UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES

BACHARELADO EM SISTEMAS DE INFORMAÇÃO

CATARINA MACEDO SCABELLI
GEOVANNY LUAN PIEDADE

Relatório: Exercício de Programação (MIPS) – OAC I

SÃO PAULO
2024

CATARINA MACEDO SCABELLI
GEOVANNY LUAN PIEDADE

Relatório: Exercício de Programação (MIPS) – OAC I

Trabalho de pesquisa realizado para a disciplina "Organização e Arquitetura de Computadores I" da Universidade de São Paulo

Professor(a): Prof. Dr. Gisele da Silva Craveiro

SÃO PAULO

2024

Sumário

1. Introdução

Nesse exercício programa implementou-se, na linguagem Assembly do MIPS, uma versão do algoritmo de Euclides para o cálculo do máximo divisor comum (MDC) de dois inteiros.

Para tanto, esse trabalho tem o objetivo de introduzir as principais características de um processador MIPS, descrevendo seu funcionamento, registradores disponíveis e os tipos e formatos das instruções que pode executar. Além disso, inicialmente, o problema foi resolvido com a utilização da linguagem de programação de alto nível C para que, com base nesse código, fosse implementado o algoritmo em Assembly.

Por fim, foi explicado todo o ciclo necessário para a execução de uma instrução, descrevendo os subciclos de busca, indireto e de interrupção, que ocorrem da mesma maneira independentemente do tipo da instrução. Porém, o ciclo de execução é único para cada instrução. Por isso, cada um deles foi descrito ao nível de suas micro operações, especificando os registradores e linhas de barramento envolvidos.

2. Organização e Arquitetura MIPS

Processadores são classificados de acordo com o número de instruções que suportam, podendo ser RISC (Reduced Instruction Set Computer) ou CISC (Complex Instruction Set Computer). Um processador CISC suporta um grande conjunto de instruções, implementando uma grande variedade de tipos, desde instruções simples até muito complexas e especializadas. Entretanto, quanto maior a quantidade de instruções que um processador suporta, mais lenta é sua execução. Já o processador RISC suporta um número limitado de instruções, porém mais simples e consequentemente mais rápidas, o que otimiza o desempenho.

Os processadores MIPS (Microprocessor without Interlocked Pipeline Stages), criados pela MIPS Computer Systems Inc, são desenvolvidos desde 1988. As primeiras versões utilizavam um conjunto de instruções de 32 bits da classe RISC e, posteriormente, versões de 64 bits foram desenvolvidas. O MIPS é uma arquitetura baseada em registrador, ou seja, a CPU usa apenas registradores para realizar as suas operações aritméticas e lógicas. [1]

As funções básicas do ciclo pelo qual passa uma instrução são:

- Buscar instruções: a CPU tem de ler as instruções a partir da memória;
- Interpretar instruções: as instruções têm de ser decodificadas para determinar a ação a executar;
- Buscar dados: a execução de uma instrução precisa ler dados dos registradores ou pode necessitar da leitura de dados da memória;
- Processar dados: realizar operações lógicas ou aritméticas sobre os dados;
- Escrever dados: os resultados de uma execução implicam escrever os dados em registradores ou na memória.

2.1 Registradores

No MIPS, estão disponíveis 32 registradores de 32 bits de propósito geral, que são livres para o uso, conforme necessidade da aplicação e são organizados em grupos. A tabela seguinte ilustra essa caracterização.

Nome	Uso
$zero	constante zero
$at	assembler temporary (reservado para o montador)
$v0, $v1	retornam resultados de funções
$a0 - $a3	argumentos de funções
$t1 - $t9	temporários, que podem ser alterados por funções (não preservados pela chamada)
$s1 - $s8	temporários, mas preservados pela chamada
$ra	return address (endereço de retorno de uma função)
$k0, $k1	reservados para o kernel
$gp	ponteiro global
$sp	stack pointer
$fp	frame pointer

2.2 Instruções MIPS

O conjunto de instruções é um fator muito importante que influencia a definição e a implementação de uma arquitetura, delimitando os tipos de operações que ela é capaz de executar e as funções que deve exercer. As instruções que o MIPS suporta, categorizadas de acordo com as operações que cada uma realiza, são as seguintes:

- Aritméticas: realizam operações aritméticas sobre números inteiros (adição e subtração);
- Lógicas: realizam operações lógicas bit a bit (AND, OR, NOR);
- Transferência de dados: movem dados entre a memória e os registradores;
- Controle de fluxo: alteram o fluxo de execução do programa e podem ser de dois tipos
    - Desvios condicionais: instruções de desvio baseada em instruções de comparação
    - Desvios incondicionais: instruções de desvio e de chamada de rotina, que transferem o controle de execução para determinada instrução
- Sistema e controle: executam operações privilegiadas e de controle do sistema.

A tabela seguinte lista essas instruções e exemplifica como são utilizadas em programas.

Categoria	Instruções	Exemplo
Aritméticas	add, addi, sub, mult, div	add $s1, $s2, $s3
Lógicas	and, andi, or, ori, sll, srl	or $s1, $s2, $s3
Transferência de dados	sw, lw, lui, mfhi, mflo	lw $s1, offset($s2)
Desvios Condicionais	beq, bne, slt, slti	beq $s1, $s2, offset
Desvios incondicionais	j, jal, jr	j label
Serviço do sistema	syscall	syscall

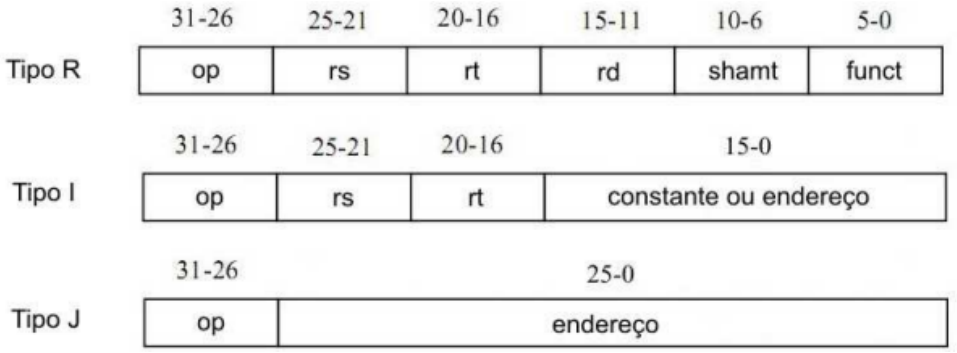
As instruções acima são aquelas que têm a execução direta. Porém, em programas que utilizam a linguagem Assembly do MIPS também existem pseudo

instruções, que são instruções que não correspondem diretamente a uma única instrução de máquina, mas são fornecidas para facilitar a programação.
Assim, O assembler as traduz em uma ou mais instruções nativas, que são suportadas pela arquitetura. A tabela abaixo elenca algumas das principais pseudo instruções do MIPS [1].

Nome	Sintaxe	Tradução
Load Address	la $rd, label	lui $t0, upper(label) ori $t0, $t0, lower(label)
Load Immediate	li $rd, immediate	addiu $t0, $zero, immediate (se o operando for menor que 16 bits) ou lui $t0, upper(label) ori $t0, $t0, lower(label) (se o operando for maior que 16 bits)
Move	move $t0, $t1	add $t0, $t1, $zero
Branch greater than	bgt $t0, $t1, label	slt $at, $t1, $t0 bne $at, $zero, label
Branch less than	blt $t0, $t1, label	slt $at, $t0, $t1 bne $at, $zero, label
Branch greater than or equal	bge $t0, $t1, label	slt $at, $t0, $t1 beq $at, $zero, label
Branch less than or equal	ble $t0, $t1, label	slt $at, $t1, $t0 beq $at, $zero, label

2.3 Formato das instruções

As instruções no MIPS podem ser de três formatos distintos [2], dependendo do seu tipo, como mostra a figura abaixo:



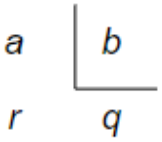
O formato do tipo R se refere às instruções lógicas e aritméticas, o tipo I se refere às de transferência de dados (como Load Word e Store Word) e de desvio condicional (como Branch on Equal) e o tipo J, às de desvio incondicional (como Jump). Os campos presentes e suas funções são:

- op (opcode): determina a operação realizada;
- rs: registrador com o primeiro operando de origem;
- rt: registrador com o segundo operando de origem;
- rd: registrador para o operando de destino;
- shamt (shift amount): quantidade de deslocamento;
- funct (function): seleciona a variante especificada no campo op;
- endereço: indica o endereço de deslocamento no caso de instruções de acesso à memória e de desvios.

3. Explicação do algoritmo matemático de Euclides

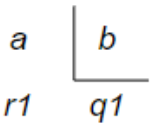
O cálculo do MDC implementado em Assembly se baseia no algoritmo de Euclides. MDC significa máximo divisor comum e, nesse caso, de dois números naturais *a* e *b*, definido como o maior número que divide tanto *a* quanto *b*. O MDC de *a* e *b* será indicado por *mdc(a,b)*. Com isso em mente, o algoritmo euclidiano traz essa solução a partir de um procedimento de divisões sucessivas, baseado no seguinte processo [3]:

Sejam *a* e *b* números naturais, tais que a>b>0.
Sejam *q* e *r* o quociente e o resto da divisão de *a* por *b*, respectivamente.
Então *mdc(a,b) = mdc(b,r)*.



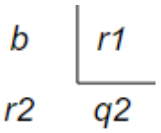
Nesse caso, não há problemas se a condição a>b>0 não for satisfeita, pois tem-se que *mdc(a,b) = mdc(b,a)* para quaisquer números naturais ambos diferentes de zero e *mdc(n,0) = n* para qualquer número natural diferente de zero.

Desse modo, segue o algoritmo de divisões sucessivas para o cálculo de *mdc(a,b)*:



Se o resto dessa divisão (*r1)* for zero, então *mdc(a,b) = mdc(b,0) = b*.

Se não, então *mdc(a,b) = mdc(b,r1)* e deve-se calcular agora *mdc(b,r1)*, como segue:



Se o resto dessa divisão (*r2)* for zero, então *mdc(b,r1) = mdc(r1,0) = r1*.

Se não, então *mdc(b,r1) = mdc(r1,r2)* e deve-se calcular agora *mdc(r1,r2)*.

E assim por diante, até que seja encontrado um resto igual a zero e, consequentemente, tem-se o MDC desejado.

4. Descrição do código em alto nível da solução

Antes da implementação do algoritmo em Assembly, o exercício foi desenvolvido na linguagem de alto nível C, para a posterior implementação comparativa do código em nível mais baixo. A figura abaixo contém o programa em C.

```c
#include <stdio.h>

// Função para calcular o MDC usando o algoritmo de Euclides
int mdc(int a, int b) {
  while (b != 0) {
    int temp = b;
    b = a % b;
    a = temp;
  }
  return a;
}

int main() {
  int num1, num2;

  // Solicita ao usuário para inserir dois números
  printf("Digite dois numeros inteiros: ");
  scanf("%d %d", &num1, &num2);

  // Calcula e exibe o MDC dos números fornecidos
  int resultado = mdc(num1, num2);
  printf("O MDC de %d e %d é: %d\n", num1, num2, resultado);

  return 0;
}

```

Em alto nível, a função baseada no algoritmo euclidiano segue o princípio de que, se *a % b == 0*, ou seja, se o resto da divisão de *a* por *b* for 0, então encontramos um divisor comum. Ou seja, caso *b*, que recebe *a % b*, seja 0, isso significa que não precisamos mais entrar no laço *while* e esse valor é o MDC.

Há, também, uma função *main* utilizada para fins de testes, que chama funções padrão da linguagem C, *printf* e *scanf* para interagir com o usuário, que preenche os números a serem utilizados. Após obter os resultados do MDC, o programa exibe o valor em questão e depois encerra.

Ou seja, esse é o resumo do funcionamento:

- O programa solicita ao usuário que digite dois números inteiros
- O usuário insere as entradas
- Após recebidas, a função MDC é chamada com esses dois parâmetros
- Se *b != 0*, inicia-se o loop
- O loop ocorre com *a* recebendo o valor antigo de *b* e com *b* recebendo *a % b*
- Quando *b == 0*, o loop se encerra e a função retorna o MDC
- O resultado é atribuído à variável resultado
- O resultado é mostrado ao usuário e o programa se encerra

5. Explicação do código em Assembly desenvolvido

O algoritmo em Assembly foi desenvolvido de maneira análoga ao código na linguagem de alto nível C, para facilitar a implementação. Alguns dos requisitos mínimos utilizados são: desvios, estrutura de repetição, chamada de subrotina, passagem de parâmetros e leitura de entradas do usuário. Além disso, uma pilha também foi usada com o objetivo de armazenar os valores de entrada lidos e depois, desempilhá-los na função *mdc* para manipulá-los e encontrar seu MDC.

O programa conta com alguns blocos importantes, cujas funções serão descritas a seguir.

5.1 Main

No bloco de execução principal, estão presentes comandos que irão preparar o programa para receber dois inteiros digitados pelo usuário, comandos que chamarão a função responsável por calcular o MDC desses números e, por fim, aqueles que imprimirão o resultado para o usuário. Essas partes envolvem os seguintes processos:

- Alocação de espaço suficiente na pilha de memória para guardar duas palavras de memória, que serão os inteiros.
- Impressão da mensagem “Digite um número”, sinalizando que o usuário pode fornecer o primeiro número escolhido ao programa.
- O processo anterior é instantaneamente seguido pela leitura da entrada do usuário e armazenamento desse valor em uma das duas posições disponíveis na pilha.
- Os dois itens acima são executados duas vezes, para a leitura das duas entradas.
- Depois de armazenados os dois valores na pilha, é feita a chamada da função que calcula o MDC, com o comando *jal mdc*.
- Quando o programa termina de executar o bloco relacionado ao cálculo euclidiano e retorna ao bloco *Main*, o valor armazenado no registrador *$v1*, responsável por retornar o valor obtido pela função MDC, é carregado no registrador *$a0* para impressão.
- Depois, é exibida a mensagem “O mdc eh:”, juntamente com o resultado final.

- Finalmente, o espaço utilizado na pilha é desalocado e a execução é encerrada.

5.2 MDC

Esse deve ser o bloco mais importante do programa, já que é responsável pelo cálculo do máximo divisor comum em si. Sua sequência de comandos pode ser facilmente comparada a do programa em C, pois ele contém o loop essencial para chegar ao resultado final. Seu funcionamento envolve os seguintes passos:

- A primeira ação é carregar os inteiros armazenados na pilha em dois registradores temporários, para sua manipulação ao longo da função.
- Depois, tem-se o loop principal *while*:
    - Nele, faz-se a comparação responsável por determinar o momento em que se deve sair do loop, *beq $t1, $zero, sair*, análoga ao comando *while(b != 0)* em C. Enquanto essa condição não for satisfeita, seguem as linhas de código necessárias para encontrar o resultado.
    - *la $t2, ($t1)* (em Assembly) é análoga a *int temp = b* (em C).
    - *div $t0, $t1* e *mfhi $t1* são as linhas análogas a *b = a % b.* Isso porque em Assembly não há uma instrução que calcule explicitamente o resto da divisão de dois inteiros, por isso, dividimos os valores armazenados pelos registradores *$t0* e *$t1* e o resto dessa razão estará no registrador *$hi*. Assim, precisamos mover esse valor para o registrador *$t1*.
    - *la $t0, ($t2)* é análoga a *a = temp*.
    - Por fim, é preciso sinalizar que o loop deve se repetir enquanto a condição acima não for satisfeita. Isso é feito com *j while*.

5.3 Sair

Esse bloco é responsável por carregar o resultado encontrado, que no momento está no registrador *$t0*, para o registrador *$v1*, utilizado para retornar o mdc calculado pela função. Além disso, também deve voltar para a função chamadora *Main* com o comando *jr $ra*.

6. Explicação detalhada das instruções utilizadas no código

A execução de um programa consiste na sequência dos ciclos de suas instruções. A execução destas envolve subciclos e estes, por sua vez, envolvem operações mais simples ainda. Dessa maneira, cada subciclo envolve um conjunto de vários passos. Esses subciclos incluem, para cada instrução, o ciclo de busca, ciclo indireto (se a instrução especificar endereçamento indireto), ciclo de execução e ciclo de interrupção. E aqueles pequenos passos, denominados microoperações, representam os processos necessários a cada um desses ciclos, que envolvem registradores da CPU [5].

No programa desenvolvido nesse exercício, foram utilizadas doze instruções do Assembly MIPS e o detalhamento de seus ciclos de execução será feito a seguir. Mas, antes disso, serão explicados também os ciclos de busca, indireto e de interrupção, que, diferentemente do ciclo de execução, são os mesmos para qualquer instrução.

6.1 Ciclo de busca

No ciclo de busca, as seguintes micro operações estão envolvidas: o processador deve buscar na memória o endereço da próxima instrução a executar, o que está no registrador PC (Program Counter). Porém, o PC não está conectado ao barramento de endereços, então o primeiro passo é mover o conteúdo do PC para o registrador MAR (Memory Access Register) e do MAR para o barramento. Juntamente com isso, a Unidade de Controle coloca no barramento de controle o comando READ. Com isso, da memória, a respectiva instrução é lida e colocada no barramento de dados e segue para o registrador MBR (Memory Buffer Register). Por fim, a instrução é colocada no registrador IR (Instruction Register) e o endereço no PC é incrementado. Tudo isso representado de maneira simbólica:

- T1: MAR ← PC
- T2:
    - MBR ← Memória
    - PC ← PC+ 1
- T3: IR ← MBR

6.2 Ciclo indireto

O ciclo indireto, sucedendo ao ciclo de busca, assume que a instrução está devidamente no IR. Com isso, ela é analisada para verificar se especifica endereçamento indireto, o que significa que, em seu campo que contém informações do operando, há o endereço de um ponteiro que aponta para um endereço em memória e este, por sua vez, aponta para o operando em si, também em memória.

Com isso, ocorrem as seguintes microoperações: o campo de endereço da instrução no IR (atualmente é o ponteiro) é colocado no MAR e segue para o barramento de endereços. Da memória, o endereço apontado por esse ponteiro é lido, passa pelo barramento de dados e chega ao MBR. Por fim, esse endereço, que ainda não é o operando em si, mas sim aponta para ele, é colocado no IR. Simbolicamente:

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

6.3 Ciclo de interrupção

Sempre e logo após se complementar o ciclo de execução de uma instrução, é feito um teste para determinar se ocorreu alguma interrupção. Se sim, é preciso tratá-la e inicia-se o ciclo de interrupção. Suas microoperações são: após o processador sinalizar que vai tratar a interrupção (acknowledgment), ele deve salvar o contexto, ou seja, armazenar o conteúdo de PC na memória. Para isso, ele move esse dado até o MBR, passando pelo barramento e chegando na memória. O endereço onde esse conteúdo deve ser armazenado é passado pela Unidade de Controle ao MAR e o comando WRITE é transmitido ao barramento de controle. Assim, o contexto é salvo em memória e o sistema operacional pode ser sobrescrito no PC para tratar a interrupção. Simbolicamente:

- T1: MBR ← (PC)
- T2:
    - MAR ← Save ADDRESS
    - PC ← Routine ADDRESS
- T3: Memória ← MBR

6.4 Ciclos de execução

O ciclo de execução difere para cada instrução [4], mas para todos os casos, assume-se que a instrução está devidamente no IR, após o ciclo de busca.

6.4.1 Instrução addi

Formato: addi $rd, $rs, operando

Descrição: Soma o valor do registrador $rs com o operando imediato, armazenando o resultado no registrador $rd. Esse tipo de instrução especifica o uso de endereçamento imediato.

Micro-operações:

- **Execução:**
    - $rd ← $rs + operando (o registrador rd recebe o conteúdo do registrador rs somado ao operando imediato)

6.4.2 Instrução add

Formato: add $rd, $rs, $rt

Descrição: É uma instrução com endereçamento direto por registrador. Ela soma os valores dos registradores $rs e $rt, armazenando o resultado no registrador $rd.

Micro-operações:

- **Execução:**
    - $rd ← $rs + $rt

6.4.3 Instrução la (Load Address)

A instrução Load Address é uma pseudo instrução, como explicado na seção 2. Assim, ela é traduzida em instruções suportadas pela arquitetura para que seja executada.

Formato: la $rd, label

Descrição: Carrega o endereço de uma etiqueta de memória no registrador $rd.

Tradução:

- lui $t0, %hi(label) (carrega os 16 bits mais significativos do endereço do rótulo)
- ori $t0, $t0, %lo(label) (combina com os 16 bits menos significativos do endereço do rótulo)

Micro-operações:

- **Execução:**
    - $t0 ← immediateHigh (primeiro $t0 recebe os bits mais significativos, operação referente à instrução lui)
    - $t0 ← $t0 OR immediateLow (depois, recebe os bits menos significativos, concatenando-os com os mais significativos com a operação OR, referente à instrução ori)

6.4.4 Instrução li (Load Immediate)

A instrução Load Immediate também é uma pseudo instrução, havendo duas possibilidades de tradução a depender do tamanho do operando imediato.

Formato: li $rd, immediate

Descrição: Carrega um valor imediato no registrador $rd.

Tradução:

- Caso 1: se o imediato cabe em 16 bits, ele pode ser carregado diretamente:
    - addiu $t0, $zero, immediate
- Caso 2: se não, outras instruções são necessárias:
    - lui $t0, %hi(immediate) (carrega os 16 bits mais significativos do valor imediato em $t0)
    - ori $t0, $t0, %lo(immediate) (carrega os 16 bits menos significativos em $t0)

Micro-operações:

- **Execução - Caso 1:**
    - $t0 ← $zero + immediate ($t0 recebe normalmente a soma do valor no registrador $zero, que é sempre 0, com o operando)
- **Execução - Caso 2:**
    - $t0 ← immediate High (primeiro $t0 recebe os bits mais significativos do valor imediato, operação referente à instrução lui)
    - $t0 ← $t0 OR immediateLow (depois, recebe os bits menos significativos, concatenando-os com os mais significativos com a operação OR, referente à instrução ori)

6.4.5 Instrução sw (Store Word)

Formato: sw $rt, offset($base)

Descrição: Armazena o valor do registrador $rt na memória, no endereço calculado como a soma do valor do registrador base com o offset.

Micro-operações:

- **Execução:**
    - T1: MBR ← (rt) (o conteúdo do registrador $rt é colocado no MBR)
    - T2: MAR ← Reg(base) + offset (o endereço de memória é calculado e colocado no MAR)
    - T3: Memória ← (MBR) (o conteúdo do MBR é escrito no endereço de memória calculado)

6.4.6 Instrução lw (Load Word)

Formato: lw $rd, offset($base)

Descrição: Carrega uma palavra (valor) de uma posição específica de memória para o registrador $rd. O endereço correto é calculado como a soma do valor de base com o offset.

Micro-operações:

- **Execução:**
    - T1: MAR ← Reg(base) + offset (o endereço da palavra de memória é calculado e colocado no MAR)
    - T2: MBR ← (Memória) (o conteúdo desse endereço é lido da memória e colocado no MBR)
    - T3: rd ← (MBR) (o conteúdo do MBR é colocado no registrador $rd)

6.4.7 Instrução beq (Branch if Equal)

Formato: beq $rs, $rt, offset

Descrição: Desvia para a posição de memória especificada pelo offset se os valores dos registradores $rs e $rt forem iguais.

Micro-operações:

- **Execução:**

- if ($rs == $rt) then PC ← PC + 4 + offset (se o valor dos registradores rs e rt forem iguais, o endereço de desvio calculado é carregado no PC)
- else PC ← PC + 4 (se os valores não forem iguais, o PC é incrementado normalmente)

6.4.8 Instrução div (Divide)

Formato: div $rs, $rt

Descrição: Divide o valor do registrador $rs pelo valor do registrador $rt e armazena o quociente e o resto em registradores diferentes. O quociente é carregado em LO e o resto, em HI.

Micro-operações:

- **Execução:**
  - LO ← $rs / $rt
  - HI ← $rs % $rt
- **Verificação de divisão por zero:** Verifica se $rt é zero e, se necessário, gera uma interrupção.
  - if ($rt) == 0 then Interruption

6.4.9 Instrução mfhi (Move from HI)

Formato: mfhi $rd

Descrição: Move o valor em HI para o registrador $rd.

Micro-operações:

- **Execução:**
  - $rd ← (HI)

6.4.10 Instrução jal (Jump and Link)

Formato: jal label

Descrição: Salta para o endereço especificado por label e armazena o endereço de retorno no registrador ra (registrador 31). É usado para chamada de sub-rotinas, permitindo que o controle do programa retorne ao ponto de chamada após a execução da subrotina.

Micro-operações:

- **Execução:**
    - T1: Reg[31] ← PC + 8 (calcula o endereço de retorno e armazena no registrador ra)
    - T2: PC ← label (atualiza o PC para o endereço da chamada da subrotina)

6.4.11 Instrução j (Jump)

Formato: j label

Descrição: Desvio incondicional que salta para o endereço especificado por label.

Micro-operações:

- **Execução:**
    - PC ← label (atualiza o PC para o endereço label)

6.4.12 Instrução jr (Jump Register)

Formato: jr $rs

Descrição: Salta para o endereço contido no registrador $rs.

Micro-operações:

- **Execução:**
    - PC ← ($rs) (atualiza o contador de programa (PC) com o valor lido do registrador rs)

6.4.13 Instrução syscall

Formato: syscall

Descrição: Gera uma interrupção do sistema para chamar uma função do sistema operacional. É usado para operações como entrada/saída, alocação de memória, etc; e a operação a ser executada é determinada pelo código de serviço no registrador v0, geralmente.

Micro-operações:

- **Execução:**
    - T1: ServiceCode ← (v0) (lê o código de serviço do registrador v0)
    - T2: SyscallHandler ← ServiceCode (executa a chamada do sistema baseado no código de serviço lido)

- T3: Result ← SyscallHandler (processa qualquer retorno ou resultado da chamada do sistema)

7. Conclusão

O exercício proposto proporcionou uma compreensão aprofundada da arquitetura MIPS e de suas instruções. Começando pela descrição detalhada do funcionamento de um processador MIPS, seus registradores e instruções, passando pela explicação do algoritmo de Euclides, foi possível desenvolver a solução tanto em C quanto em Assembly. A análise dos ciclos de execução das instruções permitiu compreender a complexidade e a sequência de microoperações envolvidas na execução do programa, reforçando a importância de cada detalhe na programação de baixo nível.

8. Referências bibliográficas

[1] ARQUITETURA MIPS. In: WIKIPÉDIA, a enciclopédia livre. Flórida: Wikimedia Foundation, 2023. Disponível em: <https://pt.wikipedia.org/w/index.php?title=Arquitetura_MIPS&oldid=65497606>

[2] FREITAS, A.; JUNIOR, C.; SOUZA, M.; GONÇALVES, R. “Arquitetura MIPS: desenvolvimento de um simulador”, Departamento de Informática – Universidade Estadual de Maringá (UEM), 2010, p.44-57. Disponível em: <https://www.din.uem.br/~wesley/wordpress/wp-content/uploads/2011/05/Anais-Fitem-2010.pdf#page=44>

[3] GIACOMO, S. “Sala de estudo: algoritmo de euclides para determinação de mdc”, Equipe COM-OBMEP, Rio de Janeiro, 2023. Disponível em: <http://clubes.obmep.org.br/blog/sala-de-estudos-algoritmo-de-euclides-para-determinacao-de-mdc/>

[4] PRICE, Charles. “MIPS IV Instruction Set”. Mountain View: MIPS Technologies, Inc, 1995. Revisão 3.2. Disponível em: <https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf>

[5] STALLINGS, W. Estrutura e função do processador. In: STALLINGS, W. Organização e Arquitetura de Computadores. 10. ed. São Paulo: Pearson, 2017. p.417-451.