UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

ANA BEATRIZ RODRIGUES CHAGAS, N° USP: 12734130
ANDRÉ PALACIO BRAGA TIVO, N° USP: 13835534

PROBLEMA 9 (SOMA DE BITS)

Exercício Programa de Organização e Arquitetura de Computadores I, 2023.1

SÃO PAULO - SP
2023

ANA BEATRIZ RODRIGUES CHAGAS
ANDRÉ PALÁCIO BRAGA TIVO

PROBLEMA 9 (SOMA DE BITS)

Exercício Programa de Organização e Arquitetura de Computadores I, 2023.1

Trabalho em dupla apresentado à disciplina “Organização e Arquitetura de Computadores I” como parte dos requisitos obrigatórios para aprovação.

Docente: Gisele da Silva Craveiro

SÃO PAULO - SP

2023

ANA BEATRIZ RODRIGUES CHAGAS
ANDRÉ PALACIO BRAGA TIVO

PROBLEMA 9 (SOMA DE BITS)

Exercício Programa de Organização e Arquitetura de Computadores I, 2023.1

Trabalho em dupla apresentado à disciplina “Organização e Arquitetura de Computadores I” como parte dos requisitos obrigatórios para aprovação.

São Paulo - SP, __ de ____ de 2023

BANCA EXAMINADORA

_____

Docente: Gisele da Silva Craveiro

RESUMO

Neste relatório, será destrinchado de forma concisa e detalhada todas as informações referentes à implementação de uma função que recebe três bits, ou seja, três inteiros entre 0 e 1, e devolve um bit soma representando a adição dos três, e um novo bit "vai-um" armazenado em um ponteiro passado por parâmetro. Além do detalhamento das etapas que envolvem a implementação da função feita para a resolução do problema passado, também será tratado sobre o conhecimento teórico em relação à organização e arquitetura MIPS, bem como a apresentação do código em Assembly e as instruções utilizadas no exercício programa.

Palavra-chave: bit, soma, MIPS, Assembly.

ABSTRACT

In this report, all the information regarding the implementation of a function that receives three bits, that is, three integers between 0 and 1, and returns a sum bit representing the addition of the three bits, and a new bit “vai-um” stored in a pointer passed by parameter. In addition to detailing the steps that involve the implementation of the function performed to solve the previous problem, the theoretical knowledge regarding the MIPS organization and architecture will also be discussed, as well as the presentation of the code in Assembly and the instructions used in the program exercise.

Keywords: bit, sum, MIPS, Assembly.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1. ORGANIZAÇÃO E ARQUITETURA MIPS

1.1. Definição de MIPS

O MIPS é uma Arquitetura de Conjunto de Instruções (*Instruction Set Architecture – ISA*), desenvolvida pela antiga empresa MIPS Computer Systems, atualmente chamada de MIPS Technologies. Seu nome é um acrônimo de *Microprocessor Without Interlocked Pipeline Stages* (Microprocessador Sem Estágios Intertravados de Pipeline).

Em 1984, um grupo de pesquisadores da Universidade de Stanford fundou a MIPS Computer Systems na intenção de lidar com microprocessadores que possuíam arquitetura RISC (*Reduced Instruction Set Computer* - "Computador com um conjunto reduzido de instruções").

Nesse sentido, John Leroy Hennessy, cientista da computação e professor universitário norte-americano, fez parte da criação da empresa, assim como da história da evolução dos microprocessadores, o que pode ser comprovado com a leitura de seus livros e artigos científicos de renome.

1.2. Princípios seguidos em um projeto que utiliza MIPS

Com a intenção de conciliar a simplicidade de hardware e a disposição de aparatos tecnológicos de qualidade aos usuários programadores, o MIPS dispõe de quatro princípios de projeto que auxiliam na definição de sua arquitetura. Sendo eles:

1.2.1. Simplicidade é favorecida pela regularidade

O primeiro princípio diz que quanto maior for a simplicidade do que é solicitado ao computador, mais simples será a arquitetura, fazendo-o mais eficiente e menos custoso. Além disso, a simplicidade das instruções acarreta na descomplexidade do seu processo de decodificação. Esse preceito acarreta em características como instruções de tamanho fixo, poucos formatos de instruções e códigos de operação que sempre utilizam os primeiros 6 bits.

1.2.2. Quanto menor, mais rápido

O segundo princípio considera a hierarquia de memória e algumas questões eletrônicas. Isso ocorre porque a velocidade de transmissão de dados e instruções está concatenada com o tamanho das conexões disponíveis.

A disposição dos componentes, aliado aos seus tamanhos, influencia na velocidade e maneira com que o computador opera. Como por exemplo, as operações lógicas e aritméticas são executadas com base em operandos, pois esses estão no topo da hierarquia de memória, logo, possuem baixa capacidade e uma alta velocidade.

Essa retórica implica em características como repertório de instruções limitado, quantidade de registradores pequena e número reduzido de modos de endereçamento. Todas essas propriedades visam um acesso individual mais rápido, já que os sinais eletrônicos levam um tempo menor para percorrer caminhos mais curtos do circuito.

1.2.3. Torne rápido o caso mais comum

O terceiro princípio tem como base a “Lei Amdahl”, que por sua vez, é utilizada para alcançar a máxima melhora possível para um sistema em geral quando apenas uma única parte dele é aprimorada.

Dessa forma, segundo esse regimento, o aperfeiçoamento da performance nos casos mais comuns costuma beneficiar o funcionamento geral do sistema em questão, não apenas o processo envolvido na ocorrência de uma atividade específica.

A aplicação desse princípio pode ser vista na existência de instruções que contém operandos, como por exemplo, o registrador fixo que armazena o número zero. Sua criação é justificada pelo fato desse algarismo ser usado frequentemente, logo, a presença de um registrador que permita o seu acesso praticamente imediato desfaz a necessidade da atribuição constante de registradores temporários à esse valor.

1.2.4. Bom projeto requer boas escolhas (compromissos)

O quarto princípio remete à ideia de determinar garantias no projeto. Dessa forma, a principal característica do MIPS que está relacionada com essa regra é o fato de que diferentes formatos de instruções com campos em comum complicam a decodificação, porém permitem a existência de instruções de tamanho fixo (todas com 32 bits). Isso é em parte favorável porque mantém o compromisso no projeto e estabelece um padrão de formato.

1.3. Organização da memória no MIPS

No MIPS, apesar de uma palavra ter 32 bits (4 bytes), o endereçamento é feito por bytes. Isso é uma vantagem, pois dessa forma é possível acessar bytes, metade de palavras ou palavras inteiras, fazendo do endereçamento altamente flexível. Porém, para que haja maior organização, é necessário que as palavras estejam escritas na memória de modo alinhado, ou seja, uma instrução ou dado por linha, nem que seja necessário complementar o conteúdo dela com zeros. Essa característica de alinhamento no MIPS faz com que os endereços estejam organizados em múltiplos de quatro, pois 2^4 é o tamanho de uma palavra. Outra propriedade interessante é o fato desse modelo ser *Big Endian*, ou seja, o byte mais significativo está no endereço menos significativo da palavra. Esse fator é importante quando está sendo feita a manipulação de endereços de memória, por exemplo.

1.4. Registradores no MIPS

1.4.1. Definição

Os registradores são as estruturas responsáveis por armazenar as informações dentro do processador. Pelo fato de serem memórias de tamanho reduzido, custo elevado e rapidez avantajada, encontram-se no topo da hierarquia de memória.

1.4.2. Utilidade

Pode-se citar duas tarefas principais como as utilidades primordiais dos registradores, sendo elas:

1.4.2.1. Diminuição do acesso à memória

Ou seja, eles fazem com que seja possível o usuário programador diminuir a quantidade de acessos à memória, já que os registradores são visíveis e acessíveis para eles.

1.4.2.2. Controle de operação e execução

Logo, eles provêm o controle da operação do processador e a execução dos programas no computador (os registradores responsáveis por essa tarefa são chamados de registradores de estado).

1.4.3. Tipos

O MIPS possui 32 registradores diferentes. Eles podem ser representados pelo símbolo “$” seguido pelo seu número, indo de 0 à 31 consecutivamente, ou então, por um nome equivalente à sua função. Quanto aos seus tipos, podem ser temporários ou executar uma tarefa específica. Dessa forma, os registradores do MIPS são:

1.4.3.1. $zero

Também chamado de $0, é um registrador acessível que retorna o valor de constante zero.

1.4.3.2. $at *(Assembler Temporary)*

Também chamado de $1, é um registrador acessível reservado pelo assembler.

1.4.3.3. $v0~$v1 *(Values)*

Conhecidos como $2 e $3, respectivamente, são registradores acessíveis que armazenam resultados de funções e valores das expressões de avaliação.

1.4.3.4. $a0~$a3 *(Arguments)*

Conhecidos por $4 a $7, respectivamente, são registradores acessíveis utilizados como argumentos para subrotinas (são os primeiros quatro parâmetros dela), mas não são preservados em chamadas de *procedures*.

1.4.3.5. $t0~$t9 *(Temporaries)*

Os registradores $t0 a $t7, conhecidos por $8 a $15, são acessíveis e utilizados para armazenar valores temporários. Sendo assim, as subrotinas podem escolher usá-los salvando-os ou não. Esses, diferentemente dos registradores $a0 a $a3, são preservados nas chamadas de *procedures.* Já os registradores $t8 e $t9, também chamados de $24 e $25, são usados em adição aos $t0 a $t7.

1.4.3.6. $s0~$s7 *(Saved Values)*

Os registradores $s0 a $s7, conhecidos como $16 a $23, são acessíveis e utilizados para valores salvos. Uma subrotina que os utiliza, deve salvar os valores originais e restaurá-los antes de terminar. Ademais, eles são preservados em chamadas de *procedures*.

1.4.3.7. $k0~$k1

Também conhecidos como $26 e $27, consecutivamente, são registradores acessíveis, porém, reservados para o uso no tratamento de interrupções .

1.4.3.8. $gp *(Global Pointer)*

Também conhecido como $28, é responsável por apontar para o meio do bloco de 64k de memória no segmento de dados estáticos.

1.4.3.9. $sp *(Stack Pointer)*

Também conhecido como $29, é responsável por apontar para o topo da pilha.

1.4.3.10. $fp *(Frame Pointer)*

Também chamado de $30, é responsável por apontar para o início do *frame* da pilha. Além disso, é preservado na chamada de *procedures.*

1.4.3.11. $ra *(Return Address)*

Conhecido como $31, é um registrador não acessível que armazena o endereço de retorno quando há uma chamada de procedimento.

1.4.3.12. MAR *(Memory Address Register)*

O MAR *(Memory Address Register)* não é um registrador acessível, porém está conectado ao barramento de endereço do sistema. Ele é o responsável por especificar o endereço no qual acontecerá uma operação de leitura ou escrita.

1.4.3.13. MBR *(Memory Buffer Register)*

O MBR (Memory Buffer Register), assim como o MAR, não é acessível. Esse, por sua vez, está conectado ao barramento de dados do sistema e é responsável por armazenar o valor a ser guardado ou lido da memória.

1.4.3.14. IR *(Instruction Register)*

Assim como os dois últimos, não é um registrador acessível. Sua função é armazenar a última instrução lida.

1.4.3.15. PC *(Program Counter)*

O PC (Program Counter) também não é um registrador acessível diretamente. Sua finalidade é guardar o endereço da próxima instrução a ser lida.

1.4.3.16. HI

O HI não é um registrador acessível. Sua finalidade é armazenar o resto de uma divisão.

1.4.3.17. LO

O LO não é um registrador acessível. Sua finalidade é armazenar o resultado de uma divisão inteira.

1.5. Modos de endereçamento no MIPS

Os modos de endereçamento detalham as maneiras distintas que os operandos podem ser informados para as instruções. Nesse sentido, o MIPS dispõe de cinco modos de endereçamento, sendo eles:

1.5.1. Endereçamento imediato

Diretamente pela instrução, o operando é informado como uma constante, sendo assim, não ocorre interação com a memória para recuperá-lo.

1.5.2. Endereçamento em registrador

É informado o *id* do registrador que contém o dado requisitado. Da mesma maneira, não é necessário acesso à memória principal para recuperá-lo, pois apenas é preciso acessar o registrador discernido.

1.5.3 Endereçamento de base ou deslocamento

A instrução contém uma constante imediata (um operando que é um valor imediato) e um registrador que armazena um endereço.

O valor da constante será utilizado como *offset* para o endereço base. Em seguida, os resultados serão somados para indicar o endereço desejado.

1.5.4 Endereçamento relativo ao Program Counter

Muito similar ao endereçamento de base, diferenciando-se pelo fato que o próprio *Program Counter* (PC) assumirá o papel de endereço base.

Esse modo de endereçamento baseia-se no princípio da localidade de referências, no sentido de que é extremamente provável que as instruções e dados requeridos por um programa estejam armazenados próximos a ele mesmo.

1.5.5 Endereçamento pseudo direto

A instrução contém somente uma constante imediata. Endereçamento dá-se por meio dos bits da constante sendo concatenados com os bits mais altos do *Program Counter* (PC).

1.6. Instruções no MIPS

A arquitetura MIPS possui instruções com tamanho igual a 32 bits. Ademais, as possibilidades para o seus formatos são três, todas elas englobando da mesma arquitetura, porém com organizações distintas, aliadas aos seus objetivos.

1.6.1. Formatos das instruções

1.6.1.1. Instruções do tipo R

São instruções que trabalham especificamente com registradores. Elas abrangem todas as instruções aritméticas. Seus campos e respectivos tamanhos dão-se de seguinte maneira:

ILUSTRAÇÃO 1 - FORMATO DE INSTRUÇÃO DO TIPO R

opcode	*rs*	*rt*	*rd*	*shamt*	*funct*
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Fonte: Autoria própria

1.6.1.2. Instruções do tipo I

São instruções que trabalham com constantes imediatas e registradores. Dentre suas tarefas, é abrangido as instruções de transferência de dados, as instruções que utilizam operandos imediatos e as *branches.* Seus campos e respectivos tamanhos dão-se de seguinte maneira:

ILUSTRAÇÃO 2 - FORMATO DE INSTRUÇÃO DO TIPO I

opcode	*rs*	*rt*	*end / const*
6 bits	5 bits	5 bits	16 bits

Fonte: Autoria própria

1.6.1.3. Instruções do tipo J

São instruções que trabalham apenas com constantes imediatas, abrangendo todas as instruções de *jump.* Seus campos e respectivos tamanhos dão-se de seguinte maneira:

ILUSTRAÇÃO 3 - FORMATO DE INSTRUÇÃO DO TIPO I

opcode	*end*
6 bits	26 bits

Fonte: Autoria própria

1.7. Aplicações do MIPS

Os processadores que utilizam a arquitetura MIPS ainda estão à venda. Eles possuem uma fatia de mercado avantajada no que diz respeito à comercialização no núcleo de embarcados.

Dentre as suas aplicações, destacam-se pela utilização mais comum o uso em eletrônica de consumo, impressoras, equipamentos de rede e de armazenamento.

Alguns aparelhos famosos que usufruíram de microprocessadores MIPS em sua criação foram o Nintendo 64, Sony PlayStation e os roteadores Cisco.

2. DESCRIÇÃO DO PROBLEMA E CÓDIGO DE ALTO NÍVEL DA SOLUÇÃO

2.1. Descrição do problema 9 (soma de bits)

A implementação deste exercício programa consiste em uma função com protótipo `void somabit (int b1, int b2, int *vaium, int *soma)`. Nela, são passados três bits por parâmetro (`int b1, int b2` e `int *vaium`). O objetivo final é devolver um bit soma, no caso o valor armazenado no ponteiro `int *soma` passado por parâmetro, contendo o resultado da adição dos três bits informados. Ademais, é retornado um novo bit "vai-um" armazenado no ponteiro `int *vaium`.

Tendo em vista o objetivo de implementar a função citada, com a finalidade de executar essa tarefa utilizando a linguagem de montagem Assembly, foi efetuada uma visualização preliminar da lógica necessária utilizando uma linguagem de alto de nível, nesse caso, a linguagem C. Tendo em vista que o código realizado em alto nível casualmente seria utilizado como arquétipo para a realização da função em nível intermediário, o código em C simula as limitações presentes no Assembly (no caso, poucas operações sendo feitas por instrução).

Levando em consideração `int b1, int b2` e `int *vaium` como os bits que participarão da operação de adição, a função executará duas linhas de código para atingir o seu objetivo final estipulado. A 1° linha de código é `*soma = (b1 ^ b2) ^ *vaium`. Nela, basicamente ocorre atribuição de um valor no endereço apontado por `soma`. Na linguagem C, o operador indicado pelo símbolo `^` representa a operação de OU exclusivo bit a bit (também conhecido como *bitwise XOR*). Portanto, na 1° linha, `(b1 ^ b2)` calcula o resultado da soma dos bits `b1` e `b2` sem considerar o bit `*vaium`. Posteriormente, `(b1 ^ b2) ^ *vaium` realiza novamente uma operação *XOR* bit a bit com o valor de `*vaium`, chegando ao resultado final da adição. Esse valor, por sua vez, é armazenado na variável apontada por `soma`.

Já a 2° linha de código é `*vaium = (b1 & b2) | ((b1 ^ b2) & *vaium)`. Nela, o operador `&` representa a operação E bit a bit (também conhecida como *bitwise AND*), enquanto o operador `|` representa a operação

OU bit a bit (da mesma maneira, conhecida como *bitwise OR*). Nesta 2° linha, `(b1 & b2)` calcula o `*vaium` consequente da soma dos bits `b1 e b2`. Logo depois, `((b1 ^ b2) & *vaium)` checa se há algum `*vaium` a ser transmitido para a próxima posição. A operação *bitwise AND* produzirá um valor diferente de zero caso exista um `*vaium` sendo transmitido. Para finalizar a função e atingir o que foi proposto, o resultado das duas operações é combinado usando o *bitwise OR* e é armazenado na variável apontada por `vaium`.

2.2. Código de alto nível em C

ILUSTRAÇÃO 4 - CÓDIGO DE ALTO NÍVEL EM C

```c
void somabit(int b1, int b2, int *vaium, int *soma) {
    *soma = (b1 ^ b2) ^ *vaium;
    *vaium = (b1 & b2) | ((b1 ^ b2) & *vaium);
}
```

Fonte: Autoria própria

3. CÓDIGO EM ASSEMBLY DESENVOLVIDO

ILUSTRAÇÃO 5 - CÓDIGO DE NÍVEL INTERMEDIÁRIO EM ASSEMBLY MIPS

```
.data

      # Os valores encontrados no .data NÃO devem ser alterados

      # Foi utilizada uma estrutura de repetição para considerar todos
      # os casos possíveis de b1 e b2, que são:

      # b1 = 0, b2 = 0;
      # b1 = 1 e b2 = 0 (o inverso segue a mesma retórica)
      # b1 = 1 e b2 = 1

      # O valor de vaiUm não é mudado (soma de BITS não se espera vaiUm = 1)

      # Durante o programa é utilizada a pilha (stackpointer)
      # para atualizar os valores de vaiUm e soma (os retorna a zero)

      # O resultado da soma dos bits é impresso durante o programa
      # Assim como o valor de vaiUm e o valor dos Bits da soma
      # A saída esperada é composta por:
```

```
        # O valor binário a ser somado é: b1 + b2
        # O resultado da soma é: valorDaSoma
        # O resultado vai um é: valorDoVaiUm

        b1: .byte 0 #byte 1
        b2: .byte 0 #byte 2
        vaiUm: .byte 0 # vai um da soma
        soma: .byte 0 # resultado da soma
        resSoma: .asciiz "\nO resultado da soma é: "
        resVaiUm: .asciiz "\nO resultado vai um é: "
        valorSer: .asciiz "\nO valor binário a ser somado é: "
        espacoSoma: .asciiz " + "

.text
.globl main

        main:
        #TODO: fazer um for que altera o valor das variáveis

        lb, $a0, b1 # carrega b1
        lb, $a1, b2 # carrega b2
        move $s3, $a0 # salva b1 para ser comparado e alterado dentro do loop
        move $s4, $a1 # salva b2 para ser comparado e alterado dentro do loop
        # ambos em $a pois serão passados como argumentos

        loop:
        # começo da rotina de impressão do loop
        la $a0, valorSer
        li $v0, 4
        syscall

        move $a0, $s3
        li $v0, 1
        syscall

        la $a0, espacoSoma
        li $v0, 4
        syscall

        move $a0, $s4
        li $v0, 1
        syscall
        # fim da rotina de impressão do loop
        lb $t2, soma # pega a soma do .data
        lb $t3, vaiUm # pega o vaium do .data
        addiu $sp, $sp, -8 # aloca duas palavras no stack
        sw $t2, 4($sp) # salva a soma
        sw $t3, 0($sp) # salva o vai um

        move $a0, $s3 # carrega os valores, atualizados pelo loop, de b1 e b2
```

```
move $a1, $s4

jal somabit # mesma coisa que passar em C

# somabit(b1, b2, &soma, &vaium)
# como está sendo pedido no enunciado
# como é uma função void, registradores de resultado não são usados

lw $t0, 4($sp) # soma
lw $t1, 0($sp) # vaium
# pega os valores salvos no stack
# a seguinte rotina imprime os valores da Soma e do VaiUm
la, $a0, resSoma
li, $v0, 4
syscall

move $a0, $t0
li $v0, 1
syscall

la $a0, resVaiUm
li $v0, 4
syscall

move $a0, $t1
li $v0, 1
syscall

beq $s3, $zero, b1zero # altera  b1 para o valor 1
beq $s4, $zero, b2zero # altera b2 para o valor 1
beq $s3, $s4, continue # ao chegar nesse ponto do código, b1 = b2 = 1
j loop
continue:

addiu $sp, $sp, +8 # encerra/desaloca o espaço no stackpointer

end:
li $v0, 10 # rotina que encerra o programa
syscall

b1zero:
li $s3, 1 # altera o valor de b1 e reinicia o loop
j loop

b2zero:
li $s4, 1 # altera o valor de b2 e reinicia o loop
j loop
somabit:

# $s0 SOMA
# $s1 VAIUM
```

```
move $t1, $a0 # salva os argumentos nos registradores temporários
move $t2, $a1

lw $s0, 4($sp) #soma
lw $s1, 0($sp) #vaium
# ^ pega os valores de soma e vaium do stackpointer

xor $t0, $t1, $t2 # (b1 ^b2) = t0
xor $t0, $t0, $s0 # t0 ^ vaium
move $s0, $t0 # salva o valor em s0

and $t0, $t1, $t2 # (b1 & b2) = $t0
xor $t3, $t1, $t2 # (b1 ^ b2) = $t3
and $t3, $t3, $s1 # ($t3 & vaium) = $t3
or $s1, $t0, $t3 # ($t0 | $t3)

# salva os valores atualizados de soma e vaium no stack
sw $s0 4($sp) # salva a soma
sw $s1, 0($sp) # salva o vai um


jr $ra
```

Fonte: Autoria própria

4. EXPLICAÇÃO DETALHADA DAS INSTRUÇÕES UTILIZADAS NO CÓDIGO FEITO PELA EQUIPE

4.1. Micro operações

Um programa contém um conjunto de instruções, que são armazenadas na memória principal e possuem um campo para opcode responsável por determinar o que será executado. Por sua vez, essas diretrizes podem ser subdivididas em etapas menores e elementares, pois, para projetar uma unidade de controle, é necessário especificar as etapas mais essenciais da execução de diretrizes que ocorrem no nível dos registros internos do processador. Essas operações básicas são conhecidas como micro operações. Cada micro operação é mínima e pode ser classificada nos seguintes tipos:

- Transferência de dados de um registrador A para um registrador B.
- Transferência de dados de um registrador para uma interface externa.
- Transferência de dados de uma interface externa para um registrador.
- Execução de uma operação lógica ou aritmética, utilizando registradores de E/S.

E, empregando essas micro operações primárias, é viável estabelecer sequências para realizar uma ampla variedade de tarefas e ciclos de instruções. Todas as instruções obrigatoriamente possuem um ciclo de busca e um ciclo de execução, além de poderem ter um ciclo indireto e um ciclo de interrupção.

4.2. Ciclos de Instrução

4.2.1 Ciclo de Busca

O ciclo de busca necessita de 4 registradores para ocorrer, sendo eles:

- Memory Address Register (MAR)
- Memory Buffer Register (MBR)
- Program Counter (PC)
- Instruction Register (IR)

Com esses registradores, a sequência de busca é executada da seguinte forma:

No começo do ciclo, o endereço da instrução posterior a ser executada é armazenada no PC.

- Primeiramente, move-se esse endereço para o MAR. A partir desse momento, MAR é o único registrador ligado às linhas de endereço do bus do sistema.

- Em segundo, é adquirida a instrução e o endereço desejado é colocado no bus de endereço. A unidade de controle envia um comando READ para o controlador do bus e o resultado aparece no bus de dados, é então copiado para o MBR. Nesse momento, também é preciso incrementar o PC em uma unidade para obter a seguinte instrução.
- O terceiro passo consiste em mover o conteúdo do MBR para o IR. Com isso, é liberado ao MBR para a utilização durante um possível ciclo indireto.

Dessa forma, um possível ciclo de busca consiste em ter 3 passos e 4 micro-operações, sendo que cada uma delas envolve o movimento de dados. Contanto que os movimentos de dados não interfiram umas nas outras, mais movimentos podem decorrer no mesmo passo, conseguindo uma economia de tempo.

Simbolicamente, os passos acima ficam:

```
A1: MAR <- PC
A2: MBR <- memória, PC <- PC + 1
A3: IR <- MBR
```

Mesmo que seja possível fazer o agrupamento de micro operações, é necessário respeitar a sequência de passos e evitar conflitos durante o processo.

4.2.2 Ciclo Indireto

Posteriormente ao ciclo de busca, talvez seja necessário buscar os operandos na memória. Se a instrução precisa de um endereço indireto, será realizado um ciclo indireto antecedente ao ciclo de execução. Ele é formado, geralmente, por três passos. Sendo eles:

```
T1: MAR <- (IR(Endereço))
T2: MBR <- Memória
T3: IR(Endereço) <- (MBR(Endereço))
```

O endereço da instrução é carregado no MAR, e há o envio de uma instrução para o barramento de controle com o intuito de obter o

endereço do operando. Esse endereço é guardado no MBR e o campo de endereço do IR é atualizado por ele.

4.2.3 Ciclo Interrupção

A partir do momento que o ciclo de execução é executado, um teste é feito para determinar se existe alguma interrupção habilitada. Esse ciclo possui, normalmente, contém três passos e quatro micro operações, sendo elas:

```
T1: MBR <- (PC)
T2: MAR <- Endereço_Salvar, PC <- Endereço_rotina
T3: Memória <- (MBR)
```

O conteúdo do PC é enviado para o MBR para que possa ser gravado e retomado após a interrupção. Depois o MAR recebe o valor do PC e o PC recebe o endereço da rotina de processamento da interrupção. Por fim, o valor de MBR é armazenado na memória.

4.2.4 Ciclo de Execução das instruções utilizadas no programa

4.2.4.1 Instrução lb

A instrução “lb” significa “load byte” e passa um byte armazenado em memória para um registrador. Deve receber um registrador, no qual o byte será armazenado e poderá ser acessado depois em outro momento do programa. A utilização da instrução é:

```
lb $t0
```

As suas micro operações são:

```
T1: MAR <- IR
T2: MBR <- Memória
T3: rl <- MBR
```

4.2.4.2 Instrução move

A instrução “move” é uma pseudo instrução que na verdade se traduz com a soma do valor de um registrador com zero e atribui esse resultado a outro registrador. Recebe dois registradores, primeiramente o registrador que irá receber o segundo.

A utilização da instrução é:

```
move $t1, $t0
```

E pode ser traduzida para:

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

Suas micro operações são:

```
t1: r1 <- (r2) + $zero
```

4.2.4.3 Instrução la

A pseudo instrução “la” significa “Load Addres” e é uma pseudo instrução que carrega em um registrador o endereço de um label na memória. Recebe um registrador e uma label, que é o nome de um dados definido da seção “.data”. A utilização da instrução é:

```
la $s1, nomeDoLabel
```

E pode ser traduzido para:

```
lui $at, 4097 (0x1001 → upper 16 bits of $at).
ori $a0,$at,disp
```

E suas micro operações são:

```
T1: MAR <- IR
T2: MBR <- Memória
T3: rl <- MBR
```

4.2.4.4 Instrução syscall

A pseudo instrução “syscall” significa “System Call” e é uma instrução que executa uma chamada de sistema

especificada previamente. Sua utilização mais comum é para realizar a impressão e/ou inserção de dados via linha de comando. Um exemplo de uso seria:

```
li, $v0, 10
syscall
```

Nesse exemplo, é carregado o valor 10 no registrador $v0, que entende esse número inteiro como um sinal específico e encerra o programa.

4.2.4.5 Instrução addiu

A instrução “addiu” significa “Add immediate unsigned” e é utilizada para realizar uma adição com um valor imediato, o tratando como um inteiro sem sinal. Recebe três operadores, um registrador de destino, um de origem e um valor imediato. Sua utilização é:

```
addiu $t0, $t1, valorImediato
```

E suas micro operações são:

```
T1:  registrador0 ← (registrador1)+(registrador2)
```

4.2.4.6 Instrução sw

A instrução “sw” significa “Store word” e é utilizada para salvar informações de registradores na pilha de memória. Recebe dois argumentos, o registrador com o valor que deve ser salvo e o endereço da pilha. Sua utilização é:

```
sw $t0, endereço
```

E suas micro operações são:

```
T1: MAR <- (IR(add))
T2: MBR <- r1
T3: Memória <- (MBR)
```

4.2.4.7 Instrução jal

A instrução jal significa “Jump and Link” e é utilizada para fazer a chamada de procedimentos e alterar o fluxo do programa, inserindo a parte do código presente no endereço passado como label. Também salve o endereço da instrução seguinte ao jump no código em um registrador de retorno, permitindo o código de continuar depois. Recebe um único argumento que é uma label/endereço. Sua utilização é:

```
jal label
```

E suas micro operações são:

```
T1: RA <- PC
T2: PC <- label
```

4.2.4.8 Instrução lw

A instrução “lw” significa “load word”. Sua função é carregar uma mensagem armazenada em memória em um registrador. Recebe um único argumento que é um registrador de destino no qual a palavra será armazenada. Sua utilização é:

```
lw $t1, add
```

E suas micro operações são:

```
T1: R1 <- $zero + valor
```

4.2.4.9 Instrução li

A instrução “li” significa “Load Immediate” e é uma pseudo interação utilizada para armazenar um valor específico em um registrador. Ela funciona através de um “addiu” e recebe dois argumentos, um registrador e um valor a ser atribuído.

```
li $t0, valor
```

E se traduz como:

```
addiu $t0, $t1, $zero
```

E suas micro operações:

```
T1: R1 <- R2 + R3
```

4.2.4.10 Instrução beq

A instrução “beq” significa “branch if equal” e é utilizada para testar uma condição de igualdade que, quando satisfeita, desvia o fluxo do programa. Recebe dois registradores com valores atribuídos e uma label. Se os valores nos registradores forem iguais, é realizado o desvio para a seção do código da label passada como argumento. Sua utilização é:

```
beq $t0, $t1, label
```

E suas micro operações são:

```
T1: SE (r1) != (r2) ENTÃO PC <- label
```

4.2.4.11 Instrução j

A instrução “j” significa “jump” e é utilizada para realizar um teste incondicional para uma label/endereço específico do programa, assim alterando o fluxo de instrução e executando a parte do código imediatamente a baixo de label passada como argumento. Recebe somente um argumento, um label/endereço. Sua utilização é:

```
j label
```

E suas micro operações são:

```
T1: PC < - label
```

4.2.4.12 Instrução xor

A instrução “xor” significa “Exclusive Logical Or” e é utilizada para realizar a operação lógica de ou exclusivo na comparação bit a bit de dois registradores. Recebe três argumentos, um registrador de destino e outros dois que serão comparados entre si. Sua utilização é:

```
xor $t0, $t1, $t2
```

E suas micro operações são:

```
T1: R1 <- R2 XOR(^) R3
```

4.2.4.13 Instrução and

A instrução “and” significa “Logical And” e é utilizada para realizar a operação lógica de “e” na comparação bit a bit de dois registradores. Recebe três argumentos, um registrador de destino e outros dois que serão comparados entre si. Sua utilização é:

```
and $t0, $t1, $t2
```

E suas micro operações são:

```
T1: R1 <- R2 AND(&) R3
```

4.2.4.14 Instrução or

A instrução “or” significa “Logical Or” e é utilizada para realizar a operação lógica de “ou” na comparação bit a bit de dois registradores. Recebe três argumentos, um registrador de destino e outros dois que serão comparados entre si. Sua utilização é:

```
or $t0, $t1, $t2
```

E suas micro operações são:

```
T1: R1 <- R2 OR(|) R3
```

4.2.4.15 Instrução jr

A instrução “jr” significa “jump return” e é utilizada para realizar um desvio de fluxo incondicional, que salta para a label/endereço armazenado no registrador de retorno (RA), é muito parecida com a instrução “return” de linguagens de mais alto nível. Recebe somente o registrador de retorno. Sua utilização é:

```
jr $ra
```

E suas micro operações são:

```
T1: PC <- RA
```

5. REFERÊNCIAS BIBLIOGRÁFICAS

CENTRO DE INFORMÁTICA - UNIVERSIDADE FEDERAL DE PERNAMBUCO. **Arquitetura e Organização de Computadores - Aula: Conjunto de Instruções**. Disponível em: https://portal.cin.ufpe.br/. Acesso em: 27 jun. 2023.

DEPARTAMENTO DE INFORMÁTICA - UNIVERSIDADE FEDERAL DO PARANÁ. **Guia Rápido - MIPS**. Disponível em: https://web.inf.ufpr.br/dinf/. Acesso em: 27 jun. 2023.

MIPS ELEVATING RISC-V. **MIPS32 Architecture**. Disponível em: https://www.mips.com/products/architectures/mips32-2/. Acesso em: 27 jun. 2023.

PATTERSON, David A.; HENNESSY, John L.. **Organização e Projeto de Computadores**: A interface Hardware/Software. 4. ed. Rio de Janeiro: Elsevier, 2014. p. 1-709.

YOUTUBE - CANAL DA UNIVESP. **Organização de Computadores - Aula 06 - Conjunto de Instruções do MIPS**. Disponível em: https://www.youtube.com/watch?v=VYSy21RwNIc. Acesso em: 27 jun. 2023.

YOUTUBE - CANAL DA UNIVESP. **Organização de Computadores - Aula 07 - Conjunto de Instruções do MIPS (Parte 2)**. Disponível em: https://www.youtube.com/watch?v=i_NeQ2hblX0. Acesso em: 27 jun. 2023.

6. GLOSSÁRIO

Arquitetura de Conjunto de Instruções: Corresponde aos níveis de linguagem de montagem (assembly) e de linguagem de máquina.

Barramento: Conjunto de linhas de comunicação que permitem a interligação entre dispositivos.

Bit: Menor unidade de informação que pode ser armazenada ou transmitida, podendo assumir os valores 0 ou 1.

Bloco: É uma sequência de bytes ou bits, normalmente contendo algum número completo de registros, que possui um tamanho máximo.

Byte: Um conjunto de 8 *bits* adjacentes.

Const: Abreviação de *"*constante".

End: Abreviação *de* "endereço".

Funct: É um complemento ao *opcode*, permitindo um maior número de operações sem a necessidade de aumentar o tamanho do campo do *opcode*.

Id: Abreviação de "identidade".

Instrução: Uma operação única executada por um processador e definida por um conjunto de instruções.

Offset: Um inteiro indicando a distância (deslocamento) entre o começo do objeto e um dado elemento ou ponto, presumivelmente dentro do mesmo objeto.

Opcode: Código de operação, comumente abreviado para "op".

Palavra: Unidade de acesso natural de um computador, normalmente um grupo de 32 bits, corresponde ao tamanho de um registrador na arquitetura MIPS.

Pipeline: Cadeia de elementos de processamento.

Procedure: Conjunto de passos computacionais a serem executados.

Rd: Código do registrador que recebe o resultado da operação.

Rs: Código do registrador que armazena o primeiro operando.

Rt: Código do registrador que armazena o segundo operando.

Shamt: Quantidade de deslocamento que deve ser usada em operações de *shift* (deslocamento).