UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO
ACH2034 - ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES I

DAVI BATISTA DE SOUZA
DIOGO DOS SANTOS DA ROCHA

RELATÓRIO DO EXERCÍCIO-PROGRAMA (EP) DA DISCIPLINA ‘ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES I’

SÃO PAULO
2024

DAVI BATISTA DE SOUZA
DIOGO DOS SANTOS ROCHA

Relatório do Exercício-Programa (EP) da disciplina ACH2034 - 'Organização e arquitetura de computadores I’

Relatório apresentado à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo como parte dos requisitos para conclusão da disciplina Organização e Arquitetura de Computadores I. Orientadora: Profa. Dra. Gisele S. Craveiro.

São Paulo
2024

SUMÁRIO

SEÇÃO 1 - ORGANIZAÇÃO E ARQUITETURA MIPS

1.1 Arquitetura do Conjunto de Instruções

A Arquitetura do Conjunto de Instruções (ISA - Instruction Set Architecture) define o conjunto de instruções de linguagem de máquina que um computador pode executar. Essa interface atua como uma fronteira entre o hardware e o software, sendo crucial para o funcionamento do computador (STALLINGS, 2017). A ISA especifica a linguagem de máquina que o processador entende, ou seja, o conjunto de instruções que podem ser utilizadas para programar a CPU. Cada instrução realiza uma operação básica, como adição, subtração, comparação ou transferência de dados. A combinação e a sequência dessas instruções formam os programas de software que executam tarefas complexas.

Uma ISA bem projetada pode aumentar o desempenho do processador, permitindo que os programas sejam executados de forma mais rápida. Algumas das arquiteturas de ISA mais conhecidas são x86, ARM e MIPS, sendo esta última o foco desta seção. Exploraremos mais sobre a arquitetura MIPS e, posteriormente, utilizaremos suas operações para implementar a solução de um problema computacional.

1.2 Arquitetura do Conjunto de Instruções MIPS

'MIPS' significa 'Microprocessor without Interlocked Pipeline Stages'. A arquitetura MIPS oferece 32 registradores de propósito geral e 32 de ponto flutuante, podendo acessar a memória apenas através de instruções de load-store (PATTERSON, HENNESSY, 2012). Por isso, ela é classificada como uma arquitetura load-store. Além disso, é considerada de fácil compreensão porque todas as instruções possuem um tamanho fixo de 32 bits. A tabela a seguir apresenta um resumo dos registradores disponíveis no MIPS (a tabela estendida, com informações adicionais como a representação binária de cada registrador, está no Apêndice A).

1.2.1 Modos de endereçamento de memória no MIPS

Os modos de endereçamento no MIPS são: registrador, imediato (para constantes) e deslocamento, em que um deslocamento constante é adicionado a um registrador para formar o endereço de memória (HENNESSY, PATTERSON, 2012).

1.2.2 Tipos e tamanhos dos operandos

O MIPS suporta operandos de 8 bits (caractere ASCII), 16 bits (caractere Unicode ou meia palavra), 32 bits (inteiro ou palavra), 64 bits (dupla palavra ou inteiro longo) e ponto flutuante IEEE 754 com 32 bits (precisão simples) e 64 bits (precisão dupla) (HENNESSY, PATTERSON, 2012).

1.2.3 Operações

As categorias gerais de operações são: (1) transferência de dados, (2) lógica e aritmética, (3) controle e (4) ponto flutuante. Cada categoria representa operações específicas que a CPU pode executar. O MIPS é uma arquitetura de conjunto de instruções simples, ideal para execução em

pipeline, representando as arquiteturas RISC utilizadas em 2011 (HENNESSY, PATTERSON, 2012). A seguir, são apresentados exemplos de instruções de cada categoria:

1.2.3.1 Instruções de transferência de dados

Essas instruções movimentam dados entre a memória e os registradores. Alguns exemplos são:

- **Instrução de transferência de valor para registrador:**
    - Formato: `mflo R` (onde R é um registrador de uso geral)
    - Exemplo: `mflo $t0` (move o valor do registrador especial LO para o registrador geral $t0)
- **Instrução de transferência de dados de registrador para registrador:**
    - Formato: `add Rd, Rs, Rt` (onde Rd = Registrador de Destino, Rs = 1º Registrador de Origem e Rt = 2º Registrador de Origem)
    - Exemplo: `add $t1, $s1, $s2` (adiciona os valores dos registradores $s1 e $s2 e armazena o resultado em $t1)
- **Instrução de carregamento de palavra (Load Word):**
    - Formato: `lw Rt, offset(Rs)` (onde Rt = Registrador de Destino, Offset = deslocamento de memória relativo à base, e Rs = Registrador de Base, usado para calcular o endereço de memória a ser carregado)
    - Exemplo: `lw $t2, 4($s3)` (carrega a palavra armazenada no endereço $s3 + 4 bytes e a armazena em $t2)
- **Instrução de armazenamento de palavra (Store Word):**
    - Formato: `sw Rt, offset(Rs)` (onde Rt = Registrador de origem, Offset = deslocamento de memória relativo à base, e Rs = Registrador de Base, usado para calcular o endereço de memória onde o valor será armazenado)
    - Exemplo: `sw $t3, 8($s4)` (armazena o valor do registrador $t3 no endereço $s4 + 8 bytes)

1.2.3.2 Instruções Lógicas e Aritméticas

- **Instrução de transferência de dados imediata (Immediate Data Transfer):**
    - Formato: `addi Rd, Rs, immediate` (onde Rd = Registrador de Destino, Rs = Registrador de Origem e immediate = constante imediata, que é adicionada ao valor do registrador de origem)
    - Exemplo: `addi $t0, $s0, 42` (adiciona a constante imediata 42 ao valor do registrador $s0 e armazena o resultado em $t0)
- **Instrução de subtração imediata (Subtract Immediate):**
    - Formato: `subi Rd, Rs, immediate` (onde Rd = Registrador de Destino, Rs = Registrador de Origem e immediate = constante imediata, que é subtraída do valor do registrador de origem)
    - Exemplo: `subi $t0, $s0, 10` (subtrai a constante imediata 10 do valor do registrador $s0 e armazena o resultado em $t0)

- **Instrução de multiplicação (Multiply):**
    - Formato: `mult Rs, Rt` (onde Rs e Rt são registradores de origem, cada um contendo um dos fatores da multiplicação)
    - Exemplo: `mult $s1, $s2` (multiplica os valores dos registradores $s1 e $s2). Essa operação salva o resultado em registradores especiais (HI e LO)
- **Instrução de divisão (Divide):**
    - Formato: `div Rs, Rt` (onde Rs e Rt são registradores de origem, com Rs recebendo o dividendo e Rt, o divisor)
    - Exemplo: `div $s3, $s4` (realiza a divisão dos valores dos registradores $s3 e $s4). Essa operação também utiliza os registradores HI e LO

1.2.3.3 Instruções de controle (desvios condicionais e incondicionais)

O MIPS utiliza endereçamento relativo ao PC (Program Counter), onde o endereço de desvio é especificado por um campo de endereço somado ao PC. Desvios condicionais no MIPS (como BEQ - Branch Equals e BNE - Branch Not Equals) testam o conteúdo dos registradores. As chamadas de procedimentos colocam o endereço de retorno em um registrador. Alguns exemplos são:

- **Instrução de salto incondicional (Jump):**
    - Formato: `j target`
    - Exemplo: `j label` (salta para a sequência de instruções identificada pela etiqueta "label")
- **Instrução de salto condicional igual a zero (Branch Equal Zero):**
    - Formato: `beqz Rs, target`
    - Exemplo: `beqz $t0, label` (salta para a instrução identificada pela etiqueta "label" se o valor do registrador $t0 for igual a zero)
- **Instrução de salto condicional não igual a zero (Branch Not Equal Zero):**
    - Formato: `bnez Rs, target`
    - Exemplo: `bnez $t1, label` (salta para a instrução identificada pela etiqueta "label" se o valor do registrador $t1 não for igual a zero)
- **Instrução de retorno de sub-rotina (Return):**
    - Formato: `jr Rs`
    - Exemplo: `jr $ra` (retorna para o endereço de retorno armazenado no registrador $ra)

1.2.3.4 Instruções de operações de ponto flutuante

Os exemplos a seguir tratam de operações de ponto flutuante com precisão simples. Para precisão dupla, adiciona-se ".d" após o opcode.

- **Instrução de adição de ponto flutuante (Floating-Point Addition):**
    - **Formato:** `add.s Fd, Fs, Ft`

    - **Exemplo:** `add.s $f0, $f2, $f4` (realiza a adição dos valores dos registradores $f2 e $f4 e armazena o resultado em $f0)
- **Instrução de subtração de ponto flutuante (Floating-Point Subtraction):**
    - **Formato:** `sub.s Fd, Fs, Ft`
    - **Exemplo:** `sub.s $f6, $f8, $f10` (realiza a subtração dos valores dos registradores $f8 e $f10 e armazena o resultado em $f6)
- **Instrução de multiplicação de ponto flutuante (Floating-Point Multiplication):**
    - **Formato:** `mul.s Fd, Fs, Ft`
    - **Exemplo:** `mul.s $f12, $f14, $f16` (realiza a multiplicação dos valores dos registradores $f14 e $f16 e armazena o resultado em $f12)
- **Instrução de divisão de ponto flutuante (Floating-Point Division):**
    - **Formato:** `div.s Fd, Fs, Ft`
    - **Exemplo:** `div.s $f18, $f20, $f22` (realiza a divisão dos valores dos registradores $f20 e $f22 e armazena o resultado em $f18)

1.2.4 Formato das Instruções

O formato de instruções define a estrutura utilizada para representar as instruções de uma determinada arquitetura do conjunto de instruções. O formato das instruções do MIPS pode ser definido de três formas diferentes: tipo R (Registrador), tipo I (Imediato) e tipo J (Jump), e os campos recebem nomes diferentes para facilitar o seu tratamento.

1.2.4.1 Formato R (Registrador)

O formato R é utilizado para operações entre registradores. Cada instrução neste formato é representada por 32 bits, divididos em campos específicos. Os campos do formato R são organizados da seguinte forma:

Onde:

- `op` significa ‘operação básica da instrução’, tradicionalmente chamado de opcode. Indica a operação a ser executada.
- `rs` representa o primeiro registrador do operando fonte.
- `rt` representa o segundo registrador do operando fonte.
- `rd` representa o registrador do operando de destino. Ele recebe o resultado da operação.
- `shamt` significa “Shift amount” (quantidade de deslocamento).
- `funct` significa “função”. Esse campo, normalmente chamado código de função, indica a função específica da operação.

1.2.4.2 Formato I (Imediato)

O formato I é utilizado para operações entre registradores e constantes imediatas. Assim como o formato R, as instruções do formato I possuem 32 bits e campos específicos. Os campos do formato I são representados da seguinte forma:

Onde:

- `op` significa ‘Opcode’ e indica a operação a ser executada.
- `rs` especifica o registrador de origem.
- `rt` especifica o registrador de destino.
- `Immediate` contém a constante imediata utilizada na operação.

1.2.4.3 Formato J (Jump)

O formato J é utilizado para instruções de salto incondicional. As instruções neste formato possuem 32 bits e são representados da seguinte forma:

Onde:

- `op` significa ‘Opcode’ e indica a operação de salto.
- `addr` significa ‘Address’ e especifica o endereço de salto.

SEÇÃO 2 - DESCRIÇÃO DO PROBLEMA E CÓDIGO ALTO NÍVEL DA SOLUÇÃO

2.1 - Descrição do problema

O problema computacional mencionado na seção 1.1 será descrito a seguir, juntamente da implementação em alto nível da solução. Compreendendo essas duas partes, conseguiremos entender a implementação no MIPS.

Este problema envolve a soma de bits individuais, onde se utiliza a técnica de soma com propagação de vai-um (carry). Vamos abordar a implementação dessa soma utilizando a linguagem C.

A ideia é resolver o seguinte problema:

- Implementar a soma de dois bits individuais e determinar o vai-um gerado pela soma.

Assumiremos que os bits e o vai-um inicial são fornecidos, e o objetivo é calcular a soma e o novo vai-um resultante da operação.

2.2 - Código em alto nível

Vamos implementar a solução utilizando a linguagem C. A função principal será responsável por calcular a soma de dois bits e o vai-um resultante.

2.2.1 Inicialização de variáveis e impressão dos valores

Vamos iniciar definindo estaticamente os bits “b1” e “b2” e o vai-um inicial “vaium”. A seguir, a implementação em alto nível da inicialização das variáveis:

```c
#include <stdio.h>
#include <stdlib.h>

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

int main() {
  int b1 = 1, b2 = 1, vaium = 1, soma;

  somabit(b1, b2, &vaium, &soma);

  printf("Soma: %d\n", soma);
  printf("Vai-um: %d\n", vaium);

  return 0;
}
```

2.2.2 - Definição, explicação e chamada da função que realiza a soma de bits

A função `somabit` tem o propósito de calcular a soma de dois bits e determinar o novo vai-um. A função

recebe quatro parâmetros: os bits `b1` e `b2`, um ponteiro para o vai-um e um ponteiro para a soma. A implementação em alto nível ficou da seguinte forma:

```
4   void somabit(int b1, int b2, int* vaium, int* soma) {
5     *soma = (b1 ^ b2) ^ *vaium;
6     *vaium = (b1 & b2) | (b1 & *vaium) | (b2 & *vaium);
7   }
```

2.2.2.1 - Explicação do código em alto nível para a função implementada

A função `somabit` calcula a soma de dois bits considerando o vai-um. A ideia geral é:

1. Calcular a soma parcial dos bits `b1` e `b2` utilizando a operação XOR e adicionar o vai-um.
2. Calcular o novo vai-um utilizando as operações AND e OR para identificar se há um carry gerado pela soma.

O uso de operações bit a bit (AND, OR, XOR) é essencial para a correta implementação da soma de bits em nível binário.

2.2.2.2 - Chamada da função `somabit` na função `main`

No `main`, iniciamos os bits `b1` e `b2`, bem como o vai-um. Em seguida, chamamos a função `somabit` e imprimimos os resultados:

```
9   int main() {
10    int b1 = 1, b2 = 1, vaium = 1, soma;
11
12    somabit(b1, b2, &vaium, &soma);
13
14    printf("Soma: %d\n", soma);
15    printf("Vai-um: %d\n", vaium);
16
17    return 0;
18  }
```

2.3 - Exemplos de entradas e saídas esperadas

Vamos considerar alguns exemplos para demonstrar a funcionalidade da função `somabit`:

Exemplo 1

Entradas: b1 = 1, b2 = 1, vaium = 0 Resultado esperado:

- Soma: 0
- Vai-um: 1

Exemplo 2

Entradas: b1 = 1, b2 = 0, vaium = 1 Resultado esperado:

- Soma: 0
- Vai-um: 1

Exemplo 3

Entradas: b1 = 0, b2 = 0, vaium = 1 Resultado esperado:

- Soma: 1
- Vai-um: 0

Exemplo 4

Entradas: b1 = 1, b2 = 1, vaium = 1 Resultado esperado:

- Soma: 1
- Vai-um: 1

Com esses exemplos, podemos verificar que a função `somabit` realiza corretamente a soma de bits e determina o novo vai-um.

SEÇÃO 3 - CÓDIGO EM ASSEMBLY DESENVOLVIDO

O código que foi desenvolvido em Assembly é composto por duas partes principais, sendo elas: “.data” e “.text”. A primeira parte abriga as variáveis que serão manipuladas com a ajuda dos registradores ofertados pelo MIPS. Já a segunda parte contém as rotinas que serão utilizadas para realizar a execução do programa. As subseções a seguir tem como objetivo apresentar o código de forma segmentada e bem explicada. Ao final desta seção, o código será disponibilizado de forma integral, para melhor visualização.

3.1 – Trecho “.data” do código em Assembly

```
67  .data
68  $b1: .word 1 # Declaração dos bits de b1 e b2
69  $b2: .word 0
70  $vaium: .word 1 # Declaração do vai-um e do resultado da soma
71  $soma: .word 0
72  $resultadoSoma: .asciiz "Resultado da soma: "
73  $resultadoVaiUm: .asciiz " Resultado do vai-um: "
```

Figura 1 – Trecho “.data” do código em Assembly

Na seção data, na linha 68, é declarada um número inteiro que representa um bit que recebe o valor 1, com o nome de “b1”. A linha 69 declara outro inteiro que representará um bit, cujo valor atribuído será 0. O objetivo dessas duas variáveis é fazer a soma entre si, utilizando-se de um arcabouço lógico relativamente simples, como será demostrado posteriormente.

O próxima linha, 70, a variável “vaium” é declarada com o valor 1. O objetivo desta variável é armazenar o valor do número que “sobrar” da adição das duas variáveis acima, realmente funcionando como um “vaium” da adição, ou conhecido mais formalmente como adição de reserva ou reagrupamento.

A variável[1] “soma”, atribuída com o valor 0 na linha 71, tem por finalidade, como pode-se supor, armazenar o valor da soma entre as variáveis “b1”, “b2” e “vaium”, que irá ser retornada junto com o “vaium” ao final da execução do algoritmo.

As variáveis subsequentes, “resultadoSoma” e “resultadoVaiUm”, são strings que servirão como apoio para melhor entendimento da saída do programa, cada uma retornando, respectivamente, o valor da variável “soma” e o valor da variável “vaium”.

1 Variáveis são rótulos dados a espaços de memória que foram alocados de forma sequencial.

3.2 – Trecho “.text” do código em Assembly

A seção “.text” armazena o código que será executado por nosso programa. Ela possui algumas subdivisões, por isso, iremos dividir sua explicação para cada rótulo que há no código. Tendo compreendido a explicação do código em alto nível, a explicação agora dada sobre a ótica do algoritmo em Assembly será facilitada para o leitor. A execução se mantém praticamente inalterada:

- Os valores dos bits são armazenados nos respectivos registradores;
- Ocorre a operação lógica XOR (que será mais aprofundada posteriormente) entre os valores “b1” e “b2”;
- Após isso o valor de “vaium” é usado com o resultado da comparação anterior, agora fazendo novamente a operação XOR;
- Em seguida, a operação lógica AND é feita entre “b1” e “b2”, depois entre “b1” e “vaium” e por fim é feita a mesma operação entre “b2” e “vaium”;
- Depois, é feita a operação OR entre o resultado de cada comparação anterior, ou seja, mais duas comparações;
- Por fim, os resultados são impressos para o usuário. Primeiro é informado o valor da soma e depois o valor do “vaium”;

3.2.1 – Diretiva “.globl main” e rótulo main

```
1  .text
2  .globl main
3
4  main:
5      lw $t0, $b1 # Carrega o valor de b1 em t0
6      lw $t1, $b2 # Carrega o valor de b2 em t1
7      jal somabit # Chama a função somabit
8      j fim # Pula para o fim
```

Figura 2 – Rótulo “main” do código desenvolvido

O rótulo “main” indica para o programa por onde deve começar a execução. É a partir dele que o compilador irá identificar por onde iniciar suas instruções. O código completo será disponibilizado junto com este relatório, no entanto, é de suma importância a explicação por completo dos pormenores da implementação requerida.

As linhas 5 e 6 tem como objeto por enviar os valores corretos como parâmetro. O intuito com isso foi o de fazer o código de alto nível e o implementado em Assembly ficassem o mais parecidos possível, para facilitar o entendimento. Desta forma, os valores dos bits “b1” e “b2” são armazenados nos registradores $a0 e $a1, que são usados para armazenar os parâmetros da

função, o que será melhor explicado posteriormente. Esses valores será utilizados pelo rótulo denominado “somabit”, chamado na linha 7 por meio da instrução jal (jump-and-link), que também será melhor explicada ao decorrer do relatório.

3.2.2 – Rótulo “somabit”

```
30  somabit:
31      xor $t2, $a0, $a1 # Faz a operação XOR entre t0 e t1 e armazena em t2
32      addi $sp, $sp, -4 # Decrementa o ponteiro de pilha
33      sw $t2, 0($sp) # Armazena o valor de t2 na pilha
34      lw $t0, $vaium # Carrega o valor de vaium em t0
35      lw $v0, 0($sp) # Carrega o valor da pilha em v0
36      addi $sp, $sp, 4 # Incrementa o ponteiro de pilha
37      xor $t7, $t0, $v0 # Faz a operação XOR entre t0 e v0 e armazena em t1
38
39      lw $t0, $b1
40      lw $t2, $b2
41      and $t3, $t0, $t2 # Faz a operação AND entre t0 e t2 e armazena em t3
42      addi $sp, $sp, -4
43      sw $t3, 0($sp)
44
45      lw $t4, $vaium
46      and $t5, $t4, $t0 # Faz a operação AND entre t4 e t0 e armazena em t5
47      addi $sp, $sp, -4
48      sw $t5, 0($sp)
49
50      and $t6, $t2, $t4 # Faz a operação AND entre t2 e t4 e armazena em t6
51      addi $sp, $sp, -4
52      sw $t6, 0($sp)
53
54      lw $t0, 0($sp)
55      addi $sp, $sp, 4
56      lw $t1, 0($sp)
57      addi $sp, $sp, 4
58      lw $t2, 0($sp)
59      addi $sp, $sp, 4
60
61      or $t4, $t0, $t1
62      or $t6, $t4, $t2 # Faz a operação OR entre t4 e t2 e armazena em t5, SOMA ESTÁ EM T5
63
64      jr $ra
```

Figura 3 – Rótulo “somabit” do código desenvolvido

Este rótulo identifica a sequência de instruções que irá somar os bits “b1” e “b2”, depois comparar com “vaium” e por fim determinar o valor da soma. Inicialmente, na linha 31, a instrução “xor $t1, $ao, $a1” faz a operação lógica XOR entre os valores de “$a0” e “$a1” e posteriormente adiciona o resultado no registrador “$t2”. A operação XOR será melhor detalhada na seção 4 deste relatório.

A linha a seguir, com a instrução “addi $sp, $sp, -4”, decrementa o ponteiro da pilha, já que iremos armazenar o valor de “$t2” na mesma, o que é realizado na linha 33. Após isso carregamos o valor da variável “$vaium” no registrador “$t0”, assim como, na linha seguinte, armazenamos o primeiro valor da pilha em “$v0”, acompanhado de um acréscimo na pilha, já que estamos removendo um valor da pilha.

Na linha 37 ocorre mais um XOR entre o valor obtido pela mesma operação anteriormente realizada e o valor da variável “$vaium”, o valor desta operação é guardado no registrador “$t7”.

Nas linhas 39 e 40 carregamos para a memória novamente os valores das variáveis “$b1” e “$b2”, com a instrução “lw $t0, $b1”. Em seguida realizamos a operação AND, por intermédio da instrução “and $t3, $t0, $t2”, que será destrinchada posteriormente, entre as variáveis citadas anteriormente. O valor resultante desta operação será armazenado no registrador “$t3”. Após isso o registrador da pilha é decrementado e o valor de “$t3” é armazenado na pilha.

Já na linha 45 ocorre o armazenamento do valor da variável “$vaium” no registrador “$t4”. Após isso, é realizada a operação AND entre as variáveis “$vaium” e “$b1”, o valor é armazenado em “$t5” e depois armazenado na pilha como antes.

Das linhas 50 a 52 é feita a última operação AND entre “$vaium” e “$b2”, com o valor sendo armazenado na variável “$t6”, e o valor é armazenado na pilha tal qual já fizemos anteriormente.

As linhas seguem da 54 a 59 desempilham os valores que foram armazenados com os resultados as operações AND, cada um para os registradores “$t0”, “$t1” e “$t2” respectivamente. Além disso a instrução “addi $sp, $sp, 4” diminui o endereço do ponteiro da pilha, cada vez que é retirado um valor da mesma.

A instrução “or $t4, $t0, $t1”, que está presenta nas linhas 61 e 62, faz a operação lógica OR primeiro entre o resultado da operação AND de “b1” e “b2” e a operação AND de “b1” e “vaium”. Por fim é feita a operação OR entre o resultado da operação anterior e o resultado da operação AND de “b2” e “vaium”.

Por fim, na linha 64 existe a instrução “jr $ra”, a qual é usada para fazer o salto incondicional para o endereço armazenado no registrador de retorno ($ra). Esta instrução retorna para a próxima instrução a partir da qual foi chamada, prosseguindo com a execução do programa.

3.2.3 – Rótulo “fim”

```
10  fim:
11      la $a0, $resultadoSoma # Printando o resultado da soma
12      li $v0, 4
13      syscall
14
15      move $a0, $t7 # Transferindo o conteúdo do registrador t6 para a0
16      li $v0, 1
17      syscall
18
19      la $a0, $resultadoVaiUm # Carrega o endereço da string em a3
20      li $v0, 4
21      syscall
22
23      move $a0, $t6 # Printando o vai-um
24      li $v0, 1
25      syscall
26
27      li $v0, 10 # Termina o programa
28      syscall
```

Figura 4 – Rótulo “fim” do código desenvolvido

A primeira linha do rótulo “fim” carrega no registrador “$a0” o endereço da string armazenada na variável “$resultadoSoma”, e após isto é realizada uma “syscall” para imprimir a string para o usuário.

Na linha 15, temos a instrução “move $a0, $t7”, que irá mover o conteúdo do registrador “$t7“ para o registrador “$a0“ e em seguida temos uma chamada “syscall” que irá retornar o valor da variável da soma dos bits.

As linhas 19 a 21 vão armazenar o valor da string “$resultadoVaiUm” no registrador “$a0“ e faz uma “syscall” para retornar a string para o usuário. As instruções contidas nas linhas 23 a 25, move o valor do registrador “$t6” para o “$a0” e em seguida imprime o valor para o usuário do valor resultante em “vaium”.

A linha 27 e 28 indicam para o compilador que o programa se encerrou, o que finaliza a execução do mesmo.

SEÇÃO 4 - EXPLICAÇÃO DETALHADA DAS INSTRUÇÕES UTILIZADAS

De acordo com o que foi discutido anteriormente, o processo de codificação de um programa em nível intermediário com o manejo das instruções disponibilizadas na arquitetura MIPS é muito mais detalhado e cuidadoso, se levarmos em conta o mesmo processo com linguagens de alto nível, como C por exemplo. Podemos ainda, com a utilização do simulador “MARS”, observar um nível ainda mais fundamental, as micro-operações.

As micro-operações são as operações mais fundamentais de execução do processador, cujas são controladas diretamente através dos sinais de controla da Unidade de Controle (UC). Elas são operações atômicas, por exemplo a transferência de dados (dentro ou fora da CPU) e a execução de operações aritméticas ou lógicas. Na imagem a seguir podemos ver mais nitidamente os breves ciclos de micro-operações realizados no decorrer de apenas uma instrução:

Figura 5 – Elementos que compõem a execução de programa (STALLINGS, 2017)

A fim de descrever de forma mais rigorosa as instruções que foram utilizadas pela dupla para desenvolver o código em Assembly, se faz necessário primeiro conhecer os ciclos que se mantém fixo não importa qual a instrução executada, sendo eles: O ciclo de busca, o ciclo indireto e o ciclo de interrupção.

4.1 – Ciclo de busca

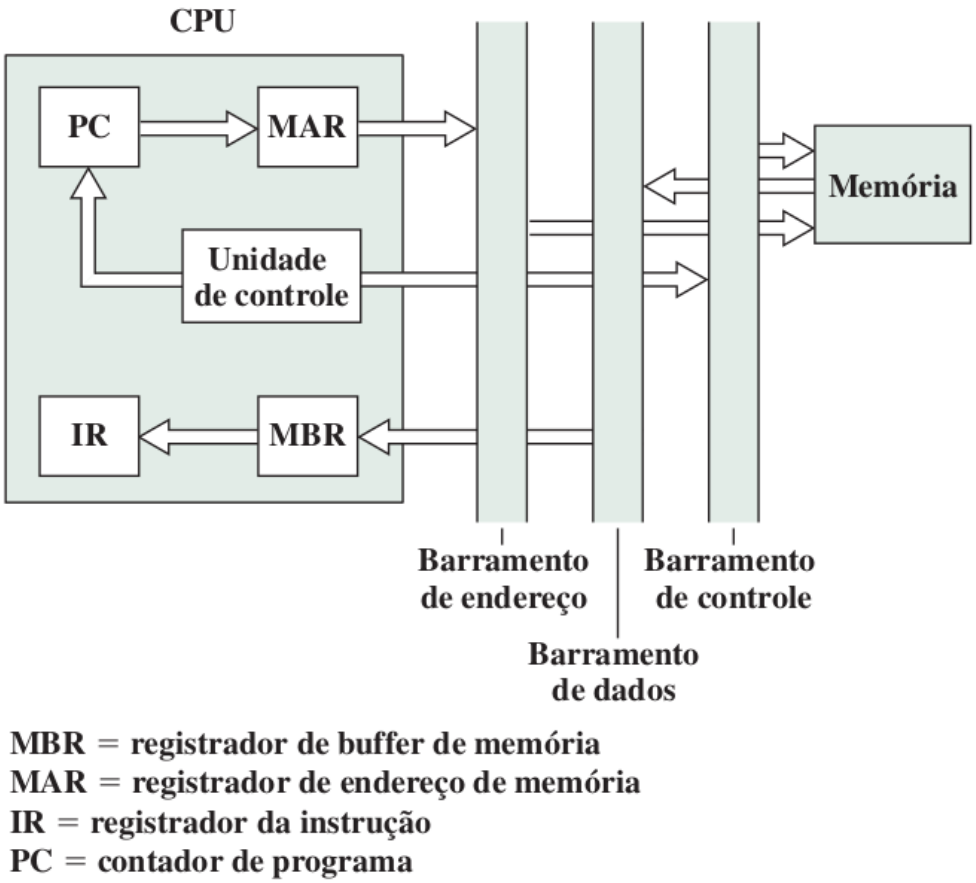


Figura 6 – Fluxo de dados do ciclo de busca. (STALLINGS, 2017)

O ciclo indicado acima é iniciado no começo de cada instrução, com o intuito de obter uma instrução da memória. Assumiremos a organização da figura acima, que possui quatro registradores relacionados:

- Contador de programa, ou *Program Counter* (PC): armazena o endereço da próxima instrução a ser lida;
- Registrador de endereço de memória, ou *Memory Address Register* (MAR): conectado às linhas de endereço do barramento do sistema. Ele diz o endereço na memória para a operação de leitura ou de escrita;
- Registrador de buffer de memória, ou *Memory Buffer Register* (MBR): conectado às linhas de dados do barramento do sistema. O mesmo contém o valor que será guardado na memória ou o último valor lido da memória;
- Registrador de instrução, ou *Instruction Register* (IR): guarda a última instrução lida.

Tal ciclo tem por finalidade, inicialmente, buscar o endereço de memória da próxima instrução a ser executada, que está localizado no PC. Então, este endereço é enviado para o MAR. Depois, o endereço desejado é enviado pelo barramento de endereços, o que faz com que a Unidade de Controle emita o comando READ no barramento de controle, e após isto o endereço é copiado para o MBR. Durante este processo o PC é incrementado. É interessante ressaltar que as operações de ler a memória e incrementar o PC não dependem entre si, desta forma, podem ser executadas ao mesmo tempo. O último passo seria mover o conteúdo do MBR para o IR.

De forma simbólica temos que;

- Primeiro passo: MAR ← (PC) - Move o conteúdo do PC para o MAR
- Segundo passo: MBR ← Memória & PC ← PC + I - Move o conteúdo na memória referenciada pelo MAR para o MBR e incrementa o PC no tamanho da instrução
- Terceiro passo: IR ← MBR - Move o conteúdo do MBR para o IR

4.2 – Ciclo Indireto

O passo seguinte é buscar os operandos fontes. Quando o ciclo de busca é finalizado, a Unidade de Controle decodifica a instrução e assume o modo de endereçamento usado pela instrução. Em MIPS, há cinco modos de endereçamento disponíveis para buscar os operandos:

- Endereçamento em registrador, no qual o operando é um registrador;
- Endereçamento imediato, no qual o operando é uma constante dentro da própria instrução;
- Endereçamento pseudo direto, no qual o endereço de salto é formado pela concatenação dos 26 bits da instrução com os bits mais significativos do PC;
- Endereçamento de base ou deslocamento, no qual o operando está em um endereço de memória que é a soma de um registrador e uma constante na instrução;
- Endereçamento relativo ao PC, no qual o endereçamento resultante é a soma do PC e uma constante na instrução.

A depender do modo de endereçamento usado pela instrução, pode ser que haja a necessidade de realizar mais um ciclo antes da execução. Aqui, vamos assumir que o formato de instrução possui um endereço, utilizando endereçamento direto e indireto.

Desta forma, se a instrução especificar um endereço indireto, então um ciclo indireto será realizado antes do ciclo de execução. Assim, o fluxo de dados irá incluir as seguintes micro-operações:

- Primeiro passo: MAR ← (IR(Endereço)) - Move o campo de endereço da instrução para o MAR;

- Segundo passo: MBR ← Memória – O endereço do operando é obtido;
- Terceiro passo: IR(Endereço) ← MBR(Endereço) – Atualiza o IR partindo do MBR, resultando no IR com o endereço direto.

Assim, agora o IR se encontra no mesmo estado, de forma que o endereçamento indireto não tivesse sido usado, mas sim o direto, possibilitando ao sistema que possa iniciar o ciclo de execução. Porém, se faz necessário notar que dados certos contextos o ciclo de interrupção pode ser iniciado após o ciclo de execução. Isso ocorre quando uma interrupção é acionada durante a execução do programa. Desta forma, faz-se necessário explorar o ciclo de interrupção, que será destrinchado a seguir.

4.3 – Ciclo de Interrupção

O ciclo de interrupção é absolutamente relevante em muitos sistemas computacionais. Este ciclo possibilita que o processador interrompa momentaneamente a execução normal do programa para tratar eventos extremos ou situações adversas. Essas paradas podem ser resultantes da entrada/saída, erros, temporizadores, solicitações do sistema operacional, entre outros.

De forma geral, enquanto ocorre um ciclo de interrupção, a CPU salva o contexto em que o programa se encontra, isso inclui os valores atuais dos registradores, salvos em um espaço na memória dedicado ao contexto de interrupção. Após isso, o fluxo de instruções é alterado, sendo direcionado para um tratador de interrupção, também chamado de rotina de serviço de interrupção, cuja função é lidar com a determinada interrupção.

A rotina de serviço de interrupção executa as ações necessárias para lidar da melhor forma com a interrupção, como por exemplo atender uma chamada do sistema operacional, uma operação de entrada/saída ou atualizar estados. Quando essa rotina é concluída, o processador então restaura o contexto anterior, retomando a execução normal do programa a partir do ponto em que foi interrompido.

O funcionamento mais detalhado deste ciclo varia muito dependendo da máquina. Desta forma, assumiremos a organização da figura 6 para descrever de forma simbólica o que ocorre neste ciclo:

- Primeiro passo: MBR ← (PC)
- Segundo passo: MAR ← Endereço_Onde_Salva, PC ← Endereço_Rotina
- Terceiro passo: Memória ← MBR

No primeiro passo o endereço do PC, que contém o endereço da próxima instrução do programa, é armazenado no MBR, para que o contexto possa ser salvo.

Já no segundo passo, temos que o MAR é carregado com o endereço onde o conteúdo do PC deve ser salvo, e o PC é carregado com o endereço do início da rotina de tratamento de interrupção.

Por fim, o último passo é armazenar o MBR, o qual contém o valor do antigo PC, em memória. Assim, o processador fica pronto para começar o próximo ciclo de instrução.

4.4 – Ciclo de execução

Os ciclos anteriormente exemplificados são caracterizados por serem simples e com padrões claros. Cada um desses ciclos executa uma sequência enxuta e simples de micro-operações que se repetem a cada vez que são executados.

A mesma coisa não ocorre com o ciclo de execução, dado a variedade imensa de opcodes que existem em cada arquitetura, existem diversas sequências de micro-operações que podem ser aplicadas no ciclo de execução. A Unidade de Controle é responsável por fazer a tradução do opcode da instrução atual e gerar a sequência correta de micro-operações com base neste valor. A esta etapa damos o nome de decodificação da instrução.

Esta etapa é do mais alto nível de criticidade, pois é ela que irá determinar quais e em que sequência ficarão as micro-operações corretas para determinada instrução e como os dados serão manipuladas. A Unidade de Controle se utiliza de uma tabela para fazer a decodificação correta dos opcodes, para garantir que as micro-operações corretas sejam executadas.

Esta abordagem mais flexível permite ao processador lidar melhor com a variedade de opcodes e instruções disponíveis. Cada instrução recebe seu próprio tratamento e forma uma sequência única de micro-operações, se adaptando e atendendo às necessidades específicas da instrução em questão.

Em seguida, explicaremos os ciclos de execução para cada instrução implementada em nosso código desenvolvido em Assembly.

4.4.1 – Ciclo de execução de cada uma das instruções no código em Assembly

Para realizar o desenvolvimento do algoritmo em Assembly aqui apresentando, fizemos o uso do simulador MARS 4.5.

4.4.1.1 - Ciclo de execução da instrução “lw”:

t1: MAR ← (Rsrc1 + offset);

t2: MBR ← Memória;

t3: Rdest ← MBR;

No primeiro passo, o endereço de memória (MAR) é atualizado adicionando o valor do registrador Rsrc1 com o valor do deslocamento (offset) especificado na instrução. Isso calcula o endereço de memória para buscar o valor requisitado.

No segundo passo, o valor da memória (MBR) é atualizado com o conteúdo da memória no endereço apontado pelo MAR. Isso envolve uma busca na memória para recuperar o valor necessário para a operação de carga.

No terceiro passo, o valor de destino (Rdest) é atualizado com o valor contido em MBR, que é o valor carregado da memória.

Essa instrução tem como finalidade carregar um valor da memória para um registrador em MIPS, onde o endereço é calculado a partir da soma de um valor do registrador Rsrc1, onde o valor é buscado na memória e por fim armazenado no registrador de destino Rdest.

4.4.1.2 - Ciclo de execução da instrução “jal”:

t1: R31 ← PC + 8;

t2: PC ← PC + (offset << 2);

O primeiro passo desta instrução é atribuir ao registrador R31 o valor atual do Program Counter (PC) acrescido de 8. Isso armazena o endereço de retorno no registrador R31, que geralmente é utilizado para retornar à instrução depois da chamada de uma sub-rotina.

No segundo passo, o valor do PC é atualizado para o valor atual do PC com o acréscimo do valor do deslocamento específico da instrução, que é deslocado para a esquerda em 2 bits. Isso possibilita que o PC prossiga para o endereço da sub-rotina, fazendo com que a execução parta a “função” desejada.

A instrução "jal" é utilizada como intuito de realizar uma chamada de sub-rotina, armazenando o endereço de retorno diretamente no registrador R31 e desviando para o endereço de destino especificado pelo deslocamento.

4.4.1.3 - Ciclo de execução da instrução “jr”:

t1: PC ← Rsrc;

Esta instrução possui um único estágio, onde o valor do Program Counter (PC) é atualizado com o valor armazenado no registrador de origem (Rsrc). Isso direcionará a execução do programa para o endereço especificado por este registrador.

O uso desta função é feito, rotineiramente, para realizar o retorno de uma sub-rotina, no qual o endereço de retorno é armazenado no registrador R31, por exemplo. Assim, quando esta

instrução é utilizada, o valor do registrador Rsrc é atualizado no PC, o que faz com que o fluxo do programa volte a um estado anterior à chamada da sub-rotina.

4.4.1.3 - Ciclo de execução da instrução “ori”:

t1: Rdest ← Rsrc1 | imm;

O valor presente no registrador Rdest é atualizado com a operação OR bit a bit entre o valor do registrador Rsrc1 e o valor imediato (imm).

Essa instrução realiza a operação lógica OR bit a bit entre os valores dos registradores, atualizando o valor do registrador de destino com o resultado. Cada bit de Rdest será igual ao valor correspondente de Rsrc1 OU imm.

Essa operação é particularmente útil para realizar combinações lógicas entre valores de registradores e valores imediatos. Ela permite definir ou modificar bits específicos em um registrador usando a operação lógica OR.

4.4.1.4 - Ciclo de execução da instrução “addiu”:

t1: MAR ← (IR(endereço));

t2: MBR ← Memória;

t3: Rdest ← (Rsrc1) + (MBR);

O primeiro passo troca o endereço de memória (MAR) com o valor do registrador Rsrc1 somado ao valor imediato (imm) da instrução. O segundo passo atualiza o valor da memória (MBR) com o conteúdo da memória no endereço indicado por MAR. Por fim o terceiro passo atualiza o registrador de destino (Rdest) com a soma do valor anterior de Rsrc1 e o valor de MBR.

4.4.1.5 - Ciclo de execução da instrução “addu”:

t1: Rdest ← Rsrc1 + Rsrc2;

Essa representação simbólica exemplifica, em um único estágio, em que o registrador de destino (Rdest) é atualizado com a soma dos valores dos registradores de origem Rsrc1 e Rsrc2. Não há a necessidade de estágios adicionais para realizar a operação de soma. Essa instrução realiza a soma sem considerar o valor imediato. Ela simplesmente adiciona os valores dos registradores de origem e armazena o resultado no registrador de destino.

4.4.1.6 - Ciclo de execução da instrução “j”:

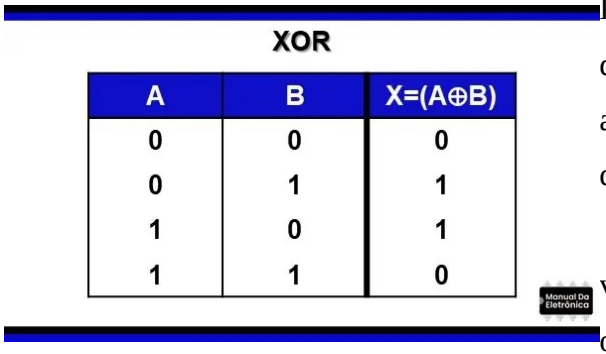
t1: PC ← PC + (offset << 2);

Esta representação, tal qual a anterior, também possui apenas uma etapa, na qual o valor do PC é atualizado com o acréscimo específico do campo da instrução, que é deslocado para a esquerda em 2 bits, o que faz com que a execução do programa prossiga de outro ponto que foi especificado na instrução.

4.4.1.7 - Ciclo de execução da instrução “xor”:

t1: Rdest ← Rsrc1 xor Rsrc2;

Aqui podemos ver mais uma instrução que possui apenas uma etapa. Nela, o conteúdo do registrador Rsrc1 é submetido à operação lógica XOR bit a bit com o conteúdo do registrador Rsrc2. Por fim o resultado de tal operação é armazenado no registrador de destino Rdest.



Abaixo podemos ver uma tabela verdade da operação XOR, que resulta em 1 se, e somente se, um dos dois valores for verdadeiro, e 0 caso ambos sejam 1 ou 0.

Figura 7 – Tabela verdade da operação lógica XOR

4.4.1.8 - Ciclo de execução da instrução “and”:

t1: Rdest ← Rsrc1 and Rsrc2;

A representação acima, tal qual as anteriores, precisa apenas de uma etapa para realizar sua função, que consiste em analisar bit a bit os valores dos registradores Rsrc1 e Rsrc2 com a operação lógica AND. No final o valor é armazenado no registrador de destino Rdest. Esta operação é comumente utilizada para verificar se dois valores são verdadeiros.

4.4.1.9 - Ciclo de execução da instrução “addi”:

t1: IR ← Memória(MAR)

t2: MBR ← sign_extend(IR(immediate))

t3: Rdest ← (Rsrc1) + (MBR)

O primeiro passo é carregar a instrução para o IR, que vem do MAR. O passo seguinte é atribuir ao MBR o valor da instrução contido no IR. Por fim, a última etapa é somar o conteúdo do registrador de origem Rsrc1 com o MBR e atribuí-lo ao registrador de destino Rdest.

4.4.1.10 - Ciclo de execução da instrução “sw”:

t1: IR ← Memória(MAR)

t2: MAR ← Rsrc1 + sign_extend(IR(offset))

t3: Memória(MAR) ← Rdest

Para a representação simbólica acima, o primeiro passo é atribuir ao IR o endereço de memória contido no MAR. Após esta operação, o valor do registrador Rsrc1 é extraído, juntamente com o offset da instrução, cujo sinal é estendido. Então, ocorre a soma do valor dos registradores para calcular o endereço efetivo, atribuindo o mesmo ao MAR. Por fim, o valor do registrador do registrador Rdest é armazenado no espaço de memória calculado.

Tal instrução permite que os valores contidos em registradores possam ser armazenados em memória, permitindo que sejam resgatados posteriormente, o que é absolutamente essencial para manipular dados, armazenar variáveis, realizar a passagem de parâmetros e realiza a manutenção de estados entre chamadas de subrotinas.

Referências Bibliográficas

Gatto, Elaine Cecília. "Primeira Instrução MIPS". Embarcados, 2016. Disponível em: https://embarcados.com.br/primeira-instrucao-mips/ . Acesso em: 24 de Junho de 2024

GEEKSFORGEEKS et al. **Computer Organization | Different Instruction Cycles**. 6.06. ed. [S. l.], 2023. Disponível em: https://www.geeksforgeeks.org/different-instruction-cycles/. Acesso em: 24 jun. 2024.

Hennessy, John LeRoy; Patterson, David A. Arquitetura de Computadores: Uma Abordagem Quantitativa. Editora: Elsevier, 2012.

MACHINE ORGANIZATION AND ASSEMBLY LANGUAGE, 2009, Washington. Slides [...]. [S. l.: s. n.], 2009. 22 p. Disponível em: https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec05.pdf. Acesso em: 25 jun. 2024.

MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual. 6.06. ed. [S. l.: s. n.], 2016. 485 p.

STALLINGS, William. Arquitetura e Organização de Computadores. 10a edição. Editora: Pearson, 2017.