UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES
ARTES, CIÊNCIAS E HUMANIDADES

MATHEUS PECORARO – Nº USP 11917271
SUNGWON YOON – Nº USP 9822261

TRABALHO FINAL - MIPS

Reconhecer se uma string é ou não palíndromo

SÃO PAULO
07 - 2021

MATHEUS PECORARO
SUNGWON YOON

TRABALHO FINAL - MIPS

Reconhecer se uma string é ou não palíndromo

Trabalho apresentado à disciplina de Organização e Arquitetura de Computadores I da Escola de Artes, Ciências e Humanidades da Universidade de São Paulo.
Orientadora: Dr. Gisele da Silva Craveiro

SÃO PAULO
2021

LISTA DE ILUSTRAÇÕES

FIGURAS

TABLEAS

SUMÁRIO

1 ORGANIZAÇÃO E ARQUITETURA MIPS

O MIPS, *Microprocessor without Interlocked Pipelined Stages* (microprocessador sem estágios interligados de pipeline), é uma arquitetura do tipo RISC e foi projetada por John L. Hennessy na década de 80 na Universidade de Stanford e atualmente é desenvolvido por *MIPS Technologies.*

Hoje em dia é largamente utilizado como processador para sistemas embarcados como Nintendo 64, Sony Playstation, roteadores da Cisco etc.

1.1 COMPONENTES BÁSICOS

O MIPS tem componentes básicos: barramento, unidade de controle, banco de registradores, unidade lógica e aritmética (ALU), contador de programa (PC), memória e registrador de instruções (IR).

A interconexão entre os componentes se dá por meio do barramento, e tem tamanho de 32 bits. A unidade de controle busca uma sequência de sinais de controle na memória e executá-las. A ALU executa operações aritméticas e lógicas. O IR é um registrador que guarda cópia de instrução mais recente buscada na memória e, segundo o conteúdo, a UC determina qual a operação a ser realizada.

Os componentes não explicados aqui serão detalhados nas seções posteriores.

1.2 TIPOS DE DADOS

A unidade básica de MIPS é um bit, podendo assumir valor de 0 ou 1. Os bits são organizados de tal maneira que 4 bits constituem um *nibble* e 8 bits constituem um byte. Assim, uma palavra é de 4 bytes (32 bits), meia palavra é de 2 bytes (16 bits) e palavra dupla, de 8 bytes (64 bits). As instruções são de 32 bits.

O processador também pode representar caractere ascii, números inteiros, com ou sem sinal, e números reais. Um caractere é de 1 byte, um inteiro é de 4 bytes (uma palavra) e um real pode ser de precisão simples (4 bytes) ou precisão dupla (8 bytes).

1.3 REGISTRADORES

O MIPS, sendo arquitetura RISC, é do tipo *load-store*, ou seja, as operações lógicas e aritméticas são computadas apenas com dados em registradores e com operandos imediatos, e não diretamente da memória. Por esse motivo, permite uma implementação rápida de instruções simples.

Para manipular os dados da memória, eles precisam ser carregados *(load)* com instrução lw *(load word)* para registradores a fim de que uma operação possa ser executada. Após a execução, o resultado pode ser armazenado (*store*) com instrução sw *(store word)* de volta na memória externa.

MIPS possui 32 registradores de propósito geral dentro do CPU .Cada registro armazena valores de variáveis e instruções lógicas e é representada começando com símbolo $. Eles podem ser utilizados com número, como $0 ou $1, ou com nomes convencionais como $zero e $t1.

A tabela 1 mostra o número, nome dos registradores e seus usos.

TABELA 1 – Registradores de propósito geral de MIPS

Número	Nome	Uso
$0	$zero	Valor constante 0
$1	$at	*Assembler temporary*
$2-$3	$v0-$v1	Valores de retorno de funções
$4-$7	$a0-$a3	Argumentos/parâmetros de funções
$8-$15	$t0-$t7	Registradores temporários
$16-$23	$s0-$s7	Registradores salvos
$24-$25	$t8-$t9	Registradores temporários
$26-$27	$k0-$k1	Reservado para kernel de SO
$28	$gp	Ponteiro global
$29	$sp	Ponteiro para pilha
$30	$fp	Ponteiro para frame
$31	$ra	Endereço de retorno

Fonte: https://www.mips.com/

Devido à alta frequência de uso, existe um registrador exclusivo para o número 0 — registrador $0 – e somente permite a leitura, não podendo ser modificado.

O registrador $at é reservado para *assembler*. Os registradores $v0 e $v1 são os valores do retorno de uma função. Os registradores $a0 a $a3 são utilizados para armazenar quatro primeiros parâmetros ou argumentos de uma função.

Os registradores $t0 a $t9 são utilizados para armazenar variáveis internos temporários, ou seja, não precisam ser preservadas por funções, ao passo que os registradores $s0 a $s7 são utilizados para armazenar variáveis globais permanentes.

Também existem registradores especiais como $gp, $sp, $fp e $ra.

Além disso, existe um coprocessador 1, também chamado de FPA (*Floating Point Accelerator*), para tratar os pontos flutuantes. Igualmente possui 32 registradores e está descrito brevemente na tabela 2.

TABELA 2 – Registradores de ponto flutuante de MIPS

Número	Uso
$f0, $2	Valores de retorno de funções
$f4, $f6, $f8, $f10	Registradores temporários
$f12, $f14	Argumentos/parâmetros de funções
$f16, $f18	Registradores temporários
$f20, $f22, $f24, $f26, $f28, $f30	Variáveis de registro

Fonte: https://www.mips.com/

1.4 TIPOS DE INSTRUÇÕES

Pode-se dividir as instruções de MIPS em principais categorias: instruções aritméticas e lógicas, instruções de acesso à memória e instruções de controle de fluxo.

As instruções de aritméticas são instruções de adição (add, addi, addu e addiu), subtração (sub e subu), divisão (div e divu), multiplicação (mult e multu) e movimentação (mfc, mfhi e mflo).

As instruções de operadores lógicos são operações que cada operando deve conter um valor booleano verdadeiro ou falso e são instruções and, andi, or e ori e de

shift (sll, srl).

As instruções de transferência de dados são *load upper imm.* (lui), *load* (lw) que traz um dado da memória para registrador e *store* (sw) que armazena dado que está no registrador na memória.

As instruções de controle de fluxo são desvio (beq e bne), *set less than* (slt, slti, sltu e sltiu) e salto incondicional (j, jr e jal)

Além disso, o MIPS possui instruções para tratar os pontos flutuantes, como add.s, sub.s, add.d, sub.d, lwc1 e swc1.

1.5 FORMATOS DE INSTRUÇÕES

As instruções de MIPS são de tamanho fixo de 32 bits e são necessários três endereços para a manipulação de dados – um para indicar o registrador de destino que irá armazenar o resultado e dois para indicar registradores fonte.

Existem três formatos de instruções: tipo R, tipo I e tipo J.

1.5.1 Tipo R

As instruções de tipo R são de *registros,* ou seja, todos os operando se situam nos registradores de propósito geral e são executadas em ULA. Assim, engloba instruções: add, addu, and, break, div, divu, jalr, jr, mfhi, mflo, mthi, mtlo, mult, multu, nor, or, sll, sllv, slt, sltu, sra, srav, srl, srlv, sub, subu, syscall e xor.

As instruções são divididas em seis campos, como na figura 1.

FIGURA 1 – Tipo R

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

Fonte: Produzido pelo autor

O campo *op,* de 6 bits, designa o código de operação e todas as instruções de tipo R têm valor 0. O campo rs, de 5 bits, significa *source register* e designa o primeiro registrador fonte. O campo rt, de 5 bits, tem sua nomenclatura devido a letra t que segue da letra s e designa o segundo registrador fonte. O campo rd, de 5 bits, designa o registrador de destino, onde o resultado da operação a ser armazenado. O campo

shamt significa *shift amount* e representa a quantidade de bits a ser deslocada. Por úlitmo, o campo funct, de 6 bits, designa um tipo de operação como add, addu etc e estende o *opcode*.

Para a instrução add $t0, $t1, $t2, por exemplo, representa-se 000000 01001 01010 01000 00000 100000 como na figura 2.

FIGURA 2 – Exemplo de tipo R

op	rs	rt	rd	shamt	funct
000000	01001	01010	01000	00000	100000

Fonte: Produzido pelo autor

E equivale a breve à representação binária de figura 3.

FIGURA 3 – Exemplo simplificado de tipo R

op	rs	rt	rd	shamt	funct
0	$t1 = $9	$t2 = $10	$t0 = $8	0	add

Fonte: Produzido pelo autor

1.5.2 Tipo I

As instruções de tipo I (*immediate*) são instruções que envolvem registradores e um operando imediato (ou constante). São instruções de aritméticas imediatas, instruções lógicas, instruções de salto condicional e instruções de movimentação de dados. Assim, engloba instruções: addi, addiu, andi, beq, bgez, bgtz, blez, bltz, bne, lb, lbu, lh, lhu, lui, lw, lwc1, ori, sb, slti, sltiu, sh, sw, swc1 e xori.

As instruções são divididas em quatro campos como na figura 4.

FIGURA 4 – Tipo I

6 bits	5 bits	5 bits	16 bits
op	rs	rt	imm

Fonte: Produzido pelo autor

As instruções do tipo I possui três operandos, sendo dois registradores e um dado imediato.

Da mesma maneira que no tipo R, o campo op tem 6 bits e designa o *opcode*

e indica a operação a ser realizada. Igualmente, os campos rs e rt tem 5 bits e indicam registrador fonte e registrador de destino, respectivamente.

O campo imm (imediato) é interpretado de diferente maneira a depender de diferentes instruções. Para um constante, pode assumir um valor entre -2^{15}e $+2^{15} - 1$. Para um endereço, o *offset* é adicionado para o endereço base em *rs.*

Para a instrução addi $s1, $s2, 10, por exemplo, representa-se 001000 10010 10001 0000000000001010 como na figura 5.

FIGURA 5 – Exemplo de tipo I

op	rs	rt	Valor imediato
001000	10010	10001	0000000000001010

Fonte: Produzido pelo autor

E que equivale a figura 6.

FIGURA 6 – Exemplo simplificado de tipo I

op	rs	rt	Valor imediato
8	$s1	$s2	10

Fonte: Produzido pelo autor

1.5.3 Tipo J

As instruções de tipo J se trata das instruções de salto e engloba instrução j *(jump)* e *jal (jump and link).* É dividido em dois campos, como mostrado na figura abaixo.

FIGURA 7 – Tipo J

6 bits	26 bits
op	endereço

Fonte: Produzido pelo autor

O campo op, de 6 bits, de maneira similar aos outros tipos, designa o código de operação; neste caso, mais especificamente, qual tipo de salto. O campo endereço de 26 bits designa o endereço alvo do salto. Quando é executado, o endereço de 32

bits é formado concatenando os quatro bits mais significativos do PC e os 26 bits do campo endereço e dois 0.

Para a instrução j 10, por exemplo, representa-se 000010 00000000000000000000001010 como na figura 8.

FIGURA 8 – Exemplo tipo J

op	endereço
000010	00000000000000000000001010

Fonte: Produzido pelo autor

1.5.4 Instruções de coprocessador

As instruções de coprocessador são instruções para a manipulação de pontos flutuantes e engloba instruções: add.s, cvt.s.w, cvt.w.s, div.s, mfc1, mov.s, mtc1, mul.s e sub.s.

Os campos são divididos em 6 campos, similarmente ao tipo R, como mostrado na figura abaixo, porém diferente no seu uso.

FIGURA 9 – Instruções de pontos flutuantes

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	format	ft	fs	fd	funct

Fonte: Produzido pelo autor

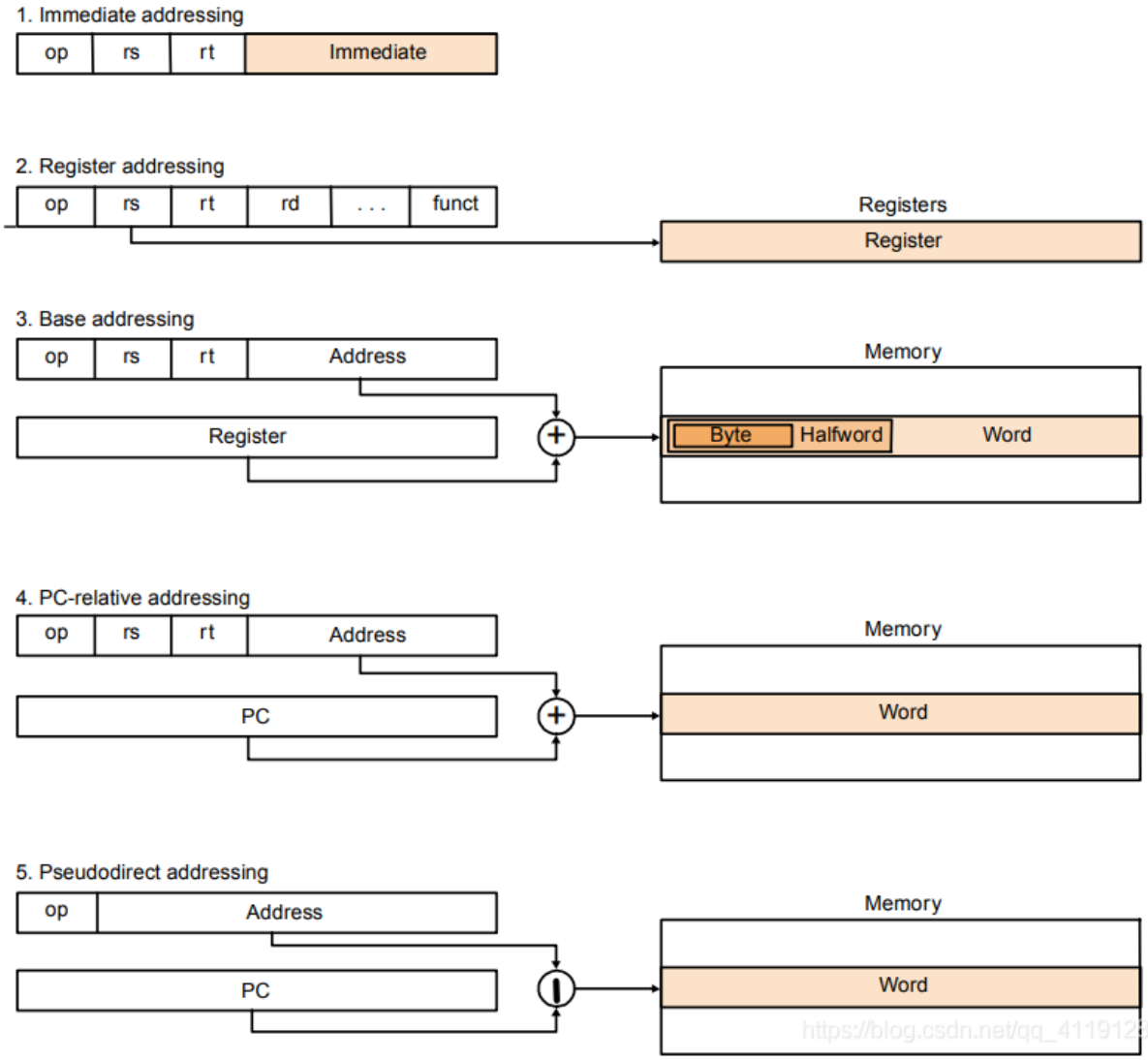
Os campos op, ft, fs, fd e funct tem funções equivalentes ao tipo R. E o campo format é utilizado para especificar se é de precisão simples, precisão dupla ou ponto fixo. As instruções mfc1 e mtc1 utilizam ele para estender o campo função.

1.6 MODOS DE ENDEREÇAMENTO

Os operandos podem ser buscados em três locais: diretamente na instrução, no registrador ou na memória. Para isso, o MIPS utiliza cinco modos de endereçamento**:** endereçamento por registradores, endereçamento imediato, endereçamento por registrador base, endereçamento PC relativo e endereçamento pseudo-direto.

São representados graficamente na figura 10.

FIGURA 10 – Modos de endereçamento MIPS



Fonte: https://blog.csdn.net/qq_41191231

Em endereçamento imediato, o operando já está explícito na instrução no campo op de 16 bits. Assim, não requer nenhum acesso à memória e é rápida. Porém o valor não cabe no tamanho de uma palavra.

O endereçamento por registradores utiliza somente registradores para encontrar os operandos das instruções. As instruções do tipo R fazem parte deste modo.

O endereçamento registrador base é utilizado nas instruções de load e store, pois há a necessidade de acessar a memória. O endereço efetivo é calculado somando o valor do registrador base e o imediato de 16 bits.

O endereçamento PC relativo é utilizado nas instruções de desvio. Para o cálculo de endereço efetivo, soma-se o valor imediato de 16 bits ao PC.

O endereçamento pseudo-direto é utilizado nas instruções de *jump*. Para o cálculo de endereço efetivo, o valor imediato é multiplicado por 4, e concatenado aos 4 bits mais significativos do PC.

1.7 MEMÓRIA

Devido à presença de pequeno número de registradores internos, alguns dados (geralmente os de menor uso) são armazenados na memória. No entanto, como RAM é externo, é necessário um número maior de ciclos de clock e maior tempo de acesso.

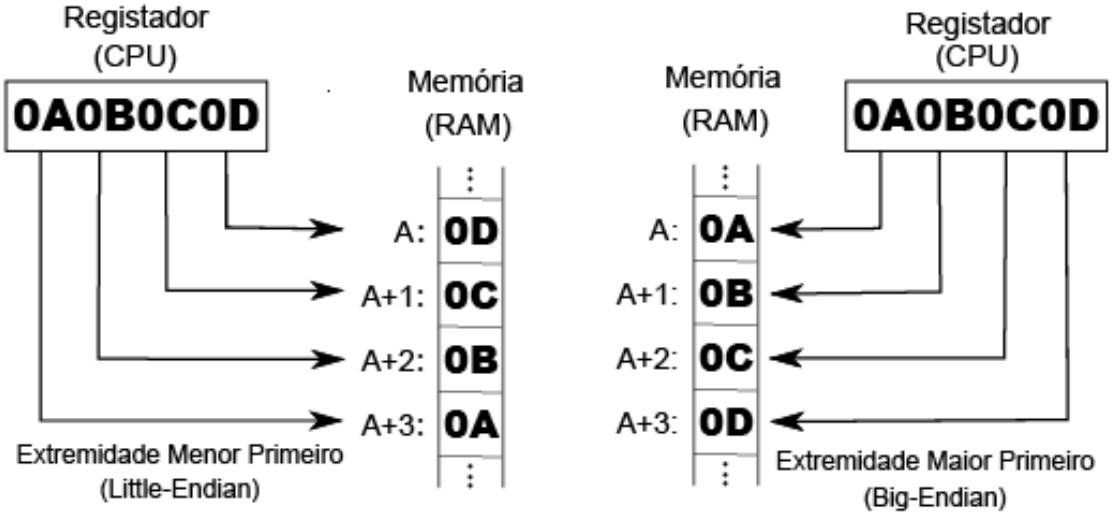
O acesso à memória ocorre somente para fazer *load* (lw) de memória para registradores e *store* (sw) de memória para registradores de dados.

Memória de MIPS é byte-endereçável, ou seja, cada endereço de memória são 8 bits. Como uma palavra (32 bits) ocupa 4 bytes, assim, os endereços devem ser múltiplos de 4.

1.8 *ENDIANNESS*

O MIPS utiliza representação *big endian,* ou seja, o byte mais significativo é armazenado no menor endereço de byte, como ilustrado na figura abaixo.

FIGURA 11 – Comparação *big endian* e *little endian*



Fonte: https://en.wikipedia.org/wiki/Endianness

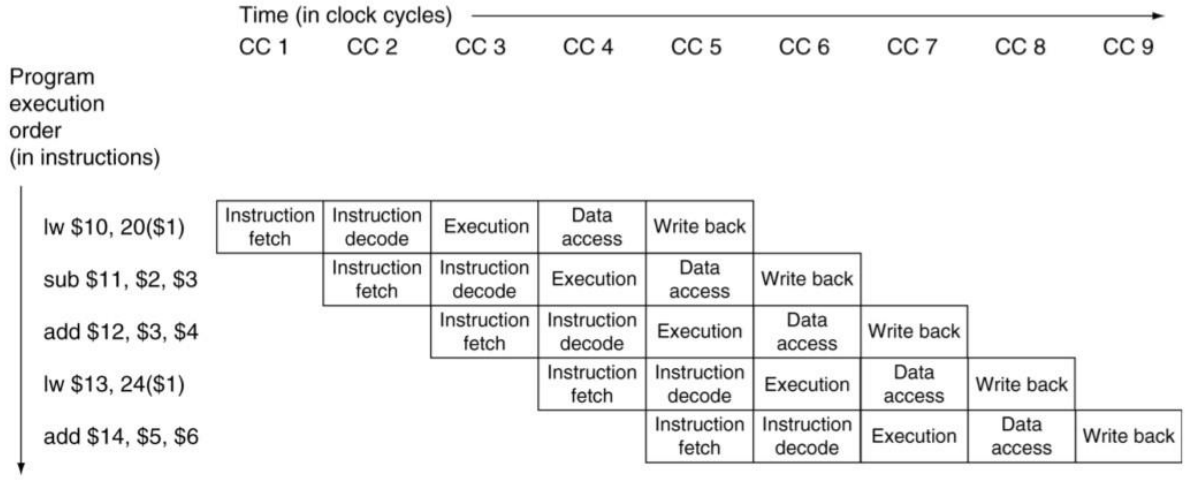
Esse formato é lenta para cálculos aritméticas como adição e subtração pois pode ocorrer um *carry* do byte menos significativo para byte mais significativo. No entanto, o *big endian* é mais rápido que *little endian* quando se trata de operações de comparação, pois a parte mais significativa do número está situada no começo da memória.

1.9. PIPELINE

Devido à ineficiência de monociclo do processador, foi introduzido o conceito de pipeline. Em uma implementação com pipeline, uma instrução é inicializada antes da outra terminar completamente. Assim, em comparação, tem um tempo total de execução menor.

O diagrama abaixo representa como uma implementação com pipeline ocorre ao decorrer do tempo.

FIGURA 12 – Diagrama de pipeline



Fonte: https://blog.kakaocdn.net/dn/cucRF0/btqEO0N4XHU/LIcVFRq8ZzKrr0mCmgsY41/img.jpg

1.9.1 Estágios

O MIPS possui cinco estágios de pipeline: IF, ID, EX, MEM e WB.

O estágio IF *(Instructioin fetch)* busca uma instrução da memória. Utilizando o PC como endereço, adiciona 4 ao PC e armazena esse valor.

O estágio ID *(Instruction decode)* lê e decodifica a instrução e sinais de controle

correspondentes são ativadas. Como as instruções do MIPS são regulares, a leitura e decodificação ocorrem ao mesmo tempo.

O estágio EX *(execution)* executa a operação. Se a instrução é de memória, o endereço efetivo é calculado e se é de salto condicional, a condição é verificada e o endereço alvo é calculado.

O estágio MEM *(memory)* acessa a memória, de acordo com a instrução load ou store.

Por último, o estágio WB *(write back)* escreve o resultado no registrador. Se a instrução for load, escreve o valor de memória para o registrador equivalente; se for uma operação de ALU, escreve o valor no registrador.

1.9.2 Características

Todas as instruções tem mesma largura pois facilita a busca de instrução no estágio IF e a decodificação no estágio ID.

Poucos formatos de instruções pois as instruções possuem mesma posição dos registradores, dessa maneira, pode executar leitura e decodificação simultaneamente.

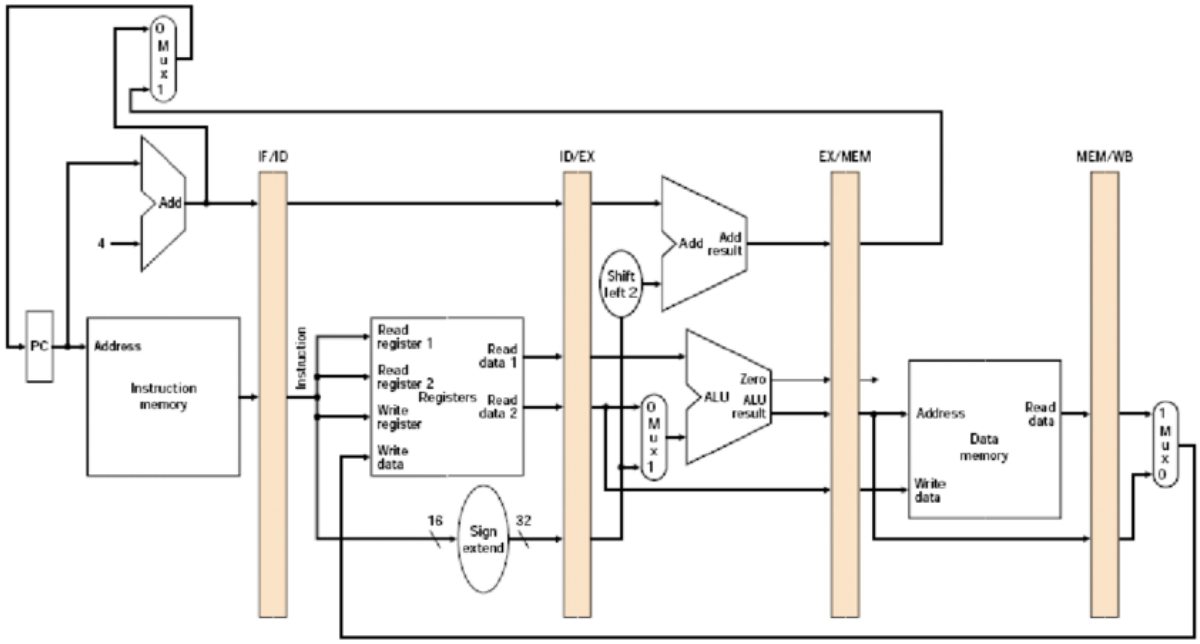
Apenas load e store podem acessar a memória pois se fosse possível de executar operações lógico-aritméticas diretamente dos operandos na memória, haveria necessidade de os estágios EX e MEM serem expandidos para estágios de endereço, memória e execução.

Os operandos estão alinhados na memória, portanto, não precisa ser acessado mais de uma vez. As transferências devem acontecer em um único ciclo.

1.9.3 *Datapath*

O MIPS apresenta caminho de dados conforme a figura 13.

FIGURA 13 – *Datapath* com pipeline



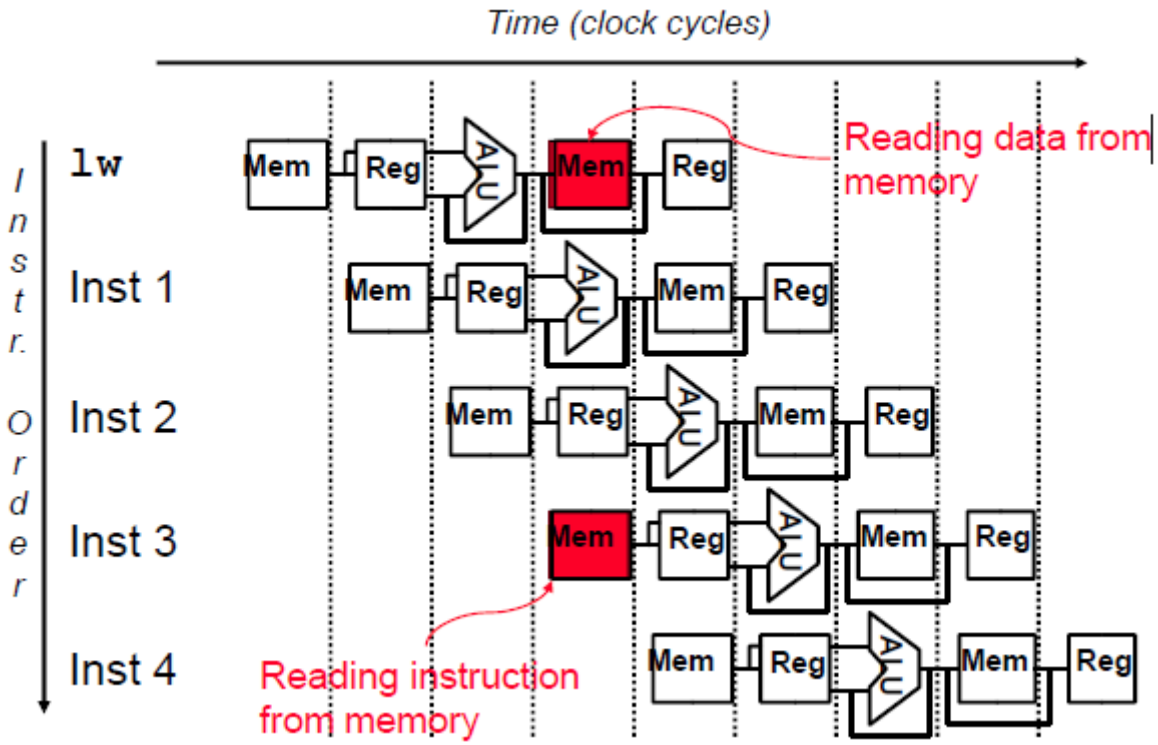
Fonte: https://www.researchgate.net/profile/Juan-Gomez-Luna/publication/252059873/figure/fig2/AS:667858463903745@1536241274055/Pipelined-version-of-the-MIPS-datapath.png

1.9.4 Hazard

Na figura 12, é assumido que todos os estágios são executados sem interrupção. No entanto, o hazard de pipeline ocorre quando o pipeline não pode executar o próximo estágio no seu ciclo de clock de forma contínua. Existem três tipos de hazard de pipeline que podem ocorrer: hazard estruturais, hazard de dados e hazard de controle.

O hazard estruturais ou hazard de recurso ocorre quando duas ou mais instruções que já estão no pipeline precisam do mesmo recurso. Pode ser solucionado gerando um sinal de stall ou adicionando mais hardwares.

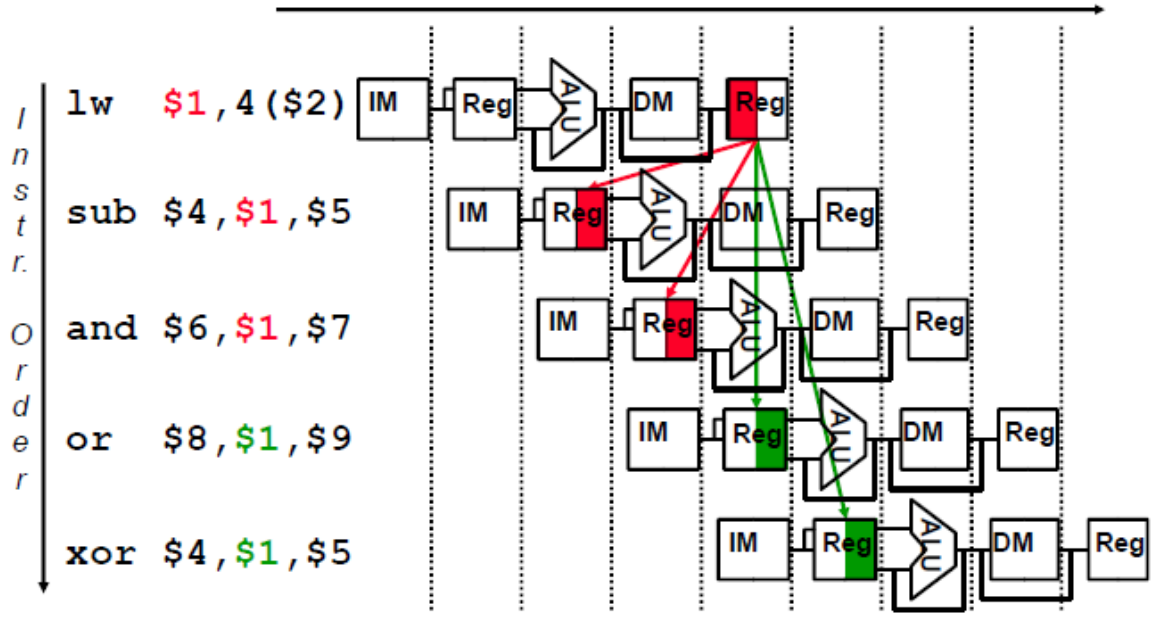
FIGURA 14 – Hazard estruturais



Fonte: https://blog.daum.net/goaxhflgoaxhfl/5

O hazard de dados ocorre quando há um conflito no acesso de um local de operando. Também pode ser solucionando esperando ou fazendo *forwarding*.

FIGURA 15 – Hazard de dados



Fonte: https://blog.daum.net/goaxhflgoaxhfl/5

O hazard de controle ou hazard de desvio ocorre quando o pipeline toma uma decisão errada ao prever um desvio ou quando há modificações no PC. E pode ser solucionado prevendo melhor o desvio.

1.10 FILOSOFIA MIPS

As características básicas de MIPS podem ser explicadas suscintamente com base na sua filosofia de design.

A simplicidade favorece a regularidade: as instruções do MIPS possuem uma regularidade. São de tamanho fixo de 32 bits, possuem mesma quantidade de operandos – duas de registrador fonte e uma de registrador de destino, e opcode sempre está situado nos 6 bits mais significativos.

Tornar o caso comum rápido: o MIPS possui instruções que são rápidas e mais utilizadas. As instruções complexas são divididas em partes e possuem operandos imediatos.

Menor é melhor: o MIPS possui quantidade de instruções, registradores e modos de endereçamento determinado.

Bom projeto demanda compromissos: há três formatos de instruções – R, I e J.

2 DESCRIÇÃO DO PROBLEMA E CÓDIGO ALTO NÍVEL DA SOLUÇÃO

2.1 DESCRIÇÃO DO PROBLEMA

O problema escolhido pelo nosso grupo é reconhecer se uma string é ou não palíndromo. Palíndromos são palavras ou frases que podem ser lidas da esquerda para direita ou da direita para a esquerda mantendo o mesmo significado. Para frases os espaços não contam logo devem ser ignorados ao percorrer a string.

2.2 CÓDIGO ALTO NÍVEL

Código em c que soluciona o problema:

```c
void removerEspacos(char* str) {
    char* aux = str;
    do {
        while (*aux == ' ') {
            *aux++;
        }
    } while (*str++ = *aux++);
}

int checarPalindromo(char* str)
{
    int start = 0;
    int end = strlen(str) - 1;

    if(end <= 1)
        return 0;

    while(start != end && start < end) {
        if(str[start] != str[end]) {
            return 0;
        }
        start++;
        end--;
    }

    return 1;
}
```

3. CÓDIGO EM ASSEMBLY

```
.text
.globl main
main:
	# Imprime $textoInicial na tela
	la $a0,$textoInicial
	li $v0,4
	syscall

	# Recebe uma string do usuário
	la $a0,$buffer
	li $a1,50
	li $v0,8
	syscall

	# Remove espaços da string
	jal removerEspacos

	# Calcula e guarda o tamanho da string em $a1
	la $t0,($a0)
	li $a1,0
	loopTamanho:
		lb $t2,($t0)
		addiu $t0,$t0,1
		addiu $a1,$a1,1
		bne $t2,'\n',loopTamanho
	subiu $a1,$a1,1			# Retira o byte de newline do contador

	ble $a1,2,menorQueDois		# Termina o programa caso a string tenha menos que
3 caracteres

	subiu $a1,$a1,1			# Deixa o contador no formato para uso em arrays
	jal checarPalindromo		# Chama a função que checa se a string é
palíndromo
	beq $a2,0,false			# Condição do resultado da função

	# Caso a string for palíndromo, retorna um texto correspondente e termina o
programa
	true:
		la $a0,$textoPalindromo
		li $v0,4
		syscall
		j end
	# Caso a string não for palíndromo, retorna um texto correspondente e termina
o programa
	false:
		la $a0,$textoNaoPalindromo
		li $v0,4
		syscall
		j end
```

```
	# Caso a string tiver menos que 3 caracteres, retorna um texto correspondente
e termina o programa
	menorQueDois:
		la $a0,$textoErro
		li $v0,4
		syscall
	# Termina o programa
	end:
		li $v0,10
		syscall

# Função que checa se a string guardada em $a0 com seu tamanho em $a1 é palíndromo,
retorna 1 caso for ou 0 caso contrário.
checarPalindromo:
	# Início da string
	la $t0,($a0)
	# Fim da string
	la $t1,($a0)
	addu $t1,$t1,$a1

	# Loop principal para checar se a string é palíndromo
	loopPalindromo:
		beq $t0,$t1,stringPalindromo
		bgt $t0,$t1,stringPalindromo

		lb $t2,($t0)
		lb $t3,($t1)

		bne $t2,$t3,stringNaoPalindromo

		addiu $t0,$t0,1
		subiu $t1,$t1,1

		j loopPalindromo

	# Caso a string for palíndromo, guarda 1 (True) em $a2 e retorna
	stringPalindromo:
		li $a2,1
		jr $ra

	# Caso a string não for palíndromo, guarda 0 (False) em $a2 e retorna
	stringNaoPalindromo:
		li $a2,0
		jr $ra




# Função que remove os espaços de uma string guardada em $a0
```

```
removerEspacos:
	# Guarda o endereço da string em dois registradores, um para percorrer os
espaços e o outro para editar a string
	la $t0,($a0)
	la $t2,($t0)

	loopEspacos:
		# Percorre pelos espaços com $t2 até chegar no próximo caractere
		condicaoEspaco:
			lb $t1,($t2)
			bne $t1,32,condicaoEspacoFim
			addiu $t2,$t2,1
			j condicaoEspaco

		condicaoEspacoFim:
		# Guarda o valor de $t2 em $t0 para todo caractere, assim removendo
todos os espaços da string
		lb $t1,($t2)
		sb $t1,($t0)

		addiu $t2,$t2,1
		addiu $t0,$t0,1

		lb $t1,($t2)
		bne $t1,0,loopEspacos

	# Adiciona o caractere null que pode faltar e retorna
	lb $t1,($t2)
	sb $t1,($t0)
	jr $ra

.data
$textoInicial: .asciiz "Digite a string: "
$textoErro: .asciiz "Uma string deve ter no mínimo 3 caracteres para ser um
palíndromo."
$textoPalindromo: .asciiz "A string inserida é palindromo"
$textoNaoPalindromo: .asciiz "A string inserida não é palindromo"
$buffer: .space 50
```

4. EXPLICAÇÃO DAS INSTRUÇÕES UTILIZADAS NO CÓDIGO ASSEMBLY

Load Address(la): Carrega o endereço de memória no registrador destino. É uma pseudo instrução traduzida em:

- **Load Upper Immediate(lui):** Carrega o valor constante recebido nos 16 bits superiores do registrador temporário.
- **OR Immediate(ori):** Realiza uma operação bitwise OR com o valor constante recebido e o registrador temporário para formar o endereço completo de 32 bits e guardá-lo no registrador destino.

Load Immediate(li): Carrega o valor constante recebido no registrador. É uma pseudo instrução traduzida em: Add Immediate Unsigned, que é explicada mais abaixo.

Load Byte(lb): Carrega um byte da memória do endereço recebido no registrador destino.

Store Byte(sb): Carrega um byte do registrador recebido no endereço de memória.

Add Unsigned(addu): Adiciona o conteúdo de ambos os registradores sem checar por overflow e guarda a resultado no registrador destino.

Add Immediate Unsigned(addiu): Adiciona o conteúdo de um registrador com o valor constante recebido sem checar por overflow e guarda o resultado no registrador destino.

Subtract Immediate Unsigned(subiu): Subtrai o conteúdo de um registrador com o valor constante recebido sem checar por overflow e guarda no registrador destino. É uma pseudo instrução traduzida em:

- **Load Upper Immediate(lui):** Carrega 16 bits superiores do valor constante no registrador temporário.
- **OR Immediate(ori):** Realiza uma operação bitwise OR com os 16 bits

inferiores do valor constante recebido e o registrador temporário para formar o valor constante completo de 32 bits e guardá-lo no registrador temporário.

- **Subtract Unsigned(subu):** Subtraí o valor do registrador temporário com o valor do registrador destino e guarda o resultado no registrador destino.

Jump(j): Pula para a posição indicada.

Jump Register(jr): Pula para a posição indicada pelo registrador.

Jump and Link(jal): Pula para a posição indicada e guarda a posição de retorno no registrador $ra.

Branch on Equal(beq): Pula para a posição indicada caso ambos os registradores forem iguais.

Branch on Not Equal(bne): Pula para a posição indicada caso ambos os registradores não forem iguais.

Branch on Less Than Equal(ble): Pula para a posição indicada caso o primeiro registrador for menor ou igual ao segundo registrador.

Branch on Greater Than(bgt): Pula para a posição indicada caso o primeiro registrador for maior que o segundo registrador.

syscall: Realiza uma chamada de sistema baseado no valor do registrador $v0.

REFERÊNCIAS

DANDAMUDI, S. **Guide to RISC Processors: For Programmers and Engineers.** 1. ed. New York, United States: Springer. 2005.

HARRIS, D.; HARRIS, S.; **Digital Design and Computer Architecture.** 2. ed. Waltham, United States: Elsevier. 2013.

PATTERSON, D.; HENNESSY, J. **Organização e Projeto de Computadores.** Tradução: Daniel V. 5. ed. Rio de Janeiro: Elsevier. 2017.

STALLINGS, W. **Arquitetura e Organização de Computadores.** Tradução: Sérgio N. 10. ed. São Paulo: Pearson. 2017.

WAVE COMPUTING**. Introduction to the MIPS32 Architecture v6.01**. Disponível em: <https://www.mips.com/?do-download=introduction-to-the-mips32-architecture-v6-01>. Acesso em 19 jul. 2021.

WAVE COMPUTING. **MIPS32 Instruction Set Quick Reference v1.01**. Disponível em: <https://www.mips.com/?do-download=mips32-instruction-set-quick-reference-v1-01>. Acesso em 19 jul. 2021.