC e Cell;

5.1 Instruções

Na arquitetura MIPS existem três tipos de instruções.Estas instruções servem têm comportamentos diferentes assim como fins diferentes.

Todas as instruções do MIPS são armazenadas em registos de 32 bits tendo o tamanho fixo.

O número de registos disponíveis é de 32 bits.Este número reduzido de registos deve-se principalmente a duas razões. Uma delas é que ao aumentar o número de registos aumenta-se o número de ciclos de relógio necessários para o acesso à memória, reduzindo a performance do processador, a outra prende-se com manter o tamanho das instruções pois aumentando o numero de registos era necessário aumentar a quantidade de bits nos campos de endereçamento da memória.

5.1.1 Instruções do tipo R

Estas instruções são responsáveis pelas operações lógicas e aritméticas.

Têm como operandos os seguintes elementos:

- **op** : mais conhecido por opcode, representa a operação da instrução que foi dada-6 bits;

- **rs** : o operando do segundo registo-5 bits;

- **rt** : operando do segundo registo-5 bits;

- **rd** : registo de destino onde é guardado o resultado da operação a executar-5bits.

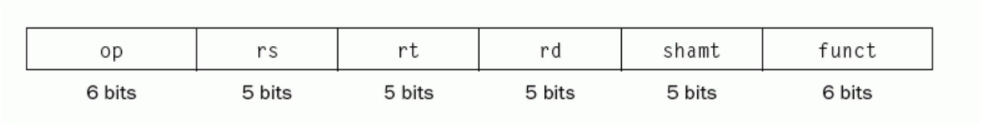

Figura 5.1: configuração das intruções do tipo R

5.1.2 Instruções do tipo I

As instruções do tipo I são usadas para exprimir as operações de leitrua e escrita na memória assim como as operações com imediatos.

Por permitir o endereçamento à memoria e representação de constantes a configuração das istruções é diferente. É de notar que a arquitetutra MIPS opta por manter o tamanho das intruções fixo (32 bits) alterando-se o tamanho de alguns dos campos da instrução..

Têm como operandos os seguintes elementos:

- **op** : mais conhecido por opcode, representa a operação da instrução que foi dada-6 bits;
- **rs** : registo sobre o qual vais ser processada a operação-5 bits;
- **rt** : opernado que armazena o registo com o valor base do endereçamento-5 bits;
- **offset** : registo no qual é armazenado o endereço da memória ou o valor da constante a ser carregado-16bits;

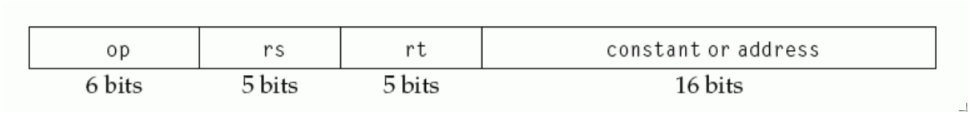

Figura 5.2: configuração das intruções do tipo I

Exemplo 1

```
lw $t0,32($s3)  #$s3 => 19
```

Ao utilizar esta operção iriamos obter uma istrução do tipo I com os seguintes valores nos seus campos:

opcode	rs	rt	offset
35	19	8	32

Ficamos com a seguinte instrução em binário 10001110011100000000000010000

5.1.3 Exemplo 2

```
A[300] = h + A[300];//sintaxe em c

lw $t0,1200($t1)
add $t0,$t0,$s2
sw $t0,1200($t1)
```

opcode	rs	rt	offset
35	19	8	1200

opcode	rs	rt	offset
43	9	8	1200

Capítulo 6

Funções

Na Arquitectura MIPS as funções seguem a seguinte sequência de execução:

1. Colocar os argumentos nos registos $a0,$a1,$sa2,$a3 (exceção será abordado abaixo);
2. Começar a execução da função, faz-se por via da instrução jal (*"jump and link"*) que guarda o endereço da instrução seginte em $ra;
3. Alocar memória para a execução do procedimento;
4. Executar a função;
5. Guardar o valor de retorno em $v0 ou $v1;
6. Finalmente usa-se a instrução de salto jr com o valor de $ra(registo que guarda o ponto a retornar) para saltar para a instrução seguinte função chamadora estava antes de iniciar a função.

Existem dois tipos de funções:

- Folha : todo o conjunto de subrotinas que não chama uma subrotina;
- Não Folha : analogamente são aquelas funções que chamam funções dentro de si mesmas.

Em casos normais a ordem de processamento segue a ordem desses pontos, porém quando os valores de return ou argumentos são mais do que os que estão convencionados torna-se necessário proceder com alguma cautela.

Relembrando que a função não pode deixar resíduos da sua execução qualquer, os registos da função chamadora que forem precisos guardar têm que ser repostos imediatamente após a execução da função e com o valor que tinham quando antes da função ser executada.

A estrutura de dados ideal para implementar o que acima foi descrito é uma stack(LIFO). A stack precisa de um ponteiro($sp, *Stack Pointer*) que referencie o endereço que foi alterado mais recentemente por forma a permitir que a funcção a executar saiba onde guardar os seus dados assim como permite restaurar os valores antigos. Por cada dado que é acrescentado é necessário dar um incremento de uma palavra ao *Stack Pointer*(4bytes). Por convenção uma pilha enche do endereço mais alto para o mais pequeno.Por isso quando se adiciona conteudo à mesma é necessário decrementar o valor do ponteiro.

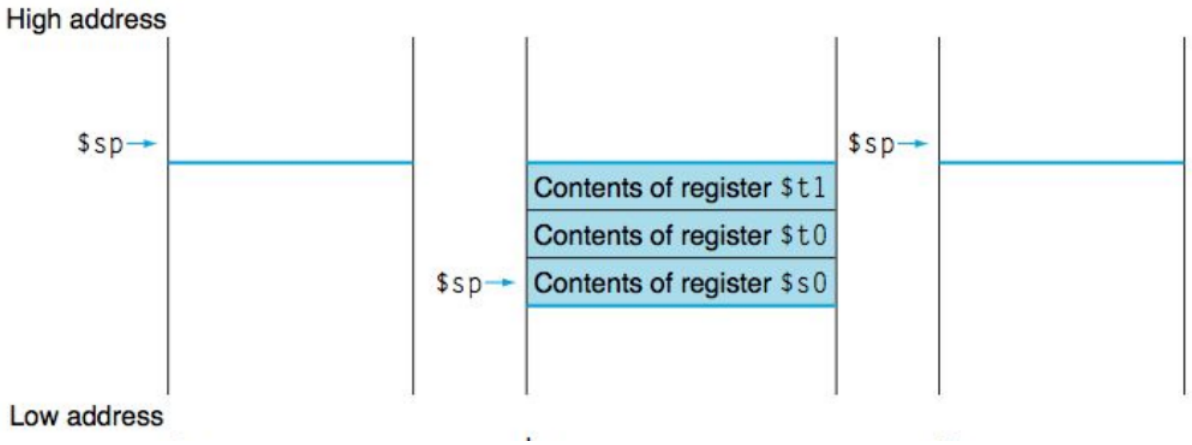

Figura 6.1: Comportamento do *Stack Pointer*

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Figura 6.2: Convenções dos Registos no MIPS

Capítulo 7

Datapath

7.1 Datapath Single Cycle

Nos capitulos acima forma descritas as instuções base do MIPS agora vamos tentar criar um processador que execute essas instruções o conjunto de blocos que as executa chama-se de datapath.

Os dois primeiros passos de execução de uma instrução consistem em :

1. Usando o endereço de memória armazenado no PC para aceder à instrução armazenada na memória de instruções;
2. Ler o conteudo de um ou dois registos(depende da instrução a executar);

Depois destes dois passos iniciais dependendo do tipo de instrução a executar os dados vão seguir caminhos diferenntes.Dentro do mesmo tipo de instruções os elementos a usar não variam muito, tornando mais fácia a implementação do Datapath.

Abaixo seguem os elementos base e as suas funcionalidades de um datapath single cycle.

PC + Memória de Instruções A junção do PC com a Memória de instruções e um adder permite-nos montar um circuito que permite efetuar o endereçamento à memória de instrução para poder dar inicio à execução de uma instrução o adder permite determinar qual o endereço da próxima instrução.Esse adder vai adicionar 4 bytes ao endereço atual do PC (PC+4).

Registos

7.2 Datapath Multi Cycle

A implementação do datapath Multi Cycle consiste em dividir a execução de uma instrução em várias fases(operações).Cada uma dessas fases usa ium dos elemntos fundamentais(memória,register file ou ALU). Em cada ciclo de relógio é possivel executar várias operações em paralelo desde que sejam independentes. Ao usar esta estratégia a frequência máxima de funcionamento depende apenas pelo maior dos tempos de atraso de cada um dos elementos.

A implementação deste datapath é feita com um máximo de 5 fases.

7.2.1 Estrutura

A versão multi-cycle do datapath em comparação ao datapath single-cycle terá apenas uma úncia memória (arquitectura *Von Neumann*) e uma única ALU.

A existência de uma única memória implica que os acessos à memória de instrução e memória de dados sejam controlados para evitar que os dados na memória se corrompam.

Porque a execução de uma instrução vai demorar mais que um ciclo de relógio vai ser necessária a existência de registos à saída dos elementos funcionais para que, caso os valores produzidos pela unidade em causa sejam necessários no ciclo seguinte estes estejam resguardados. Foram então adicionados cinco registos, memória de dados, memória de instruções, A e B(encontram-se à saída do Register File) e ALUOut.

A utilização de apenas uma alu em comparação ao uso de uma alu mais dois somadores obriga a que se alterem o número de entradas dos multiplexer que se encontra à segunda entrada da ALU assim como adicionar um à primeira. Na primeira entrada o mux vai permitir selecionar a saída do registo A e a saída do Registo PC.No segundo mux é possível escolher entre o registo B a constante 4, o sinal proveniente do signal extender e por fim o sinal que vem do shift left 2.

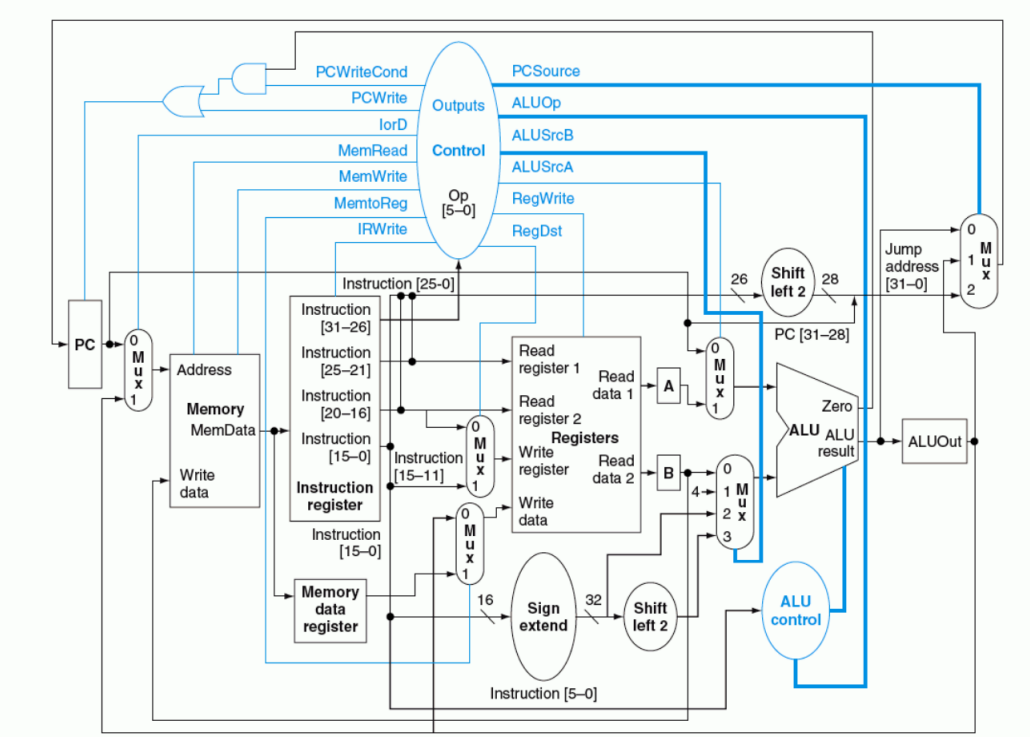

Figura 7.1: Datapath multi-cycle completo

7.2.2 Fases de Execução

O datapath abordado pela cadeira reconhece 5 fases de processamento. Sendo que as duas primeiras são comuns a qualquer instrução.A única instrução que faz uso das cinco fases de execução do MIPS é a instrução se LW, sendo que as operações de salto condicional(branch) e jump demoram apenas 3 fases(ou ciclos) e as instruções de escrita na memória(SW) e do tipo R demoram 4 fases(ou ciclos de relógio) a serem executadas. Abaixo são descritas as operações executadas e os sinais de controlo usados em cada fase de execução mediante a instrução a executar.

Fase 1

É nesta fase que se efetua o instruction fetch(IF).

É feita a escrita no instruction register através do endereçamento à memória(o PC é o endereço usado) e o PC é incrementado por quatro(PC +4).

Sinais de Controlo

- MemRead = '1';
- IRWrite = '1';
- IorD != '0'; -> garante que o PC é usado como endereço da memória;
- ALUSrcA = '0' -> efetua seleção do valor de PC;
- ALUSrcB = "01-> Seleção da constante 4(PC+4);
- PCSource = "00-> é a saída da ALU;
- PCWrite = '1' -> escrita do PC

Fase 2

É nesta fase que se efetua o instruction decode(ID),Operand Fetch e Cálcuo do Branch Target Address(BTA).

Neste passo ainda se desconhece qual a instrução que se vai executar. Para não por em causa a correta execução da instrução, apenas são efetuadas as operações que não influenciam a execução natural de uma instrução.

É nesta operação que se acede aos registos ,cujos endereços são fornecidos pelos campos rs e rt, e se guardam os mesmos nos registos A e B. é também calculado o BTA sendo este guardado no registo ALUOut.

Sinais de Controlo

- ALUSrcA = '0' -> efetua seleção do valor do PC;
- ALUSrcB = "11-> Seleção do sinal que sai do shift;
- ALUOp = "00-> é efetuada a soma;

Fase 3

A partir desta fase a sequência de operações a executar depende diretamente da instrução a executar.

Instruções de Referência à Memória(LW,SW) Soma dos operandos(registoA e Signal Extender) na alu para formar o endereço.

Sinais de Controlo

- ALUSrcA = '1' -> efetua seleção do registo A;
- ALUSrcB = "10-> Seleção do sinal que sai do signal extender;
- ALUOp = "00-> é efetuada a soma;

Instruções do tipo R É executada a operação designada pelo campo funct ou funct + shmnt da instruçãp.

Sinais de Controlo

- ALUSrcA = '0' -> efetua seleção do registo A;
- ALUSrcB = "00-> Seleção do sinal que sai do registo B;

Instrução Branch A alu é usada para verificar uma igualdade(efetua uma subtração) sendo o sinal zero(uma saída da ALU) é usado para determinar se o branch é válido ou não, isto é se a condição de salto se verifica)

Sinais de Controlo

- ALUSrcA = '1' -> efetua seleção do registo A;
- ALUSrcB = "0-> Seleção do sinal que sai do registo B;
- ALUOp = "01-> subtração;
- PCWriteCond = 1/0 dependente da verificação da condição de salto;
- PCSource = "01"valor escrito no PC virá do registo ALUOut;

Jump O valor do PC é substituido pelo endereço de salto. O PCSource é alterado para "10"para direcionar o JA(Jump Address) para o PC e o PCWrite é ativado para que o novo valor do PC seja escrito na próxima transição ativa de relógio.

Fase 4

Nesta fase é feito o acesso à memória e concluem-se as instruções do tipo R e store-word (SW).

Referência à Memória Quando um valor é devolvido pela memória este é guardado no MDR(*Memory Data Register*) para se usado no próximo ciclo de relógio.

Sinais de Controlo Se a operação for de leitura:

- MEMRead = 1;

Se for de escrita :

- MEMRead = 1;

Em ambas o IorD é colocado a 1 para que o endereço seja fornecido pela saída da ALU.

Conclusão das instruções do tipo R Os conteúdos do registo ALUOut são guardados no registo de destino.

Sinais de Controlo

- RegDST = '1' para usar o endereço fornecido por RS
- RegWrite = '1';
- MemtoReg = '0' -> assegura que a saída da ALU é escrita no registo;

Fase 5

Última fase de execução do datapath na qual é feito o Write-Back da operação LW, isto é, a escrita no registo de destino. Os sinais de controlo comutados são o MEMtoReg para 1 e o RegDst para '0'.

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]),2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

Figura 7.2: Resumo de Operações

7.2.3 Unidade de Controlo

No datapath Syngle-Cycle, no qual as instruções se executam num único ciclo de relógio e por isso a unidade de controlo é apenas responsável pela geração de sinais que se mantêm inalterados ao longo da execução de uma determinada instrução,sendo a lógica de controlo é puramente combinatória.

A unidade de controlo do datapath Multi-Cycle passa a ser uma máquina de estados, já que é necessário gerar sinais diferentes para as várias fases de execução das diferentes instruções suportadas pelo datapath, o que impossibilita a utilização de apenas lógica combinatória.

A máquina de Estados

A máquina de estados da unidade de controlo do datapath contem dez estados diferentes, abaixo explicados.

Como os dois primeiros ciclos de execução são comuns a todas as instruções, correspondem a dois estados de execução. O primeiro estado corresponde à primeira fase de processamento(*Instruction Fetch*), o segundo como seria expectável é a fase de *Instruction Decode*, este estado tem cinco destinos diferentes. Os possiveis estados de destino serão:

- SLTI-> encaminha para os estados responsáveis pela execução destas operações;
- ADDI,SW,LW;
- Instruções do tipo R;
- beq;
- jump;

Instruções do tipo R Para executar as operações de tipo R são necessários dois estados adicionais, um deles para contolar a ALU para que esta desempenhe a operação designada pelas operações fornecidas, assimcomo assegurar que os dados que entram na ALU são os corretos.

O ultimo estado grava o valor calculado no registo de destino indicado pela instrução.

ADDI A execução da instrução ADDI exige que sejam criados dois estados adicionados, sendo que o primeiro tem como função definir a operação a re alizar pela ALU e assegurar a correto encaminhamento dos operandos para a mesma.

O último tal guarda o valor calculado pela ALU num registo.

SLTI Para realizar esta operação é necessário assegurar o encaminhamento do segundo operando e garantir que ALU efetue a operação correta, o estado de escrita no registo é garantido pelo mesmo estado da instrução anterior(???).

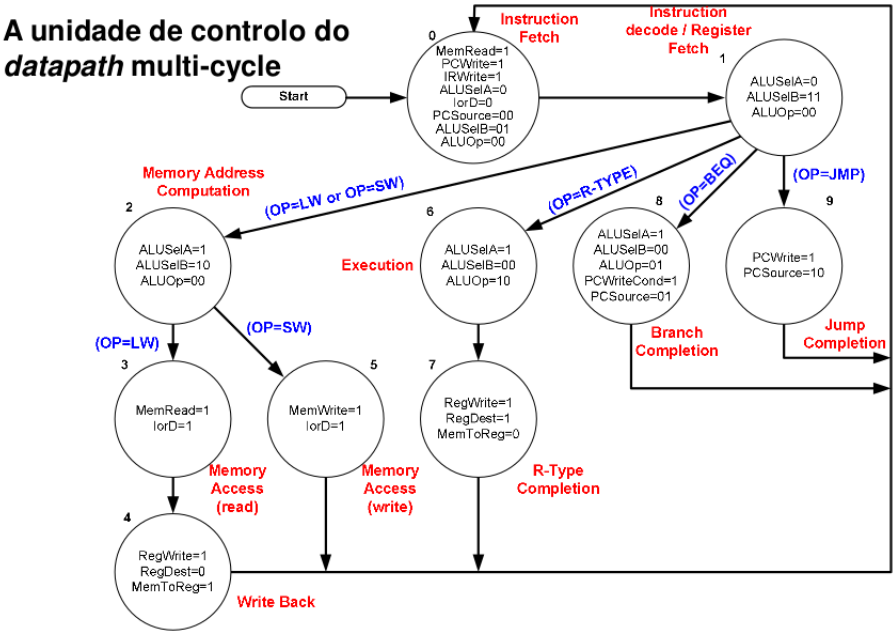

Figura 7.3: Estados da Unidade de Controlo

7.3 Pipeline

Esta estratégia para a implementação do datapath permite aumentar o numero de instruções executadas num determinado momento para o (número de instruções*numero de fases(F) do pipeline)-1. Aprofundando, ao invés de executar uma instrução exclusiva em cinco ciclos de relógio(numero de fases do pipeline apresentado), aproveita os elementos lógicos(que se enquadram numa determinada fase de execução) que vão sendo libertadas na execução de uma primeira instrução, quando a primeira instrução liberta uma das fases uma segunda instrução entra no datapath e dá-se inicio à sua execução na fase que fora libertada anteriormente.

7.3.1 Estrutura

Pegando no datapath Single-Cycle, é possível dividi-lo em fases de execução.

1. Instruction Fetch;
2. Insruction Decode;
3. Execute;
4. Memory Access;
5. Write Back;

Como as instruções são executadas ao longo de vários ciclos de relógio é necessário guardar os valores que vão passando de uma fase para a outra do pipeline. Para isso foram criados registos que se encontram entre as fases de processamento do pipeline.

- IF/ID;
- ID/EX;
- EXE/MEM;
- MEM/WB;

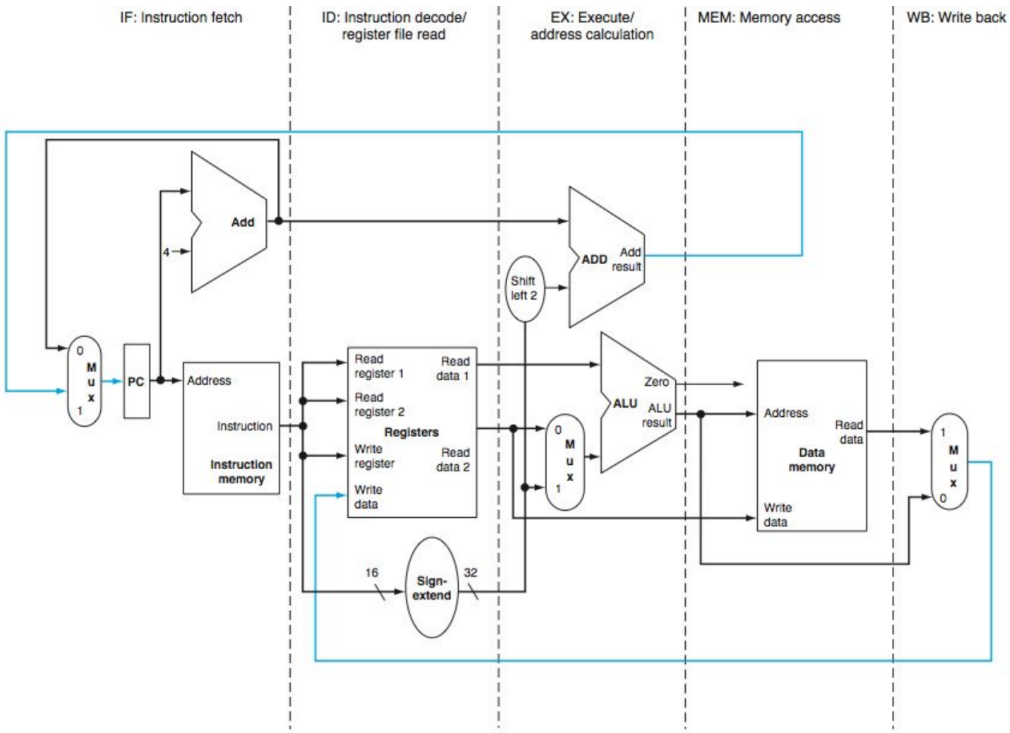

Figura 7.4: Datapath Single-Cycle dividio nas fases de execução

Conforme as instruções vão sendo executadas e dependendo do tipo de instruções, os dados necessários são retransmitidos de fase para fase, ou seja, transitam de registo para registo.

Assim obtemos um datapath Pipelined muito simplificado, no final da secção de hazards é possível encontrar um datapath mais avançado que suporta a resolução de hazards.

Pode-se concluir que os elementos lógicos existentes em cada fase só podem ser usados uma vez por ciclo de relógio.

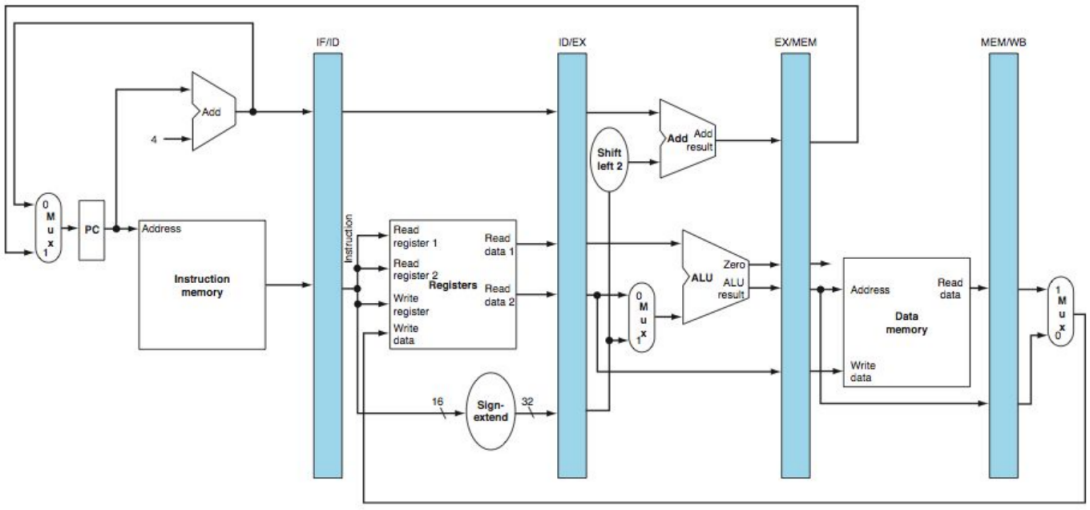

Figura 7.5: Datapath Pipelined simples

7.3.2 Unidade de Controlo

A unidade do datapath pipelined é semelhante à do datapath Single-Cycle. É uma unidade combinatória que gera os sinais de controlo em função da instrução a executar. os sinais de controlo avançam ao longo das fases do pipeline.

Os sinais de controlo utilizados são em tudo semelhantes aos usados no datapath ao single cycle.

É possível dividir as linhas de controlo pelas várias fases de processamento.

Instructon Fetch Os sinais de controlo utilizados para ler a memória de instruções e o PC estão sempre assegurado, não é necessário nenhum sinal de controlo adicional.

Instruction Decode/register file read Tal como na fase anterior não é necessário nenhum sinal de controlo especifico para esta fase.

Execution Nesta fase já são necessários sinais de controlo especificos:

- RegDst;
- ALUOp;
- ALUsrc;

Estes sinais tal como no datapath Single Cycle permitem selecionar o registo de destino, a operação a executar na alu e direcionar,conforme a instrução a ser executada, os sinais provenientes do signal extender ou da saida ReadData2.

Memory Access Os sinais de controlo nesta fase são os sinais de Branch, MemRead e MemWrite. Sinais que variam conforme a instrução a executar.

Write Back As linhas de controlo existentes nesta fase são o MemtoReg e o RegWrite. O primeiro decide entre mandar o resultado proveniente da ALU ou o valor de memória para um Registo, já o segundo escreve o valor escolhido anteriormente.

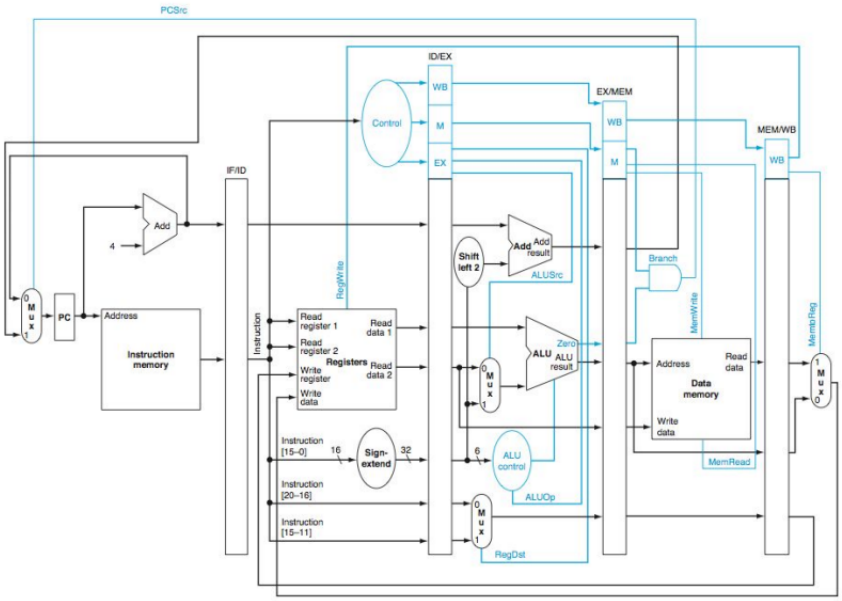

Figura 7.6: Datapath Pipelined com sinais de controlo

7.3.3 Hazards

No seguimento da execução de várias instruções podem haver problemas tais como acessos simultâneos à memória, ou sobreposição de registos, por exemplo quando se efetua o instruion fetch e em simultâneo outra fase mais adiantada de execução faz um acesso à memória.

Outro exemplo

```
add $s0,$t0,$t1$
sub $t2,$s0,$t3
lw $t4,0($t2)
```

Neste exemplo vemos que o registo s0 é usado pela instrução seguinte, o problema surge porque a instrução add não é concluida a tempo de forncer o resultado para correto. para que a operção seguinte(sub).

Aos problemas que surgem durante a execução das intruçoes ao longo dos vários ciclos de relógio no pipeline dá-se o nome de Hazards.

Existem três tipos de Hazards:

- Hazards Estruturais;
- Hazards de Dados;
- Hazards de Controlo;

Hazards Estruturais

Este tipode hazards ocorre quando o Hardware não suporta a execução de uma combinação de instruções num único ciclo de relógio, ou seja, quando uma instrução tem que aceder ao mesmo hardware no mesmo ciclo de relógio. Surgem associados a acessos simultâneos à memória, problema que é resolvido usando duas memŕorias no datapath, uma para os dados e outra para o programa.

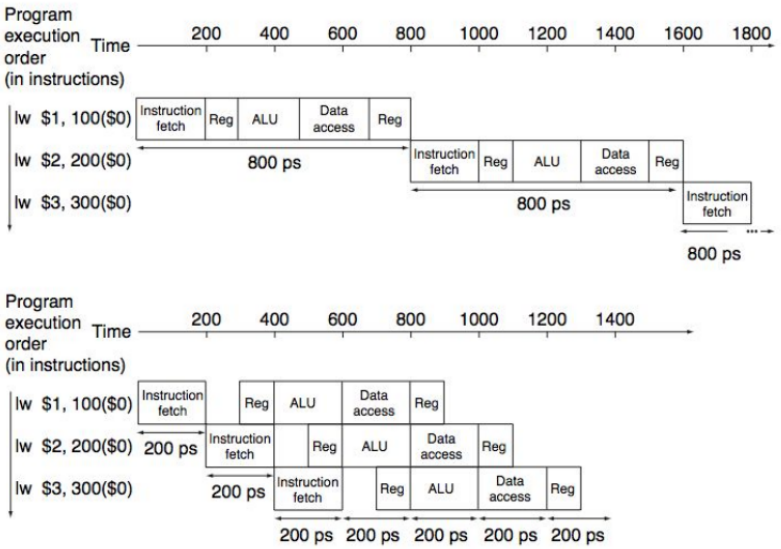

Figura 7.7: Execução de instruções: Outros vs Pipeline

Hazard de Controlo

Este tipo de hazards ocorre quando é necessário parar a execução das instruções no pipeline. Quando é executado um instruction fetch para uma instrução, mas existe uma instrução,mais avançada no pipeline, que pode pudar o curso de execução do programa. Conclui-se que estes hazards surgem associados às instruções que envolvem branches e jumps.

Existem três maneiras de lidar com Hazards de Controlo:

- Stalling;
- Previsão;
- Delayed Branch;

Stalling O Stalling consiste em atrasar a entrada de uma nova instrução no pipeline até que se saiba o resultado da instrução anterior. Assim que é detetado um branch a instrução que se encontra a seguir é substituida por um nop(por exemplo um sll 0,0,0)

Esta alternativa é eficaz mas não eficiente, uma vez que implica a paragem do datapath. Revela-se,assim, demasiado lenta para ser adotada, subretudo em pipelines com várias fases. Torna-se imperativo encontrar uma outra solução.

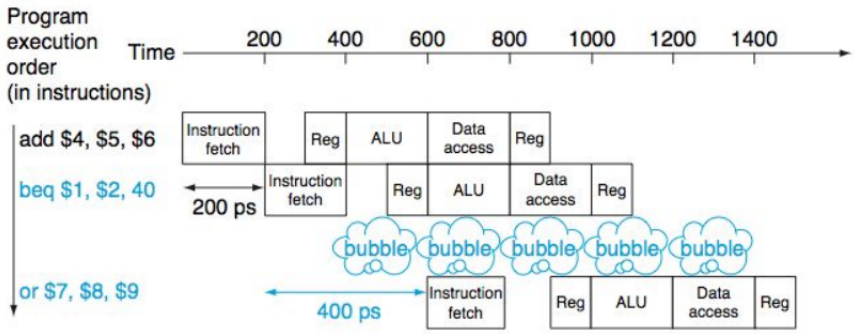

Figura 7.8: Stalling em todos os Branches

Previsão Nesta aboragem quando os branches são detetados são considerados como *not taken*, isto é, o resultado do branch é false.

Caso a previsão esteja errada a unidade de contolo do pipeline força o datapath a reiniciar a execução da instrução, substitui a instrução a seguir ao branch por uma NOP e continua o Instruction Fetch da operação correcta.

Quando se usam previsores estáticos(previsores taken, ou not-taken) o resultado da previsão não depende do resultado das instruções.

Nesta sebenta apenas iremos abordar a previsão estática. É possível fazer previsões mais acertadas usando previsores dinâmicos para que, ao longo da execução as previsões se alterem mediante os resultandos que vão sendo obtidos.Este tipo de previsores permite optimizar a previsão por forma a acertar em 90 das situações de branch.

Comparado com o stalling a previsão permite optimizar o pipeline, msmo assim quando a previsão está errada vão-e perder ciclos de processamento.

Branch Delay Slot O Branch Delay Slot consiste em executar a instrução imdeiatamente a seguir ao branch, quer este seja taken or not taken.

Esta técnica é aplicada ao nivel do software, neste caso o compilador que organiza as as instruções por forma a reposicionar o branch para aumentar a eficiência. Esta troca só é feita quando não existem implicações na execução do programa. Quando o compiilador não consegue aplicar estas mudanças é introduzida uma NOP.

Hazards de Dados

Quando há dependência do resultado de uma instrução com o operando de uma instrução que se encontra mais atrasada no pipeline estamos perante um hazard de dados.

Se o resultado que vai ser necessário pela instrução atrasada ainda não tiver sido armazenado em memória não se pode prosseguir a execução, já que vai ser usado como operando um valor destualizado e, por isso ,errado.

Podemos concluir que estes hazards ocorrem quando uma instrução opera sobre dados, dados estes que estão a ser manipulados por instruções mais à frente no pipeline.

Há duas soluções para este problema :

- Forwarding
- Stalling

Forwarding Ao observar a execução das várias instruções suportadas pelo pipeline é possível concluir que não é preciso esperar cinco fases para obter o resultado válido de uma instrução, muitos hazards podem ser resolvidos antecipando a aquisição dos resultados necessários. Por exemplo, as instruções do tipo R calculam o resultado na terceira fase de execução(EX), podendo o resultado ser disponibilizado para a instrução seguinte. A este processo dá-se o nome de Forwarding, ou Bypassing

Este método só pode ser aplicado se houver uma relação de causalidade entre as operações, isto é, só se pode fazer forwarding de dados para uma fase de dados subsequente, ou seja, que não tenha ocorrido.Quando não se verifica a causalidade temos que recorrer a soluções como o stalling.

Exemplos...

Por forma a resolver um hazard de de dados é ncessário que primeiro, sejam detetados os Hazards e segundo, é preciso garantir que se faz encaminhamento do valor ou valores pelas diferentes fases de execução do pipeline que se encontram mais avançadas. A maior parte dos hazards são resolvidos encaminhando os valores que se

encontram em fases mais avançadas do pipeline para, normalmente, a fase EX, fase em que são necessários, com excepção dos *branches*, os valores corretos para os registos a manipular pela ALU.

Forwarding Control Unit As situações de hazard que carecem de rencaminhamento(forwarding) podem ser facilmente identificadas:

1. Instrução na fase MEM cujo destino é o registo de uma instrução que se encontra na fase de EX.
2. Uma instrução que se encontra em WB cujo registo de destino seja um registo de uma instrução na fase de EX

1 De forma simplificada temos que:
EX/MEM.RDD == **ID/EX.RS** ou
EX/MEM.RDD == **ID/EX.RT**
Exemplificando

```
MEM add $1,$2,$3
EX  sub $4,$1,$5
```

2 De forma simplificada temos que:
MEM/WB.RDD == **ID/EX.RS** ou
MEM/WB.RDD == **ID/EX.RT**
Exemplificando

```
WB  add $1,$2,$3
ME; add $6,$2,$3
EX  sub $4,$5,$1
```

Exemplo Slides(61)
Embora as situações acima referidas permitam fazer uma deteção de Hazards esta não é suficiente, para o ser, é necessário avaliar o valor do sinal que controla a escrita nos registos(RegDst)
Ficamos com:
(**EX/MEM.RegWrite** == 1) and(**MEM/WB.RDD** == **ID/EX.RS**) ou
(**EX/MEM.RegWrite** == 1) and ()**MEM/WB.RDD** == **ID/EX.RT**)
(**MEM/WB.RegWrite** == 1) and (**EX/MEM.RDD** == **ID/EX.RS**) ou
(**MEM/WB.RegWrite** == 1) and (**EX/MEM.RDD** == **ID/EX.RT**)

Stalling O Stalling, que já fora abordado levemente na introdução pipeline, consiste em parar a execução das instruções durante um ou mais ciclos de relógio.

Esta é uma alternativa que torna o cpu pouco eficiente. Atualmente ,para evitar este tipo de situação(stalling) os compiladores tentam efetuar uma reordenação de instruções sem alterar o resultado final do programa. Vejamos o seguinte exemplo:

```
lw $t0,0($t1)
            lw $t2,4($t1)
            sub $s0,$t1,$t2 #Aqui ocorre stalling causado por um Hazzard de dados
            sw  $t0,4($t1)
            #efetuando uma troca ficamos com:
            lw $t0,0($t1)
            lw $t2,4($t1)
            sw  $t0,4($t1)
            sub $s0,$t1,$t2 #efetivamente a troca da instru  o de sub com a sw n o
```

Outro exemplo

```
            lw  $4,20($1)
            sub $2,$4,$3 #Aqui ocorre stalling causado por um Hazzard de dados
            add $3,$3,$2
```

Na primeira parte do exemplo dado acima podemos concluir que o stalling é desencadeado quando uma instrução aritmética(ou lógica) é executada a seguir a uma instrução de load com a qual estabelece dependência, registo 4. A situação de stalling é desencadeada quando sub(instrução do tipo R) se encontra na sua fase de ID e pode ser detetada se atendermos à seguinte condição:

(**MEM/WB.RegWrite** == 1) and (**EX/MEM.RDD** == **RS** or **EX/MEM.RDD** == **RT**).

Quando é detetada a situação de stalling è inserida uma *bubble* na fase EX fazendo um reset síncrono ao registo ID/EX. De seguida param-se as fases IF e ID para impedir a escrita no registo IF/ID e o incremento do PC.