

ARQUITETURA DE COMPUTADORES I

2º Ano 1º Semestre

TPI • PIA

• José Luís Azevedo (jla@ua.pt)

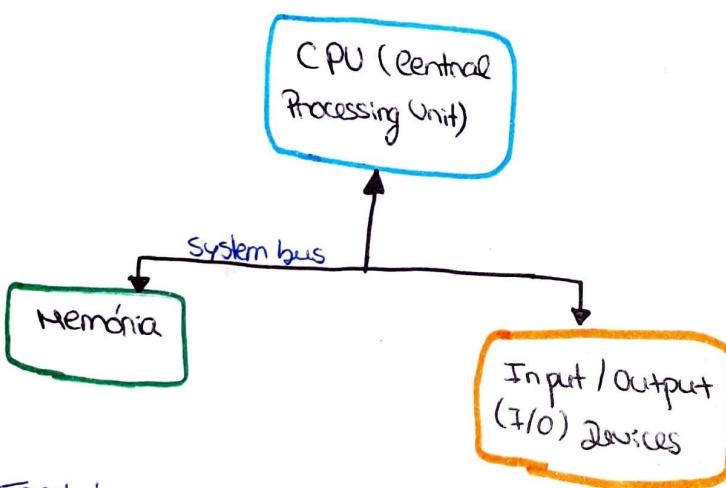
• Avaliação: Prática \Rightarrow 40%.

TPI: 40% (17/11 e 18/11)

TP2: 60% (em conjunto com o exame final técnico)

Teórica \Rightarrow 60%
exame final

① L Computador: the big picture



- CPU \rightarrow Também chamado de microprocessador. Executa instruções sequencialmente.
- Memória \rightarrow Armazena o programa (conjunto de instruções) e dados.
- I/O devices \rightarrow Comunicação com o exterior.
- System bus \rightarrow Interliga os sistemas.

L Versão Simplificada do CPU

- 3 blocos fundamentais:

ALU:

Realiza operações aritméticas e lógicas (p. ex. $+$, $-$, \times , $:$, And, Or, Not, Xor, ...)

Registros:
elementos de armazenamento localizados dentro do CPU

Unidade de Controle:
Responsável pela coordenação dos blocos do CPU, durante a execução de uma instrução.

- Sobre os registos ; os mais importantes são :

Program Counter (PC) :

- Guarda o endereço da memória onde se situa a próxima instrução a executar
- No CPU, após a leitura do código de uma instrução, o PC é actualizado.

Registos de utilização geral :

- São habitualmente referenciados por nomes (exemplo: \$0, \$1, ..., \$31)

Níveis de Representação :

Alto nível de programação (em C)

↓ Compiler

Linguagem Assembly (instruções)

↓ Assembler

Código de máquina em binário

Sobre Assembly :

- É a linguagem básica de programação de processadores, legível por humanos.
- Conjunto de instruções que realizam operações simples :
 - Somar o conteúdo de 2 registos
 - Inicializar um registo com um valor
 - Transferir um valor de um registo interno para a memória

Exemplos :

add \$1, \$5, \$7 # \$1 = \$5 + \$7

ori \$6, \$0, 0x1234 # \$6 = \$0 | 0x1234 = 0x1234

↳ O MIPS

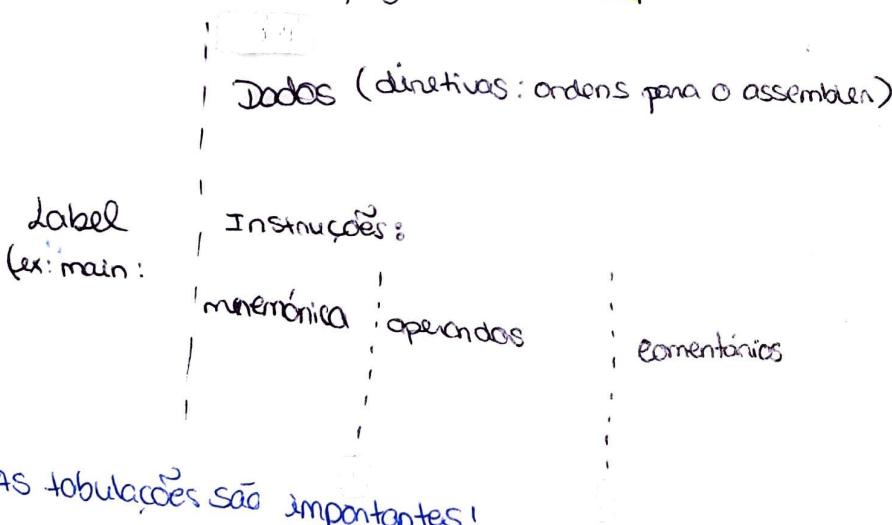
- É um microprocessador de 32 bits, isto é:
 - Cada registo interno armazena uma word de 32 bits
 - A ALU opera sobre quantidades de 32 bits.
- Tem 32 registos internos de uso geral, com a designação nativa \$0, \$1, ..., \$31.
- Estes registos são normalmente referenciados nas programações por um nome lógico:
 - \$a0, \$a1, \$a2, \$a3
 - \$t0, ..., \$t9
 - \$s0, ..., \$s7
 - \$v0, \$v1
 - \$ra

- O registo \$0 é um caso particular, uma vez que não permite armazenamento, e retorna sempre o valor 0.

Instruções do MIPS:

- Operações aritméticas: add, sub, addi (add immediate)
- Operações lógicas bitwise: and, or, ori

↳ Anatomia de um programa Assembly



* AS tabulações são importantes!

↳ Ambiente de simulação para o MIPS - MARS:

- MIPS Assembler and Runtime Simulator

- É um IDE com editor, assembler e simulador.

Permite

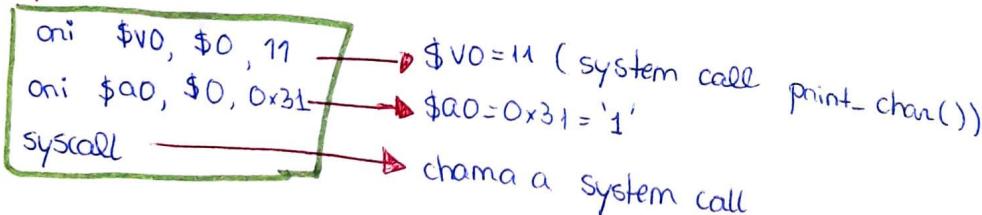
- Execução do programa assembly de 1 só vez, ou instrução a instrução (single step execution)
- Acesso aos registos internos da CPU
- Acesso à memória
- Interagir com o exterior (system calls)

↳ System calls

- São funções do sistema operativo que implementam serviços básicos de I/O.
 - Imprimir strings, ler inteiros, ler strings,...

- O MARS disponibiliza cerca de 50 system calls.
 - Registo \$v0 → identifica a system call
 - \$a0 a \$a3 → argumentos da system call
 - O SC pode usar \$v0 para devolver um valor.

Exemplo:



Como funciona um system call?

- O SO verifica \$v0 para saber qual a tarefa a realizar
- Se necessário, o SO lê os valores dos registos de argumento.
- O SO executa a tarefa
- O SO coloca o resultado no registo \$v0

↳ A Máquina e a sua linguagem

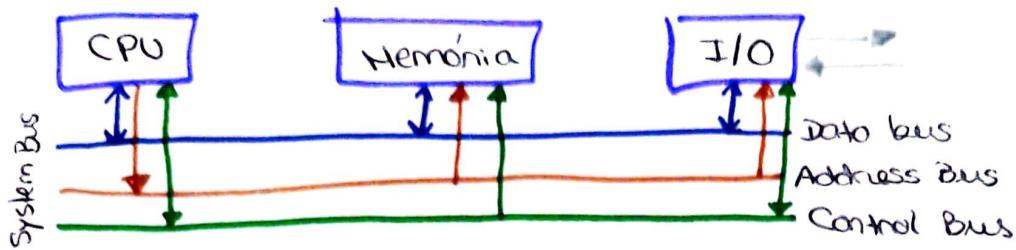
- Nas computadoras atuais, as instruções são representadas da mesma forma que os números.

- Os programas são armazenados em memória, para serem lidos e escritos, tal como os números.

- Estes princípios formam os fundamentos do conceito da arquitetura "stored-program"

↳ Na memória pode residir informação de vários tipos: código fonte de um programa em C, um editor de texto, um compilador,....

↳ Modelo computacional de von Neumann



- Data bus: transferência de informação (dados) $\text{CPU} \leftrightarrow \text{Memória} / \text{CPU} \leftrightarrow \text{I/O}$
- Address Bus: identifica a origem ou destino da informação
- Control Bus: sinais de protocolo que especificam o modo como a transferência de informação deve ser feita (enable, etc)

No CPU:

→ Datapath (seção de dados): elementos operativos / funcionais, para encominhamento, processamento e armazenamento de informação:

- Multiplexers
- ALU
- Registros internos

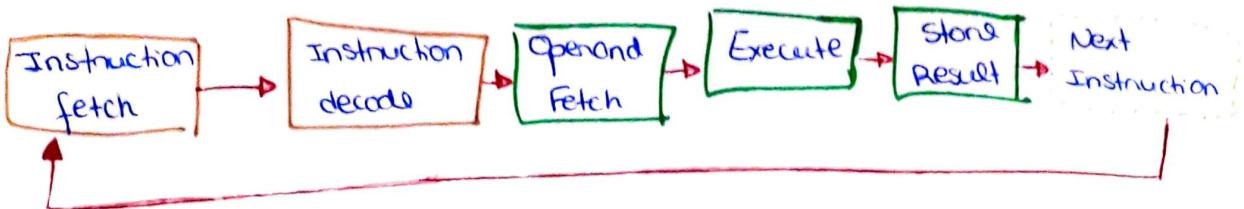
→ Controlpath (unidade de controlo): responsável pela coordenação dos elementos do datapath.

- Gera sinais de controlo
- Pode ser uma máquina de estados ou um elemento meramente combinatório

Independentemente, o CPU é sempre uma máquina de estados síncrona.

↳ Ciclo-base de execução de uma instrução:

- Instruction fetch: leitura do código máquina da instrução
- Instruction decode: decodificação da instrução pela unidade de controlo
- Operand fetch: leitura dos operandos
- Execute: execução da operação especificada pela instrução
- Store result: armazenamento do resultado da operação no destino especificado na instrução



↳ Codificação das instruções:

- Tem de conter toda a informação de que o CPU necessita para a execução da instrução:

→ Qual a operação a realizar?

→ Qual a localização dos operandos?

→ Onde colocar o resultado?

→ Qual a próxima instrução a executar?

• Normalmente o fluxo é sequencial, por isso não é necessário mencionar

• Há instruções que alteram a sequência de execução, logo a instrução deverá fornecer o endereço da próxima instrução.

Registros → número do registo

Memória → endereço

Registros

Memória

↳ Arquitetura do Conjunto de Instruções:

- Instruction Set: Coleção de todas as operações suportadas pelo processador.
- Instruction Set Architecture (ISA): "os atributos de um sistema computacional tal como são vistos pelo programador, i.e. a estrutura conceitual e o comportamento funcional" (Amdahl, 1964).
- Arquitetura de Computadores = ISA + organização da máquina
- A ISA é também designada por "modelo de programação".
- Descreve tudo o que o programador precisa de saber para programar corretamente, em assembly, um determinado processador.
- É uma abstração importante que representa a interface entre o nível mais básico de software e o hardware. Assim, descreve de forma independente do hardware que a implementa.
- Podemos falar em "arquitetura" e "implementação de uma arquitetura"
- ↳ Os processadores AMD são compatíveis com Intel x86, pois têm a mesma arquitetura, mas implementações diferentes.

- Requisitos básicos da ISA:

- Fácil de entender / programar
- Compilações eficientes
- Implementação simples e eficiente em hardware
- Melhor desempenho
- Energeticamente eficiente
- Menor custo

↳ Classes de Instruções:

- Processamento: Aritméticas e lógicas
- Transferência de informação: Cópia registos ↔ Registos ou Registos ↔ Memória
- Controle de fluxo de execução: Alteração da sequência de execução (estruturas condicionais, ciclos, chamadas a funções, ...)

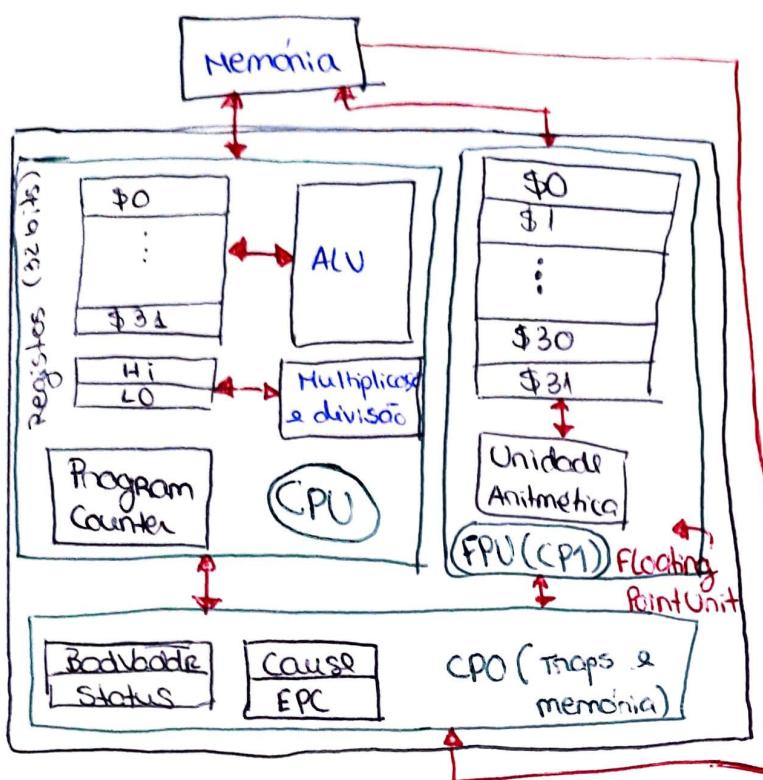
↳ Instruções e implementação de hardware:

- No projeto de um processador, a definição do instruction set exige um delicado compromisso entre múltiplos aspectos, nomeadamente:
 - as facilidades oferecidas aos programadores
 - a complexidade do hardware envolvida na sua implementação
- Princípios básicos ligados a um bom design ao nível do hardware:
 - A regularidade favorece a simplicidade
 - Quanto mais pequeno mais rápido (nº de portas lógicas)
 - O que é mais comum deve ser mais rápido
 - Um bom design implica compromissos adequados.

Nota:

- A nível de codificação de instruções, nº de registos internos, localização dos operandos das instruções, entre outros aspectos, a arquitetura de um processador pode variar muito (dai a grande variedade no mercado). Assim, o foco agora é o microprocessador **MIPS**, por ser muito funcional e simples (apesar de rudimentar e da pouca aplicação em produtos atuais).

↳ Aspetos-chave da arquitetura do MIPS



- 32 registos de uso geral, de 32 bits cada (word de 32 bits)
- ISA baseado em instruções de dimensão fixa (32 bits)
- Arquitetura Load-store (Register-Register). Isto significa que só há 2 instruções que interferem com a memória: Load e Store.
- Memória armazenada em bytes (byte addressable)
- Espaço de endereçamento de 32 bits (2^{32} endereços), 4GB de memória
- Armazenamento de dados externo de 32 bits.

• Instruções aritméticas

SOMA: add a, b, c $\rightarrow a = b + c$

SUBTRAÇÃO: sub a, b, c $\rightarrow a = b - c$

↳ Codificação de instruções no MIPS - formato R

• É um dos 3 formatos de codificação de instrução no MIPS

• Campos da instrução:

- op: opcode (é sempre zero nas instruções do tipo R)
- RS: endereço do registo do 1º operando
- RT: endereço do registo do 2º operando
- Rd: endereço do Registo onde o resultado é armazenado
- shamt: shift amount (utilizado apenas nas operações de deslocamento)
- funct: código da operação a realizar.

31

op	rs	rt	rd	shamt	func
6bits	5bits	5bits	5bits	5bits	6bits

em R é sempre zero

• Operações lógicas no MIPS (bitwise)

- and Rd, RS, RT
- or Rd, RS, RT
- nor Rd, RS, RT
- xor Rd, RS, RT

$$\begin{aligned} \text{Rd} &= \text{RS} \& \text{RT} \\ \text{Rd} &= \text{RS} \mid \text{RT} \\ \text{Rd} &= \sim(\text{RS} \mid \text{RT}) \\ \text{Rd} &= \text{RS} \wedge \text{RT} \end{aligned}$$

Note: negação bitwise:

NOR com o Registro ZERO

a	b	a nor b
0	0	1
0	1	0
1	0	0
1	1	0

• Operações de deslocamento

- << shift left
- >> shift right (lógico ou aritmético)

• Instruções de deslocamento

- SLL (shift left logical)
- SRL (shift right logical)
- SRA (shift right arithmetic)

Também são codificadas com o formato R

Exemplo:

SLL Rd, RT, Shamt

o resultado fica zero
o shamt é preenchido
com o tamanho do deslocamento

• Transferência entre registros internos:

- OR Rd, RS, \$0 \rightarrow Rd = RS

↳ Existe uma instrução virtual que melhora a legibilidade dos programas: move (as instruções virtuais decomponem-se em instruções nativas).

move Rd, RS \rightarrow Rd = RS

↳ Instruções de controle de fluxo de execução

- Necessidade de tomar decisões dentro dos programas

- No MIPS, as instruções básicas são:

- beq Rs1, Rs2, Label \rightarrow branch if equal
- bne Rs1, Rs2, Label \rightarrow branch if not equal

↳ São conhecidas como "branches" (saltos) condicionais

nome que identifica
o local do programa
para onde o salto deve
ser realizado

endereço-alvo / branch target
address (bita)

No caso do beq:

- Se o conteúdo dos registros Rs1 e Rs2 forem iguais, é realizado um salto e a execução continua na instrução situada no endereço representado por "Label".

↳ Branch taken

- Caso contrário, a execução continua na instrução seguinte

↳ Branch not taken

Na BNE, a situação ocorre igualmente

Nota: O salto pode ser feito para a frente ou para trás da instrução de branch.

- No PC, o valor é substituído pelo endereço-alvo

Outras instruções de branch:

- bltz RS, Label → Branch if RS < 0 (less than zero)
- blez RS, Label → Branch if RS ≤ 0 (less or equal than zero)
- bgtz RS, Label → Branch if RS > 0 (greater than zero)
- bgez RS, Label → Branch if RS ≥ 0 (greater or equal than zero)

Nota: Nestas instruções, o registo \$0 está implícito para a comparação.

- SLT rd, rs1, rs2 → set if less than : set rd if RS1 < RS2

↳ o registo rd toma o valor 1
Se RS1 < RS2, caso contrário,
toma o valor zero

set = por a 1
reset = por a 0

- SLT_i rd, rs1, imm → set if less than immediate
constant set rd if rs1 < imm

→ A utilização destas instruções em conjunto com o registo \$0 permite a implementação de todas as condições de comparação entre 2 registo, ou entre 1 registo e 1 constante:

$$A = B \quad | \quad A \neq B \quad | \quad \begin{array}{c} A > B \quad / \quad A \geq B \quad / \quad A < B \quad / \quad A \leq B \\ \downarrow \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \end{array}$$

BEQ BNE SLT & SLTI

Instruções virtuais de branch do MIPS

- São decompostos pelo assembler em instruções nativas

- blt RS1, RS2, Label → Branch if RS1 < RS2 less than
 - ble RS1, RS2, Label → " " RS1 ≤ RS2 less/equal than
 - bgt RS1, RS2, Label → " " RS1 > RS2 greater than
 - bge RS1, RS2, Label → " " RS1 ≥ RS2 greater/equal than
- este valor pode ser um registo ou uma constante

↳ Instrução de salto incondicional

- Não existe teste de qualquer condição: o salto é sempre realizado.
- Instrução "jump".

j label → O fluxo de execução é desviado, de forma incondicional, para "label"

↳ Introdução C → Assembly

- If()... then ... else

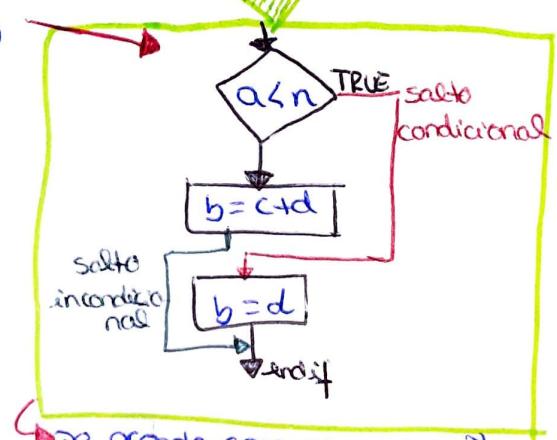
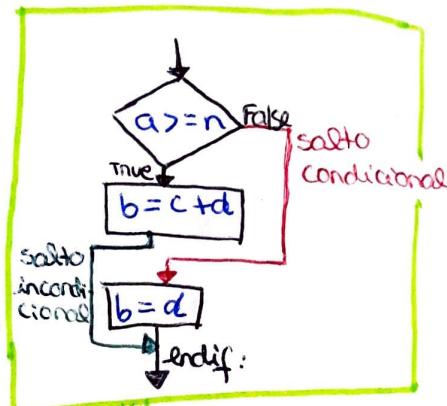
Exemplo:

```
if (a >= n){  
    b = c + d; }  
else {  
    b = d; }
```

→ Ocorrência de um salto condicional e um salto incondicional

→ Fluxograma equivalente:

O salto condicional é adaptado (complemento lógico) para que este ocorra quando a condição for verdadeira (tal como nos branches)



→ De acordo com as instruções de branch

- Ciclos for() e while()

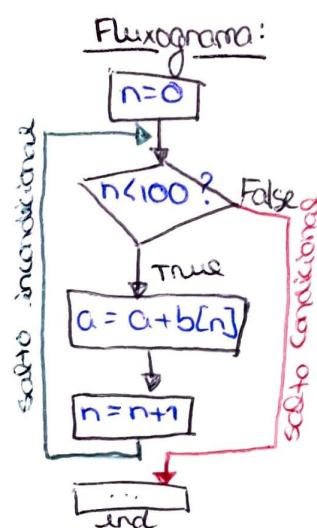
- Um ciclo for() é equivalente a um while()

```
int n;  
for (n=0; n < 100; n++)  
    a = a + b[n];
```

```
n=0;  
while(n < 100){  
    a = a + b[n];  
    n++;}
```

* O salto condicional necessita de ser alterado, de forma a ser efectuado quando a condição for verdadeira.

Complemento lógico: $n \geq 100$



- Círculo do...while()

- O corpo do círculo do...while() é executado incondicionalmente pelo menos uma vez.

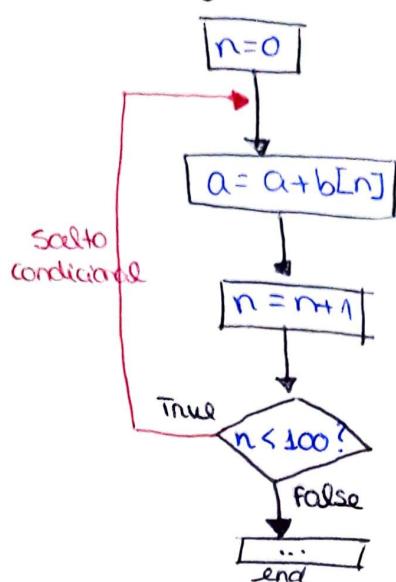
Exemplo: $n = 0;$

do {

$a = a + b[n];$

$n++; } \text{ while}(n < 100);$

Fluxograma:



↳ Armazenamento de informação

• A arquitetura do MIPS é do tipo load-store, ou seja, não é possível operar diretamente sobre o conteúdo da memória externa.

↳ Terão que existir instruções para transferir informação entre os registos da CPU e a memória externa.

• Modo de endereçamento - método usado pela arquitetura para aceder ao elemento que contém a informação que irá ser processada por uma dada instrução.

• Nas instruções aritméticas e lógicas (formato R), ambos os operandos residem em registos internos.

↳ especificados diretamente nos campos RS e RT da própria instrução.

↳ endereçamento tipo Registo

• Acesso à memória externa: implica especificar o endereço da posição que se quer ler/escravar.

32 bits no MIPS

→ cada instrução do MIPS ocupa 32 bits, como é que é possível codificar o endereço, sendo que este também tem 32 bits?

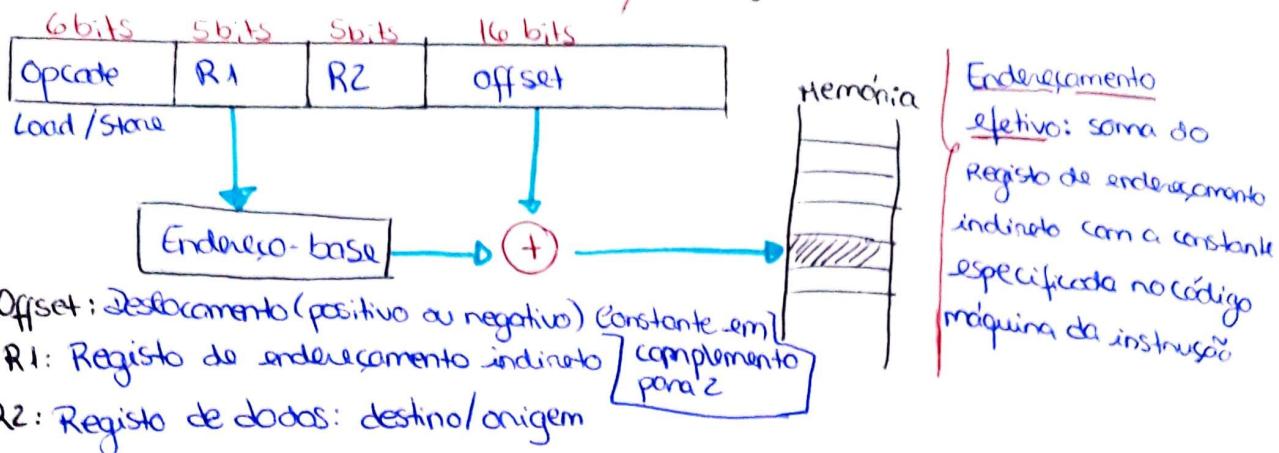


Em vez do endereço, a instrução indica um registo que contém o endereço de memória a aceder.

↳ Endereçamento indireto por Registo

(é necessário inicializar o registo com o valor do endereço)

Código máquina das instruções:



Ler da memória - LW (Load word)

- Cópia da memória uma palavra de 32 bits para um registo interno do CPU.

Formato:

LW \$Reg-dest, const (\$Reg-endereço),
 destino no CPU constante de 16 bits ($\pm 32k$) Reg. endereçamento indireto

Escrever na memória - SW (Store word)

- Transfere uma palavra de 32 bits de um registo para a memória (1 word ocupa 4 posições de memória).

Formato:

SW \$R-origem, const (\$Reg-endereço),
 Registo de origem do CPU constante de 16 bits ($\pm 32k$) Reg. endereçamento indireto

Nota: O acesso a words só é possível em endereços múltiplos de 4, uma vez que cada word ocupa 4 posições de memória.

- Se o endereço for inválido, gera uma exceção e a instrução é ignorada.
- Um número múltiplo de 4 em binário tem os 2 bits menos significativos a zero.
- Generalizando, um n-múltiplo de 2^k tem os k bits menos significativos a zero.

• Leitura de memória - LBU (Load byte unsigned)

- Transfere um byte da memória para um registo interno.
- Os 24 bits mais significativos do registo de destino são colocados a 0.

Formato: lbu \$reg-dest, const (\$reg-endr)

Registo destino do CPU
Constante de deslocamento (16 bits)
Registo de endereçamento indireto

• Leitura de memória - LB (Load byte)

- Transfere um byte da memória para um registo interno
- Faz extensão de sinal do valor lido de 8 para 32 bits (complemento p/2)

Formato: lb \$reg-dest, const (\$reg-endr)

Registo destino
Constante de deslocamento (16 bits)
Registo de endereçamento indireto

• Escrita de 4 bytes - SB (Store byte)

- Transfere um byte de um registo interno para a memória
- Só são usados os 8 bits menos significativos

Formato: sb \$reg-orig, const (\$reg-endr)

Notas: - Se as operações envolverem 32 bits de dados, então o MIPS tem de fazer um alinhamento da memória. Isto significa que os endereços são agrupados de 4 em 4. Podemos assim "excluir" (ignorar) os 2 últimos bits do endereço, visto que estes vão sempre ser 00 (múltiplos de 4). Assim, o endereço inicial do conjunto (array) é que indica o endereço da word

0x00000020	word: 0x20
0x00000021	
0x00000022	word: 0x24
0x00000023	
0x00000024	word: 0x24
0x00000025	
0x00000026	word: 0x24
0x00000027	

↳ O acesso a words só é possível em endereços múltiplos de 4.

• Restrição de alinhamento

- Se as operações forem de 1 byte (LBU, LB e SB), o problema do alinhamento não se coloca
- O MIPS gera o endereço múltiplo de 4 (EN4) que, no acesso a uma word de 32 bits inclui o endereço pretendido

- No caso de leitura (lbe/lbe), é realizada uma leitura do EN4 e retira os 8 bits dessa posição, correspondentes ao endereço pretendido.

- No caso de escrita (sb):
 - É feita uma leitura de 1 word do EN4
 - De entre os 32 bits lidos, substitui os 8 bits que correspondem ao endereço pretendido
 - Escreve a word em EN4

Read - Modify - Write

L 4 Organização das words de 32 bits na memória

- 2 opções:

Exemplo: 0x01 23 8f A5

Address	Data
0x1001000C	0x01
0x1001000D	0x23
0x1001000E	0x8f
0x1001000F	0xA5

Big Endian

OU

Address	Data
0x1001000C	0xA5
0x1001000D	0x8f
0x1001000E	0x23
0x1001000F	0x01

Little - Endian

Para o programador, em geral, é irrelevant

- O MARS implementa o Little - Endian

L 5 Diretivas do Assembly

- Comandos especiais colocados num programa em Assembly, destinados a instruir o assembly a executar uma determinada tarefa ou função.
- Não são instruções, logo não geram código máquina, nem fazem parte do ISA
- Podem ser usadas para diversas finalidades:
 - Reservar e inicializar espaço em memória para variáveis
 - Controlar os endereços reservados para variáveis em memória (endereços M4, por exemplo)
 - especificar os endereços de colocação de código e dados na memória.
 - definir valores simbólicos
- As diretivas são específicas de um dado assembly.

- Diretivas do Assembler do NARS

- .ascii z str → Reserva espaço e armazena o string str em sucessivas posições de memória, acrescentando o terminador '\0' (null)

Ex:

msg1: .ascii "AC-1"
↓
Label

- .space n → Reserva n posições de endereços de memória, sem inicialização

Ex: array: .space 20

- .byte b₁, b₂, ..., b_n → Reserva espaço em memória e armazena os bytes b₁, b₂, ..., b_n em sucessivas posições

Ex: array: .byte 0x41, 0x43, 0x31, 0x00

- .word w₁, w₂, ..., w_n → Reserva espaço e armazena as words w₁, w₂, ..., w_n em sucessivas posições de memória (cada word em 4 endereços consecutivos)

Ex: array: .word 0x012387A5, 0xF34, 0x678AC

- .align n → alinha o próximo item num endereço múltiplo de 2ⁿ.

Ex: .align 2 (próximo item está alinhado num endereço múltiplo de 4)

- .equ symb, val → atribui a um símbolo um valor. No programa, o assembler substitui ocorrências de symb por val. Permite melhorar a legibilidade do programa, uma vez que o nome da constante simbólica diz algo sobre o seu significado.

Ex: .equ print_int10, 1 (na utilização de system calls)

- .data → indica ao assembler que a zona do programa que vem a seguir é uma zona de dados (seção de dados)

(0x16050000) é o endereço desta seção

- .text → indica ao assembler que a zona do programa seguinte é uma zona de código (seção de código) (0x00400000)

Nota: Durante a interpretação e execução das ações associadas a cada uma das diretivas, o assembler vai constituindo uma tabela com 2 colunas, uma com o nome do Label que encontra, outra com o endereço que atribuiu esse Label. → Tabela de símbolos

↳ Assim, sempre que encontram uma referência a esse Label, substitui pelo endereço da tabela.

L_D Ponteiros em C

- Um ponteiro é uma variável que contém o endereço de outra variável.
- Assim, o acesso à segunda variável pode ser feito indiretamente através do ponteiro.

C_D Pode-se fazer uma analogia com o endereçamento indireto por registo do MIPS.

- Em C, var é uma variável, &var é o endereço da variável.

$$px = \&var \rightarrow px \text{ é um ponteiro que aponta para var}$$

operador de indireção

- O operador \star trata o operando como um endereço.
- Permite o acesso para obter ou alterar o conteúdo.

Ex: $a = *px \rightarrow$ Atribui o conteúdo da variável apontada por px a a

Aritmética de ponteiros:

- Se pa é um ponteiro:
 - $*pa = *pa + 1 \rightarrow$ altera a variável apontada por pa.
 - $pa = pa + i \rightarrow$ incrementa pa de modo a apontar para $(pa + i)$ o elemento seguinte.

Nota: Em Assembly, a tradução da expressão anteriores leva em conta o tipo de variável:

Ex: - Um inteiro que seja definido com 4 bytes(32 bits): pa implica adicionar 4 ao valor inicial.

L_D Acesso sequencial a elementos de um array

① Acesso indexado:

Endereçamento a partir do nome do array e de um índice que indica o elemento a que se pretende aceder.

$$v = a[i]$$

② Utilização de um ponteiro:

Identifica em cada instante o endereço do elemento a que se pretende aceder

$$v = *p \rightarrow p = \&a[i]$$

Implementações distintas
em Assembly

Em assembly:

① Acesso indexado:

$$V = a[i]$$

Para aceder ao elemento i do array a , o programa começa por calcular o respetivo endereço, a partir do endereço inicial do array

$$\text{endereço} = \text{endereço inicial} + i \times \text{size}$$

$\approx a[0]$

i indice

em bytes,
de cada posição
do array

② Acesso por ponteiro

$$V = *p$$

O endereço do elemento a aceder está armazenado num registo

$$\text{endereço do elemento seguinte} = \text{endereço atual} + \frac{\text{size}}{\text{dimensão de cada posição do array (em bytes)}}$$

↳ FORMAÇÕES DE CODIFICAÇÃO DO MIPS

→ Instruções aritméticas e lógicas: Formato R

Op	RS	RT	Rd	shamt	funct
6b	5b	5b	5b	5b	6b

31

→ Instruções de salto: Formato I

Op	RS	RT	offset
6b	5b	5b	16 bits

↳ Constante

Ex: beq, RS, RT, offset

OFFSET → O campo offset pode ser usado para codificar a diferença entre o valor do endereço-alvo e o endereço onde está armazenada a instrução de branch ↳ "label"

↳ É interpretado como um valor em complemento para dois, permitindo saltos para endereços anteriores (offset negativo) ou posteriores (offset positivo)

↳ Durante a execução da instrução de branch, o seu endereço está disponível no registo PC, pelo que o processador pode calcular o endereço-alvo como: $\boxed{\text{Endereço-alvo} = PC + \text{offset}}$

↓ Endereçamento Relativo

NO MIPS...

• A referência para o cálculo do offset é o endereço da instrução seguinte, → porque o PC incrementa após o fetch.

• As instruções estão armazenadas em endereços múltiplos de 4 (cada instrução é uma word (32 bits), pelo que o offset é também um valor múltiplo de 4 (2 bits menos significativos são 0). → Os 2 bits menos significativos não são representados na instrução)

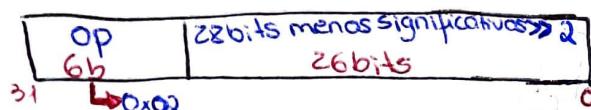
Assim: O campo offset do código máquina da instrução de branch é então usado para codificar a diferença entre o valor do endereço-alvo e o valor do endereço seguinte ao da instrução de branch, dividida por 4 (2 shifts à direita, para ignorar os 2 bits menos significativos):

$$\text{Endereço-alvo} = \underbrace{\text{PC-atual}}_{\text{a seguir ao branch}} + (\text{offset} * 4) \quad \ll 2$$

$$\text{Offset} = \frac{\text{Endereço-alvo} - \text{PC-atual}}{4}$$

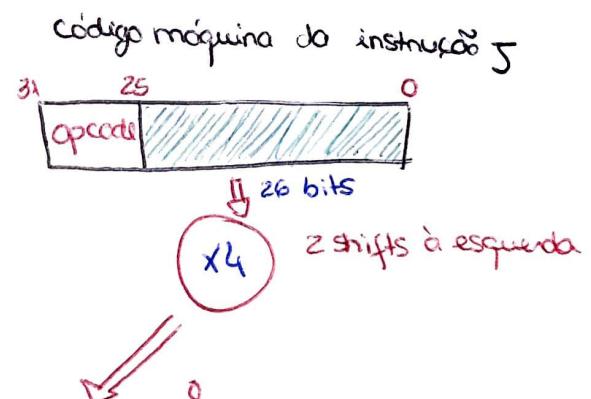
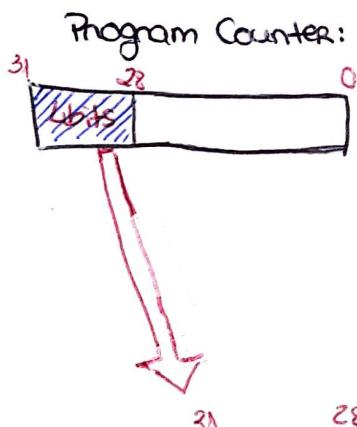
Lp Instrução de Salto Incondicional

- Endereçamento pseudo-direto → o código máquina da instrução codifica diretamente ponto do endereço-alvo
- Formato J:



- O endereço-alvo da instrução J é sempre múltiplo de 4 (2 bits menos significativos são sempre 0)
- 28 bits menos significativos → 26 de forma explicita
2 de forma implícita

- Cálculo do endereço-alvo de uma instrução J:

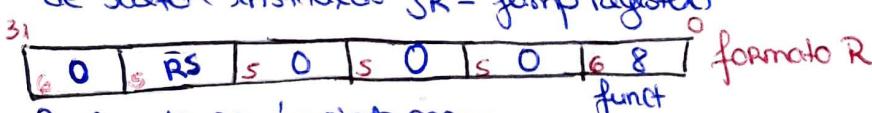


Endereço-alvo:
(32 bits)



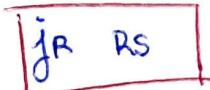
- Salto incondicional - Endereçamento indireto por registo:

- Um registo interno (de 32 bits) armazena o endereço-alvo da instrução de salto (instrução JR - jump register)



- O valor de RS é copiado para o

Program Counter, de modo a conter o endereço da instrução desejada



↳ Manipulação de constantes

- Uma constante é um valor determinado com antecedência (quando o programa é escrito) e que não pretende que seja ou possa ser alterada durante a execução do programa.
- As constantes podem ser armazenadas na memória externa. Mas como a arquitetura do MIPS é do tipo "Load-Store", a sua utilização implicaria o recurso a 2 instruções:

↳ Leitura da constante e cópia para um registo interno

↳ Operação com essa constante

- Para aumentar a eficiência, as arquiteturas disponibilizam um conjunto de instruções em que as constantes se encontram armazenadas na própria instrução.

↳ A constante faz parte do código máquina

- Assim, diz-se que o endereçamento é imediato, uma vez que não há necessidade de recorrer a uma operação de leitura prévia

↳ As instruções aritméticas / lógicas deste tipo são identificadas pelo sufixo ":" :

addi
andi
ori
slti

} Formato I → Temos 16 bits para codificar a constante
Generalmente é suficiente para armazenar as constantes mais usadas

Qual é a gama de representação? (valores pequenos)

↳ No caso mais geral, a constante representa uma quantidade inteira codificada em complemento para 2:

$$\rightarrow [-32768, +32767] \text{ } \begin{matrix} +2^{15} \\ -2^{15} \end{matrix}$$

(a constante é estendida para 32 bits preservando o sinal)

↳ Nas instruções lógicas, a constante deve ser entendida como uma quantidade inteira sem sinal:

→ andi, ori, etc

$$\rightarrow [0, 65535] \text{ } +2^{16}-1$$

(a constante é estendida para 32 bits, sendo os 16 mais significativos 0x0000)

Codificação de instruções que usam constantes

31

opcode	5	RS	5	RT	16	immediate (constante)	0
--------	---	----	---	----	----	-----------------------	---

(Tipo I)

Em Assembly:

Ex: addi RT, RS, immediate
op

↳ Manipulação de constantes de 32 bits - LUI (Load Upper Immediate)

- Esta instrução é usada quando é necessário 32 bits para a constante.

- Formato I: lui \$Reg, immediate

↳ Coloca a constante "immediate" nos 16 bits mais significativos do registo de destino
↳ Os 16 bits menos significativos ficam a zero

Codificação:

- lui RT, immediate →

31	001111	0000	5	RT	16	immediate	0
	OP	(RS)					

Exemplo:

Como inicializar o registo \$6 com o valor 0xF32864D9 (32 bits):

① "Partir" a constante em duas:

0x F328



mais significativo

0x 64D9



menos significativo

② Copiar para \$6:

- lui \$6, 0xF328

- ori \$6, 0x64D9

↳ Manipulação de constantes de 32 bits - LA/LI

- Instrução virtual "load address" e "load immediate":

- la RT, imm → é decomposta em lui + ori

→ Pode (e costuma ser)

um label de segmento
de dados

- li RT, imm → é decomposta em lui + ori

Nota: O registo RT do lui é o \$1 (\$at) por definição do MIPS, o permite este tipo de decomposição.

↳ Sub-Rotinas

- Blocos de instruções que realizam uma determinada tarefa / funcionalidade
- É o equivalente ao conceito de "função" numa linguagem de alto nível
- Pode se dizer apenas rotina
- Pode ser chamada em qualquer ponto do programa
 - ↳ Chamar a sub-rotina;
 - ↳ Tem de haver um mecanismo que permite regressar à instrução imediatamente à seguir à instrução que fez a chamada.

→ Razões que justificam a existência de funções / subrotinas:

- Reutilização de código de uma função no contexto de um determinado programa

↳ Aumento da eficiência na dimensão do código
↳ Substituição por um único trecho evocável.

- Reutilização no contexto de um conjunto de programas

↳ O mesmo código pode ser Reproveitado (bibliotecas)

- Organização e estruturação do código

• Chamar uma sub-rotina significa desviar o fluxo de execução do programa para o endereço onde está armazenada a 1^ª instrução desta.

• A última instrução de uma sub-rotina devia novamente o fluxo de execução para o endereço a seguir à instrução chamadora.

↳ Retorno

• Uma sub-rotina pode chamar outra sub-rotina.

Mecanismo de chamada:

- A instrução que chama a sub-rotina tem que guardar o endereço da instrução para onde deve ser feito o retorno
- No MIPS, a ligação (link) entre o chamador e o chamado (sub-rotina) é feita pela instrução **JAL** (jump and link)
 - ↳ É uma instrução de salto incondicional, que armazena o valor atual do PC no registo **RA** (return address)

Mecanismo de retorno:

- O valor do PC é incrementado durante a fase fetch das instruções, por isso na fase de execução, o PC já tem o endereço da instrução seguinte à instrução jal (Endereço guardado \$RA)
- Por isso é necessário apenas fazer um salto incondicional no final da sub-rotina, para o endereço localizado em \$RA ($JR \RA)
- A instrução jal é codificada do mesmo modo que a instrução j.

Instrução jump and link:

jal target-address

A codificação do endereço alvo é igual à feita nas instruções j.

PC4msb | (28LsbAdd) <52

Na fase fetch:

IR = MEM[PC] (Instruction register)

PC = PC + 4 (valor é incrementado)

Na fase execute:

\$RA = PC (já incrementado) → O Registo \$RA (\$31) armazena o

PC = target-address endereço de retorno

Após a sub-rotina ser executada:

- Aproveita-se o endereço de retorno armazenado em \$RA durante a execução da instrução jal, para usar o salto incondicional JR \$RA
- Execução da instrução jr:

No fase fetch:

IR = MEM[PC] (Instruction Register)

PC = PC + 4 (incremento do PC)

Fase execute:

PC = \$RA (neste caso de $JR \$RA$)

A próxima instrução é a que se encontra no endereço \$RA

Nota:

- O registo \$RA não deve ser usado como registo de armazenamento. (\$31)

↳ Chamada de uma sub-rotina a partir de outra sub-rotina

- Neste caso, o valor do registo \$Ra é alterado (pela instrução jal), pendendo-se a ligação com o primeiro chamador.

• A solução encontrada é salvaguardar o valor inicial de \$Ra, no ^{em memória} início da subrotina, e reponer mesmo ^{antes} do retorno.

↳ Instrução jump and link Register

- Especificação de um endereço-alvo de 32 bits.
- Funciona de modo idêntico à instrução "jal", exceto na obtenção do endereço-alvo
 - O endereço da subrotina é lido do registo especificado na instrução (endereçamento indireto por registo), como na instrução jr
- Codificada com o formato R

↳ Aspetos conceptuais sobre os sub-rotinos:

- A reutilização de rotinas é essencial em programação, em especial quando suportam funcionalidades básicas, quer do ponto de vista computacional como do ponto de vista da interface entre o computador, os periféricos e o utilizador humano.
- As subrotinas surgem frequentemente agrupadas em bibliotecas, a partir das quais podem ser evocadas por qual programa.
Ex: printf(...) → biblioteca stdlib (em C)
- Utilização de subrotinas escritas por outros para serviço dos nossos programas, não deverá implicar o conhecimento dos detalhes da sua implementação.
- Geralmente, o acesso ao código-fonte da sub-rotina (conjunto) de instruções escritas por um programador não é sequer possível, o menos que o autor tenha público.

↳ Na perspectiva do programador, o objetivo é escrever um trecho de código isolado, com uma funcionalidade bem definida, e com uma interface que ele próprio pode determinar em função das necessidades.

- Uma sub-rotina pode ser reutilizada, logo o programador não conhece as características do programa que a aí chaman.

↳ Torna-se necessário definir um conjunto de regras que regulem a relação entre o programa "chamador" e a sub-rotina "chamada".

- definição da interface entre ambos:

- Quais os parâmetros de entrada? Como os passar?
- Como devolver os resultados ao programa chamador?

- princípios que assegurem uma "só convivência" entre os dois, de modo a que um não destrua os dados do outro.

→ Que registos da CPU podem usar o "chamador" e o sub-rotina usá-los, sem que haja indevida alteração de informação entre os dois.

→ Como partilhar a memória usada para armazenar dados, Sem risco de sobreposição (e consequente perda de informação)

↳ Convenções do MIPS

① Passagem e devolução de valores:

• Os parâmetros que podem ser armazenados na dimensão de um Registo (32 bits, i.e. char, int; ponteiros) devem ser passados à Sub-Rotina nos registos \$00 a \$03 (\$4 a \$7), por esta ordem.

↳ Caso o número de parâmetros seja superior a 4, os 4 primeiros são passados nos registos \$00; e os restantes devem ser passados na Stack (pilha).

• A sub-rotina pode devolver um valor de 32 ou 64 bits:

- 32 bits: é usado o registo \$V0

- 64 bits: os 32 bits mais significativos são armazenados no \$V1 e os 32 bits menos significativos no registo \$V0

② Utilização e salvaguarda de Registos

• Atribuir a uma das entidades a responsabilidade de copiar previamente para a memória externa o conteúdo de qualquer registo que pretenda usar (salvaguardar o registo) e nepon, posteriormente, o valor original lá armazenado.

↳ Estratégia "caller-saved": Deixa-se ao cuidado do programa "chamador" a responsabilidade de salvaguardar o conteúdo da totalidade dos registos, antes de chamar a sub-rotina.

- Cabe-lhe também a tarefa de nepon posteriormente o valor

- É admissível que se guarde apenas os registos de cujo conteúdo venga a precisar mais tarde.

↳ Estratégia "callee-saved": Entrega-se à sub-rotina a responsabilidade de salvaguardar os registos que vai usar.

- Assegura igualmente a tarefa de repor o seu valor imediatamente antes de regressar ao programa "chamador".
- ⊕ Funciona para todos os níveis de sub-rotinas

• Salvaguarda de registos no Mips:

- Os registos \$t0 a \$t9, \$v0 a \$v1, e \$a0 a \$a3 podem ser usados livremente e alterados pelas sub-rotinas

→ Para estes registos, aplica-se a regra "callee-saved", cabe ao programa principal salvaguardar os Registos.

- Na perspetiva do chamador, os registos \$s0 a \$s7 não podem ser alterados pelas Sub-Rotinas.

→ Se uma dada Sub-Rotina precisar de usar qualquer um dos Registos, compite a essa sub-rotina salvaguardar previamente o seu conteúdo, repondo-o imediatamente antes de terminar. "callee-saved"

↳ É seguro para o programa chamador usar um registo \$sn, uma vez que tem a garantia que estes não vão ser alterados.

Considerações práticas sobre a utilização da convenção

→ Sub-Rotinas terminais (Sub-Rotinas folha, i.e., que não chamam qualquer sub-rotina):

- Só devem usar registos que não têm a responsabilidade de salvaguardar (\$t0 a \$t9, \$v0, \$v1, \$a0 a \$a3)

→ Sub-Rotinas intermédias (que chamam outras sub-rotinas):

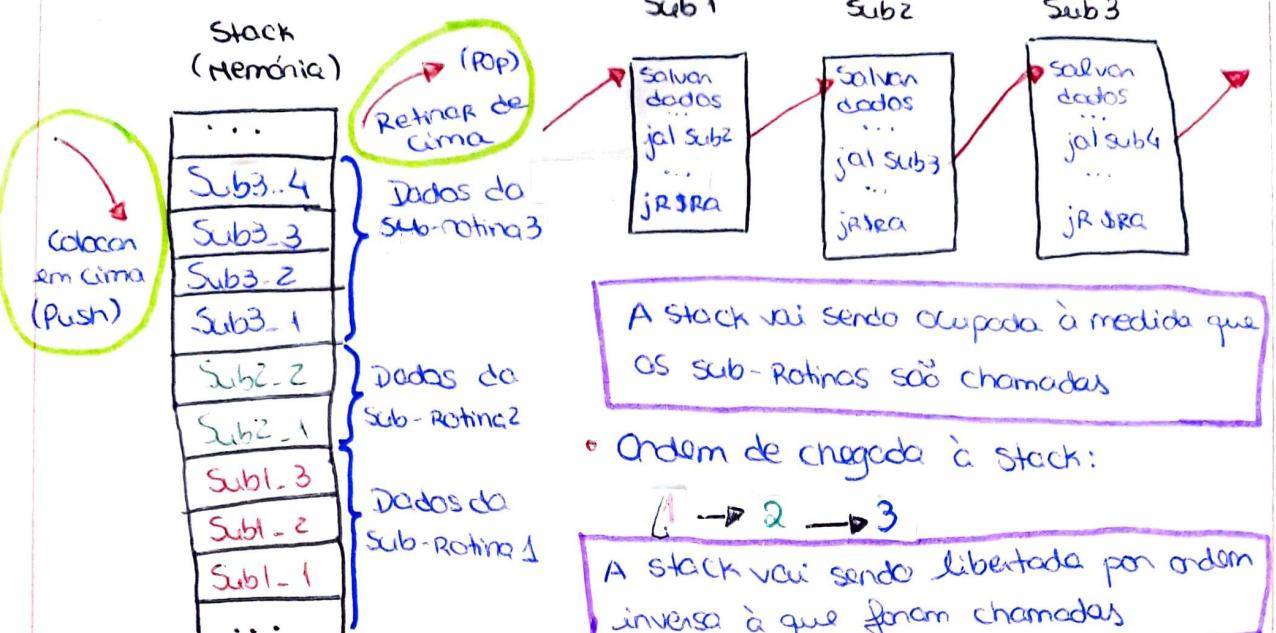
- Devem usar os registos \$s0... \$s7 para o armazenamento de valores que pretendem preservar durante a chamada à sub-rotina
↳ Implica a sua prévia salvaguarda e reposição no final.
- Devem usar os registos \$t0 a \$t9, \$v0, \$v1, \$a0 a \$a3 para os restantes valores

↳ Armazenamento temporário de memória - Stack (pilha)

- Estrutura de dado LIFO (Last In, First Out)

- A ordem de retorno é contrária à ordem de chamada





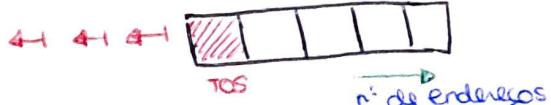
Nota: As stacks são estruturas de dados muito importante. A arquitetura Intel x86 suporta no seu ISA instruções específicas para a sua manipulação.

Operações push & pop : stack

- Estas operações têm associado um registo designado por **stack pointer (sp)**
 - As operações usam este registo para saber de onde retirar ou colocar informação.
- O registo SP mantém, de forma permanente, o endereço do topo da stack (tos - top of stack) e aponta sempre para o último endereço ocupado.
 - Numa operação de **push** é necessário pré-atualizar o stack pointer antes de copiar informação para a stack.
 - Numa operação de **pop** é feita uma leitura da stack, do endereço atual do SP, seguida de uma atualização do valor do SP.
- A atualização do SP, numa operação de push, pode seguir uma de 2 estratégias:
 - Ser incrementado, fazendo crescer num sentido crescente dos endereços.

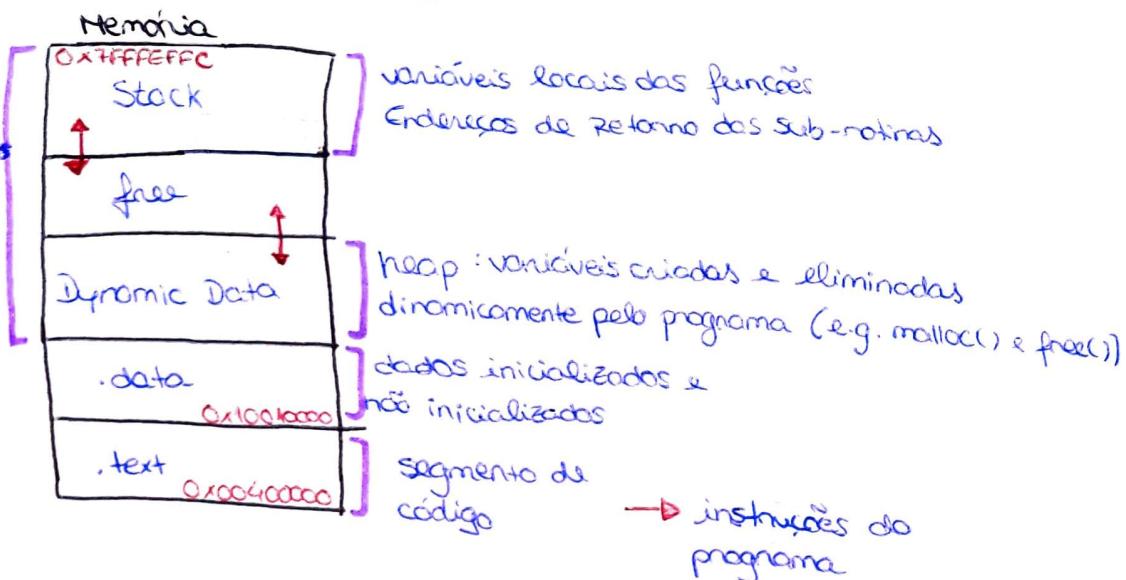


- Ser decrementado, fazendo crescer no sentido decrescente dos endereços.



- A estratégia de crescimento da stack no sentido dos endereços mais baixos é, geralmente, a preferida:

- Permite uma gestão simplificada da fronteira entre os segmentos de dados e stack.



Regras da utilização da stack na arquitetura MIPS:

- O registo \$Sp = \$29 contém o endereço da última posição ocupada da stack
- A stack cresce no sentido decrecente dos endereços de memória
- A stack está organizada em words de 32 bits
- A gestão de espaço na stack pode ser feita somando constantes ao registo \$Sp

Ex: `addiu $sp, $sp, -16` → Reserva espaço para 4 words na stack

`sw $ra, 0($sp)` → Copia o valor dos registos para o stack (push)

`sw $s0, 4($sp)`

`sw $s1, 8($sp)`

`sw $s2, 12($sp)`

(...)

→ Proólogo

`lw $ra, 0($sp)` → Repõe o valor dos registos

`lw $s0, 4($sp)`

`lw $s1, 8($sp)`

`lw $s2, 12($sp)`

`addiu $sp, $sp, 16` → Liberta espaço na stack

→ Epílogo

↳ Representação de inteiros

- Um computador é um sistema digital binário, logo a representação de inteiros faz-se sempre em base 2 (usando 0 e 1)
 - Tipicamente, um inteiro pode ocupar um número de bits igual à dimensão de um registo interno da CPU.
 - Assim, a gama de valores representáveis é finita, e corresponde ao número máximo de combinações que é possível obter com o número de bits de um registo interno.
 - No MIPS, um inteiro ocupa 32 bits, pelo que o número de inteiros representáveis é: $N_{int} = 2^{32} = \underline{4.294.967.296_{10}}$
 - Os circuitos que realizam operações aritméticas estão igualmente limitados a um número finito de bits, geralmente igual à dimensão dos registos internos do CPU.
 - ↳ Os circuitos aritméticos operam em aritmética modular, ou seja em mod(2ⁿ) em que n é o nº de bits da representação.
 - O maior valor que um resultado aritmético pode tomar será portanto $2^n - 1$, sendo o valor interno imediatamente a seguir o valor zero (representação círcula).

Numa ALU:

De 8 bits,

$$11001011 + 00110111 = \begin{array}{|c|c|} \hline & 1 \\ \hline \end{array} 00000010 \rightarrow \text{result has 8 bits}$$

↓
carry / transporter

Se os operando forem do tipo unsigned, o bit carry sinaliza que o resultado não cabe num registo de 8 bits, ou seja, há

overflow é um signed (em complemento para 2),

Se os operadores forem do tipo signed, o bit de carry, por si só, não tem qualquer significado, e não faz parte do resultado.

LD Representação em complemento para dois

- É o método usado em sistemas computacionais para a codificação de quantidades inteiros com sinal (signed)

Definição: Se K é um número positivo, então K^* é o seu complemento para 2 (complemento verdadeiro) e é dado por:

$$K^* = 2^n - K$$

$n \rightarrow$ n.º de bits da representação

Ex: Representação de -5 , com 4 bits:

$$\rightarrow N = 5_{10} = 0101_2$$

$$\rightarrow 2^n = 2^4 = 10000$$

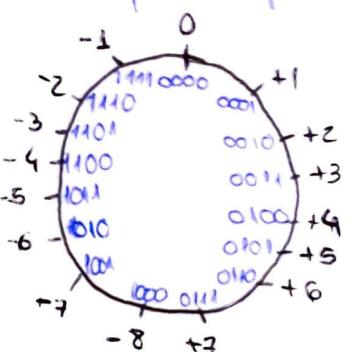
$$\rightarrow N^* = -5_{10} = 2^n - N = 10000 - 0101 = 1011$$

Método prático: inverter todos os bits do valor original e somar 1:

$$0101 \rightarrow 1010 \rightarrow 1011$$

→ Este método é reversível.

- O bit mais significativo pode ser interpretado como sinal
 - ↳ 0 → positivo
 - ↳ 1 → negativoTem peso negativo $\rightarrow -2^n$
- Uma única representação para o zero
- Codificação assimétrica (mais um negativo do que positivos)
- A subtração é realizada através de uma operação de soma com o complemento para 2 do segundo operando: $a - b = a + (-b)$



Representação circular

↳ Overflow em complemento para 2

- Ocorre overflow quando é ultrapassada a gama de representação.
Isto acontece quando:

- ① → Somam-se dois positivos e o resultado é negativo
- ② → Somam-se dois negativos e o resultado é positivo

Analise do bit de carry

1: Quando o bit é 1.

2: Comparação com os bits mais significativos dos operandos

De um modo geral: A situação de overflow ocorre quando o carry-in do bit mais significativo não é igual ao carry-out, ou seja, quando:

$$C_{n-1} \oplus C_n = 1$$

A Lembra:

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

→ Em operandos interpretados em complemento para 2:

- quando $A+B > 2^{n-1}-1$ ou $A+B < -2^{n-1}$

$$\text{↳ OVF} = (C_{n-1} \cdot \overline{C_n}) + (\overline{C_{n-1}} \cdot C_n) = C_{n-1} \oplus C_n$$

- não tendo acesso aos bits intermédios de carry ($R = A+B$):

$$\text{↳ OVF} = R_{n-1} \cdot \overline{A_{n-1} \cdot B_{n-1}} + \overline{R_{n-1}} \cdot \overline{A_{n-1} \oplus B_{n-1}}$$

resultado negativo com
operandos positivos

resultado positivo com operandos
negativos

→ Em operandos interpretados sem sinal:

- (Quando $A+B > 2^n-1$ ou $A-B < 0$, c/ $B > 0$)
- O bit de carry $C_n = 1$ sinaliza a ocorrência de overflow

- O MIPS apenas deteta overflow nas operações de adição e subtração com sinal (ADD, SUB, ADDI) e, quando isso acontece, gera uma exceção. ADDU, SUBU e ADDIU não detetam overflow.

unsigned

L) Multiplicação de inteiros

- Devido ao aumento da complexidade que deu resultado, nem todas as arquiteturas suportam, no nível de hardware, a capacidade para efetuar operações aritméticas de multiplicação e divisão para inteiros.
- Multiplicação de quantidades unsigned: algoritmo clássico que é usado na multiplicação em decimal
- Multiplicação de quantidades signed: em complemento para dois. É usado o algoritmo de Booth
- Uma multiplicação que envolva 2 operandos de N bits consome de um espaço de armazenamento, para o resultado, de 2xN bits.

L) Multiplicação de inteiros no Mips

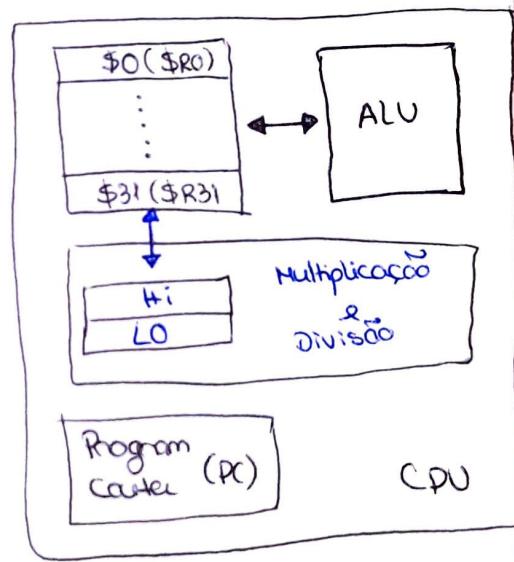
- No Mips, a multiplicação e a divisão são asseguradas por um módulo independente da ALU.

- Os operandos são registros de 32 bits. Na multiplicação, isso implica que o resultado tem de ser armazenado com 64 bits

- Os resultados são armazenados num par de registros especiais designados por Hi e Lo, cada um com 32 bits.

32 HSB

32 Lsb



- A transferência de informação entre os registros Hi e Lo e os restantes registros de uso geral faz-se através das instruções mfhi e mflo.

mfhi Rdst # copia hi para Rdst
mflo Rdst # copia lo para Rdst

move from move from
high low

- A unidade de multiplicação pode operar considerando os operandos signed ou unsigned:

- mult - multiplicação com sinal
- multu - multiplicação sem sinal

- Em Assembly, a multiplicação é efetuada pelas instruções:

→ mult RSRC1, RSRC2

→ multu RSRC1, RSRC2

- O resultado fica armazenado nos registros **Hi e Lo**

⚠ Não esquecer guardar os resultados Hi e Lo nos registros "normais".

- Instruções virtuais de multiplicação:

➊ Multiplicação signed
mul Rdst, RSRC1, RSRC2

- mult RSRC1, RSRC2
- mflo Rdst

➋ Multiplicação unsigned
mulu Rdst, RSRC1, RSRC2

- multu RSRC1, RSRC2
- mflo Rdst

➌ Multiplicação unsigned com detecção de overflow

mulou Rdst, RSRC1, RSRC2

- multu RSRC1, RSRC2
- mfhi \$1
- beq \$1, \$0, cont
- break
- Cont: mflo Rdst

➍ Multiplicação signed com det de overflow

mulo Rdst, RSRC1, RSRC2

- mult RSRC1, RSRC2
- mfhi \$1
- mflo Rdst
- sra Rdst, Rdst, 31
- beq \$1, Rdst, cont
- break
- Cont: mflo Rdst

↳ Divisão de inteiros com sinal

- A divisão de inteiros com sinal faz-se, do ponto de vista algorítmico, em sinal e em módulo.

- Aplicam-se as seguintes regras:

→ Divide-se dividendo pelo divisor, em módulo.

→ O quociente tem sinal negativo se os sinais do dividendo e do divisor forem diferentes.

→ O resto tem o mesmo sinal do dividendo.

$$\text{Ex: } -7 / 3 = -2 \text{ resto} = -1$$

$$7 / -3 = -2 \text{ resto} = 1$$

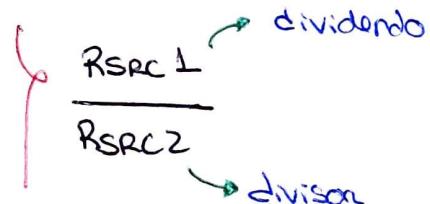
⚠ $\text{Dividendo} = \text{Divisor} * \text{Quociente} + \text{Resto}$

- Tal como na multiplicação, é necessário usar os registos Hi e LO para armazenar o resultado final na forma de um quociente e de um resto.

- Hi → armazena o resto da divisão inteira
- LO → armazena o quociente da divisão inteira

- No MIPS, usam-se as seguintes instruções:

- `div Rsrc1, Rsrc2` → Signed
- `divu Rsrc1, Rsrc2` → Unsigned



Ex: Obter o resto da divisão inteira:

$$\begin{array}{l} \text{div \$t0, \$t5} \\ \text{mfhi \$ao} \end{array} \rightarrow \begin{array}{l} hi = \$t0 \% \$t5 \\ lo = \$t0 / \$t5 \\ \$ao = hi \end{array}$$

Instruções virtuais de divisão

Divisão Signed

- ④ `div Rdst, Rsrc1, Rsrc2`
- ↳ `div Rsrc1, Rsrc2`
- ↳ `mflo Rdst`

Divisão unsigned

- ④ `divu Rdst, Rsrc1, Rsrc2`
- ↳ `divu Rsrc1, Rsrc2`
- ↳ `mflo Rdst`

Resto da divisão inteira Signed

- ④ `nem Rdst, Rsrc1, Rsrc2`
- ↳ `div Rsrc1, Rsrc2`
- ↳ `mfhi Rdst`

Resto da divisão inteira unsigned

- ④ `remu Rdst, Rsrc1, Rsrc2`
- ↳ `divu Rsrc1, Rsrc2`
- ↳ `mfhi Rdst`

Note!: Todas estas instruções poderão ser ajustadas para

- o segundo registo ser substituído por uma constante (immediate)

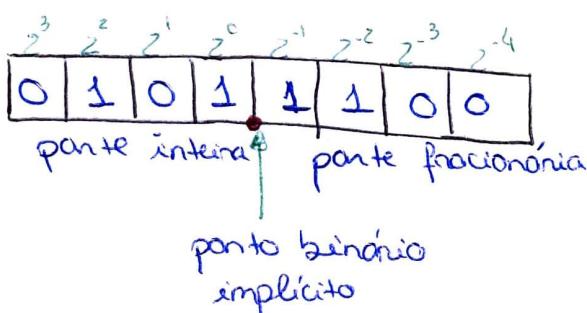
↳ Representação de quantidades fracionárias

- A codificação de quantidades numéricas com que trabalhamos até agora esteve sempre associada à representação de nº inteiros.
- A representação posicional de inteiros pode também ser usada para representar números nacionais considerando-se potências negativas da base.
 - ↓
 - ↓ A posição de um algarismo determina o seu peso no valor

Ex: 8 bits

↳ 4 para a parte inteira	↳ 4 para a parte fracionária
--------------------------	------------------------------

$$5,75 = 4 + 1 + 0,5 + 0,25 = 2^3 + 2^2 + 2^{-1} + 2^{-2}$$



Representação
em Vírgula
fixa

- A representação de quantidades em vírgula fixa coloca a questão da divisão do espaço de armazenamento para cada uma das partes.

- O espaço de armazenamento é limitado, por isso quantos bits devem ser reservados para a parte inteira e quantos para a parte fracionária?

→ O nº de bits da parte inteira determina a gama de valores representáveis (2^4 , no ex. anterior)

→ O nº de bits da parte fracionária determina a precisão da representação ($2^{-4} = 0,0625$, no ex. anterior).

- Representação em vírgula flutuante (VF)

- A posição da vírgula pode ser deslocada sem alterar o valor representado, ponderando-se pelo valor do expoente de base 10 representado.

$$\text{Ex: } 23,45129 = 0,2345129 \times 10^2$$

- Tem a vantagem de não desperdiçar espaço de armazenamento com os zeros à esquerda.

Em sistemas computacionais digitais:

$$N = \pm 1.f \times 2^{\text{Exp}}$$

- f - parte fracionária representada por n bits
 - $1 \cdot f$ - mantissa ou significando
 - Exp - expoente da potência de base 2 representado por m bits

→ O problema da divisão do espaço de armazenamento coloca-se também neste caso, mas agora na determinação do nº de bits ocupados pela ponte fracionária e pelo expoente.

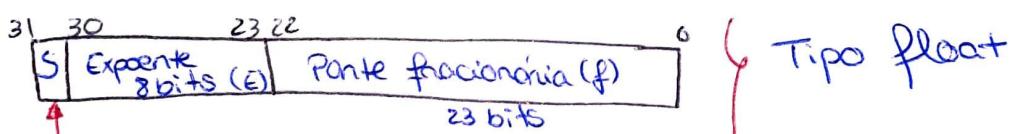
→ Essa divisão é um compromisso entre gama de representação e precisão:

- Aumento do n.º de bits da parte fracionária \Rightarrow maior precisão
 - Aumento do n.º de bits do expoente \Rightarrow maior gama de Representação

Um bom design implica compromissos adequados

↳ Norma IEEE 754 (preciso simples)

- 32 bits divididos em 3 campos:



Sinal da quantidade:

O - positivo

1- negativo

- A representação é **normalizada**: o bit à esquerda do ponto binário é sempre 1, e por isso não é explicitamente representado (hidden bit)
 - Ponte fracionária (23 bits) pode tomar valores entre:

$$+0000(\dots) \quad \text{e} \quad +1111(\dots)$$

- E os limites de representação da mantissa são: (1.f)

1.0000 (...) e 1.1111 (...)

- O exponte é codificado em excesso de 127 ($2^{n-1} - 1 \Rightarrow n = 8$ bits). Ou seja, é somado ao expoente verdadeiro (exp) o valor 127 para obter o código de representação.

$$\begin{array}{l} E = \text{Exp} + 127 \\ \downarrow \\ \text{expoente codificado} \end{array} \quad \left\{ \begin{array}{l} E = 127 \rightarrow \text{Exp} = 0 \\ E > 127 \rightarrow \text{Exp} > 0 \\ E < 127 \rightarrow \text{Exp} < 0 \end{array} \right.$$

- Sendo assim, temos a seguinte representação:

$$N = (-1)^s \cdot 1.f \times 2^{\text{Exp}} = (-1)^s \cdot 1.f \times 2^{E-127}$$

- Na conversão binário → decimal, qual é o nº de dígitos à direita da vírgula na representação em casas decimais?
- representação com "n" dígitos fracionários na base "R", o nº máximo de dígitos na base "s" que garante que a mudança de base não acrescenta precisão à representação original.

$$m = \lfloor n \frac{\log n}{\log s} \rfloor$$

Neste formato podem ocorrer:

- Overflow → Quando o expoente do resultado não cabe no espaço que lhe está reservado:

$$N_{\text{resultado}} > 1,111(\dots) \times 2^{+127} \quad | \quad E > 254$$

- Underflow → caso em que o expoente é tão pequeno que também não é representável:

$$0 < N_{\text{resultado}} < 1,0000(\dots) \times 2^{-126} \quad | \quad E < 1$$

Operações de Adição / Subtração:

- 1º Passo → Igualar os expoentes ao maior
 - ↳ Desnormalizar o nº com menor expoente

- 2º Passo → Somar / Subtrair os mantissas, mantendo os expoentes

- 3º Passo → Normalizar o resultado

- 4º Passo → Amedondar o resultado e renormalizar (se necessário)

- ↳ Truncar
- ↳ Amedondar → somarmos 1 ao bit imediatamente à direita do ponto binário virtual, colocado no local onde se quer arredondar

Operações de multiplicação:

1º Passo → Somar os expoentes

2º Passo → Multiplicar as mantissas

3º Passo → Normalizar o resultado

4º Passo → Arredondar e renormalizar o resultado (da mesma forma que na adição/subtração)

Operações de divisão:

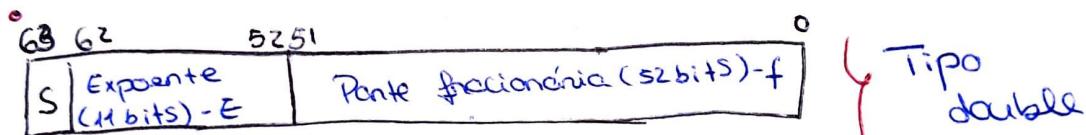
1º Passo → Subtrair os expoentes

2º Passo → Dividir as mantissas

3º Passo → Normalizar o resultado

4º Passo → Arredondar o resultado

↳ Norma IEEE 754 (precisão dupla)



$$N = (-1)^S \cdot 1.f \times 2^{(E-1023)}$$

O exponente pode tomar valores entre -1022 e +1023

A gama de representação:

$$\pm [1,000(\dots) \times 2^{-1022}] ; [1,111(\dots) \times 2^{+1023}]$$

De modo a não exceder a precisão da representação original, a representação em decimal deve ter, no máximo, $\lfloor 52/\log_2(10) \rfloor = 15$ c.d.

Casos particulares:

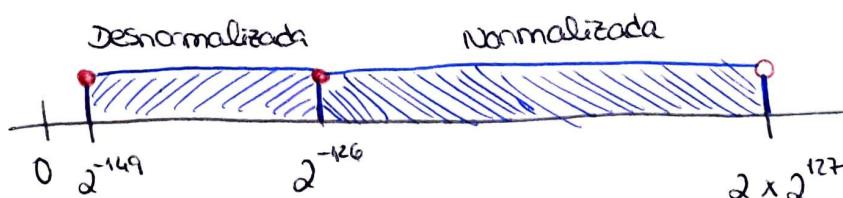
- Quantidade zero: esta quantidade não seria representável (0×2^{126} - 0 e +0)
- $\pm\infty$: Gama de representação excedida. Divisão por zero
- Resultados não numéricos (NaN - Not a Number): Exemplos: $\frac{0.0}{0.0}$, $\frac{\inf}{\inf}$, non²
- Afim de aumentar a resolução (menor quantidade representável), é ainda possível usar um formato de mantissa desnormalizada, no qual o bit à esquerda do ponto binário é zero

Precisão simples	Precisão dupla		Significado	
Expoente	P. frac.	Expoente	P. frac.	
0	0	0	0	Zero
0	$\neq 0$	0	$\neq 0$	Quantidade desnormalizada
1 a 254	qualquer	1 a 2046	qualquer	Nº em UF normalizada
255	0	2047	0	Infinito
255	$\neq 0$	2047	$\neq 0$	NaN (Not a Number)

Representação desnormalizada:

- Assume-se que o bit à esquerda do ponto binário é 0.
- O expoente codificado é zero \rightarrow o expoente verdadeiro é -126 (precisão simples) ou -1022 (precisão dupla)
- Permite a representação de quantidades cada vez mais pequenas (underflow gradual)
- Gama de representação, em precisão simples:

$$\pm [0,000(\dots)01 \times 2^{-126}; 0,1111(\dots)111 \times 2^{-126}] = \pm [2^{-23} \times 2^{-126}; 1 \times 2^{-126}]$$



Ld Arredondamento na FP

- As operações aritméticas são efetuadas com um número de bits da parte fracionária superior ao disponível no espaço de armazenamento.
- Assim, na conclusão de qualquer operação aritmética, é necessário proceder ao arredondamento do resultado, de forma a assegurar a sua adequação ao espaço que lhe está destinado.
- As técnicas mais comuns neste processo (que introduz um erro) são:
 - ① → Truncatura
 - ② → Arredondamento simples
 - ③ → Arredondamento para o par / ímpar mais próximo.

① Truncatura :

Exemplo 2 bits na parte fracionária : d=2

valor	Trunc.	erro
x.00	x	0
x.01	x	- 1/4
x.10	x	- 1/2
x.11	x	- 3/4

- Mantém - se a parte inteira, desprezando qualquer informação que exista à direita do ponto binário
- Erro médio: $\frac{0 - \frac{1}{4} - \frac{1}{2} - \frac{3}{4}}{4} = -\frac{3}{8}$

② Arredondamento simples :

Exemplo: 2 bits na parte fracionária, d=2

valor	Arredond.	Erro
x.00	x	0
x.01	x	$x - x.25 = -\frac{1}{4}$
x.10	x+1	$(x+1) - x.5 = +\frac{1}{2}$
x.11	x+1	$(x+1) - x.75 = +\frac{1}{4}$

- Soma - se 1 ao 1º bit à direita do ponto binário e trunc - se o resultado:
- arredond = trunc (valor + 0,5)

$$\begin{array}{r} x.00 \\ + 0,1 \\ \hline x.10 \end{array} \quad \begin{array}{r} x.01 \\ + 0,1 \\ \hline x.11 \end{array} \quad \begin{array}{r} x.10 \\ + 0,1 \\ \hline x+1.00 \end{array} \quad \begin{array}{r} x.11 \\ + 0,1 \\ \hline x+1.01 \end{array}$$

• Erro médio: $\frac{0 - \frac{1}{4} + \frac{1}{2} + \frac{1}{4}}{4} = +\frac{1}{8}$

↳ Mais próximo de zero do que no caso da truncatura, mas ligeiramente mais polarizado do lado positivo.

③ Arredondamento para o par mais próximo

Exemplo: 2 bits na parte fracionária, $d=2$

val	Arred	Enro
x0.00	x0	0
x0,01	x0	-1/4
x0.10	x0	-1/2
x0.11	x1	+1/4

Número par
arredonda para baixo

val	Arred	Enro
x1.00	x1	0
x1.01	x1	-1/4
x1.10	x1+1	+1/2
x1.11	x1+1	+1/4

Número ímpar
arredonda para cima

$$\begin{array}{r} \times 0.10 \\ + 0.0 \\ \hline \times 0110 \end{array}$$

$$\begin{array}{r} \times 1.10 \\ + 0.1 \\ \hline \times 1110 \end{array}$$

Nota: Estes arredondamentos servem para restringir a parte fracionária do resultado a 23 bits, na precisão simples, ou a 52 bits, na precisão dupla.

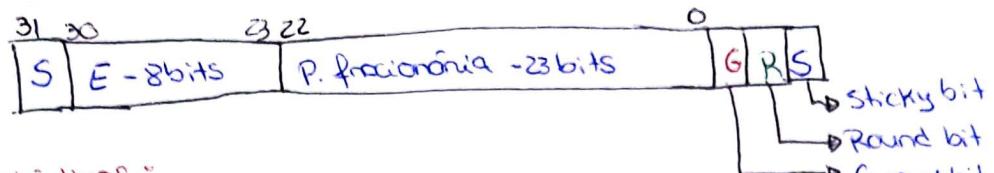
- O arredondamento é feito no bit menos significativo da p. fracionária (bit 23 / 52)

O que fica à direita de b_{23}	Resultado	
$b_{24} = 0 \quad (< 0.5)$ e restantes $\neq 0$	Round down: bits à direita de b_{23} são descartados	Arredondamento Simples
$b_{24} = 1 \quad (> 0.5)$ e restantes $\neq 0$	Round up: soma-se 1 a b_{23} (propagando o carry)	
$b_{23} = 1 \quad \quad b_{24} = 1 \quad e$ restantes $= 0 \quad \quad = 0.5$	$\dots b_{22} 1100$ Round up: soma-se 1 a b_{23} (propagando carry)	Arredondamento para o par mais próximo
$b_{23} = 0 \quad \quad b_{24} = 1 \quad e$ restantes $= 0 \quad \quad = 0.5$	$\dots b_{22} 0100$ Round down: bits à direita de b_{23} são descartados	
$b_{23} = 1 \quad \quad b_{24} = 1 \quad e$ restantes $= 0 \quad \quad = 0.5$	$\dots b_{22} 1100$ Round down: bits à direita de b_{23} são descartados	Arredondamento para o ímpar mais próximo
$b_{23} = 0 \quad \quad b_{24} = 1 \quad e$ restantes $= 0 \quad \quad = 0.5$	$\dots b_{22} 0100$ Round up: soma-se 1 a b_{23} (propagando o carry)	

↳ Norma 754 - Arredondamentos

- Os valores resultantes de cada fase intermédia do cálculo de uma operação aritmética são armazenados com 3 bits adicionais à direita do bit menos significativo da mantissa

↳ Precisão simples: bits com pesos 2^{-24} , 2^{-25} e 2^{-26}



objectivos:

- Ter bits suplementares para a pós-normalização
- Minimizar o erro introduzido pelo processo de arredondamento

G → Guard bit - bit imediatamente à seguir à parte fracionária

R → Round bit

S → Sticky bit - resultado da soma lógica (or) de todos os bits à direita do bit R (se houver pelo menos um bit a '1', então S = '1').

↳ Cálculo em vírgula flutuante no MIPS

FPU (floating point unit)

- O MIPS inclui um coprocessador aritmético (Coprocessador 1 - (CP1)) capaz de efetuar operações aritméticas em vírgula flutuante, usando a norma IEEE 754.

- Esse coprocessador tem o seu próprio espaço de armazenamento composto por um conjunto de 32 registos, de 32 bits cada, e o seu próprio ISA.

→ Registros

- Os registos do coprocessador 1 são designados por \$fn, em que n toma valores entre 0 e 31 (\$f0, \$f1, ...)

- Cada par de registos consecutivos [\$fn, \$fn+1] (com n par) pode funcionar como um registro de 64 bits para armazenar valores em precisão dupla.

- A referência ao conjunto de 2 registos faz-se sempre indicando o registro par (\$f0, \$f2, ...)

Nota: Apenas os registos par podem ser usados no contexto das instruções.

(Só os 16 de indice par)

Instruções aritméticas do CPU:

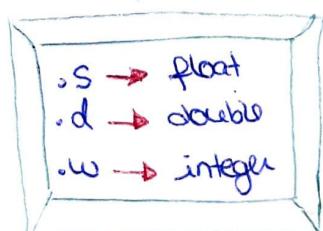
- abs.p FPdst, FPsnc # Absoluto
- neg.p FPdst, FPsnc # Negado
- div.p FPdst, FPsnc1, FPsnc2 # Divisão
- mul.p FPdst, FPsnc1, FPsnc2 # Multiplicação
- add.p FPdst, FPsnc1, FPsnc2 # Adição
- sub.p FPdst, FPsnc1, FPsnc2 # Subtração

Sufixo → representa a precisão com que é efectuada a operação (simples ou dupla).

Na instrução, este sufixo é substituído por .s, na precisão simples, ou .d, na precisão dupla.

↳ Conversões entre tipos de variáveis

- Cvt.d.s FPdst, FPsnc # Float → Double
- Cvt.d.w FPdst, FPsnc # Int → double
- Cvt.s.d FPdst, FPsnc # double → float
- Cvt.s.w FPdst, FPsnc # int → float
- Cvt.w.d FPdst, FPsnc # double → int
- Cvt.w.s FPdst, FPsnc # float → int
 - ↓ ↓
formato original
 - do resultado



Nota: Estas conversões entre tipos de representação são efectuadas na FPU, logo os operandos são obrigatoriamente registos da FPU

↳ Instruções de transferência - VF no MIPS

→ Transferência de informação entre registos do CPU e da FPU, e entre registos da FPU

- mtc1 CPU\$nc, FPdst

FPU
Move to coprocessor 1
CPU → FPU

- mfcl CPU\$dst, FPSnc

FPU
Move from coprocessor 1
FPU → CPU

Nota: Estas instruções copiam o conteúdo integral dos registos, não há qualquer tipo de conversão.

- mov.s FPdst, FPSnc

Move from FPSnc to FPdst com precisão simples (float)

- mov.d FPdst, FPSnc

Move from FPSnc to FPdst com precisão dupla (double)

→ Transferência de informação entre registos da FPU e a memória

- l.s FPdst, offset(CPUneg)

load float from memory

- s.s FPSnc, offset(CPUneg)

store float into memory

- l.d FPdst, offset(CPUneg)

load double from memory

- s.d FPSnc, offset(CPUneg)

store double into memory

Instruções nativas:

Não usadas

- lwc1 FPdst, offset(CPUneg)
- swc1 FPSnc, offset(CPUneg)
- ldc1 FPdst, offset(CPUneg)
- sdc1 FPSnc, offset(CPUneg)

load float from memory

store float into memory

load double from memory

store double into memory

↳ Manipulação de constantes - VF no NIPS

- Nas instruções do FPU do NIPS, não há suporte para a manipulação de constantes, por isso os operandos têm que residir em registos internos.

Método 1: (+ trabalho e sujeito a erros)

- Determinar, manualmente, o valor que codifica a constante (32 bits para precisão simples ou 64 bits para precisão dupla)
- Carregar essa constante em 1 ou 2 registos do CPU e copiar o seu valor para o/s registro(s) da FPU

Método 2: (+ simples, a codificação da constante fica ao trabalho do MARS)

- Usar as diretivas ".float" ou ".double" para definir em memória o valor da constante: 32 bits (.float) ou 64 bits (.double)
- Ler o valor da constante da memória para um registo da FPU usando as instruções de acesso à memória (l.s ou l.d)

Para este método, o MARS disponibiliza 2 instruções virtuais que facilitam a sua utilização:

- l.s FPdst, label
 - l.d FPdst, label
- label representa o endereço onde a constante está armazenada em memória
- Cada uma destas instruções é decomposta em duas instruções nativas:

Ex: l.s. \$f0, k1 (sendo k1 = 0x100000C)

- lei \$1,0x1001 (16 bit mais significativos do endereço)
- l.s \$f0, 0x000C(\$1)

↳ Instruções de decisão - VF no NIPS

- As tomadas de decisão do coprocessador 1 são realizadas de forma distinta da utilizada para o mesmo tipo de operações no CPU (quantidades inteiros)
- Para quantidades em vírgula flutuante, só necessárias 2 instruções em sequência:
 - Comparação das 2 quantidades
 - Decisão (usa a informação produzida pela comparação)

① A instrução de comparação coloca a True ou False numa flag (1 bit), dependendo da condição em comparação ser verdadeira ou falsa.

② A instrução de decisão (instrução de salto) pode alterar o fluxo de execução, em função do estado dessa flag

Instruções de comparação:

- C. xx . S FPU neg¹, FPU neg² # comparação com floats
- C. xx . d FPU neg¹, FPU neg² # comparação com doubles

em que xx é substituído por uma das seguintes condições:

- EQ - equal
- LT - less than
- LE - less or equal

Ex: C.eq.Sf0,\$f2

C.le.d \$f4,\$f8

Instruções de salto:

- bc1t label # branch if true
- bc1f label # branch if false

↳ Convénção de utilização de registos - VF no MIPS

- Registos para passar parâmetros para sub-rotinas (do tipo save):
→ \$f12(\$f13) e \$f14(\$f15)

- Registos para devolução de resultados das subrotinas:
→ \$f0(\$f1)

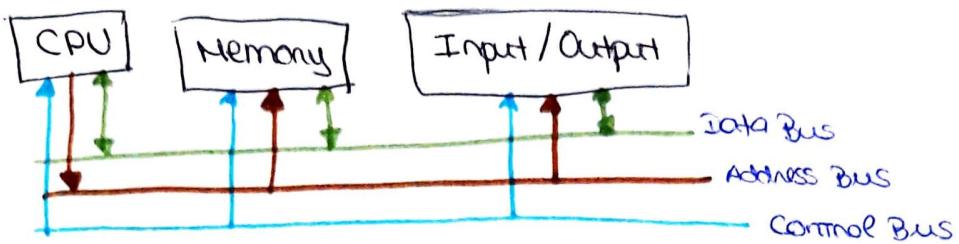
- Registos que podem ser livremente usados e alterados pelas sub-rotinas ("caller-saved"):
→ \$f0(\$f1) a \$f18(\$f19)

- Registos que não podem ser alterados pelas sub-rotinas ("callee-saved"):
→ \$f20(\$f21) a \$f30(\$f31)

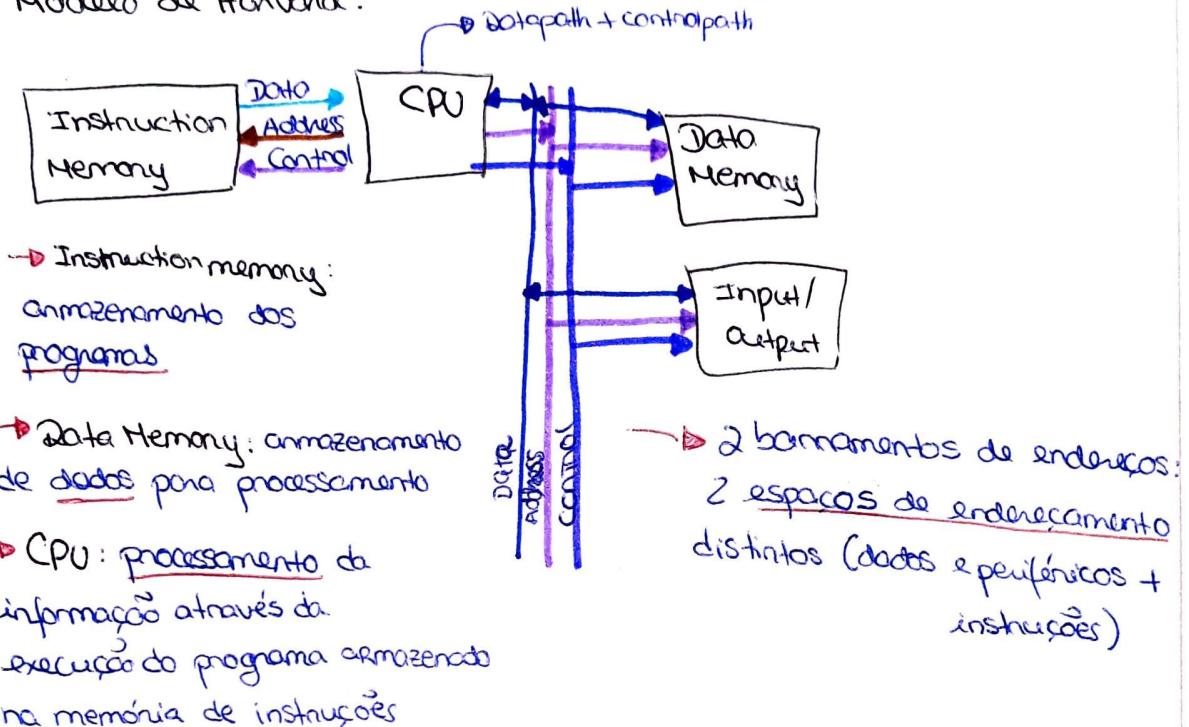
L Modelos Computacionais

(ver página 5)

- Modelo de von Neumann:



- Modelo de Harvard:



Von Neuman	vs	Harvard
<ul style="list-style-type: none"> • Um único espaço de endereçamento para instruções e dados (1 única memória) • Acesso a instruções e dados é feito em ciclos de relógio distintos 		<ul style="list-style-type: none"> • 2 espaços de endereçamento separados: um para dados e outro para instruções (2 memórias independentes) • Possibilidade de acesso, no mesmo ciclo de relógio, a dados e instruções (a CPU pode fazer o fetch da instrução e ler os dados que a instrução vai manipular no mesmo ciclo). • Memórias de dados e instruções podem ter comprimentos de palavra diferentes

SINGLE CYCLE

↳ Implementação de um datapath

- O CPU consiste, fundamentalmente, em 2 seções:

→ Secção de dados (datapath):

Elementos operativos / funcionais

para armazenamento, processamento
e encaminhamento da informação:

- Registros (banco de registos, p.u.)

- Unidade Aritmética e Lógica (ALU)

- Elementos de encaminhamento
(multiplexers)

→ Unidade de Controlo (controlpath):

Responsável pela comandação

dos elementos da secção de
dados, durante a execução
de cada instrução

Tipos de elementos do datapath:

- elementos combinatórios: p.ex., ALU;

- elementos de estado: têm capacidade de armazenamento (p.ex. banco de registos ou outros registos internos (PC))

Nota: Considera-se também a memória externa como um elemento operativo do datapath (elemento de estado), por uma questão de legibilidade dos diagramas.

- Um elemento de estado possui, pelo menos, 2 entradas:

- Uma para os dados a serem armazenados

- Uma para o clock, que determina o instante em que os dados são armazenados (interface síncrona)

- Um elemento de estado pode ser lido em qualquer momento.

- A saída do elemento de estado disponibiliza a informação armazenada na última transição ativa do relógio

- Além do sinal de relógio, um elemento de estado pode ainda ter sinais de controlo adicionais:

→ Sinal de leitura (read): permite a leitura (assíncrona) da informação armazenada.

→ Sinal de escrita (write): autoriza a escrita de informação na próxima transição ativa do relógio (escrita Síncrona).

↳ Caso algum destes sinais não estiver explicitamente representado, a respetiva operação é sempre realizada:

- Escrita realizada 1 vez por ciclo, na transição ativa do clock

- Leitura sempre ativa, c/ a informação na saída.

↳ Implementação de um Datapath no MIPS

- Consideremos um subconjunto de instruções do MIPS:

- Instruções aritméticas e lógicas: add, addi, sub, and, or,slt,slti
- Instruções de acesso à memória: lw e sw
- Instruções de controlo de fluxo de execução: beq e j

Nota: Independentemente da quantidade e do tipo de instruções suportadas por uma dada arquitetura, a infra-estrutura do CPU necessária para executar essas instruções é comum a praticamente todas elas.

No caso do MIPS:

- Para qualquer instrução, as 2 primeiras operações necessárias à sua execução são sempre as mesmas!

① Usar o conteúdo do registo PC (Program Counter) como endereço de memória do qual vai ser lido o código máquina da próxima instrução e efectuar essa leitura (incremento do PC)

② Ler dois registos internos, usando para isso os índices nos respetivos campos da instrução (rs e rt)

→ Nas instruções lw e que operam com constantes (imediatos), operar o conteúdo de um registo é necessário (rs)

→ Em todas as outras é sempre necessário o conteúdo de dois registos, exceto na instrução "j".

(Pode acontecer: ler os 2 registos e usá-los, ou ler os 2 registos e descartar um dos conteúdos)

③ Depois destas operações genéricas, realizam-se as ações específicas para completar a execução da instrução em causa.

- As ações específicas de cada instrução são, em grande parte,

semelhantes, independentemente da instrução em causa:

→ Todas as instruções (exceto "j") utilizam ALU depois da leitura dos registos;

- instruções aritméticas e lógicas usam a operação correspondente (campo func)

- instruções de acesso à memória usam para calcular o endereço de memória (soma)

- branch, efectua a subtração para determinar se os operandos são iguais ou diferentes

- A execução da instrução de salto incondicional (".") resume-se à alteração incondicional do registo PC com o endereço-alvo
- Endereço-alvo → 26 LSBits do código máquina da instrução e dos 4 MSB bits do atual PC
- Depois de usar a ALU, os acdes que se seguem diferem:
 - Inst. Anit. e Lógicas: armazenam o resultado à saída da ALU no registo destino da instrução.
 - SW: Acede à memória para escrita do valor do registo RT
 - LW: Accede à memória para leitura do valor que, de seguida é escrito no destino especificado em rt.
 - BEQ: Depende do resultado da ALU: pode ter que alterar o conteúdo do PC, se a condição em teste for verdadeira (ALU=0)

Fase de leitura de uma instrução:

Instruction Fetch:

- As instruções que compõe um programa são armazenadas sequencialmente na memória:

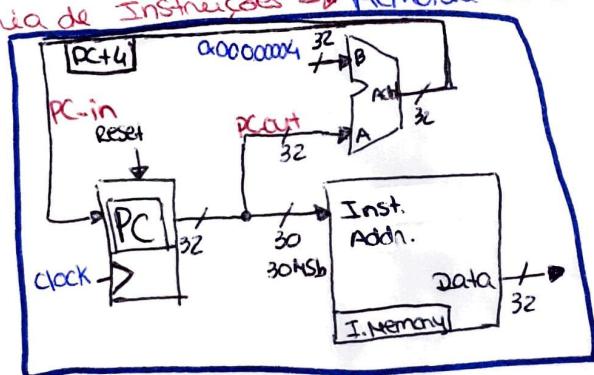
→ Logo, para obter o endereço da instrução seguinte, basta somar Size ao endereço da instrução atual.

- ↳ Dimensão da instrução, em bytes
- ↳ No MIPS, a dimensão é fixa e igual a 4, logo os endereços são sempre múltiplos de 4.

- O processo de Instruction Fetch deverá, uma vez concluído, deixar o conteúdo do PC pronto para endereçar o próximo instrução, i.e., incrementar o PC.

↳ No MIPS, isto corresponde a somar 4 ao valor atual do PC.

Nemória de Instruções → Memória só de leitura



Configuração da ponte necessária à execução do Instruction Fetch

④ Execução de uma instrução do tipo R:

- add, Sub, And, Or, Slt : Inst. aritméticas e lógicas que operam sobre 2 registos

Ex: add $\frac{E2}{Rd}, \frac{B3}{RS}, \frac{B4}{RT}$

32

opcode 6bits	RS 5 bits	RT 5 bits	Rd 5 bits	shamt 5 bits	funct 6 bits
\$3	11	92			
000000 00011	00100	00010	00000	100000	

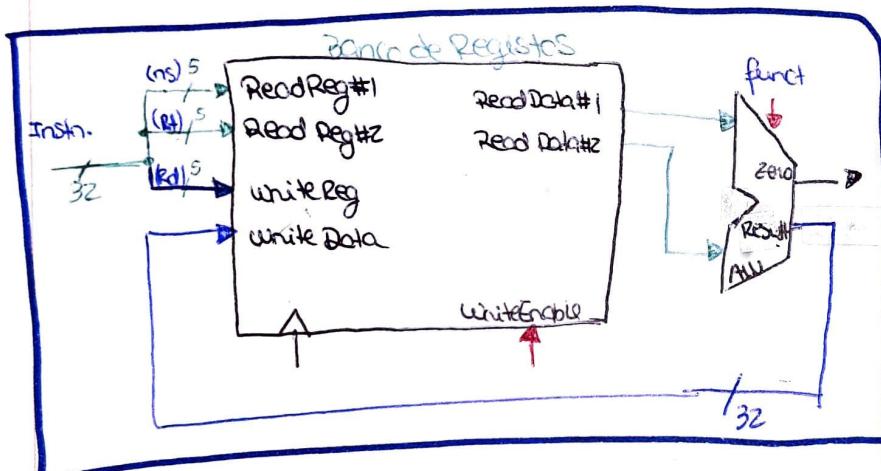
codigo máquina:

0x00641020

O operações realizadas:

- Instruction Fetch: leitura da instrução, cálculo do PC+4
- Leitura dos registos: registos operando especificados nos campos "rs" e "rt" da instrução.
- Realização da operação na ALU: especificada no campo "funct"
- Escrita do resultado: o registo de destino é especificado no campo "rd".

Elementos necessários à execução destas instruções



Configuração dos elementos para a execução de uma inst. do tipo R

Banco de Registros:

- 32 registos de 32 bits cada
- 2 pontos de leitura assíncrona (dual-port memory)
- 1 ponto de leitura síncrona
- Zer0 = 1 quando o resultado da operação for zero (0x00000000)

ALU:

- Operandos de 32 bits
- Resultado de 32 bits
- Operação a realizar depende do campo "funct" da instrução

① Execução de uma instrução store word (sw)

Formato I

Ex: sw \$2, 0x24(\$4)

opcode 6bits	RS 5bits	RT 5bits	offset
0x2B	4	2	0x24

Operações realizadas:

- Instruction Fetch: leitura da instrução, cálculo de PC+4
- Leitura dos registos: contém o endereço-base e o valor a transferir, especificados nos campos "RS" e "RT" da instrução
- Cálculo do endereço de acesso: realiza-se, na ALU, a soma algébrica entre o conteúdo do registo RS e o offset especificado na instrução
- Escrita na memória

② Execução de uma instrução load word (lw)

Formato I

Ex: lw \$4, 0x2F(\$15)

opcode	RS	RT	offset
0x23	15	4	0x2F

Operações realizadas:

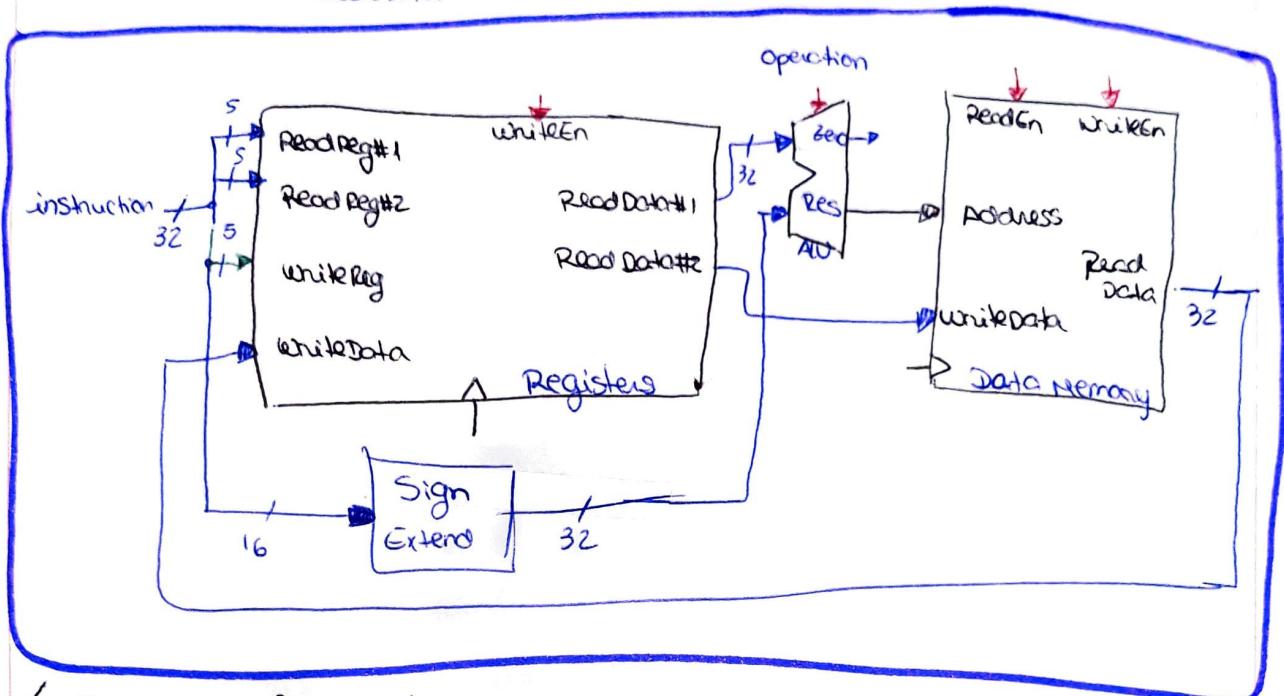
- Instruction Fetch
- Leitura dos registos: tal como nas instruções store word
- Cálculo do endereço de acesso: analogamente às instruções store word
- Leitura da memória

Escrita do valor lido da memória: no registo de destino (campo "rt" da instrução)

Nota: Além da ALU e do Banco de Registros, também são necessários outros elementos para a execução de instruções lw e sw:
→ Memória externa (de dados)
→ Extensor de Sinal

Extensão de Sinal - Cria uma constante de 32 bits, em complemento para dois, a partir dos 16 bits menos significativos da instrução. O bit 15 é replicado nos 16 mais significativos da constante de saída.

Memória de dados: leitura e escrita síncronas, especificados por 2 sinais de enable (\downarrow para leitura e \uparrow para escrita)



↳ Configuração necessária para a execução das instruções `lw` e `sw`

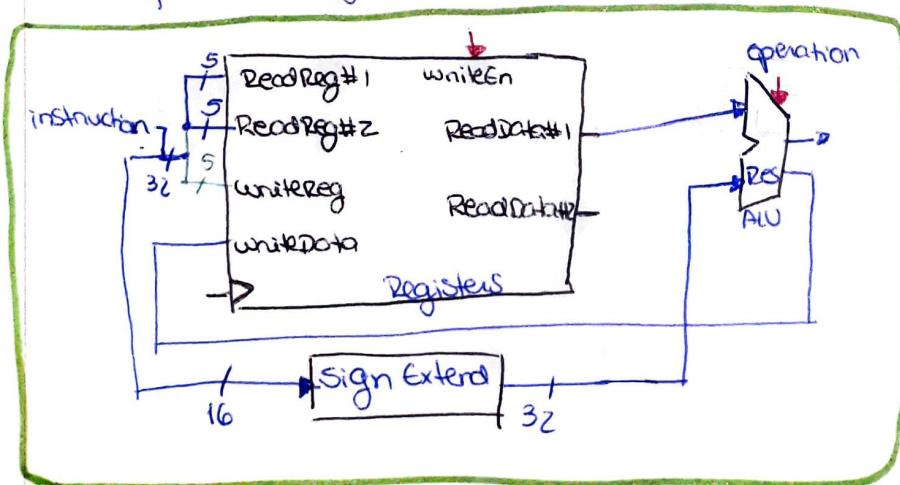
★ Execução de uma instrução do tipo I (immediate) :

→ addi e slli

Exemplo: addi \$4, \$15, 0x2F } 0x21E4002F

opcode	rs	rt	offset
8	15	4	0x2F

• A configuração é semelhante às instruções do tipo R. A única alteração é a fonte do segundo operando.

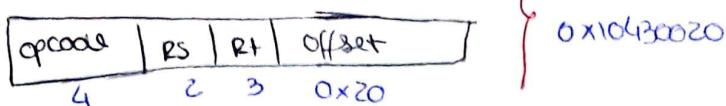


Configuração necessária para a execução de instruções com constantes

① Execução de instruções de branch

Exemplo: `beq $2, $3, 0x20`

Formato I



Operações realizadas:

- Instruction Fetch
- Leitura de dois registos: campos rs e rt
- Comparação dos dois registos: é realizada uma operação de Subtração na ALU. Pode-se usar a saída zero da ALU
- Cálculo do endereço-alvo: Branch Target Address (BTA)

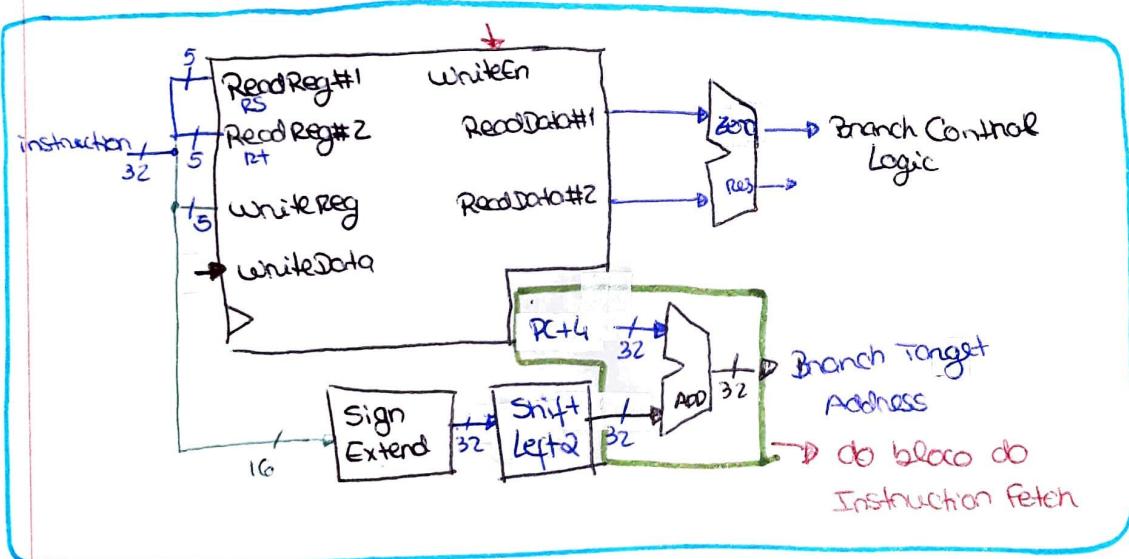
$$\text{BTA} = (\text{PC} + \text{U}) + (\text{instruction-offset} \ll 2)$$

equivalente a multiplicar por 4

- Alteração do valor do register PC

- Se a condição testada pelo branch for verdadeira, então

$$\text{PC} = \text{BTA}$$



Configuração necessária para a execução da instrução de salto condicional (`beq`)

No final :

- É necessário combinar as configurações ora's vistas

 - Utilizam-se 3 multiplexers para encaminhamento de informação.
 - Os sinais de controlo são definidos pelo controlpath em função da instrução a executar.

Multiplexer 1: Escolhe o registo de destino para escrita (unidade reg):
Rd nas instruções tipo R e Rt nas instruções lw e sw.
anit./lógicas imediatas.

Multiplexer 2: Escolher o 2º operando da ALU: conteúdo de um registo (tipo R e branches) ou offset extendido (lw, sw e imediatas)

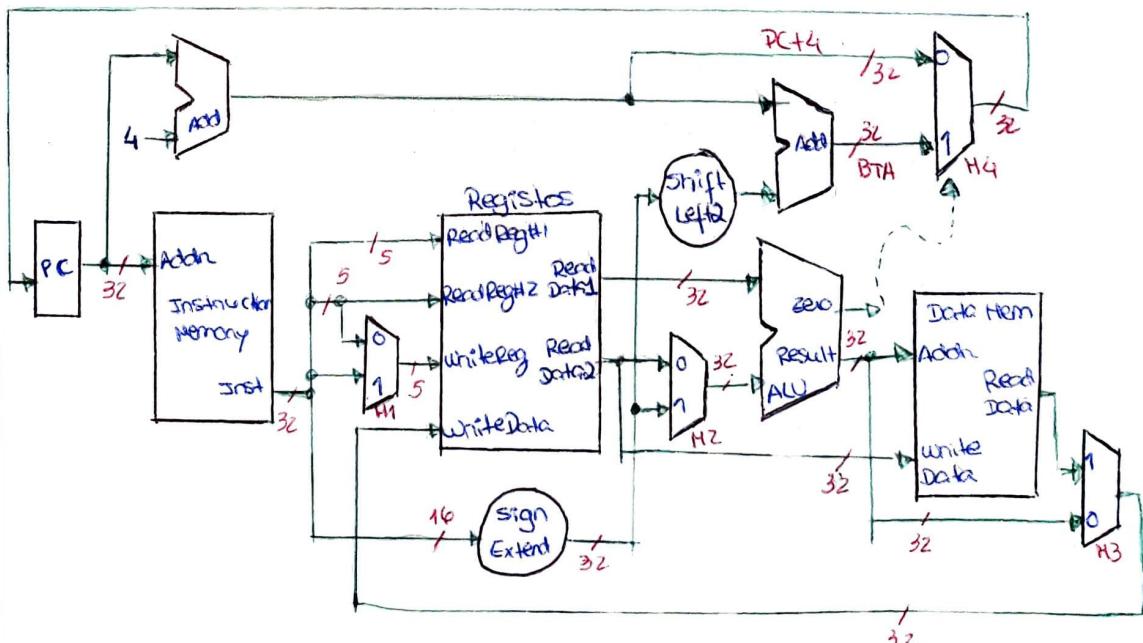
Multiplexor 3: Seleciona o valor a escrever no banco de registos
(write data): valor lido da memória (lw) ou o resultado da ALU (tipo R e imediatas)

→ Faltava ainda incluir o bloco do Instruction Fetch

→ Integrar a configuração das instruções de salto condicional (bif) (cálculo do BTA)

Multiplexor 4: Escolhe o valor com que o PC é atualizado (BTAC) Dependendo da instrução a executar e do valor PC+4

No instrução `bne` } da saída zero de ALU
 Zero = '0' → PC+4 (condição falsa)
 Zero = '1' → BTA (condição verdadeira)



④ Execução da instrução de salto incondicional ("j")

- No formato j existem apenas 2 campos:
- opcode (31-26)
- campo de endereço (25-0)

Tipo 5

- O endereço-alvo é denominado de jump target address (JTA), e obtém-se pela concatenação
 - dos bits 31-28 do PC+4, com
 - os bits do campo de endereço da instrução multiplicados por 4 (2 shifts à esquerda)

$$JTA = \boxed{\begin{array}{c|c} PC+4 & \text{inst-const } \ll 2 \\ \hline 31 & 28 \ 29 \\ & 0 \end{array}}$$

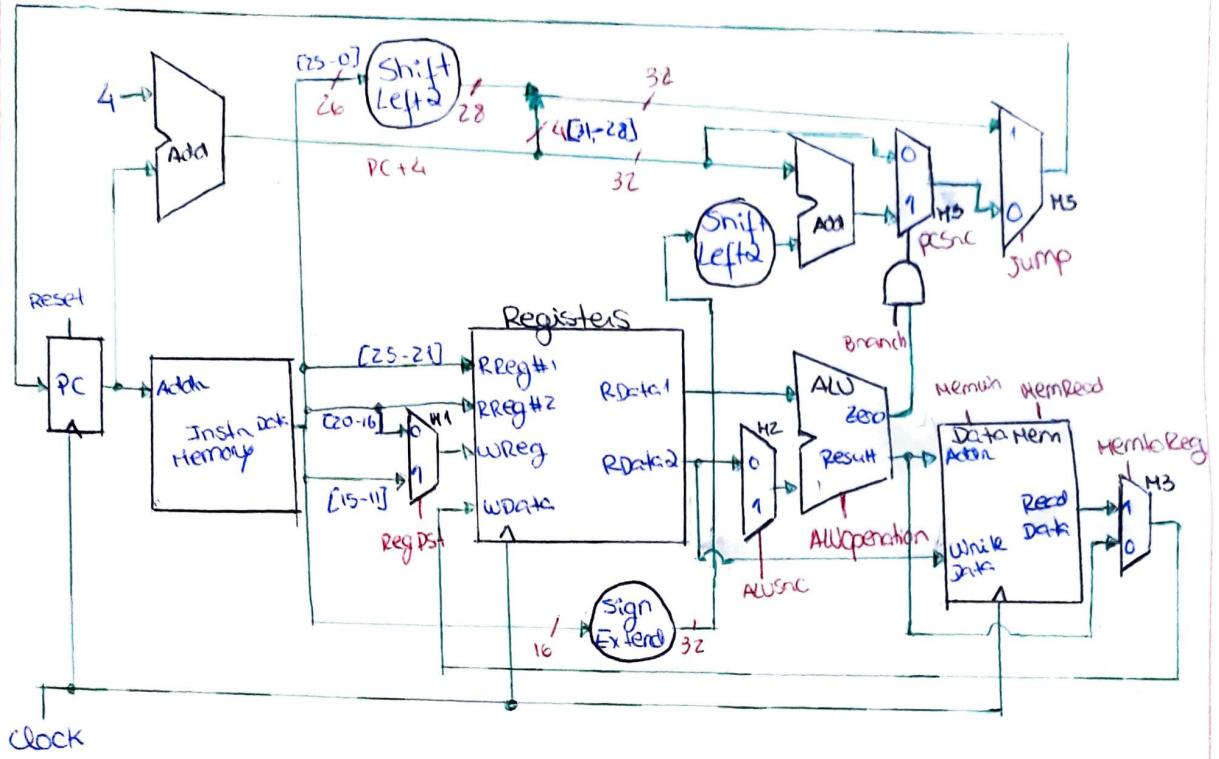
- No próximo flanco ativo do clock, o valor do PC será incondicionalmente alterado com o valor do JTA.

Alterações no datapath de modo a suportar a instrução "j":

→ Cálculo do JTA:

- Shift Left2 → recebe na sua entrada os 26 bits menos significativos da instrução.
- Concatenação dos 4 bits mais significativos do PC+4 com a saída do módulo ShiftLeft2
- Multiplexor 5 → A instrução j seleciona nesse multiplexor a entrada 1, o que faz com que o valor que é encaminhado para a entrada do program counter seja o JTA. Nas restantes instruções, é selecionada nesse multiplexor a entrada 0, e isso corresponde à situação do datapath anterior.

Diagrama single-cycle completo:



Unidade de Controlo - do datapath single cycle

- Deve gerar sinais de controlo, de modo a:
 - Controlar a escrita e/ou leitura em elementos de estado (registos e memória de dados)
 - Definir a operação da ALU e o encaminhamento nos mux's
- Alguns dos elementos de estado são acedidos em todos os ciclos de relogio (PC e memória de instruções), logo não há necessidade de explicitar um sinal de controlo.
- Outros elementos de estado podem ser lidos ou escritos dependendo da instrução (memória de dados e banco de registos), neste caso, é necessário explicitar os respetivos sinais de controlo.
- Dos elementos de estado:
 - Escrita síncrona
 - Leitura assíncrona

Sinais de controlo:
sinais a vermelho no diagrama acima

ALU

- Todas as instruções, exceto o "j", usam este componente:
- **LW e SW**: Soma - para calcular o endereço da memória externa
- **Beq / Bne**: para determinar se os operandos são iguais ou não - Subtração
- **Aritméticas e Lógicas**: para efetuar a respetiva operação
 - ↳ Operação definida pelo opcode e funct.

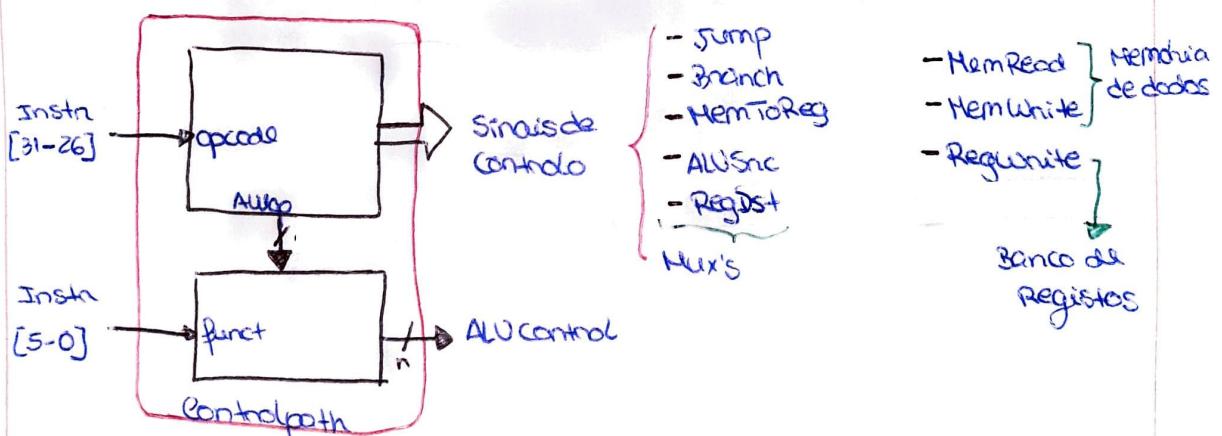
→ A operação a realizar na ALU depende:

- dos campos opcode e funct nas instruções aritméticas e lógicas de tipo R (opcode = 0)
 - ↳ $ALUControl = f(opcode, funct)$
- do campo opcode nas restantes operações:
 - ↳ $ALUControl = f(opcode)$

— || —

→ A unidade de controlo pode ser subdividida em duas:

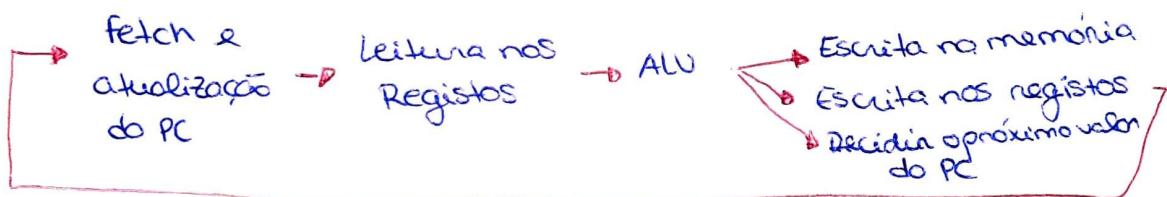
- 1 - Controlo de Multiplexes e elementos de estado
- 2 - Controlo da ALU



Notas sobre o single cycle:

- A execução de uma instrução ocorre no intervalo de tempo correspondente a um único ciclo de relógio.
- Para simplificar a análise, considera-se que a utilização dos vários elementos operativos ocorre em sequência.
- A sequência de operações de uma instrução finaliza com:
 - escrita no banco de registos : tipo R, LW, imediatas
 - escrita na memória de dados : SW
 - ...
- O PC é sempre atualizado:
 - endereço-alvo de branch
 - PC+4
 - endereço-alvo de jump

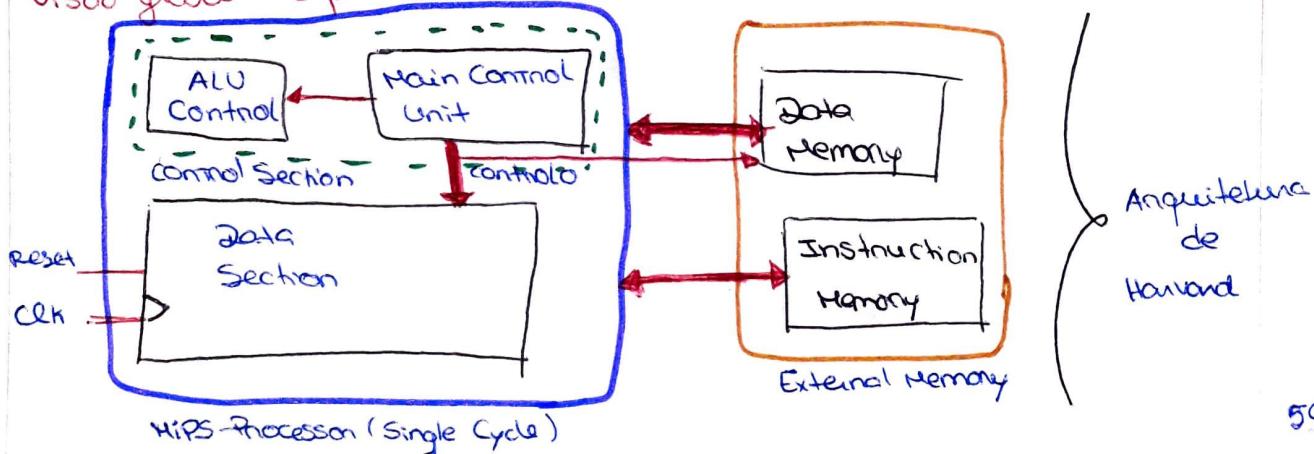
Operações no datapath:



↳ Todas estas fases têm um tempo de execução, o que gera latências, por isso operas na transição ativa podemos dizer que a instrução foi efetuada e usou o seu resultado.

- t_{RH} - Tempo de acesso à memória para leitura
- t_{RRF} - Tempo de acesso ao Register File para leitura
- t_{ALU} - Tempo de operação da ALU
- t_{WRF} - Tempo de acesso ao Register File para preparar a escrita

Visão global do processador:



L) Limitações das arquiteturas single-cycle

• Tempo de execução das instruções:

- A frequência máxima do relógio está limitada pelo tempo de execução da instrução mais longa.
- Tempo de execução de uma instrução \rightarrow soma de todos os atrasos introduzidos por cada um dos elementos envolvidos na execução.
 - L) Só os elementos que se encontram em série contribuem para aumentar o tempo de execução (caminho crítico)

Tempos de atraso no datapath single-cycle:

- t_{RM} - acesso à memória p/ leitura
- t_{WM} - acesso à memória p/ escrita
- t_{RFF} - acesso ao Register File para leitura
- t_{WRF} - acesso ao Register File para escrita
- t_{ALU} - operação da ALU
- t_{ADD} - operação de um somador
- t_{CTRL} - unidade de controlo
- t_{TSE} - Extensor de Sinal
- t_{SL2} - Shift Left 2
- t_{SPPC} - tempo de setup do PC

Exemplos:

\rightarrow de um modo simplificado:

$$\text{Tipo R: } t_{\text{exec}} = t_{RM} + \max(t_{RFF}, t_{CTRL}) + t_{ALU} + t_{WRF}$$

$$\text{Instrução SW: } t_{\text{exec}} = t_{RM} + \max(t_{RFF}, t_{CTRL}, t_{TSE}) + t_{ALU} + t_{WM}$$

$$\text{Instrução LW: } t_{\text{exec}} = t_{RM} + \max(t_{RFF}, t_{CTRL}, t_{TSE}) + t_{ALU} + t_{RM} + t_{WRF}$$

$$\text{Instrução BEQ: } t_{\text{exec}} = t_{RM} + \max(\max(t_{RFF}, t_{CTRL}) + t_{ALU}, t_{TSE} + t_{SL2} + t_{ADD}) + t_{SPPC}$$

$$\text{Instrução J: } t_{\text{exec}} = t_{RM} + \max(t_{CTRL}, t_{SL2}) + t_{SPPC} + t_{TSE} + t_{SL2} + t_{ADD} + t_{SPPC}$$

Notas:

- Desprezou-se os atrasos dos multiplexers
- Desprezou-se o t_{SPPC} nas instruções de controle de fluxo
- Só se considera o t_{SPPC} nas instruções de controle de fluxo

Conclusão:

→ Com esta tecnologia, uma operação de multiplicação ou divisão poderia demorar muito tempo, o que diminuiria drasticamente a frequência máxima do datapath.

→ Tempo de execução de um programa:

$$\textcircled{4} \quad T = N_{\text{instruções}} \times CPI \times \text{Clock-cycle CPU}$$

• Sendo CPI o n.º médio de ciclos de relógio por instrução.
↳ Em single-cycle, $\boxed{CPI = 1}$

$$\textcircled{5} \quad \text{Desempenho} = \frac{1}{T_{\text{exec}}}$$

• O desempenho de um CPU ($\text{CPU}_{\text{ANALISE}}$) relativamente a outro ($\text{CPU}_{\text{referência}}$) pode ser expresso por:

$$\textcircled{6} \quad \frac{\text{D}_{\text{ANALISE}}}{\text{D}_{\text{ref}}} = \frac{T_{\text{exec-ref}}}{T_{\text{exec-análise}}}$$

↳ O Datapath Multi-cycle

- As instruções podem ser executadas em mais que um ciclo de relógio
- A execução da instrução é decomposta num conjunto de operações, independentes entre si.
↳ Cada uma dessas operações faz uso de um elemento operativo fundamental:
 - Memória
 - Register File
 - ALU

Objetivo: Limitar o período de relógio operado pelo máx. dos tempos de atraso de cada um dos elementos operativos fundamentais

- Anitaekura Multi-cycle do MIPS → adota um ciclo de instrução composto no máximo por 5 passos distintos, cada um deles executado num ciclo de relógio

- Durante um ciclo de Relógio, apenas é possível efetuar uma das seguintes ações :

- Acesso à memória (R/w)
- Acesso ao Register File (R/w)
- Operação na ALU

→ No mesmo ciclo de clk podem ser realizadas operações em elementos operativos distintos, desde que sejam independentes

- Datapath multi-cycle :

- Uma única memória para programa e dados (arquitetura de Von Neumann)
- Uma única ALU
- Necessidade de mais elementos de encaminhamento.

→ Um mesmo elemento operativo pode ser utilizado mais que uma vez, no contexto de execução de uma mesma instrução, desde que em ciclos de clk distintos.

- L Ex: Mesma memória para dados e instruções
 L Ex: ALU é usada para calcular PC+4 / BTA

FASES DE EXECUÇÃO

Fase 1:

- Instruction fetch
- Cálculo de PC+4 (atualização em \uparrow)

Memória
ALU

Fase 3:

- Execução da operação (funct)
- ou
- Cálculo do endereço da memória (sw/w)
- ou
- Comparação de operandos (Branch) (conclusão da instrução de branch)

ALU

Fase 2:

- Operando fetch (WRF)
- Cálculo do BTA
- Instruction Decode

Register File
ALU
Unidade de controlo

Fase 4:

- Acesso à memória para leitura (lw)
- ou
- Acesso à memória para escrita (conclusão da instrução sw)
- ou
- Escrita no RF (conclusão das instruções write-back (R/addi/sti))

Memória
ou
RF

Fase 5:

- Escrita no register file (conclusão lw)
write-back

RF

Resumo:

- Branch/j → 3 ciclos de clk
- R/immediate/sw → 4 ciclos de clk
- Lw → 5 ciclos de clk

No datapath é necessário:

- Um multiplexor no bombardeamento de endereços da memória para selecionar entre:

- O conteúdo do PC (leitura de instruções)

ou

- O valor calculado na ALU (leitura / escrita de dados (Lw/Sw))

- Registros na saída dos elementos operativos fundamentais (Mem, ALU e RF), para armazenamento de informação obtida / calculada durante o ciclo de clock corrente e que será utilizada no ciclo de relógio seguinte.

- Instruction Register (IR)

- Data Register (MemDR)

- Registros A e B para armazenar os dados do RF (1 e 2, respetivamente)

- ALUOut (saída da ALU)

Respetivos sinais de controlo para escrita

- Multiplexor na 1^a entrada da ALU, que encaminha a saída do Registo A ou a saída do Registo PC

Sinal de Controlo → ALUSelA

- Multiplexor da 2^a entrada da ALU é aumentado (4:1) para poder suportar:

- incremento do PC (constante 4)

Sinal de controlo

- cálculo do BTA



- Constantes do SE

ALUSelB

- Registo B

- Atualização do PC (Multiplexor 3:1)

Sinal de Controlo → PCSource

- Saída da ALU (PC+4)

- Saída da ALUOut (BTA)

- 26 LSB c/ shift left 2 concatenados com 4 MSB do PC (JTA)

↳ Unidade de Controlo

Single-Cycle:

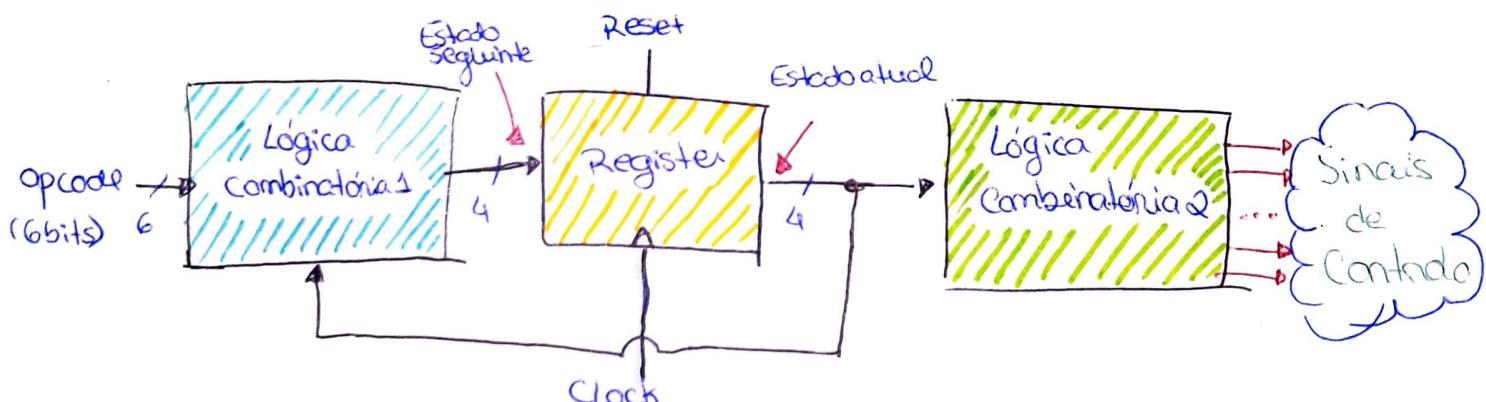
- Controlpath responsável pela geração de um conjunto de sinais que não se alteram durante a execução de cada instrução.
- Pode ser descrito por um circuito meramente combinatório

Multi-cycle:

- Cada instrução é decomposta num conjunto de ciclos de execução
- Os sinais de controlo diferem de ciclo em ciclo e de instrução para instrução (a partir da 2^a fase)
- Para gerar estes sinais, é necessário recorrer a uma máquina de estados
- Sinais de controlo são função da instrução e do estado atual

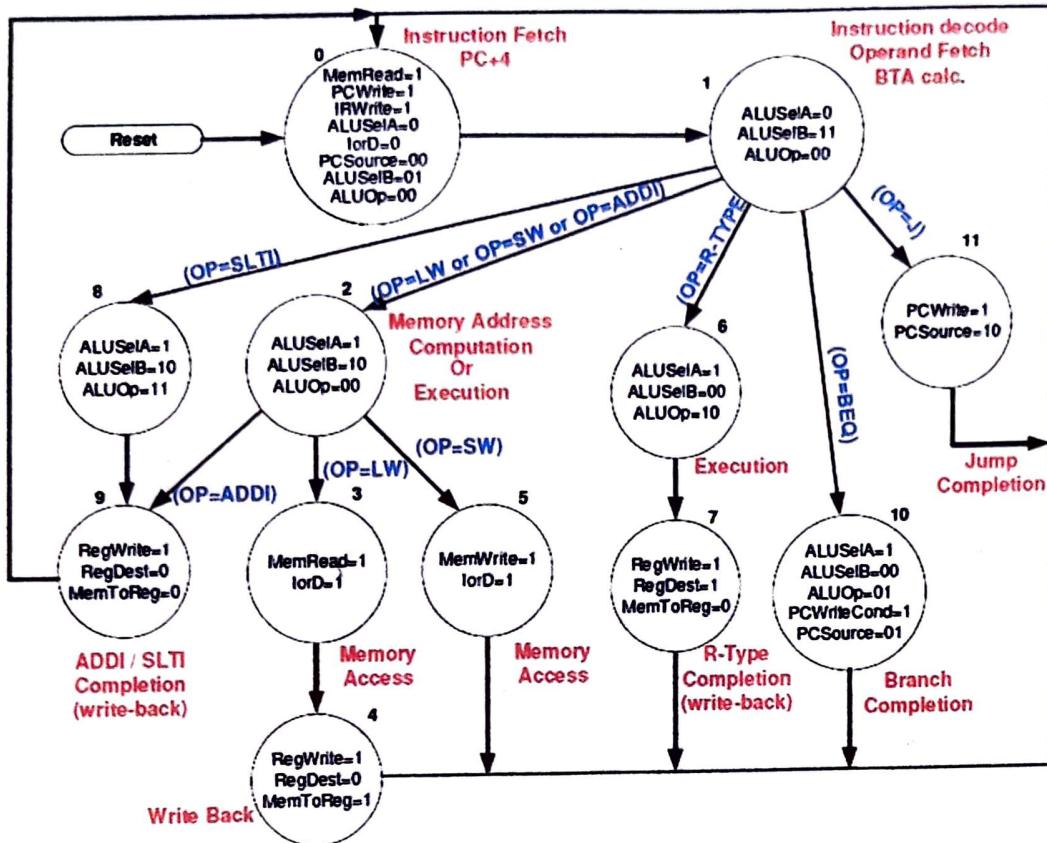
Modelo da FSM de Moore:

- As saídas só dependem do estado atual



↳ O estado seguinte é função do estado atual e das entradas (opcode)

- Diagrama de estados completo do datapath multicycle:



Outra implementação hardware possível:

L ▷ Pipelining

- Técnica de implementação do ISA, através da qual múltiplas instruções são executadas com algum grau de sobreposição temporal.
- Objetivo → aproveita, de forma mais eficiente possível, os recursos disponíveis pelo datapath, de modo a maximizar a eficiência global de processamento.

Analogia: Secções de uma lavandaria:

Lavagem → secagem → Engomar/Sobrar → Annular

- É possível ter várias ondas de roupas em diferentes fases, otimizando assim o trabalho da lavandaria.
- Cada máquina fica menos tempo parada
- Aumento do nº total de ondas processadas por unidade de tempo

→ **Throughput**

Ganho de desempenho:

→ Sistema com F fases e N cargas:

idealmente: (admitindo que cada fase demora 1 unidade de tempo).

• Sist. não pipelined $\Rightarrow T = N \times F$

• Sist. pipelined $\Rightarrow T = F + (N-1) = N \times (F-1)$

$$\hookrightarrow \text{Ganho} = \frac{T_{\text{non pipelined}}}{T_{\text{pipeline}}} = \frac{N \times F}{(F-1) \times F}$$

Nota: Para Pipelines muito longos (com muitos estágios) pode limitar drásticamente a eficiência global

{ Se $N \gg (F-1)$, então
Ganho $\approx F$
 \hookrightarrow Ganho em velocidade é igual ao nº de estágios do pipeline (F)

No MIPS

• Estes princípios podem ser aplicados aos processadores.

• Para o MIPS, a execução da instrução mais longa (lw) pode ser dividida genericamente em 5 fases \hookrightarrow instr. de referência

• Solução pipelined no MIPS \Rightarrow 5 estágios distintos:

1 - Instruction Fetch (IF) - leitura da instrução, incremento do PC

2 - Operand Fetch / Instruction Decode (ID) - leitura dos registos e descodificação da instrução.

3 - Execute (EX) - executar a operação ou calcular um endereço

4 - Memory access (MEM) - acesso à memória de dados para leitura ou escrita.

5 - Write-Back (WB) - escrita do resultado no registo destino.

Problemas da implementação Pipelined:

① As vezes é necessário efetuar, simultaneamente, um acesso à memória para leitura de dados e para o instruction fetch.

 É inevitável que esta arquitetura necessite de 2 memórias, uma para armazenar instruções e outra para dados (como na implementação single-cycle)

② Na instrução BEQ, a unidade de controlo tem de esperar até à fase execute, para que se saiba qual a instrução a realizar a seguir (condição em teste verdadeiro ou falso).

- Nosso admitindo que existe hardware dedicado para avaliar a condição do branch logo no 2º estágio, a unidade de controlo terá sempre de esperar pela execução desse estágio para saber qual a próxima instrução a ler da memória.

③ Utilização de valores de registos em utilização:

Exemplo:

add \$3, \$4, \$5
sub \$2, \$3, \$6
lw \$4, 0(\$5)

A instrução sub não pode avançar para o estágio seguinte (ex), uma vez que o valor de um dos seus operandos ainda não foi calculado e armazenado no registo destino, \$3, pela instrução anterior.

LW	Sub	ADD	(...)	(...)
IF	ID	EX	MEM	WB

- Há uma dependência de dados da instrução sub, relativamente à instrução add, no registo \$3.

↳ Datapath pipelined para o MiPS

Conjunto de instruções consideradas anteriormente:

- Conjunto de instruções consideradas anteriormente:
 - Acesso à memória: LW e SW
 - Tipo R: add, sub, and, or e slt
 - Imediatas: addi e slli
 - Alteração de fluxo de execução: beq e j

O instruction set do MiPS (Microprocessor without Intelocked Pipeline Stages) foi concebido para uma implementação em pipeline.

Os aspectos fundamentais a considerar são:

- Instruções de comprimento fixo: IF e ID podem ser feitos em estágios sucessivos (a unidade de controlo não tem de ter em consideração a dimensão da instrução).
- Poucos formatos de instrução: com a referência aos registos a ler sempre nos mesmos campos. Isto permite que os registos sejam lidos no segundo estágio ao mesmo tempo que a instrução é descodificada pela unidade de controlo.

- Referências à memória só aparecem em instruções de load/store: o 3º estágio pode ser usado para calcular o resultado da operação na ALU ou para calcular o endereço de memória, permitindo o acesso à memória no estágio seguinte.

- A solução pipelined para o MIPS parte do modelo do datapath single-cycle.
- A divisão em fases de execução é realizada com a divisão feita anteriormente (5 estágios).
- No datapath, estes estágios são separados entre si por registos, denominados registos de pipeline. São eles os registos:

- IF / ID
- ID / EX
- EX / MEM
- MEM / WB

Estes registos servem para guardar dados que sejam importantes para a fase seguinte, estes valores ficam assim disponíveis para serem usados no estágio seguinte, no ciclo de relojio a seguir.

L7 Unidade de Controlo

- A implementação pipeline do MIPS usa os mesmos sinais de controlo da versão single-cycle.
- Sendo assim, a unidade de controlo é uma unidade combinatoria que gera os sinais de controlo em função do código-máquina da instrução, presente na fase ID.
- Certos sinais são ligados a registos de pipeline de modo a preservar o seu valor até ao estágio em que são usados.
 - Mem Read e Mem Write \rightarrow MEM
 - Regwrite \rightarrow WB
 - Branch \rightarrow EX

↳ Pipeline Hazards

- São conjuntos de situações particulares que podem condicionar a progressão das instruções no pipeline no próximo ciclo de relógio.
- Podem ser agrupados em 3 classes distintas:
 - Hazards estruturais
 - Hazards de controle
 - Hazards de dados

① Hazard estrutural

- Ocorre quando mais do que uma instrução necessita de aceder ao mesmo hardware:
 - Apenas existe uma memória para dados e instruções.
 - Há instruções no pipeline com diferentes tempos de execução.
- No 1º caso, o problema é resolvido duplicando a memória, i.e., uma para instruções e outra para dados.

② Hazard de Controle

- Ocorre quando é necessário fazer o instruction fetch de uma nova instrução e existe, numa etapa mais avançada do pipeline, uma instrução que pode alterar o fluxo de execução e que ainda não terminou.

Exemplo:
beq \$5, \$6, next
add \$2, \$3, \$4
(...)
next: lw \$3, 0(\$4)

No caso do MIPS:

As situações de hazard de controle surgem com as instruções de salto; jumps e branches: j, jal, jale, jr, beq, ...

① Resolução: A resolução dos

branches no estágio ID minimiza o problema, e por isso a comparação dos operandos passa a ser feita no 2º estágio (ID), através do hardware adicional

⇒ Do mesmo modo, o cálculo do Branch Target Address passa também a ser efectuado em ID

Resolução 2: Para lidar com horizontes de controlo, pode ser usada uma técnica denominada por "stalling" ("parou o progresso de...")

- Nesta estratégia, a unidade de controlo atrasa a entrada no pipeline da próxima instrução até saber o resultado do branch condicional.

- Parar o progresso = pipeline stall, também conhecido por bubble.

- Quando a instrução está na fase IF, no ciclo de relógio seguinte

terá o seu código-máquina no registo de pipeline IF/ID. Mas se

o branch for taken e a instrução seguinte tiver de ser interrrompida,

o seu código-máquina guardado no registo IF/ID será substituído por uma bubble, ou seja, será convertida numa No Operation (NOP), por exemplo ($\text{SII } \$0, \$0, 0 \Rightarrow 0x00000000$)

Instrução que não realiza qualquer operação.

Nota: Na arquitetura MIPS, qualquer instrução que armazene o resultado no registo \$0 é uma NOP

- NOP "oficial" $\Rightarrow 0x00000000$

Isto é feito colocando uma entidade de reset síncrono no registo IF/ID

Resolução 3: Técnica do Prediction

- Provê-se que a condição do branch é falsa (not taken) pelo que a próxima instrução a ser executada será a que estiver em PC+4 (estratégia designada por previsão estática not taken)

- Se a previsão falhou, a instrução anterior lida (a seguir ao branch) é anulada (convertida em NOP), continuando o instruction fetch na instrução correta

tipos de previsões:

- Estáticos: o resultado da previsão não depende da história da execução das instruções de branch / jump:

- Not taken

- Taken

- Backward taken, Forward not taken (BT FNT)

- Dinâmicos: o resultado da previsão depende da história de branches anteriores

- Guardam informação do resultado taken/not taken de branches anteriores e do target address

- A previsão é feita com base na informação guardada

Resolução 4: (a solução do MIPS) O delayed branch.

- Nesta solução, o processador executa sempre a instrução que se segue ao branch (ou jump), independentemente da condição ser verdadeira ou falsa. (a instr. nunca é anulada)
- Esta técnica é implementada com a ajuda do compilador / assembler, que:
 - Reorganiza as instruções do programa para forma a trocar a ordem do branch com uma instrução anterior (desde que não haja dependência entre as duas), ou
 - Não sendo possível efectuar a troca de instruções, o assembler introduz um NOP a seguir ao branch.

Nota: Não é uma técnica comum nos processadores modernos.

- É feita de forma automática pelo compilador / assembler, escorrida da programação.

③ Hazard de dados

- Resultado da dependência existente entre o resultado calculado por uma instrução e o operando usado por outra que se segue mais atrás no pipeline (i.e., mais recente)

Ex: add \$2, \$1, \$3
sub \$3, \$1, \$2

Solução 1: stall do Pipeline

- Parar a progressão no pipeline (stall) da instrução que necessita do valor (e das anteriores), no estágio ID, até que a instrução que produz o resultado chegue ao estágio WB.

• Pode minornar-se o problema se a escrita no banco de registos for feita a meio do ciclo de relojio, ou seja, na transição ascendente.

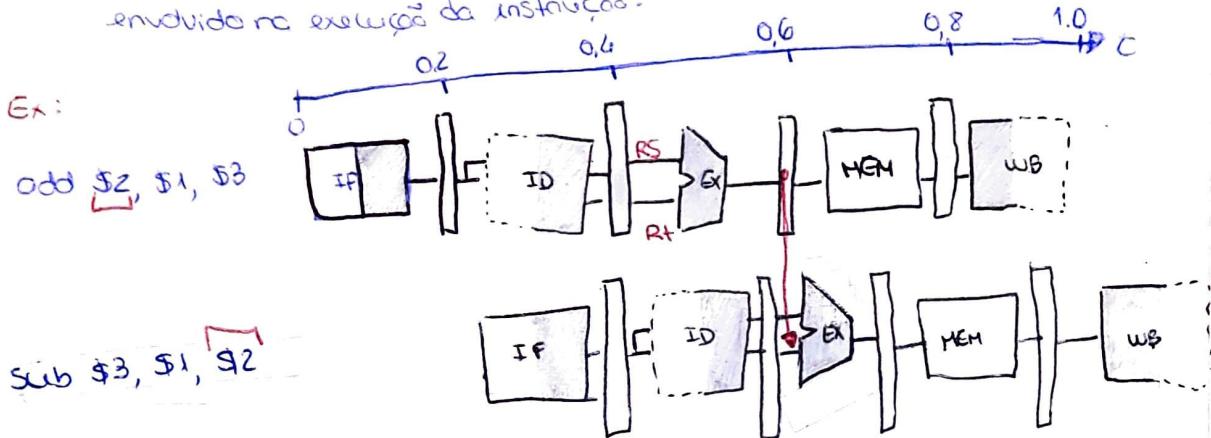
• Impacto elevado no desempenho \Rightarrow cada instrução com dependência atrasa a progressão do pipeline em 1 ou 2 ciclos de relojio.

Solução 2: Forwarding / bypassing

- Às vezes é possível disponibilizar um resultado de uma instrução que ainda não terminou a uma instrução que vem a seguir na cadeia de pipelining, que necessite deste valor.
- Por exemplo, para instruções de tipo R, pode ser disponibilizado o valor na saída da ALU \Rightarrow necessidade de escolher a origem dos operandos, na unidades de controlo

Representação gráfica da codificação de pipeline

- São usados símbolos para representar os recursos físicos
 - Quadrado cinzento: elementos de estado dos estágios IF (memória de instruções), ID (banco de registos), EX (execução), MEM (memória de dados), WB (banco de registos)
 - Metade cinzenta à esquerda: indica uma operação de escrita no elemento de estado.
 - Metade cinzenta à direita: indica uma operação de leitura no elemento de estado.
 - Quadrado branco: indica que o elemento de estado não está envolvido na execução da instrução.



- A seta indica que é usado um forwarding de EX/MEM para EX
- A técnica de forwarding em conjunto com o pipeline stalling consegue resolver grande parte dos hazards de dados.

Solução 3: Reordenação de instruções

Algumas situações de hazards de dados podem ser atenuadas ou resolvidas pelo compilador / assembler através da reordenação de instruções, desde que não comprometa o resultado final.

↳ Forwarding

- Para resolver um hazard de dados através de forwarding é necessário:

- Detectar a situação de hazard

- Encaminhar o valor que se encontra num estágio mais avançado do pipeline para onde ele é necessário

↳ Na maioria das vezes, de **Ex/MEM para Ex** e de

MEM/WB para Ex

↳ Nas instruções de branch, é necessário ter os valores corretos dos registos no estágio ID (forwarding de **Ex/MEM para ID**)

- O hardware a implementar baseia-se em multiplexers, pelo que a unidade de controlo terá a responsabilidade de escolher o encaminhamento correto.

Detectão das situações:

- ① Encaminhar valores para o estágio Ex:

→ Instrução na fase MEM cujo Rdst é um registo operando de uma instrução que se encontra na fase EX.

Ex: add \$1, \$2, \$3
sub \$4, \$1, \$5

~~~~~ → EX/MEM → EX

Acontece quando:

•  $\text{EX/MEM. RDD} == \text{ID/EX. RS}$   
e/ou

•  $\text{EX/MEM. RDD} == \text{ID/EX. RT}$

→ Instrução na fase WB cujo Rdst é um registo operando de uma instrução que se encontra na fase EX

Ex: add \$1, \$2, \$3  
add \$6, \$2, \$3  
sub \$4, \$5, \$1

~~~~~ → MEM/WB → EX

Acontece quando:

• $\text{MEM/WB. RDD} == \text{ID/EX. RS}$
e/ou

$\text{MEM/WB. RDD} == \text{ID/EX. RT}$

Ld Unidade de Controlo de Forwarding

- A simples comparação dos registos não é suficiente para a correta deteção das situações de hazard de dados.

↳ O sinal de controlo que permite a escrita no banco de registos (RegWrite) tem igualmente de ser analisado



↳ Assim, para ser detetada uma situação de hazard de dados, é necessário acrescentar uma nova condição $\Rightarrow \text{Ex/MEM.Regwrite} = 1$

Resumo das situações de hazard de dados de encaminhamento para EX

- $\text{Ex/MEM.Regwrite} = 1 \text{ AND } \text{Ex/MEM.RDD} = \text{ID/Ex.RS}$
- $\text{Ex/MEM.Regwrite} = 1 \text{ AND } \text{Ex/MEM.RDD} = \text{ID/Ex.RT}$
- $\text{MEM/WB.Regwrite} = 1 \text{ AND MEM/WB.RDD} = \text{ID/Ex.RS}$
- $\text{MEM/WB.Regwrite} = 1 \text{ AND MEM/WB.RDD} = \text{ID/Ex.RT}$

Nota: Se $\text{Ex/MEM.RDD} = \$0$ ou $\text{MEM/WB.RDD} = \$0$ estaremos perante uma instrução POP na prática, mas o detector dirá em certas situações que estaremos perante uma situação de hazard.

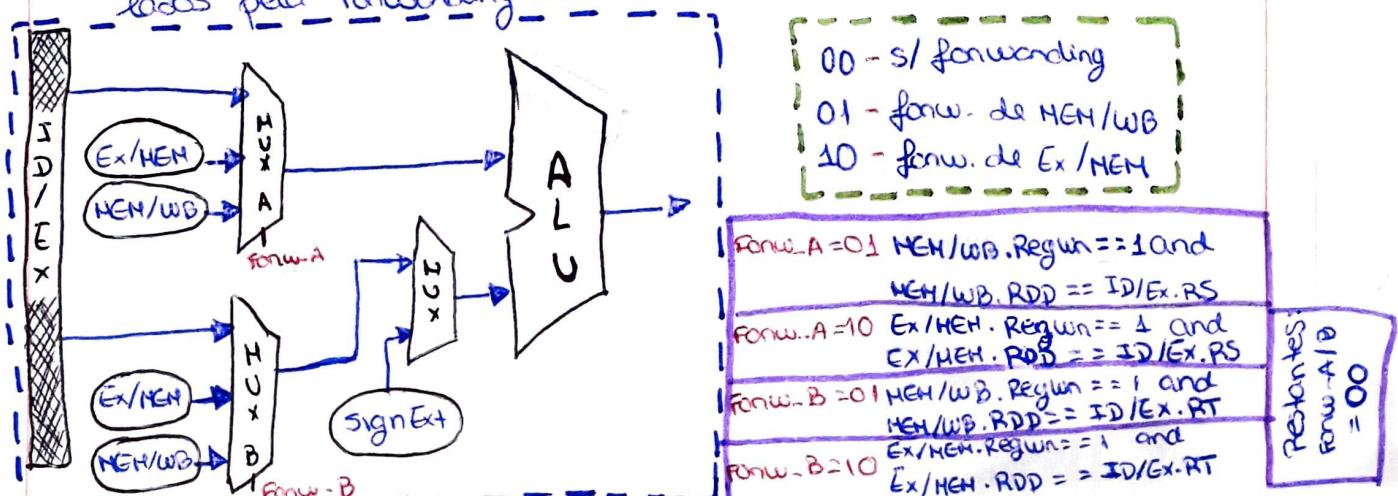
Ex: add \$1, \$2, \$4

sub \$0, \$1, \$3

sw \$2, 0x7FF0(\$0)

É necessário hardware adicional para que não considere esta situação como sendo um hazard

- ALU** → São colocados 2 multiplexers nas respetivas entradas dos operandos, tendo como sinais de escolha "Fonw-A" e "Fonw-B", controlados pela Forwarding Unit.



↳ Dependência que obriga a stalling

- Resulta de uma instrução aritmética ou lógica executada a seguir a uma instrução lw. (entre outras situações)

Ex: $lw \$4, 2(\$1)$ $sub \$2, \$4, \$3$ $add \$3, \$3, \$2$ $\left[\begin{array}{l} \text{stall } 1T, \text{ FW MEM/WB} \rightarrow EX \\ \text{FW } EX/\text{MEM} \rightarrow EX \end{array} \right]$

- A situação de stalling tem de ser detetada quando a instrução tipo R está na sua fase ID (e o lw na fase Ex)

- De forma simplificada,

$$\underbrace{(ID/Ex.\text{MemRead} == 1)}_{\text{Detecção de LW}} \text{ and } \underbrace{(ID/Ex.RDD == RS \text{ or } ID/Ex.RDD == RT)}_{\text{Detecção de igualdade de registos}}$$

- Como é feito o stall do pipeline?

- Inserir bubble na etapa Ex, fazendo o reset síncrono do registo ID/Ex
- Pongelau, durante 1T, as etapas IF e ID, impedindo a escrita no registo IF/ID e a atualização do PC

↳ O controle é feito por uma **unidade de controlo de stalling**

Nota: Existem muitas mais situações que precisam de ser resolvidas com Forwarding e Stalling, pelo que seria necessário never todos os casos e aperfeiçoar as unidades de stalling e de Forwarding.

Forwarding para Ex que obriguem stall do pipeline:

$lw \$1, 0(\$3)$ $sw \$4, 8(\$1)$ $\left\{ \begin{array}{l} \text{stall } 1T, \text{ FW MEM/WB} \rightarrow EX(RS) \end{array} \right.$

$lw \$2, 0(\$3)$ $lw \$4, 8(\$2)$ $\left\{ \begin{array}{l} \text{stall } 1T, \text{ FW MEM/WB} \rightarrow EX(RS) \end{array} \right.$

$lw \$3, 0(\$6)$ $addi \$4, \$3, 0x12$ $\left\{ \begin{array}{l} \text{stall } 1T, \text{ FW MEM/WB} \rightarrow EX(RS) \end{array} \right.$

$lw \$4, 0(\$5)$ $sw \$4, 4(\$4)$ $\left\{ \begin{array}{l} \text{stall } 1T, \text{ FW MEM/WB} \rightarrow EX(RT) \end{array} \right.$

Forwarding EX/MEM \rightarrow ID

- Instruções de branch
- Exemplos:

→ add \$1, \$2, \$3

sub \$2, \$4, \$6

beq \$1, \$5, lab

} FW EX/MEM \rightarrow ID (entrada do comparador)

→ addi \$1, \$3, 0x25

sub \$2, \$1, \$4

beq \$5, \$1, lab

} FW EX/MEM \rightarrow Ex (RS) (entrada da ALU)

} FW Ex/MEM \rightarrow ID (RT) (entrada do comparador)

Stalling seguido de forwarding para ID

- Exemplos:

→ add \$1, \$2, \$3 } Stall 1T, FW EX/MEM \rightarrow ID (RS)
beq \$1, \$5, lab

(→ o mesmo com addi

→ lw \$1, 0(\$5)
beq \$1, \$2, lab } Stall 2T

Forwarding para MEM (MEM/WB \rightarrow MEM)

- Exemplo:

→ lw \$1, 0(\$5)
sw \$1, 4(\$4) } Stall 1T, FW MEM/WB \rightarrow Ex

(→ A instrução SW só necessita do valor de \$1 no estágio MEM (\$4 é necessário em Ex), situação em que a instrução LW já se encontra em WB



Pode ser resolvido sem stalling, com FW MEM/WB \rightarrow MEM