

## *Apostila de MIPS*



Universidade Estadual de Londrina  
Curso de Ciência da Computação

Larissa Dantas Vilaca

Rafael Lucien Bahr Arias

Orientador: Prof. Fábio César Martins

## Apostila de MIPS

Versão 1

Universidade Estadual de Londrina  
Curso de Ciência da Computação

---

# Lista de ilustrações

Figura 1.1 – Formato R de Instrução . . . . .	14
Figura 1.2 – Formato I de Instrução . . . . .	15
Figura 1.3 – Formato J de Instrução . . . . .	15
Figura 1.4 – Simplificação da arquitetura MIPS . . . . .	18
Figura 2.1 – Tela inicial do MARS . . . . .	20
Figura 2.2 – Coprocessadores no MARS . . . . .	22
Figura 2.3 – Representação Numérica Binária . . . . .	23
Figura 2.4 – Menu de Ajuda do MARS . . . . .	25
Figura 2.5 – Montagem do programa . . . . .	31
Figura 2.6 – Painel de execução . . . . .	32
Figura 2.7 – String no segmento de dados . . . . .	34
Figura 4.1 – Chamada e retorno de sub-rotina . . . . .	65
Figura 4.2 – Segmento de Pilha . . . . .	70
Figura 5.1 – Dados na memória dinâmica heap . . . . .	94

---

# Lista de tabelas

Tabela 1.1 – Registradores do MIPS . . . . .	10
Tabela 2.1 – Diretivas Principais . . . . .	26
Tabela 2.2 – Principais Chamadas de Sistema . . . . .	29
Tabela 3.1 – Operações Aritméticas . . . . .	38
Tabela 3.2 – Operações Lógicas . . . . .	44
Tabela 3.3 – Instruções de Conversão . . . . .	49
Tabela 4.1 – Instruções de Bifurcação . . . . .	54
Tabela 4.2 – Instruções de Comparação (P. F) . . . . .	59
Tabela 4.3 – Instruções de Bifurcação (P. F) . . . . .	60

---

# Lista de programas

2.1	Olá Mundo . . . . .	30
3.1	Operações com Números Inteiros . . . . .	39
3.2	Operações com Números Reais . . . . .	41
3.3	Potência de dois com shift . . . . .	46
4.1	Classificação de Idade . . . . .	57
4.2	Leitura enquanto N está no intervalo . . . . .	62
4.3	Cálculo de Somatório . . . . .	62
4.4	Maior valor com procedimentos . . . . .	66
4.5	Fibonacci Recursivo . . . . .	75
5.1	Leitura e Escrita de Vetor . . . . .	80
5.2	Intercalação de Strings . . . . .	84
5.3	Leitura e Escrita de Matriz . . . . .	90
5.4	Matriz Real Dinâmica . . . . .	95
6.1	Contagem de caracteres em arquivo . . . . .	105
6.2	Leitura e soma de valores em arquivo . . . . .	107
6.3	Escrita de valores ímpares em arquivo . . . . .	111

---

# Sumário

<b>Introdução</b>	<b>6</b>
<b>1 A Arquitetura MIPS</b>	<b>8</b>
1.1 Registradores	9
1.2 Unidade Lógica e Aritmética	11
1.3 Contador de Programa	12
1.4 Registrador de Instrução	13
1.5 Unidade de Controle	13
1.6 Conjunto de Instruções	13
1.7 Endereçamento	16
<b>2 A IDE do MARS</b>	<b>19</b>
2.1 Visão Geral e Registradores	20
2.2 Menu de Ajuda	24
2.3 Olá, Mundo!	30
2.4 Depuração	31
<b>3 Aritmética e Operadores Lógicos</b>	<b>35</b>
3.1 Endereçamento e Atribuição de Valores	35
3.2 Operações Aritméticas	38
3.3 Operações Lógicas	43
3.4 Conversão de Tipos	48
<b>4 Estrutura de Controle e Sub-rotinas</b>	<b>51</b>
4.1 Rótulos	51
4.2 Desvio Incondicional	52
4.3 Desvio Condicional	53
4.3.1 Comparação de Inteiros	54

4.3.2	Comparação de Ponto Flutuante . . . .	58
4.4	Iteração . . . . .	61
4.5	Sub-rotinas e Procedimentos . . . . .	63
4.6	Pilha e Recursividade . . . . .	68
<b>5</b>	<b>Arranjos N-Dimensionais . . . . .</b>	<b>77</b>
5.1	Vetores . . . . .	77
5.2	Strings . . . . .	82
5.3	Matrizes . . . . .	87
5.4	Memória Dinâmica ( <i>Heap</i> ) . . . . .	92
<b>6</b>	<b>Manipulação de Arquivos . . . . .</b>	<b>100</b>
6.1	Leitura . . . . .	103
6.2	Escrita . . . . .	109
	<b>Referências . . . . .</b>	<b>114</b>

---

# Introdução

**E**STA APOSTILA tem como objetivo auxiliar estudantes na programação com a linguagem assembly MIPS no ambiente do MARS. Será feita uma abordagem mais prática, apresentando os conceitos e recursos da linguagem e exemplificando com instruções MIPS, podendo utilizar também o código em C correspondente para melhor entendimento. A linguagem C foi escolhida para auxiliar pois acreditamos que ela é um ponto em comum para a maioria dos estudantes que farão uso desta apostila. Da mesma forma, assume-se que os leitores desta apostila já dominam lógica de programação e desenvolvimento de algoritmos em linguagens de alto nível. Assim, compreendemos que devido à experiência, muitas vezes o estudante de MIPS já possui em mente “o que quer fazer” ao programar, ou seja, a lógica para executar determinada tarefa em outra linguagem, mas não sabe como reproduzi-la em MIPS.

Pretende-se transmitir todo o conhecimento necessário



para programar em MIPS com eficiência, usufruindo de todas as funcionalidades oferecidas pela linguagem e o ambiente do MARS. Através da experiência nos estudos de MIPS, constatamos que apesar de existirem vários materiais e livros sobre o assunto, o conteúdo ainda é escasso se comparado às linguagens de programação mais comuns, e a maior parte está na língua inglesa. Além disso, muitas das questões se encontram de maneira difusa na internet, e algumas mais específicas podem ser extremamente difíceis ou impossíveis de se obter uma explicação ou exemplo satisfatórios. Por isso, espera-se que a apostila seja útil no sentido de reunir os tópicos da linguagem de forma clara e organizada, servindo como um material de apoio de alta qualidade e cumprindo o papel de facilitar o aprendizado de MIPS.

Este trabalho deverá ser continuado com o aprimoramento do conteúdo apresentado bem como o acréscimo de outros mais avançados e exemplos.

# A Arquitetura MIPS

MIPS (Microprocessor without Interlocked Pipeline Stages) é uma arquitetura de microprocessadores RISC (Reduced Instruction Set Computing) desenvolvida pela MIPS Technologies, antigamente conhecida como MIPS Computer Systems. As primeiras arquiteturas de MIPS foram de 32 bit, com versões de 64 bit lançadas posteriormente. O funcionamento da arquitetura se baseia no fato de que conjuntos pequenos e simples de instrução, com aproximadamente o mesmo tempo de execução, fornecem melhor desempenho quando combinados com arquiteturas de microprocessadores capazes de executar instruções com um número menor de ciclos por instrução, como no caso do MIPS.

Devido à natureza desse tipo de conjunto de instruções, o modelo é denominado “reduzido”. Nos processadores baseados em RISC, não há microprogramação, ou seja, as instruções são executadas diretamente pelo hardware. A arquitetura do tipo oposto é a CISC (Complex Instruction Set Computing), que usa microprogramação. Os processadores baseados em

MIPS são usados em inúmeras aplicações, como sistemas embarcados, alguns dispositivos portáteis, computadores da Silicon Graphics, roteadores da Cisco, e até mesmo videogames como Nintendo 64 e PlayStation.

Uma das vantagens de aprender e escrever em linguagens assembly é entender melhor como funcionam compiladores, e como computadores operam em seu nível mais fundamental de bit. Outro benefício em compreender o funcionamento de programas em baixo nível é entender como escrever códigos de alto nível mais eficientes, uma vez sabendo que toda linha de código é transformada em uma ou mais instruções simples de baixo nível. Em termos de mercado, o conhecimento em MIPS também pode ser útil para a área promissora de processadores integrados.

Para compreender melhor o funcionamento da arquitetura MIPS, é necessário saber que a arquitetura de qualquer computador é composta pelos registradores que estão disponíveis para a linguagem assembly, o conjunto de instruções, os modos de endereçamento de memória, e os tipos de dados. A seguir, serão descritos os componentes básicos da arquitetura MIPS, que também incluem além das partes citadas, uma unidade lógica e aritmética (ALU), um contador de programa, um registrador de instrução (IR) e uma unidade de controle.

## 1.1 Registradores

A arquitetura MIPS possui um arquivo contendo 32 registradores, onde cada registrador é um componente com capacidade para armazenar 32 bits, que são equivalentes a 4 bytes ou

uma *word*. Logo, um registrador pode representar um valor no intervalo de -2,147,483,648 a +2,147,483,647.

Existe uma convenção que especifica quais registradores são adequados para se utilizar em cada situação. A tabela 1.1 apresenta os 32 registradores (de r0 a r31) divididos em grupos, cujo nome assembly indica sua função específica. Ao programar em MIPS, as instruções sempre referenciam os registradores pelo nome, como \$t0, \$v0 e \$a0.

Tabela 1.1 – Registradores do MIPS

<i>Registrador</i>	<i>Nome Assembly</i>	<i>Comentário</i>
\$0	\$zero	Valor constante zero, apenas leitura
\$1	\$at	Assembler temporary, reservado para o assembler para implementar macro-instruções
\$2-\$3	\$v0-\$v1	Armazenam valores retornados de sub-rotinas
\$4-\$7	\$a0-\$a3	Armazenam argumentos para sub-rotinas
\$8-\$15	\$t0-\$t7	Registradores de valor temporário
\$16-\$23	\$s0-\$s7	Registradores de valor salvo
\$24-\$25	\$t8-\$t9	Mais registradores de valor temporário
\$26-\$27	\$k0-\$k1	Reservados para o kernel do sistema operacional
\$28	\$gp	Ponteiro global, endereça variáveis globais estáticas
\$29	\$sp	Ponteiro de pilha
\$30	\$fp	Ponteiro de frame
\$31	\$ra	Endereço de retorno na chamada de uma sub-rotina

Fonte: <<https://courses.cs.washington.edu/courses/cse410/09sp/examples/MIPSCallingConventionsSummary.pdf>>

Vale notar que a especificação é apenas uma convenção, então no caso dos registradores do tipo \$t e \$s, por exemplo,

não há diferenças práticas ao utilizá-los para guardar valores do programa. A especificação apenas *recomenda* que, ao construir o código MIPS, ele dê garantia de que nas chamadas de sub-rotinas os registradores do tipo `$s` possuam os mesmos valores armazenados que continham antes da sub-rotina ser chamada (mesmo que a sub-rotina modifique os valores, eles devem ser restaurados nos respectivos registradores antes da sub-rotina terminar); enquanto que para os do tipo `$t`, não há nenhuma garantia que os valores sejam mantidos.

O cumprimento da convenção pode ser útil para o compartilhamento de código entre programadores. No contexto de compiladores, os valores nos registradores do tipo `$s` são inseridos na pilha para que possam ser restaurados após a execução de uma sub-rotina.

Já para os registradores do tipo `$v` e `$a`, existe uma utilidade real em preservá-los, pois eles são os únicos que podem ser usados para argumentos e retornos de chamadas de sistema, como será visto em capítulos mais adiante. No entanto, nada impede que o programador também utilize-os no armazenamento de valores para outras finalidades.

## 1.2 Unidade Lógica e Aritmética

A Unidade Lógica e Aritmética (ALU - Arithmetic and Logic Unit) é um circuito lógico digital projetado para executar operações aritméticas binárias (como soma e subtração) e operações lógicas binárias (como AND, OR e XOR). A operação que será executada pela ALU irá depender do código de operação no registrador de instrução.

## 1.3 Contador de Programa

O código assembly de um programa escrito em um editor de texto é convertido ou “montado” em linguagem de máquina por um programa utilitário chamado *assembler*. O código em linguagem de máquina fica armazenado em um arquivo no disco. Para executá-lo, outro programa utilitário chamado de *linking loader* carrega e conecta na memória principal todos os módulos de linguagem de máquina necessários. As instruções são armazenadas de forma sequencial na memória.

O contador de programa (PC - Program Counter) é um registrador que é inicializado pelo sistema operacional para endereçar a primeira instrução do programa na memória. Quando uma instrução é buscada na memória e carregada no contador de programa, ele é incrementado de forma que aponte para a próxima instrução. Como toda instrução na arquitetura MIPS tem o tamanho de 32 bits, o contador de programa é incrementado em 4 a cada vez que uma instrução é movida. Um nome mais apropriado para este registrador seria “apontador de programa”.

É possível acessar e modificar o contador de programa por meio das instruções de *branch* e *jump*, que executam um desvio condicional ou incondicional, respectivamente. Essencialmente, o que as instruções fazem é alterar o valor no contador de programa para o endereço de memória especificado.

## 1.4 Registrador de Instrução

O registrador de instrução (IR - Instruction Register) armazena uma cópia da última instrução buscada na memória. Como os outros registradores, ele também possui capacidade de 32 bits.

## 1.5 Unidade de Controle

A unidade de controle é implementada em hardware como uma máquina de estados finitos, que gera sinais em uma sequência específica para os outros componentes realizarem determinada tarefa. Cada componente possui um input específico. No caso dos registradores, a ativação do input implica que um novo valor será carregado da memória. Para a ALU, os sinais irão especificar qual operação deve ser executada. A memória cache precisa receber sinais para especificar quando uma operação de leitura ou escrita deve ser realizada. A velocidade de execução da unidade de controle irá depender da frequência do gerador de clock; os computadores modernos funcionam com taxas na faixa de mega-hertz.

## 1.6 Conjunto de Instruções

A arquitetura MIPS possui dezenas de instruções, mas a maioria delas é apenas uma variação de determinada instrução. Por exemplo: para a instrução de soma, existe uma versão para cada um dos tipos de dado numérico (inteiro, float e double), mais as variações *signed* e *unsigned*, com operando de valor imediato ou de registrador, etc. Logo, apenas algumas variações principais de cada instrução serão vistas nesta

apostila. Porém, uma lista completa com todas as instruções disponíveis pode ser consultada no menu de ajuda do ambiente do MARS, que será apresentado no próximo capítulo.

As instruções básicas podem ser classificadas em alguns tipos, como as instruções de operação aritmética (*add*, *sub*, *mul*, *div*), operação lógica bit a bit (*and*, *or*, *nor* e *xor*) ou de deslocamento de bits (*shift*), como *sll*, *slr* e *slt*. Elas possuem o formato denominado “R” de instruções MIPS, utilizando 3 registradores.

No formato R, o espaço de 32 bits das instruções é distribuído da forma apresentada na figura 1.1, onde “op” indica a operação básica da instrução, chamada tradicionalmente de *opcode*; “rs” e “rt” são respectivamente o primeiro e segundo operando de origem, “rd” é o operador de destino, “shamt” (*shift amount*) a quantidade de deslocamento e “funct” o código de função da instrução.

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

Figura 1.1 – Formato R de Instrução

Há as instruções de *branch* ou “bifurcação”, usadas para criar estruturas de controle com desvios condicionais, como *bne* (diferente de), *beq* (igual a) e *bgtz* (maior que zero); elas são do formato I representado na figura 1.2, junto das versões com operador imediato das instruções aritméticas e lógicas, envolvendo 2 registradores. Os últimos 16 bits são usados para especificar o endereço do desvio condicional ou o valor



imediato da instrução.

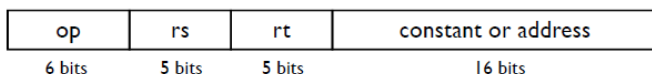


Figura 1.2 – Formato I de Instrução

Outras instruções executam desvio incondicional como *j* (jump), *jal* (jump and link) e *jr* (jump register), utilizando um único valor ou registrador, sendo classificadas em um formato diferente denominado “J”. Possui uma separação mais simples como pode ser visto na figura 1.3, com apenas 6 bits para indicar a operação básica, e 26 bits para o endereço de destino do desvio.

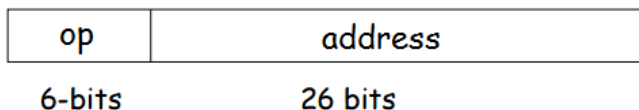


Figura 1.3 – Formato J de Instrução

As instruções citadas são executadas com alguns dos 32 registradores disponíveis na arquitetura MIPS, manipulando os valores que neles estão armazenados. As únicas instruções que acessam a memória principal são as instruções de *load* e *store*, do formato I, como *lw* e *sw* (load word e store word) e suas variações.

Também existem instruções especiais que não utilizam registradores, como *break* para causar uma exceção de inter-

rupção e *eret* para retornar da exceção. Outra instrução desse tipo que é extremamente importante é a *syscall*, que realiza uma chamada de sistema (*system call*); ela não especifica um registrador pois a operação executada será sempre definida pelo valor no registrador *\$v0*, com os argumentos nos registradores do tipo *\$a*. Esta instrução será vista com mais detalhes em capítulos adiante.

## 1.7 Endereçamento

Apenas um modo de endereçamento foi implementado em hardware para buscar valores da memória principal, ou armazenar valores na memória principal. Ele consiste em acessar um endereço base somado a um deslocamento.

Como foi dito, a arquitetura MIPS só acessa a memória principal por meio de instruções *load* ou *store*. As instruções de *load* acessam um valor da memória, e armazenam uma cópia deste valor no arquivo de registradores. Então, considerando a seguinte instrução:

---

```
lw $s0, 4($t0)
```

---

Esta instrução irá computar o endereço de memória a ser acessado adicionando o conteúdo do registrador *\$t0*, que será o endereço base, com a constante 4, que é o offset representando 4 bytes. Uma cópia do valor acessado na memória no endereço calculado será carregada no registrador *\$s0*. Um pseudocódigo equivalente seria “*\$s0* = memória[*\$t0* + 4]”.

Em todo acesso à memória, o endereço base será sempre o valor armazenado em um dos registradores disponíveis da

arquitetura MIPS. O deslocamento é sempre uma constante, cujo valor pode ser representado em uma capacidade de 16 bits, estando portanto na faixa de -32768 a +32767.

No caso da instrução *lw*, o endereço final deve ser um valor múltiplo de 4, pois uma word possui 4 bytes. Se fosse a instrução *lb* (load byte), o endereço poderia ser qualquer valor indexável, pois é múltiplo de 1 byte. A instrução *lw* pode ser usada para manipular números inteiros, já que possuem 4 bytes, enquanto a instrução *lb* é mais usada para manipular caracteres, que são formados por 1 byte ou 8 bits.

A instrução *sw* (store word) é bastante similar, mas o valor no primeiro registrador é que será carregado na memória, no endereço calculado. Então considerando a instrução:

---

*sw \$s0, (\$t0)*

---

O valor armazenado em \$s0 será carregado exatamente no endereço base em \$t0, pois não há offset nesta instrução. A notação 0(\$t0) resultaria na mesma ação. Um pseudocódigo equivalente seria “memória[\$t0] = \$s0”. Vale ressaltar que não há nenhum motivo especial para a utilização ou disposição específica dos registradores do tipo \$s e \$t nestes exemplos.

Para um programador da linguagem assembly, a memória pode ser entendida como um longo vetor, onde o endereço é o índice ou ponteiro para um campo que contém algum dado. Uma seção deste vetor é designada pelo sistema operacional como o segmento de dados. O contador de programa também é um ponteiro para este vetor, mas em um segmento diferente denominado segmento do programa. O sistema operacional aloca outra seção na memória, ou no “vetor”, denominada

segmento de pilha.

O diagrama da figura 1.4 representa o fluxo de dados entre os componentes da arquitetura MIPS descritos neste capítulo.

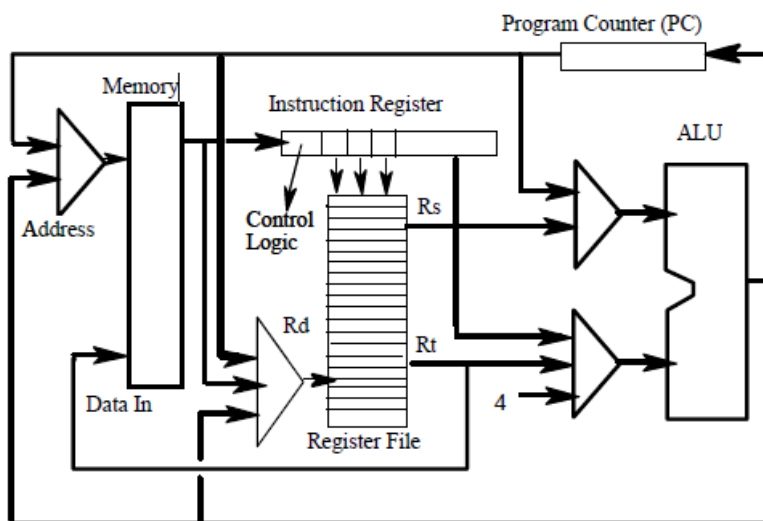


Figura 1.4 – Simplificação da arquitetura MIPS

## A IDE do MARS

Existem inúmeros simuladores de MIPS disponíveis na internet, tanto para uso educacional quanto comercial. Nesta apostila, será usada uma IDE chamada *MIPS Assembler and Runtime Simulator* (MARS), que é completamente gratuita, fácil de utilizar e possui excelentes ferramentas para quem deseja aprender a programar em MIPS. Outra boa alternativa muito semelhante ao MARS é o SPIM MIPS Simulator.

O MARS foi desenvolvido por Pete Sanderson e Kenneth Vollmar, com artigo publicado em 2005, cuja documentação e download do software podem ser encontrados no site da Missouri State University<sup>1</sup>. Como o programa foi feito em Java, é necessário possuir o JRE (Java Runtime Environment) instalado para executar o arquivo jar do MARS.

Neste capítulo, falaremos sobre os principais recursos disponíveis no ambiente do MARS, que tendo sido criado com propósitos educacionais, possui várias ferramentas de visu-

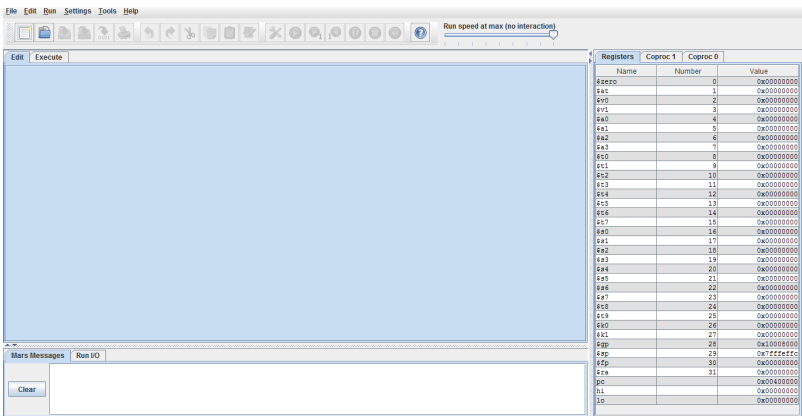
---

<sup>1</sup> <<http://courses.missouristate.edu/KenVollmar/MARS/>>

alização para melhor entendimento do funcionamento da arquitetura, assim como outras de depuração (debugging) para facilitar a detecção e remoção de erros durante a programação.

2.1 Visão Geral e Registradores

Ao iniciar o MARS, pode-se constatar que o ambiente simples é similar a qualquer outra IDE comum, com uma barra de ferramentas e uma janela para edição de texto e saída, mas já é possível identificar um aspecto importante da arquitetura MIPS: o arquivo de 32 registradores, exibidos no lado direito da tela, como pode ser visto na figura 2.1.



não se tem acesso direto na programação. Um deles é o \$pc (Program Counter) visto na seção 1.3. Os outros dois registradores, \$hi (high) e \$lo (low), são usados em operações de multiplicação e divisão, cujos resultados podem ser maiores que 32 bits; no caso da multiplicação, o produto resultante pode possuir até 64 bits, e para a divisão, a operação resulta em um quociente e resto, onde o primeiro resultado é armazenado no registrador \$lo e o último em \$hi. Os valores nestes 2 registradores podem ser buscados ou atribuídos a partir das instruções mflo e mfhi (move from) ou mtlo e mthi (move to), respectivamente. Um exemplo de uso para a divisão será visto no capítulo 3.

Cada linha na aba *Registers* representa um dos registradores, onde a primeira coluna mostra o seu nome de acordo com a convenção estabelecida; a segunda coluna indica o número do registrador (de 0 a 31) ou nada, no caso dos 3 registradores especiais; e a terceira coluna mostra o valor armazenado no registrador em tempo real. Por padrão, ele é representado na forma hexadecimal, mas é possível alterar para base decimal no menu *Settings*, com a opção de converter apenas valores ou endereços de memória separadamente.

Originalmente, valores de ponto flutuante foram implementados em um chip separado denominado coprocessor 1, também chamado de FPA (Floating Point Accelerator). Chips modernos de MIPS já incluem operações de ponto flutuante no processador principal. Porém, geralmente as instruções executam como se estivessem em um chip separado. Já o coprocessor 0, CP0 ou System Control Coprocessor é parte essencial da arquitetura e possui a função de configurar o gerenciamento de exceções ou relatar o estado de exceções atuais do programa. Cada um destes coprocessadores possui

seus próprios registradores, que no MARS estão dispostos em 2 abas separadas da aba *Registers*, nomeadas *Coproc 1* e *Coproc 0*, como mostrado na figura 2.2.

Registers Coproc 1 Coproc 0			Registers Coproc 1 Coproc 0		
Name	Float	Double	Name	Number	Value
\$f0	0x00000000	0x0000000000000000	\$8 (vaddr)	8	0x00000000
\$f1	0x00000000		\$12 (status)	12	0x0000ff11
\$f2	0x00000000	0x0000000000000000	\$13 (cause)	13	0x00000000
\$f3	0x00000000		\$14 (epc)	14	0x00000000
\$f4	0x00000000	0x0000000000000000			
\$f5	0x00000000				
\$f6	0x00000000	0x0000000000000000			
\$f7	0x00000000				
\$f8	0x00000000	0x0000000000000000			
\$f9	0x00000000				
\$f10	0x00000000	0x0000000000000000			
\$f11	0x00000000				
\$f12	0x00000000	0x0000000000000000			
\$f13	0x00000000				
\$f14	0x00000000	0x0000000000000000			
\$f15	0x00000000				
\$f16	0x00000000	0x0000000000000000			
\$f17	0x00000000				
\$f18	0x00000000	0x0000000000000000			
\$f19	0x00000000				
\$f20	0x00000000	0x0000000000000000			
\$f21	0x00000000				
\$f22	0x00000000	0x0000000000000000			
\$f23	0x00000000				
\$f24	0x00000000	0x0000000000000000			
\$f25	0x00000000				
\$f26	0x00000000	0x0000000000000000			
\$f27	0x00000000				
\$f28	0x00000000	0x0000000000000000			
\$f29	0x00000000				
\$f30	0x00000000	0x0000000000000000			
\$f31	0x00000000				
Condition Flags					
<input type="checkbox"/> 0	<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3		
<input type="checkbox"/> 4	<input type="checkbox"/> 5	<input type="checkbox"/> 6	<input type="checkbox"/> 7		

Figura 2.2 – Coprocessadores no MARS

Pode-se observar que em *Coproc 1* também há 32 registradores, nomeados de \$f0 até \$f31, que podem representar valores de ponto flutuante com *single precision* (float) ou *double precision* (double float). Deve ser ressaltado que apenas os registradores de número par devem receber valores do tipo double. Isto se deve ao fato de que todo registrador na arquitetura MIPS pode armazenar no máximo 32 bits ou 4 bytes, que é a capacidade para float; logo, como um double exige



8 bytes de espaço, são necessários 2 registradores. Então, os registradores disponíveis para o armazenamento de um único double são separados em conjuntos de 2, como \$f0 e \$f1, \$f2 e \$f3, \$f4 e \$f5, etc. Porém, os bytes do valor iniciarão sempre nos registradores pares \$f0, \$f2, \$f4, e assim por diante. A figura 2.3 mostra como os 32 ou 64 bits são organizados para armazenar o tipo inteiro e ponto flutuante de precisão simples e dupla, respectivamente. A representação binária está de acordo com o padrão IEEE 754 sobre *Binary Floating-Point Arithmetic*.

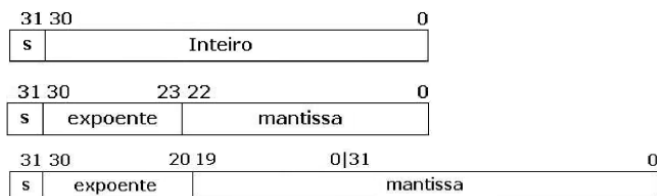


Figura 2.3 – Representação Numérica Binária

Os registradores do tipo float possuem suas próprias instruções. Elas não podem ser usadas com os registradores básicos, assim como as instruções básicas não podem ser usadas com registradores float. Além disso, será visto no capítulo 4 que o mecanismo para desvio condicional das instruções de float funciona de forma um pouco diferente das tradicionais instruções de bifurcação (branch) dos registradores básicos. Em vez de desviar diretamente dependendo do resultado de uma condição, a instrução altera o valor booleano de uma das *Condition Flags* exibidas na figura 2.2, e então outra instrução executa o desvio baseando-se no valor da respectiva *flag*.

Cada um dos 4 registradores em *Coproc 0* reporta um tipo de informação sobre uma exceção ocorrida no programa. Alguns exemplos de exceções que devem ser tratadas são: *arithmetic overflow exception*, *store address exception*, *misaligned load address exception* e *breakpoint exception*.

1. *vaddr (\$8)*: contém o endereço de memória inválido da exceção causada por load, store ou fetch.
2. *status (\$12)*: contém a máscara da interrupção e o estado dos bits.
3. *cause (\$13)*: contém o tipo da exceção e os bits pendentes.
4. *epc (\$14)*: contém o endereço da instrução onde a exceção ocorreu.

As instruções para tratamento de exceções fazem parte de um tópico mais avançado e ficarão para outra ocasião.

## 2.2 Menu de Ajuda

O MARS possui um menu de ajuda excepcional, que na maioria das vezes é desconhecido por quem começa a programação em MIPS. Ele contém praticamente todas as informações necessárias para programar sem problemas.

O menu de ajuda pode ser acessado pelo menu *Help* ou clicando-se no ícone do ponto de interrogação, como pode ser visto na figura 2.4. Outra forma é pressionar a tecla de atalho F1. Vale destacar que o MARS é um ambiente responsivo, no sentido de que praticamente todos os recursos possuem

uma tecla de atalho, e além disso, no caso de dúvida sobre algum item, pode-se posicionar o cursor sobre o mesmo para que apareça uma descrição clara da sua funcionalidade. Até mesmo no caso dos registradores, é exibida uma breve descrição relembrando a sua função especificada na convenção.

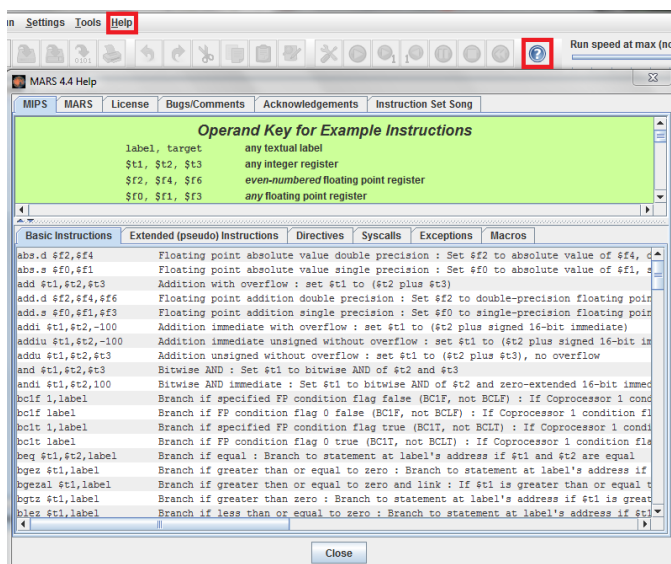


Figura 2.4 – Menu de Ajuda do MARS

Na primeira aba do menu de ajuda, podem ser consultadas todas as instruções básicas da arquitetura MIPS, como foi mencionado na seção 1.6. Elas muito provavelmente estarão disponíveis em qualquer simulador da arquitetura MIPS.

Na segunda aba, há uma lista com instruções “estendidas”, ou “pseudo-instruções”. São macros implementadas na maio-

ria das IDEs com uma sequência de instruções básicas, logo, não aumentam o poder computacional da arquitetura; elas apenas facilitam a programação, podendo economizar várias linhas de instruções com uma única instrução estendida. As pseudo-instruções são substituídas pelas instruções básicas correspondentes no momento da montagem.

Na terceira aba se encontram as diretivas para o *assembler* (montador). Elas executam algumas tarefas em relação à memória, como por exemplo, a alocação das variáveis ou dos segmentos do programa (dados e código). Na tabela 2.1 foram selecionadas as diretivas principais utilizadas no código MIPS para descrição da sua função.

Tabela 2.1 – Diretivas Principais

<i>Diretiva</i>	<i>Descrição</i>
.data	declaração de variáveis: itens armazenados no segmento de dados, no próximo endereço disponível
.globl	igual .data, mas permite referência aos dados por outros arquivos
.text	código do programa: instruções armazenadas no segmento de texto, no próximo endereço disponível
.byte	armazena os valores listados com 8 bits de espaço
.word	armazena os valores listados com 32 bits ou 4 bytes de espaço
.float	armazena os valores listados como pontos flutuantes de precisão simples (também 32 bits ou 4 bytes)
.double	armazena os valores listados como pontos flutuantes de precisão dupla (64 bits ou 8 bytes)
.space	aloca o número de bytes especificado como espaço para uma variável
.asciiz	aloca espaço para uma <i>string</i> terminada em nulo (código ASCII zero)

As 3 primeiras diretivas da tabela 2.1 são responsáveis por identificar em qual segmento da memória os itens seguintes

devem ser armazenados. Na maioria dos códigos MIPS é utilizada pelo menos a diretiva *.data* ou *.globl* para a “declaração” das variáveis, e a diretiva *.text* para indicar o início das instruções do programa.

As outras diretivas determinam que cada item listado em seguida é uma variável do tipo especificado. Isto significa que o montador vai automaticamente alocar espaço para elas e armazenar o endereço base no “rótulo” indicado. No caso da diretiva *space* não há tipo especificado, logo, no caso da instrução:

---

```
var:  .space 40
```

---

O espaço disponível em “var” pode ser utilizado tanto como um vetor de 40 caracteres, quanto como um vetor de 10 inteiros (*word*).

A diretiva *.ascii* é muito usada para declarar todas as *strings* que serão utilizadas no programa. Para uma instrução da forma:

---

```
string: .ascii "Teste\n"
```

---

Será automaticamente alocado espaço para uma cadeia de 7 caracteres - “\n” também conta como um caractere, e o caractere extra “null” é inserido no final da *string*. Portanto, 7 bytes serão alocados. A diretiva *.ascii* realiza o mesmo processo mas não termina a cadeia com nulo.

Na quarta aba do menu de ajuda, está uma tabela com todos os códigos de chamadas de sistema disponíveis no MARS. Uma chamada de sistema executa um serviço de entrada ou

saída de dados (*input* e *output*), equivalente às funções *scanf* e *printf* da linguagem C. Os primeiros códigos de chamadas de sistema, que serão mais utilizados nesta apostila, foram colocados na tabela 2.2. Eles incluem leitura e escrita de valores inteiros, float, double, caractere e string, alocação dinâmica de memória, manipulação de arquivo e finalização do programa.

Assim como para as instruções, alguns códigos são “universais”, ou seja, representam as mesmas funções nos outros simuladores de MIPS. Outros oferecem serviços que podem variar muito; no caso do MARS, que foi implementado em Java, algumas chamadas de sistemas acessam métodos de alto nível como os da classe Dialog.

Pode-se observar que cada chamada de sistema segue um protocolo: o código da função a ser executada é sempre armazenado no registrador \$v0, e os argumentos nos registradores do tipo \$a, ou em \$f12, no caso da impressão de float ou double. A função é então executada com a instrução *syscall*; valores retornados são sempre armazenados em \$v0, ou em \$f0, no caso da leitura de float ou double.

As duas últimas abas no menu de ajuda do MARS, sobre exceções e macros, não serão tratadas aqui pois fazem parte de um conteúdo mais avançado.

Tabela 2.2 – Principais Chamadas de Sistema

<i>Serviço</i>	<i>Código</i> <i>\$v0</i>	<i>Argumentos</i>	<i>Resultado</i>
Impressão de Inteiro	1	\$a0 = inteiro	
Impressão de Float	2	\$f12 = float	
Impressão de Double	3	\$f12 = double	
Impressão de String	4	\$a0 = endereço de string terminada em nulo	
Leitura de Inteiro	5		\$v0 = inteiro lido
Leitura de Float	6		\$f0 = float lido
Leitura de Double	7		\$f0 = double lido
Leitura de String	8	\$a0 = buffer de entrada \$a1 = número máximo de caracteres para leitura	
sbrk (alocação dinâmica)	9	\$a0 = número de bytes para alocar	\$v0 = endereço da memória alocada
Sair	10		
Impressão de Caractere	11	\$a0 = caractere (valor ASCII)	
Leitura de Caractere	12		\$v0 = caractere lido
Abertura de Arquivo	13	\$a0 = nome do arquivo \$a1 = <i>flags</i> (0 = leitura, 1 = escrita) \$a2 = modo	\$v0 = <i>file descriptor</i> (negativo em caso de erro)
Leitura de Arquivo	14	\$a0 = <i>file descriptor</i> \$a1 = buffer de entrada \$a2 = número máximo de caracteres para leitura	\$v0 = número de caracteres lidos (zero para fim de arquivo, negativo em caso de erro)
Escrita em Arquivo	15	\$a0 = <i>file descriptor</i> \$a1 = buffer de saída \$a2 = número de caracteres para escrita	\$v0 = número de caracteres escritos
Fechamento de Arquivo	16	\$a0 = <i>file descriptor</i>	
Sair com retorno	17	\$a0 = valor de retorno	

## 2.3 Olá, Mundo!

A seguir será testado o primeiro código em MIPS com o tradicional programa “Hello World”, que imprime uma mensagem na tela e finaliza o programa. Primeiro, é necessário abrir uma nova aba de edição de texto, e então salvar o arquivo. A extensão de arquivo padrão utilizada pelo MARS é “.asm”.

---

### Programa 2.1 – Olá Mundo

---

```
.data # Declaração das variáveis
olaMundo: .asciiz "Olá, Mundo!" # String para impressão

.text # Instruções do programa
main: # Rótulo para definir o módulo principal do
      programa
      li $v0, 4 # Carrega o valor 4 em $v0 (código de
                impressão de string)
      la $a0, olaMundo # Carrega o endereço da string em
                        $a0 (argumento)
      syscall # Chamada de sistema: imprime a string
      li $v0, 10 # Carrega o valor 10 em $v0 (código
                  para terminar a execução do programa)
      syscall # Termina o programa
```

---

Diferentemente das linguagens de programação de alto nível, o código MIPS não é compilado, e sim “montado” (*assembled*). Para realizar a montagem, basta clicar no ícone com as chaves de boca destacado na figura 2.5, ou pressionar a tecla de atalho F3.

Caso nenhum erro seja acusado, o programa será montado e estará pronto para execução. A aba ativa será automaticamente trocada do modo de edição de código para a aba de



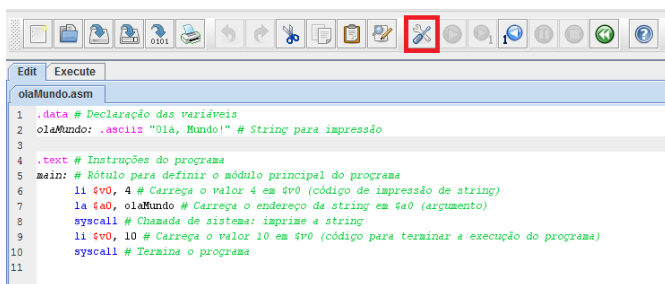


Figura 2.5 – Montagem do programa

execução, que será descrita na próxima seção. Para executar o programa, basta clicar no ícone com uma seta apontada para a direita, ou pressionar a tecla de atalho F5. Todas as ações relacionadas a montagem e execução também podem ser acessadas no menu “Run”. Se tudo tiver ocorrido corretamente, a mensagem “Olá, Mundo!” deverá aparecer na caixa de Run I/O do MARS, seguida de uma mensagem do sistema indicando que a execução do programa foi finalizada.

## 2.4 Depuração

Ao lado do botão de execução normal do código, existe também a opção para executar com “Step” e “Backstep”, que possui a simples mas conveniente funcionalidade de realizar ou desfazer uma instrução de cada vez, respectivamente. Isto pode facilitar para entender exatamente em qual instrução determinado valor é modificado em um registrador, podendo inclusive agilizar a detecção de erros. Também é importante notar o botão “Stop”, que deve ser utilizado caso o programa entre em *loop* infinito.

A aba “Execute”, ativada no momento de montagem do programa, é extremamente útil para visualizar como está ocorrendo a execução das instruções, e como os valores estão sendo alterados nos endereços de memória do segmento de dados e do segmento de texto do programa.

Na figura 2.6 foram destacadas as partes essenciais do painel de execução, que serão descritas a seguir. A caixa “Labels” é oculta por padrão; para visualizá-la é necessário marcar a opção “Show Labels Windows” no menu “Settings”. Além disso, por padrão todos os valores e endereços são representados em hexadecimal. Para alterar para o formato decimal, deve-se desmarcar as opções “displayed in hexadecimal” no mesmo menu.

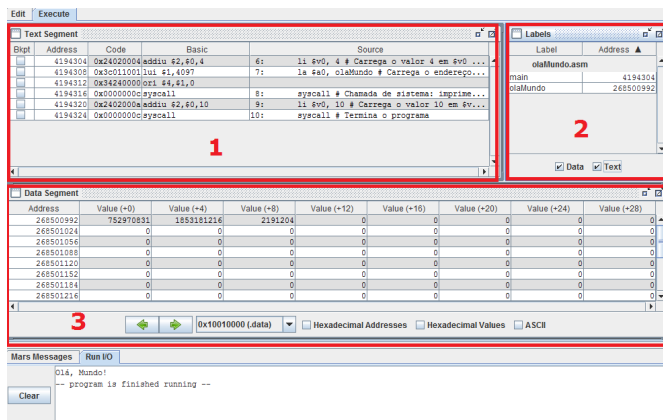


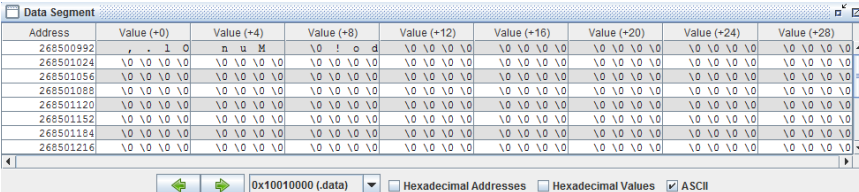
Figura 2.6 – Painel de execução

1. *Text Segment*: ao executar o programa com “Step”, a próxima linha de instrução a ser executada é destacada em amarelo. A tabela deste bloco possui uma coluna exibindo o endereço de cada instrução do segmento de texto na memória, uma coluna com o valor que a representa contido no mesmo endereço, uma mostrando a instrução original no código e outra coluna contendo as instruções básicas traduzidas - aqui os registradores são referenciados pelo seu número original, os rótulos são trocados pelo seu endereço de memória e as “pseudo-instruções” do MARS são substituídas pela sequência de instruções que compõem sua macro.
2. *Labels*: exhibe cada rótulo definido no programa e o seu endereço na memória. Isto inclui tanto rótulos de variáveis declaradas no segmento de dados quanto rótulos de procedimentos definidos no segmento de texto.
3. *Data Segment*: mostra os valores contidos nos endereços do segmento de dados na memória com um intervalo de 32 bytes para cada linha, exibindo o valor armazenado considerando desde um *offset* de zero até 28 bytes, com cada coluna variando o *offset* em 4 (o espaço de uma *word*). Por padrão, são mostrados os dados referentes às variáveis em *.data*, mas outras seções também podem ser selecionadas, como os dados em *.kdata* (memória do *kernel*), *.text*, *.extern*, na memória *heap* ou no segmento de pilha acessado pelo registrador *\$sp*, entre outros.

Em (1), existe uma coluna especial denominada “Bkpt” (*breakpoint*), que pode ser usada para selecionar uma instrução, fazendo o programa pausar antes de executá-la. Então, se for necessário parar o programa na centésima instrução

para se observar o *status* do painel de execução, em vez de se pressionar a tecla F7 (tecla de atalho para “Step”) noventa e nove vezes, pode-se apenas selecionar a instrução e executar o programa normalmente com F5.

Como exemplo da interconexão dos componentes do painel de execução, pode-se observar que o endereço do rótulo “olaMundo” em (2) é o mesmo valor que termina armazenado no registrador \$a0 após a execução do programa 2.1. O mesmo endereço em (3) contém o valor correspondente aos bytes que compõem a *string* “Olá Mundo!”, como pode ser visto na figura 2.7, onde a opção *ASCII* foi ativada na janela do segmento de dados. Assim, como cada coluna representa uma word ou 4 bytes de um endereço, é possível visualizar 4 caracteres da *string* por célula.



Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	r	1	0	n	u	d		
268501024	\0	\0	\0	\0	\0	\0	\0	\0
268501056	\0	\0	\0	\0	\0	\0	\0	\0
268501088	\0	\0	\0	\0	\0	\0	\0	\0
268501120	\0	\0	\0	\0	\0	\0	\0	\0
268501152	\0	\0	\0	\0	\0	\0	\0	\0
268501184	\0	\0	\0	\0	\0	\0	\0	\0
268501216	\0	\0	\0	\0	\0	\0	\0	\0

0x10010000 (.data)    ☐ Hexadecimal Addresses    ☐ Hexadecimal Values    ☒ ASCII

Figura 2.7 – String no segmento de dados

# Aritmética e Operadores Lógicos

Neste capítulo, será visto como declarar e carregar variáveis numéricas ou valores em registradores, utilizando-os em instruções aritméticas, instruções lógicas *bitwise* (bit a bit) e *shift*.

Serão abordadas instruções tanto para números inteiros quanto para reais (ponto flutuante), e as instruções para conversão entre os tipos. Também será mostrado nos programas construídos como realizar a leitura e impressão de valores numéricos.

## 3.1 Endereçamento e Atribuição de Valores

Existem várias formas de se carregar um valor específico em um registrador para que ele possa ser acessado e utilizado pelo programa. Considerando o código C a seguir:

---

```
int a = 1;
```

---

Ele poderia ser traduzido para MIPS da seguinte forma:

---

```
.data  
a: .word 1
```

---

Como descrito na tabela 2.1, a declaração das variáveis é precedida pela diretiva *.data*, e *.word* indica que deve-se alocar o espaço de uma word (4 bytes) para o rótulo “a”, carregando o valor “1” no endereço da memória alocada. Como as diretivas já foram apresentadas, não iremos nos ater aos seus significados.

Para que o valor da variável seja carregado em um registrador, pode-se utilizar a instrução *lw* (load word) com o rótulo indicado diretamente na instrução, mas também é possível carregar primeiro o endereço de “a” em um registrador com a pseudo-instrução *la* (load address), e em seguida acessar o valor contido no endereço armazenado no registrador com *lw*.

---

```
lw $t0, a  
# ou  
la $t0, a  
lw $a0, ($t0)
```

---

Caso o objetivo seja apenas carregar o valor constante 1 no registrador, a maneira mais fácil é com a pseudo-instrução *li* (load immediate). Na montagem ela é substituída pela instrução básica *addiu*, a versão *unsigned* de *addi*, que soma o valor que se deseja atribuir com o registrador \$zero, que contém o valor constante zero.

---

```
li $t0, 1
# ou
addiu $t0, $zero, 1
```

---

Em MIPS, a única diferença entre instruções *signed* e *unsigned* é que as instruções *signed* podem gerar uma exceção de *overflow*.

Segue um exemplo do mesmo código para ponto flutuante. Vale notar que *l.s* e *l.d* também são pseudo-instruções do MARS, traduzidas na montagem para as instruções *lui* e *lwc1* ou *ldc1*. O MARS não possui instruções para carregamento de valor imediato em registradores de ponto flutuante.

```
.data
f: .float 1.0
d: .double 1.0

.text
main: l.s $f2, f
      l.d $f0, d
```

---

A atribuição do valor de um registrador em outro também possui uma pseudo-instrução, denominada *move*; de forma similar à atribuição de constantes, ela é substituída pela instrução *addu*, somando zero com o valor no registrador.

```
move $t0, $t1
# ou
addu $t0, $t1, $zero
```

---

Para ponto flutuante, a atribuição possui instruções semelhantes à *move*. Porém, ao contrário de *move*, elas fazem parte das instruções básicas.

---

```
mov.s $f0, $f1
mov.d $f2, $f4
```

---

## 3.2 Operações Aritméticas

A seguir, serão mostrados instruções e programas de exemplo para operações aritméticas, incluindo o procedimento para leitura e impressão de valores numéricos no ambiente do MARS.

A tabela 3.1 mostra as instruções básicas para cada um dos tipos numéricos. No caso da operação módulo para inteiros, a instrução *mfhi* deve ser usada após a operação de divisão correspondente, pois o valor do resto da divisão é armazenado no registrador especial *hi*. Já as instruções para valor absoluto e negação são exclusivas do tipo de ponto flutuante.

Tabela 3.1 – Operações Aritméticas

Operação	Inteiro	Float	Double Float
$r0 = r1 + r2$	add \$t0, \$t1, \$t2	add.s \$f0, \$f1, \$f2	add.d \$f0, \$f2, \$f4
$r0 = r1 - r2$	sub \$t0, \$t1, \$t2	sub.s \$f0, \$f1, \$f2	sub.d \$f0, \$f2, \$f4
$r0 = r1 * r2$	mul \$t0, \$t1, \$t2	mul.s \$f0, \$f1, \$f2	mul.d \$f0, \$f2, \$f4
$r0 = r1 / r2$	div \$t0, \$t1, \$t2	div.s \$f0, \$f1, \$f2	div.d \$f0, \$f2, \$f4
$r0 = r1 \% r2$	mfhi \$t0	-	-
$r0 =  r1 $	-	abs.s \$f0, \$f1	abs.d \$f0, \$f2
$r0 = -r1$	-	neg.s \$f0, \$f1	neg.d \$f0, \$f2

Foi visto na seção 3.1 que as instruções de soma podem ser utilizadas como operação de atribuição caso um dos operandos seja zero, tanto para atribuir valores em registradores quanto constantes, no caso de números inteiros. Para ponto



flutuante, isto não funciona dado que o registrador \$zero não é um operando aceitável para as instruções de *float*.

Pela tabela 3.1, pode-se observar que é ainda mais simples executar as operações aritméticas, pois as instruções seguem um padrão coerente e com a mesma ordem de operandos.

No caso dos inteiros, como exemplificado na seção 3.1 com as instruções *addi* e *addu*, o acréscimo da letra *i*, *u* ou *iu* nas instruções *sub*, *mul* ou *div* indica que o segundo operando é uma constante, que a instrução não gera exceção de *overflow*, ou ambos, respectivamente.

Para aplicar o que foi visto até aqui, segue o programa 3.1 que faz a leitura de 2 valores inteiros A e B e imprime o resultado da soma, subtração, multiplicação, divisão e resto de A e B. As diretivas utilizadas podem ser conferidas na tabela 2.1, e os códigos das chamadas de sistema na tabela 2.2.

---

#### Programa 3.1 – Operações com Números Inteiros

---

```
.data
MsgA: .asciiz "    Insira o valor de A: "
MsgB: .asciiz "    Insira o valor de B: "
Res1: .asciiz "\n  Resultado de A + B: "
Res2: .asciiz "\n  Resultado de A - B: "
Res3: .asciiz "\n  Resultado de A * B: "
Res4: .asciiz "\n  Resultado de A / B: "
Res5: .asciiz "\n  Resultado de A % B: "

.text
main: la $a0, MsgA # Carrega o endereço da string MsgA
      li $v0, 4 # Código de impressão de string
      syscall # Imprime a string
```

```
li $v0, 5 # Código de leitura de inteiro
syscall # Faz a leitura de A
move $t0, $v0 # $t0 armazena o valor de A
la $a0, MsgB # Carrega o endereço da string MsgB
li $v0, 4 # Código de impressão de string
syscall # Imprime a string
li $v0, 5 # Código de leitura de inteiro
syscall # Faz a leitura de B
move $t1, $v0 # $t1 armazena o valor de B
la $a0, Res1 # Carrega o endereço da string Res1
li $v0, 4 # Código de impressão de string
syscall # Imprime a string
add $a0, $t0, $t1 # $a0 = A + B
li $v0, 1 # Código de impressão de inteiro
syscall # Imprime o resultado
la $a0, Res2 # Carrega o endereço da string Res2
li $v0, 4 # Código de impressão de string
syscall # Imprime a string
sub $a0, $t0, $t1 # $a0 = A - B
li $v0, 1 # Código de impressão de inteiro
syscall # Imprime o resultado
la $a0, Res3 # Carrega o endereço da string Res3
li $v0, 4 # Código de impressão de string
syscall # Imprime a string
mul $a0, $t0, $t1 # $a0 = A * B
li $v0, 1 # Código de impressão de inteiro
syscall # Imprime o resultado
la $a0, Res4 # Carrega o endereço da string Res4
li $v0, 4 # Código de impressão de string
syscall # Imprime a string
div $a0, $t0, $t1 # $a0 = A / B
li $v0, 1 # Código de impressão de inteiro
syscall # Imprime o resultado
la $a0, Res5 # Carrega o endereço da string Msg5
li $v0, 4 # Código de impressão de string
```

```
syscall # Imprime a string
mfhi $a0 # $a0 = resto da divisão executada
li $v0, 1 # Código de impressão de inteiro
syscall # Imprime o resultado
li $v0, 10 # Código para finalizar o programa
syscall # Finaliza o programa
```

---

Segue um exemplo de execução do programa 3.1:

---

```
Insira o valor de A: 7
Insira o valor de B: 4

Resultado de A + B: 11
Resultado de A - B: 3
Resultado de A * B: 28
Resultado de A / B: 1
Resultado de A % B: 3
-- program is finished running --
```

---

Para exemplificar as instruções de *float*, será apresentado o mesmo programa mas com valores reais de A e B, utilizando ponto flutuante de precisão dupla. Pode-se observar que a lógica do programa 3.2 é a mesma, mas os registradores, instruções, e códigos de chamadas de sistema são diferentes, e também não há mais resto de divisão.

---

### Programa 3.2 – Operações com Números Reais

---

```
.data
MsgA: .asciiz "  Insira o valor de A: "
MsgB: .asciiz "  Insira o valor de B: "
```

```
Res1: .asciiz "\n  Resultado de A + B: "  
Res2: .asciiz "\n  Resultado de A - B: "  
Res3: .asciiz "\n  Resultado de A * B: "  
Res4: .asciiz "\n  Resultado de A / B: "  
  
.text  
main: la $a0, MsgA # Carrega o endereço da string MsgA  
      li $v0, 4 # Código de impressão de string  
      syscall # Imprime a string  
      li $v0, 7 # Código de leitura de double  
      syscall # Faz a leitura de A  
      mov.d $f2, $f0 # $f2 armazena o valor de A  
      la $a0, MsgB # Carrega o endereço da string MsgB  
      li $v0, 4 # Código de impressão de string  
      syscall # Imprime a string  
      li $v0, 7 # Código de leitura de double  
      syscall # Faz a leitura de B  
      mov.d $f4, $f0 # $f4 armazena o valor de B  
      la $a0, Res1 # Carrega o endereço da string Res1  
      li $v0, 4 # Código de impressão de string  
      syscall # Imprime a string  
      add.d $f12, $f2, $f4 # $f12 = A + B  
      li $v0, 3 # Código de impressão de double  
      syscall # Imprime o resultado  
      la $a0, Res2 # Carrega o endereço da string Res2  
      li $v0, 4 # Código de impressão de string  
      syscall # Imprime a string  
      sub.d $f12, $f2, $f4 # $f12 = A - B  
      li $v0, 3 # Código de impressão de double  
      syscall # Imprime o resultado  
      la $a0, Res3 # Carrega o endereço da string Res3  
      li $v0, 4 # Código de impressão de string  
      syscall # Imprime a string  
      mul.d $f12, $f2, $f4 # $f12 = A * B  
      li $v0, 3 # Código de impressão de double
```

```
syscall # Imprime o resultado
la $a0, Res4 # Carrega o endereço da string Res4
li $v0, 4 # Código de impressão de string
syscall # Imprime a string
div.d $f12, $f2, $f4 # $f12 = A / B
li $v0, 3 # Código de impressão de double
syscall # Imprime o resultado
li $v0, 10 # Código para finalizar o programa
syscall # Finaliza o programa
```

---

E um exemplo de execução para o programa 3.2:

---

Insira o valor de A: 1.2345

Insira o valor de B: 6.7890

Resultado de A + B: 8.0235

Resultado de A - B: -5.5545

Resultado de A \* B: 8.3810205

Resultado de A / B: 0.1818382677861246

-- program is finished running --

---

### 3.3 Operações Lógicas

As operações lógicas possuem forte relação com as operações aritméticas. Ambas são executadas pela *ALU* (Unidade Lógica e Aritmética) da arquitetura, e de fato, a nível de bit as operações aritméticas são implementadas como operações lógicas binárias. Em MIPS, muitas das pseudo-instruções são formadas por instruções básicas de operação lógica.

A tabela 3.2 apresenta algumas operações lógicas com o operador correspondente em C, Java e a instrução MIPS. Todas as operações booleanas podem ser implementadas com as 3 operações AND, OR e NOT, mas a arquitetura também possui as instruções *xor* e *xori* que executam a operação XOR (OR exclusivo), que pode ser útil para expressar certas expressões lógicas com mais facilidade.

Tabela 3.2 – Operações Lógicas

Operação	C	Java	MIPS
Shift à esquerda	<<	<<	sll
Shift à direita	>>	>>>	srl
AND bit a bit	&	&	and, andi
OR bit a bit			or, ori
NOT bit a bit	~	~	nor, nori

Pode-se constatar que as instruções também são bastante similares às utilizadas nas operações aritméticas, possuindo a instrução básica que opera sobre dois registradores e armazena o resultado em um registrador de destino, e outra variante que possui um valor imediato como um dos operandos (formatos R e I de instrução respectivamente, descritos na seção 1.6).

Será feita uma breve descrição para cada uma das instruções de *shift* e *bitwise* mencionando algumas funcionalidades práticas, pois normalmente o funcionamento das operações lógicas é conhecido, mas na maioria das vezes não se sabe sobre a sua utilidade na resolução de problemas em MIPS e até mesmo em outras linguagens.

- *Shift left logical (sll)*: deslocamento lógico à esquerda,

desloca o número de bits indicado pelo valor imediato para a esquerda, preenchendo as posições deslocadas com zero. Possui correspondência direta com a multiplicação por 2 - a cada posição que os bits são deslocados para a esquerda, o valor resultante é multiplicado por 2, logo, deslocando N bits, o resultado é igual a  $x * 2^N$ . É bastante utilizado quando se quer multiplicar um número por uma potência de 2, como por exemplo, para computar endereços de memória para inteiros, o deslocamento de 2 bits para a esquerda equivale a multiplicar por 4.

- *Shift right logical (srl)*: deslocamento lógico à direita, análogo ao *sll*, mas o deslocamento de N bits corresponde à divisão por 2, N vezes.
- *Bitwise AND (and)*: conjunção lógica, o resultado será 1 se e somente se os bits na mesma posição forem 1. É útil para desativar bits específicos, e comparar bits, como por exemplo para verificar se um número é par ou ímpar; em vez do método tradicional de divisão e resto, pode-se utilizar uma instrução da forma *andi \$t1, \$t0, 1*, que irá comparar o bit de menor significância do valor em \$t0 com 1, resultando em 0 no registrador \$t1 caso o número seja par, ou 1 caso seja ímpar.
- *Bitwise OR (or)*: disjunção lógica, o resultado será 1 se pelo menos um dos bits na mesma posição for 1. Ele pode ser utilizado para atribuir o valor de um registrador em outro com uma instrução da forma *or \$t0, \$t1, \$zero*, similar à *add \$t0, \$t1, \$zero*, a diferença é que não possui *carry bit*, ou seja, não propaga bit para a próxima posição, e por isso a atribuição pode ser executada com complexidade de  $O(1)$ , enquanto a adição possui

complexidade de  $O(\log n)$  a  $O(n)$ . Entretanto, no MARS a operação OR não é mais rápida que a adição, mas em outras arquiteturas ela de fato é mais veloz. Outro exemplo de utilidade é converter caracteres maiúsculos para minúsculos (se já estiver minúsculo, não realiza alteração).

- *Bitwise NOR (nor)*: a negação ou o contrário da instrução *or*, o resultado será 1 apenas se nenhum dos bits na mesma posição for 1, ou seja, se ambos forem zero. Assim, a operação de negação lógica pode ser aplicada se a instrução for utilizada com zero como um dos operandos, como por exemplo em *nor \$t0, \$t1, \$zero*, todos os bits do valor carregado em *\$t1* serão invertidos. Vale notar que a pseudo-instrução *not* do MARS é implementada dessa forma.
- *Bitwise XOR (xor)*: disjunção lógica exclusiva, o resultado será 1 se apenas um dos bits na mesma posição for 1, ou seja, se os bits forem diferentes. Pode ser utilizada para atribuição assim como a instrução *or*, para reverter operações (sendo por isso bastante utilizada em criptografia), e trocar dois valores sem usar uma variável temporária.

Para demonstrar o funcionamento de uma das instruções menos compreendidas, segue um programa simples que faz a leitura de um valor *N* e imprime o resultado de  $2^N$ , utilizando apenas a instrução *sllv* de deslocamento de bits à esquerda.

---

#### Programa 3.3 – Potência de dois com shift

---

```
.data
MsgN: .asciiz "    Insira o valor de N: "
```



```
.text
main: la $a0, MsgN # Carrega o endereço da string MsgN
      li $v0, 4 # Código de impressão de string
      syscall # Imprime a string
      li $v0, 5 # Código de leitura de inteiro
      syscall # Faz a leitura de N
      li $a0, 1 # Inicializa o registrador com o valor 1
      sllv $a0, $a0, $v0 # Desloca o bit N vezes para a
      esquerda = 2^N
      li $v0, 1 # Código de impressão de inteiro
      syscall # Imprime o resultado de 2^N
      li $v0, 10 # Código para finalizar o programa
      syscall # Finaliza o programa
```

---

O registrador \$a0 que armazena o resultado é inicializado em 1, o que equivale ao bit de menor significância setado em 1, e todos os outros bits em zero. Se  $N = 0$ , o bit não é deslocado e o valor permanece o mesmo ( $2^0 = 1$ ); se  $N = 1$ , o bit é deslocado uma posição para a esquerda e o resultado é 2, pois o número binário correspondente é 10. Se  $N = 2$ , o bit é deslocado duas posições e o resultado é 4 (100), e assim por diante.

Pode-se constatar que o programa só funciona até  $N = 31$ , pois o registrador armazena uma *word* de 4 bytes ou 32 bits, então se  $N = 32$ , o bit volta à posição inicial. Acima de 32, o bit começa a “dar voltas”; logo, o valor de  $N$  para o resultado pode ser definido como  $N = N\%32$ . No caso de valores negativos para  $N$ , a instrução será executada como se fosse um deslocamento à direita. Segue um exemplo de execução para o programa 3.3.

---

```
Insira o valor de N: 12
4096
-- program is finished running --
```

---

### 3.4 Conversão de Tipos

Muitas vezes é necessário realizar conversões entre tipos numéricos, como por exemplo, quando se tem um conjunto de números inteiros e deseja-se computar a média em ponto flutuante. Outro exemplo é o cálculo de notas com média ponderada, onde cada nota armazenada em ponto flutuante possui um peso diferente armazenado como inteiro. Em linguagens de alto nível, uma operação simples de multiplicação entre inteiro e ponto flutuante é executada sem problemas, com conversão implícita do valor inteiro; em MIPS, o processo não é tão trivial.

Foi visto na seção 2.1 que os registradores básicos e de ponto flutuante possuem cada um suas próprias instruções, que não podem ser usadas com os do outro tipo. Por isso, são providas instruções básicas que possibilitam que um valor carregado em um registrador básico seja movido para um de ponto flutuante, e vice-versa. O processo de conversão pode ser executado em 2 instruções, não necessariamente nessa ordem:

1. *Mover para o registrador do outro tipo*: instruções *mfc1* (Move from Coprocessor 1) e *mtc1* (Move to Coprocessor 1), que indicam se vão mover um registrador da *FPU*, ou para a *FPU*, respectivamente. Também é pos-

sível carregar a *word* diretamente da memória ou para a memória, com as instruções *lwc1* e *swc1*, respectivamente.

2. *Conversão*: instruções *cvt*, que possuem 2 letras indicando o tipo de destino e o tipo de origem.

A tabela 3.3 apresenta todas as instruções de conversão para as combinações possíveis dos tipos numéricos, considerando que o valor se encontra no registrador \$f0 e que o resultado da conversão deve ser carregado em \$f2 (também poderia ser armazenado no próprio registrador). O formato da instrução é sempre *cvt.[tipo\_destino].[tipo\_origem]*.

Tabela 3.3 – Instruções de Conversão

<i>Dest./Or.</i>	<i>Inteiro</i>	<i>Float</i>	<i>Double</i>
<i>Inteiro</i>	-	cvt.w.s \$f2, \$f0	cvt.w.d \$f2, \$f0
<i>Float</i>	cvt.s.w \$f2, \$f0	-	cvt.s.d \$f2, \$f0
<i>Double</i>	cvt.d.w \$f2, \$f0	cvt.d.s \$f2, \$f0	-

As instruções para transferência entre registradores de tipos diferentes possuem o mesmo formato; considerando \$t0 e \$f0 como registrador de origem ou destino, a instrução é da forma *mfc1 \$t0, \$f0* ou *mtc1 \$t0, \$f0*. Estas instruções não são necessárias no caso da conversão entre pontos flutuantes de precisão simples e dupla.

É importante notar que *mtc1* é utilizada *antes* de uma conversão de inteiro para ponto flutuante, enquanto *mfc1* deve executar *depois* de uma conversão de ponto flutuante para inteiro, movendo o valor convertido para um registrador básico. É mais rara a necessidade de conversão para inteiro,

pois pode haver perda de informação, já que apenas a parte inteira é mantida; mesmo que o valor original seja 7.999, o resultado após a conversão será 7.

# Estrutura de Controle e Sub-rotinas

Tudo que será visto neste capítulo, no nível mais elementar, se baseia em uma simples operação: a alteração do endereço armazenado no registrador especial *pc*, que é o Contador de Programa, descrito na seção [1.3](#).

Os recursos essenciais das linguagens de alto nível para construir estruturas de desvio do fluxo de controle e sub-rotinas, como *if-then-else*, *switch-case*, *go to*, *for*, *while*, funções e procedimentos, se resumem a instruções de baixo nível que *desviam* o endereço da próxima instrução a ser executada, seja de forma condicional ou incondicional.

## 4.1 Rótulos

Uma *label* ou rótulo é um jeito mais simples de referenciar endereços de memória. Como foi visto anteriormente, um

rótulo pode identificar o endereço de memória de uma variável definida no segmento de dados (*.data*). Mas ela também pode ser usada para identificar uma linha de instrução do programa, ou seja, um endereço do segmento de texto do programa (*.text*).

Nos exemplos de programa dos capítulos anteriores, foi utilizado o rótulo “main” para identificar o endereço da primeira instrução do programa. Deve ser ressaltado que isto foi apenas uma “boa prática”; o uso do rótulo não era necessário nos exemplos, pois o Contador de Programa já é inicializado com o endereço da primeira instrução. A escolha do nome do rótulo também é arbitrária. Em linguagens de alto nível, *main* normalmente é uma palavra reservada da gramática, mas em MIPS não há essa distinção.

## 4.2 Desvio Incondicional

Sem nenhuma dúvida, as instruções mais simples da arquitetura MIPS são as de desvio incondicional, que como foi visto na seção 1.6, possuem o formato J de instrução que exige um único “argumento”. São apenas 4 instruções básicas, descritas a seguir:

- *j target (jump)*: altera incondicionalmente o valor no contador de programa para o endereço do rótulo *target*.
- *jal target (jump and link)*: igual *j*, mas também salva no registrador \$ra o endereço da próxima instrução (que seria executada em seguida), sendo especialmente útil para a chamada e retorno de procedimentos.

- *jalr \$t0 (jump and link register)*: igual *jal*, mas utiliza o endereço armazenado no registrador especificado em vez de rótulo.
- *jr \$t0 (jump register)*: igual *j*, mas utiliza o endereço armazenado no registrador especificado em vez de rótulo. Normalmente é usado com o registrador *\$ra* após *jal*, para retornar à próxima instrução após a chamada de um procedimento.

Pode-se constatar que as instruções acima são o mais puro comando *goto*, bastante demonizado na área de programação, dado que atualmente existem formas melhores (e mais elegantes) de se programar. No entanto, em MIPS e outras linguagens *assembly*, essas instruções são absolutamente essenciais.

A utilização das instruções de *jump* será demonstrada em exemplos das próximas seções.

### 4.3 Desvio Condicional

O objetivo desta seção é mostrar como construir estruturas de seleção ou expressão condicional em MIPS, o que inclui comandos como *if-then-else* e *switch-case*, cuja lógica em baixo nível também é similar ao comando *goto*. A diferença é a existência de uma expressão lógica que define se o desvio será executado.

Em MIPS, as instruções básicas proveem apenas a comparação entre valores numéricos, como inteiros ou reais (ponto flutuante). Caracteres também são representados por um número da tabela *ASCII*, portanto a comparação utiliza este

valor. A seguir, o processo de desvio condicional para inteiros e ponto flutuante será visto separadamente, pois eles diferem em alguns aspectos importantes.

### 4.3.1 Comparação de Inteiros

As instruções de comparação para valores inteiros ou *word* começam com a letra *b* de *branch*, ou *bifurcação*. Diferentemente das instruções para ponto flutuante, a mesma instrução que especifica a condição a ser satisfeita já indica o endereço para desvio. A tabela 4.1 apresenta o operador de comparação da linguagem C correspondente para cada instrução de MIPS.

Tabela 4.1 – Instruções de Bifurcação

<i>Instrução</i>	<i>Comando</i>
beq \$t0, \$t1, label	if(t0 == t1) goto label
bne \$t0, \$t1, label	if(t0 != t1) goto label
bgez \$t0, label	if(t0 >= 0) goto label
bgtz \$t0, label	if(t0 > 0) goto label
blez \$t0, label	if(t0 <= 0) goto label
bltz \$t0, label	if(t0 < 0) goto label
*beqz \$t0, label	if(t0 == 0) goto label
*bnez \$t0, label	if(t0 != 0) goto label
*bge \$t0, \$t1, label	if(t0 >= t1) goto label
*bgt \$t0, \$t1, label	if(t0 > t1) goto label
*ble \$t0, \$t1, label	if(t0 <= t1) goto label
*blt \$t0, \$t1, label	if(t0 < t1) goto label

Vale notar que as instruções marcadas com \* são pseudo-instruções, pois o conjunto reduzido de instruções básicas não oferece uma “tradução” para todo operador lógico de comparação possível. Obviamente, é possível implementar



qualquer expressão de desvio condicional com as instruções básicas, visto que elas são usadas para implementar as pseudo-instruções. No caso das destacadas na tabela, a implementação é feita em duas instruções: primeiro *slt* ou similares para setar 0 ou 1 no registrador \$at (*assembler temporary*) dependendo se o valor é maior ou menor, e em seguida compará-lo a zero com uma das instruções básicas.

Assim como para as instruções de operação aritmética, o MARS oferece uma infinidade de variações para as instruções da tabela 4.1. Por exemplo, em vez de utilizar dois registradores (no caso das instruções sem “z”), é possível substituir o segundo registrador por uma constante, sem alterar o nome da instrução. Há também as versões de comparação *unsigned*, com a adição da letra *u* no final da instrução; neste caso, ela significa de fato que a comparação será realizada *ignorando* o sinal negativo, caso possua.

Existem versões especiais das instruções *bgez* e *bltz*, denominadas *bgezal* e *bltzal*, respectivamente. Assim como *jal* (*jump and link*), elas armazenam em \$ra o endereço da próxima instrução, como descrito na seção 4.2.

Para entender melhor o funcionamento desse tipo de instrução, analisemos o seguinte código em C:

---

```
if (i == j) f = g + h;  
else f = g - h;
```

---

Há basicamente duas formas de traduzi-lo para MIPS. Elas consistem em escolher um dos blocos de código - os comandos executados caso a condição seja verdadeira ( $f = g + h$ ), ou os outros comandos executados caso ela seja falsa ( $f = g - h$ ) -

para executar imediatamente após a instrução de bifurcação, caso a condição falhe.

O outro bloco deve ter a primeira instrução indicada por um rótulo, que é colocado na instrução de bifurcação para desvio, caso a condição seja verdadeira. Se o bloco “alternativo” (aquele identificado pelo rótulo) estiver logo após o bloco “padrão”, deve-se garantir que ele não execute caso a condição falhe e o bloco padrão seja executado primeiro. Por isso, é necessário “pular” o bloco alternativo utilizando um segundo rótulo de “saída” ou “continuação”, como no exemplo a seguir:

---

```
bne $s3, $s4, Else # vá para Else se i != j
add $s0, $s1, $s2 # f = g + h
j Exit # vá para Exit
Else: sub $s0, $s1, $s2 # f = g - h
Exit:
```

---

Na prática, o bloco de *else* é construído como um procedimento que deve executar caso ( $i == j$ ) não seja verdade; por isso, a condição é invertida para ( $i != j$ ) com a instrução *bne*. Neste caso, o bloco de *then* é completamente ignorado, pois se encontra antes do rótulo “Else”.

Se a condição invertida falhar, o programa continua normalmente e executa *then*, mas quando terminar, não pode executar *else*, e por isso desvia para a instrução após o bloco de *else* com *jump*.

Para exemplificar de outra forma, segue um programa com vários blocos de *else*. Ele faz a leitura da idade inserida

pelo usuário e imprime uma classificação de acordo com o intervalo em que ela se encontra: “inválido” se a idade for negativa, “criança” até 12 anos, “adolescente” se estiver entre 13 e 17 anos, “adulto” entre 18 e 59 anos, e “idoso” para 60 anos ou mais.

---

#### Programa 4.1 – Classificação de Idade

---

```
.data
Insira: .asciiz "   Insira a sua idade: "
Res0: .asciiz "   Idade inválida"
Res1: .asciiz "   Criança"
Res2: .asciiz "   Adolescente"
Res3: .asciiz "   Adulto"
Res4: .asciiz "   Idoso"

.text
main: la $a0, Insira # Carrega o endereço da string
      li $v0, 4 # Código de impressão de string
      syscall # Imprime a string
      li $v0, 5 # Código de leitura de inteiro
      syscall # Faz a leitura da idade
      move $t0, $v0 # Salva a idade em $t0
      li $v0, 4 # Código de impressão de string
      bgez $t0, c1 # se idade >= 0, continua a análise
      la $a0, Res0 # senão, carrega a string Res0
      j res # e pula para dar o resultado
c1:   bge $t0, 13, c2 # se idade >= 13, continua a
      análise
      la $a0, Res1 # senão, carrega a string Res1
      j res # e pula para dar o resultado
c2:   bge $t0, 18, c3 # se idade >= 18, continua a
      análise
      la $a0, Res2 # senão, carrega a string Res2
      j res # e pula para dar o resultado
```

```
c3:  bge $t0, 60, c4 # se idade >= 60, continua a
    análise
    la $a0, Res3 # senão, carrega a string Res3
    j res # e pula para dar o resultado
c4:  la $a0, Res4 # carrega a string Res4
res: syscall # Imprime a classificação da idade
    li $v0, 10 # Código para finalizar o programa
    syscall # Finaliza o programa
```

---

Segue um exemplo de execução para o programa 4.1.

---

```
Insira a sua idade: 30
Adulto
-- program is finished running --
```

---

### 4.3.2 Comparação de Ponto Flutuante

Como as instruções de operação aritmética, a diferença da comparação de inteiros para ponto flutuante é majoritariamente no nome e quantidade das instruções; a lógica para construção das estruturas de controle continua a mesma. O processo é realizado em duas etapas ou instruções:

1. *Comparação*: instrução para comparar dois registradores de ponto flutuante e ativar uma *condition flag* (“bandeira de condição”) do *Coprocessor 1* caso a condição seja verdadeira, ou desativá-la caso a condição seja falsa.
2. *Bifurcação*: instrução para desviar para o endereço de um rótulo especificado caso a *condition flag* indicada esteja ativada ou desativada.

O desvio condicional é realizado de forma indireta: as instruções para comparação de ponto flutuante não podem indicar um endereço do programa para desviar, elas podem apenas alterar o valor das *flags*. E a bifurcação em si só é feita por meio *flags*.

O formato das instruções básicas de comparação disponíveis é *c.[operador de comparação].[precisão do ponto flutuante]*, como pode ser conferido na tabela 4.2. É possível especificar o número da *flag* com um argumento extra na instrução, como por exemplo *c.eq.d 3, \$f0, \$f2*, que ativa a *flag* 3 caso o valor armazenado em \$f0 seja igual ao valor em \$f2. Se nenhuma for especificada, a *flag* 0 será a modificada.

Tabela 4.2 – Instruções de Comparação (P. F)

<i>Precisão Simples</i>	<i>Precisão Dupla</i>	<i>Comando</i>
<i>c.eq.s \$f0, \$f2</i>	<i>c.eq.d \$f0, \$f2</i>	if(f0 == f2) flag0 = true; else flag0 = false;
<i>c.le.s \$f0, \$f2</i>	<i>c.le.d \$f0, \$f2</i>	if(f0 <= f2) flag0 = true; else flag0 = false;
<i>c.lt.s \$f0, \$f2</i>	<i>c.lt.d \$f0, \$f2</i>	if(f0 < f2) flag0 = true; else flag0 = false;

Pode-se observar que não existem instruções básicas ou mesmo pseudo-instruções correspondentes aos operadores *!=*, *>=*, *>* e outros. Porém, é possível simulá-los apenas invertendo a ordem dos registradores na instrução.

Após alterar o valor de uma *flag* com alguma das instruções de comparação, pode-se executar a instrução de bifurcação. O seu formato é um pouco diferente, mas novamente começa com a letra *b* indicando *branch*. São basicamente 2 instruções, mais a variante que especifica a *flag* que será

avaliada, apresentadas na tabela 4.3.

Tabela 4.3 – Instruções de Bifurcação (P. F.)

Instrução	Comando
bc1f label	if(flag0 == false) goto label
bc1f N, label	if(flagN == false) goto label
bc1t label	if(flag0 == true) goto label
bc1t N, label	if(flagN == true) goto label

Um erro comum é confundir o número “1” na instrução com a letra “l”, por isso deve ser ressaltado que o correto é *bc1*, onde *c1* se refere à *Coprocessor 1*. *N* pode ser um número de 0 a 7, visto que existem 8 *flags* disponíveis no MARS.

Para exemplificar o que foi descrito nesta seção, considere a seguinte linha de código com a instrução *branch* para inteiros, retirada do programa 4.1:

---

```
bge $t0, 18, c3 # se idade >= 18, continua a análise
```

---

Se o programa tivesse que ser modificado para utilizar ponto flutuante com precisão dupla, considerando que a idade está salva no registrador \$f0 e o valor 18 em \$f2, o código poderia ser construído da seguinte forma:

---

```
c.lt.d $f0, $f2 # se $f0 < $f2, flag0 = true, senão
    flag0 = false
bc1f c3 # se flag0 == false, goto c3
```

---

Sabe-se que a instrução *c.ge.d* não existe na arquitetura, por isso, deve-se utilizar a instrução *inversa* que é *c.lt.d*. Se a condição for verdadeira (idade < 18), *flag 0* é alterada para *true*. Porém, na instrução de bifurcação, não se pode desviar

para o rótulo *c3* caso a *flag* seja verdadeira, pois a condição está invertida. Logo, o programa deve desviar apenas quando ela for *false*, por isso utiliza-se *bc1f* e não *bc1t*.

## 4.4 Iteração

Nesta seção, será mostrado como executar comandos de *loop* ou iteração em MIPS, como *while* e *for* em linguagens de alto nível.

Ao contrário das seções anteriores, não há mais novas instruções para as próximas seções deste capítulo; as estruturas de controle também são elaboradas por meio das instruções de desvio condicional e incondicional. Construir um *loop* com a instrução *jump* é bastante intuitivo, como no exemplo a seguir:

---

```
loop: ...  
    ...  
    j loop
```

---

Pode-se constatar que, a menos que exista outra instrução de desvio dentro do bloco, o programa entrará em *loop* infinito. Logo, inserindo uma instrução de bifurcação, o resultado será a simulação de um bloco *while*. Como exemplo, considere o código C a seguir:

---

```
do {  
    scanf("%d", &n);  
} while(n > 0 && n <= 10);
```

---

Ele faz a leitura de um valor *N* enquanto *N* estiver dentro do intervalo (0, 10]. Uma possível tradução para MIPS é o

programa 4.2 a seguir.

---

Programa 4.2 – Leitura enquanto N está no intervalo

---

```
while: li $v0, 5 # Código de leitura de inteiro
      syscall # Faz a leitura de N
      blez $v0, sair # Se N <= 0, encerra o loop
      bgt $v0, 10, sair # Se N > 10, encerra o loop
      j while # Continua o loop
sair:  li $v0, 10 # Código para finalizar o programa
      syscall # Finaliza o programa
```

---

Como há duas condições a serem satisfeitas de maneira conjunta para que a iteração continue, a validade de uma das condições não é suficiente para que se desvie para *while*. Mas a falsidade de uma das condições já é suficiente para sair do *loop*. Por isso, toda a condição é negada:  $!(n > 0 \ \&\& \ n \leq 10) = n \leq 0 \ || \ n > 10$ ; logo, caso uma das condições invertidas seja verdade, o *loop* é encerrado. Se as duas condições falharem, o valor pertence ao intervalo  $(0, 10]$  e a instrução de desvio para *while* é executada.

Pode-se observar que não existe segredo para programar iteração em MIPS. É tudo uma questão de dispor instruções de desvio em uma sequência lógica. No caso do comando *for*, sabe-se que ele executa como *while*, mas com valor inicial e incremento bem definidos. Como exemplo, o programa 4.3 calcula e imprime o valor do somatório *S* definido a seguir, cujo resultado é 880:

$$S = \sum_{k=1}^{20} (4k + 2)$$



---

Programa 4.3 – Cálculo de Somatório

---

```
.data
Res: .asciiz "O valor de S é: "

.text
main: li $t0, 1 # k = 1
for:  mul $t1, $t0, 4 # k = k * 4
      addi $t1, $t1, 2 # k = k + 2
      add $s0, $s0, $t1 # S = S + termok (4k + 2)
      addi $t0, $t0, 1 # k++
      ble $t0, 20, for # if(k <= 20) goto for
      la $a0, Res # Carrega o endereço da string Res
      li $v0, 4 # Código de impressão de string
      syscall # Imprime a string Res
      move $a0, $s0 # move o valor de S
      li $v0, 1 # Código de impressão de inteiro
      syscall # Imprime o valor de S
      li $v0, 10 # Código para finalizar o programa
      syscall # Finaliza o programa
```

---

Na seção 5.3 e adiante sobre matrizes podem ser conferidos alguns exemplos mais complexos, como a programação de *for*s e *ifs* aninhados.

## 4.5 Sub-rotinas e Procedimentos

A este ponto, o leitor da apostila pode imaginar que não há muito o que ser acrescentado em uma seção sobre “procedimentos”. Pois, de fato, as estruturas de controle vistas anteriormente não deixam de ser procedimentos: um bloco de código relativo a *then*, *else*, *while* ou *for* e identificado por um rótulo também pode ser considerado um procedimento em

MIPS.

Entretanto, alguns conceitos devem ser levados em consideração, como parâmetros, retorno e utilização de registradores. Deve-se ressaltar também que as instruções *jal* e *jr* vistas na seção 4.2 serão de suma importância para o conteúdo a seguir.

Essencialmente, partindo da visão do programa em execução, as etapas para efetuar um procedimento são as seguintes:

1. Colocar parâmetros em um lugar onde o procedimento possa acessá-los.
2. Transferir o controle para o procedimento.
3. Adquirir os recursos de armazenamento necessários para o procedimento.
4. Realizar a tarefa desejada.
5. Colocar o valor de retorno em um local onde o programa que o chamou possa acessá-lo.

A atribuição de parâmetros ou valor de retorno se dá por meio de registradores específicos, visto que em MIPS não há “variáveis”; a manipulação de valores só pode ser feita através de registradores de acesso global. Para selecioná-los, vale lembrar a convenção apresentada na tabela 1.1.

O processo de transferência do controle para uma sub-rotina e posterior recuperação pelo módulo principal ou de nível superior está ilustrado na figura 4.1. Em MIPS, não é necessário inserir um rótulo para indicar o ponto de retorno; como foi visto na seção 4.2, a instrução *jal* (*jump and link*) se

encarrega de armazenar o endereço da instrução seguinte no registrador \$ra. Então, no final do procedimento, o endereço no contador de programa pode ser alterado com a instrução *jr \$ra*, para que a execução retorne exatamente ao ponto após a chamada da sub-rotina.

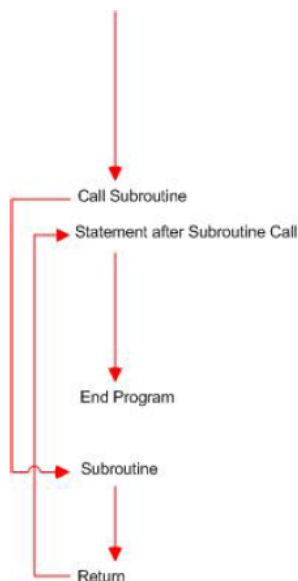


Figura 4.1 – Chamada e retorno de sub-rotina

Se outra sub-rotina for chamada dentro de um procedimento, o valor atual de \$ra deve ser salvo em outro registrador, caso contrário o endereço de retorno para o módulo principal será perdido. Se o procedimento chama a si mesmo, temos um caso de *recursividade*, e o endereço de retorno assim como os parâmetros da função e outras variáveis importantes devem

ser inseridos na *pilha*, como será visto na próxima seção.

O programa 4.4 é um exemplo de utilização de procedimentos simples, que não chamam outros procedimentos ou utilizam os registradores salvos (do tipo \$s). Ele faz a leitura de dois inteiros e imprime o maior valor na tela, com todas as ações estruturadas em procedimentos.

---

#### Programa 4.4 – Maior valor com procedimentos

---

```
.data
Ent: .asciiz "Insira um valor: "
Res: .asciiz "Maior valor: "

.text
main: jal leitura # int leitura()
      move $s0, $v0 # salva o retorno em $s0
      jal leitura # int leitura()
      move $s1, $v0 # salva o retorno em $s1
      move $a0, $s0 # parâmetro a = $s0
      move $a1, $s1 # parâmetro b = $s1
      jal maior # int maior(int a, int b)
      move $a0, $v0 # parâmetro i = retorno de maior(a,b)
      jal imprime # void imprime(int i)
      j sair # void sair()

leitura: la $a0, Ent # Carrega o endereço da string
        li $v0, 4 # Código de impressão de string
        syscall # Imprime a string
        li $v0, 5 # Código de leitura de inteiro
        syscall # Valor lido já em $v0 para retorno
        jr $ra # Retorna para a main

maior: bgt $a0, $a1, retA # Se a > b, return a
      move $v0, $a1 # senão, return b
```

```
        jr $ra # Retorna para a main
retA:   move $v0, $a0 # return a
        jr $ra # Retorna para a main

imprime: move $t0, $a0 # Move o parâmetro i para $t0
        la $a0, Res # Carrega o endereço da string
        li $v0, 4 # Código de impressão de string
        syscall # Imprime a string
        move $a0, $t0 # Move i novamente para impressão
        li $v0, 1 # Código de impressão de inteiro
        syscall # Imprime i
        jr $ra # Retorna para a main

sair:   li $v0, 10 # Código para finalizar o programa
        syscall # Finaliza o programa
```

---

A leitura de cada valor é realizada com o mesmo procedimento *leitura*, que coloca o resultado (valor lido) em \$v0 para retorno, que é então salvo em outro registrador por *main*. Os valores lidos são passados como parâmetro em \$a0 e \$a1 para *maior*, que verifica e retorna o maior valor em \$v0 novamente. O valor retornado é passado como parâmetro em \$a0 para *imprime*, que exibe o resultado na tela e não retorna valor. Por último, é chamado o procedimento *sair*, que finaliza o programa.

No exemplo dado, *main* é denominado *caller*, pois estabelece os argumentos para os outros procedimentos (nos registradores \$a0-\$a3), enquanto *leitura*, *maior* e *imprime* podem ser chamados de *callee*, visto que realizam as operações necessárias, colocam os resultados em \$v0-\$v1 e retornam o controle para o *caller* com a instrução *jr \$ra*.

## 4.6 Pilha e Recursividade

Registradores são limitados. Em algum momento, não haverá registradores suficientes para mapear todas as variáveis de um programa; por isso, é inevitável que valores devam ser movidos para armazenamento em memória.

Para o programador, há 3 tipos de memória disponível em MIPS: memória estática, memória dinâmica *heap* e dinâmica de pilha (*stack*). A memória estática é a mais simples, definida quando o programa é montado e alocada no começo da execução, e onde se encontram as “variáveis” estáticas definidas com a diretiva *.data*, como foi visto nos capítulos anteriores.

A memória dinâmica é alocada *durante* a execução do programa, o que a torna mais difícil de acessar, mas também bastante útil. Nesta seção, será visto como utilizar a memória dinâmica de pilha, enquanto a do tipo *heap* será analisada na seção 5.4 utilizando arranjos.

Como se sabe, pilha ou *stack* é uma estrutura de dados do tipo *LIFO* (*Last In, First Out*). Mas diferentemente do que se pode imaginar, em MIPS ela não é referenciada como uma estrutura “abstrata” ou uma “caixa preta”, que faz todo o trabalho de ajuste de índices, verificações, etc. Novamente, é o programador quem deve cuidar da sua correta utilização.

O registrador *\$sp* armazena o endereço para acesso à memória no segmento de pilha. Usualmente, para cada inserção de um elemento na pilha, é atribuído um valor de *offset* ou deslocamento ao endereço inicializado em *\$sp*, correspondente ao tamanho do tipo inserido. Logo, o armazenamento na pilha de uma *word* que se encontra no registrador *\$t0* po-

deria ser feito da seguinte forma:

---

```
sw $t0, -4($sp)
```

---

Porém, recomenda-se que o valor em  $\$sp$  seja manipulado de forma que contenha o endereço do valor que está no “topo” da pilha; do contrário, o programador poderá não saber qual *offset* deve atribuir para acessar o último item ou um espaço não ocupado. Então, considerando que é necessário salvar 4 registradores na pilha, de  $\$t0$  a  $\$t3$ :

---

```
subi $sp, $sp, 16  
sw $t0, 12($sp)  
sw $t1, 8($sp)  
sw $t2, 4($sp)  
sw $t3, ($sp)
```

---

Diz-se que a instrução *subi \$sp, \$sp, 16* “alocou” espaço para 4 itens (*words*), ou que a pilha foi “ajustada” para 4 itens. Também é comum que se utilize a instrução *addi* junto com um valor imediato negativo, mas não há diferença no resultado.

Pode-se observar que o item inserido na última instrução - no “topo” da pilha - encontra-se armazenado do endereço  $e - 16$  ao endereço  $e - 12$ , onde  $e$  é o valor do endereço original, antes do “ajuste”. Já o item na primeira instrução se encontra na “base” da pilha, cujos dados estão contidos no endereço  $e - 4$  até  $e$ . Se denominarmos o novo endereço como  $e_n$ , então  $e_n = e - 16$ , e os itens citados se encontram nos endereços  $e_n$  e  $e_n + 12$ , respectivamente.

A figura 4.2 ilustra a disposição dos elementos no segmento de pilha, bem como o conteúdo no registrador  $\$sp$

indicando o endereço do último item carregado na memória.

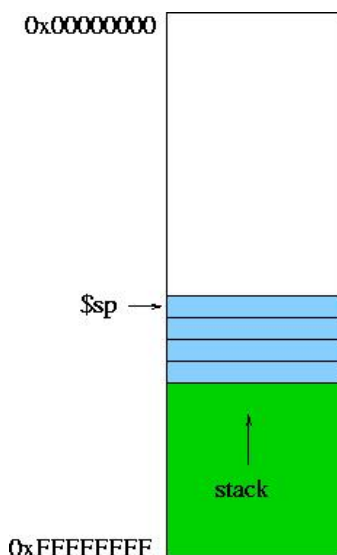


Figura 4.2 – Segmento de Pilha

É responsabilidade do *callee* (descrito na seção anterior) determinar quais registradores salvos (do tipo \$s) ele pretende utilizar, movê-los para a pilha, fazer uso dos registradores no corpo principal do procedimento, e desempilhar os valores em seus respectivos registradores logo antes de retornar o controle para o *caller*.

Para demonstrar o processo de salvar (empilhar) e restaurar (desempilhar) registradores, a função em C a seguir será traduzida para MIPS, considerando que as variáveis de parâmetro *g*, *h*, *i* e *j* correspondem aos registradores \$a0, \$a1, \$a2



e \$a3, e a variável *f* corresponde a \$s0.

---

```
int exemplo_folha(int g, int h, int i, int j){
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

---

Assim, é necessário salvar \$s0 na pilha. Os registradores \$t0 e \$t1 utilizados também serão salvos, apesar da convenção não garantir isso. Como se sabe da seção anterior, o retorno é armazenado em \$v0.

---

```
exemplo_folha:
addi $sp, $sp, -12 # Ajusta a pilha criando espaço para
                    3 itens
sw $t1, 8($sp) # Salva $t1 para usar depois
sw $t0, 4($sp) # Salva $t0 para usar depois
sw $s0, 0($sp) # Salva $s0 para usar depois
add $t0, $a0, $a1 # $t0 contém (g + h)
add $t1, $a2, $a3 # $t1 contém (i + j)
sub $s0, $t0, $t1 # $s0 recebe (g + h) - (i + j)
add $v0, $s0, $zero # $v0 = $s0 + 0
lw $s0, 0($sp) # Restaura $s0 para o caller
lw $t0, 4($sp) # Restaura $t0 para o caller
lw $t1, 8($sp) # Restaura $t1 para caller
addi $sp, $sp, 12 # Ajusta a pilha para excluir 3 itens
jr $ra # Desvia de volta à rotina que o chamou
```

---

Como foi mencionado anteriormente, outra utilidade essencial para a pilha é na recursividade, quando o procedimento é *caller* e *callee* ao mesmo tempo. Em linguagens de alto nível, podemos utilizar funções recursivas de forma intuitiva.

tiva, e toda a preservação do *contexto* de cada procedimento chamado é feita inteiramente pelo compilador, o que pode ser bastante custoso em termos de memória e tornar a execução mais lenta. Por isso, caso um tempo menor de execução seja desejável, pode ser mais vantajoso construir o algoritmo de forma iterativa.

Em MIPS, a obrigação de salvar e restaurar os contextos de cada chamada recursiva também é do programador. *Contexto* pode ser entendido como todas as informações que serão necessárias para a execução do procedimento. Para exemplificar, considere o código em C para a função recursiva que retorna o *n*ésimo elemento da sequência de *Fibonacci*:

---

```
int fib(int n){
    if(n == 0)
        return 0;
    else if(n == 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

---

Embora possa parecer simples em alto nível, traduzir este código para MIPS pode ser uma tarefa árdua para iniciantes sem antes receber algumas orientações. A questão principal é entender como lidar com a troca de contextos - quando o procedimento está *entrando* em um novo contexto, pois então o contexto anterior precisa ser *salvo*; e quando está *saindo* de um contexto, onde o contexto anterior precisa ser *recuperado*. Então, basta saber *o que* salvar do contexto, e o código pode ser construído sem maiores problemas.

Para os casos básicos de  $n == 0$  e  $n == 1$ , não há nenhuma

chamada de função (recursiva ou não), apenas desvio condicional e retorno. Logo, esta seção pode ser programada apenas com *branch*, armazenamento do valor de retorno em  $\$v0$  (0 ou 1) e instrução de retorno para o *caller* com *jr \$ra*, sem utilizar pilha ou salvar registradores.

Para valores diferentes de  $n$ , são feitas duas chamadas recursivas. Isto significa que o procedimento está saindo do seu contexto atual e indo para um novo contexto, pois uma nova *instância* da função será criada. A convenção para procedimentos continua válida: o procedimento atual deve preparar os argumentos carregando o valor de  $n - 1$  ou  $n - 2$  em  $\$a0$ .

Na preservação de contextos, um dos itens obrigatórios para inserção na pilha é o endereço de retorno. Para retornar ao contexto do *caller*, este endereço será desempilhado e utilizado na instrução *jr \$ra*. Se o contexto atual for relativo a uma chamada recursiva (a mesma sub-rotina é *caller*), o programa irá continuar a partir de uma instrução no mesmo procedimento. Se for o contexto da primeira chamada da função, o valor salvo é o endereço da próxima instrução do bloco *main*.

Outro elemento do contexto que deve ser salvo é o valor de  $n$ , pois o argumento passado na chamada recursiva é uma nova “declaração” da variável  $n$ , e uma alteração no seu valor não terá relação com o valor da variável do contexto anterior. Porém, especificamente para este código, não é necessário salvar  $n$  na pilha, pois como o seu valor é alterado apenas na passagem de parâmetro, ele pode ser restaurado para o contexto anterior apenas incrementando o mesmo valor que foi decrementado. Ou seja, após subtrair 1 de  $\$a0$  e executar o procedimento para  $n - 1$ , basta adicionar 1 novamente em  $\$a0$  para voltar ao valor de  $n$ . O “correto”, seria salvar o con-

texto de  $n$  na pilha, pois é o que um compilador faria; mas deixar de inserir mais um item na pilha simplifica bastante as instruções e diminui o número de registradores utilizados no procedimento, o que facilita o entendimento do código pelo leitor da apostila, além de economizar memória e otimizar o programa.

Considerando que  $\text{fib}(n-1)$  irá executar antes de  $\text{fib}(n-2)$  - o que não é obrigatório - o procedimento retornará primeiro o resultado de  $\text{fib}(n-1)$  em  $\$v0$ . Este valor faz parte do contexto atual e também deve ser salvo na pilha na troca de contexto, pois  $\text{fib}(n-2)$  ainda será executado; se fosse armazenado o resultado de  $\text{fib}(n-1)$  em um registrador auxiliar, ele poderia ser mantido caso o parâmetro  $n - 2$  seja 0 ou 1, mas caso seja diferente,  $\text{fib}(n-1)$  seria executado novamente no novo contexto e o valor no registrador auxiliar seria comprometido. No caso de  $\text{fib}(n-2)$ , o valor retornado será utilizado imediatamente, pois será somado com o resultado salvo de  $\text{fib}(n-1)$  e carregado em  $\$v0$  para retorno.

Logo, concluímos que devem ser salvos no mínimo o endereço de retorno em  $\$ra$  e o resultado de  $\text{fib}(n-1)$ . Estes valores devem ser inseridos na pilha após as duas instruções de *branch* para verificação de  $n == 0$  e  $n == 1$  falharem, pois então começarão as chamadas recursivas e o contexto será trocado. Após computar o resultado de  $\text{fib}(n-1) + \text{fib}(n-2)$  e armazená-lo em  $\$v0$  para retorno, o contexto anterior deve ser restaurado recuperando da pilha o resultado de  $\text{fib}(n-1)$  e o endereço de  $\$ra$  para uso na instrução *jr \$ra*. No caso do retorno de 0 ou 1, o contexto atual não foi substituído e portanto não precisa ser recuperado.

Vale notar que apenas na chamada recursiva de  $\text{fib}(n-2)$  o

valor do contexto inserido na pilha para  $\text{fib}(n-1)$  é válido. Ou seja, em menos da metade das vezes, na chamada de  $\text{fib}(n)$  ou na troca de contexto para  $\text{fib}(n-1)$ , um valor inválido que não é utilizado é inserido na pilha para  $\text{fib}(n-1)$ , pois é justamente a execução de  $\text{fib}(n-1)$  que irá computá-lo e retorná-lo em  $\$v0$ .

O programa 4.5 apresenta a codificação em MIPS da função recursiva de Fibonacci analisada, junto com procedimentos para a leitura do valor de  $n$  e a impressão do resultado.

---

#### Programa 4.5 – Fibonacci Recursivo

---

```
.data
EntN: .asciiz "Insira o valor de N: "
Res: .asciiz "Fib(N) = "

.text
main: jal N # Leitura do valor de N
      move $a0, $v0 # Move N para $a0 (parâmetro)
      jal fib # Cálculo de Fib(N)
      move $a0, $v0 # Move o resultado de Fib(N) para $a0
      jal res # Imprime o resultado
      j sair # Finaliza o programa

N: la $a0, EntN # Carrega o endereço da string
   li $v0, 4 # Código de impressão de string
   syscall # Impressão da string
   li $v0, 5 # Código de leitura de inteiro
   syscall # Leitura de N (retorna em $v0)
   jr $ra # Retorna para a main

fib: beq $a0, $zero, r0 # if(N == 0), return 0
     beq $a0, 1, r1 # if(N == 1), return 1
     subi $sp, $sp, 8 # Aloca espaço na pilha para $ra e
                      # resultado de fib(N-1)
```

```
sw $ra, ($sp) # Salva o retorno na pilha
sw $t0, 4($sp) # Salva o resultado de fib(N-1) na
pilha
subi $a0, $a0, 1 # parâmetro N-1
jal fib # fib(N-1)
move $t0, $v0 # $t0 = fib(N-1)
subi $a0, $a0, 1 # parâmetro N-2
jal fib # fib(N-2)
addi $a0, $a0, 2 # Volta o valor de N (N + 2)
add $v0, $v0, $t0 # Valor para retorno: fib(N-2) +
fib(N-1)
lw $ra, ($sp) # Recupera da pilha o retorno para o
contexto anterior
lw $t0, 4($sp) # Recupera da pilha o resultado de
fib(N-1) do contexto anterior
addi $sp, $sp, 8 # Libera o espaço dos 2 valores na
pilha
jr $ra # Retorna para fib(N-1), fib(N-2) ou main
r0: li $v0, 0 # Valor para retorno: 0
    jr $ra # Retorna para fib(N-1), fib(N-2) ou main
r1: li $v0, 1 # Valor para retorno: 1
    jr $ra # Retorna para fib(N-1), fib(N-2) ou main

res: move $t0, $a0 # Move o parâmetro temporariamente
    la $a0, Res # Carrega o endereço da string
    li $v0, 4 # Código de impressão de string
    syscall # Impressão da string
    move $a0, $t0 # Move o parâmetro de volta para $a0
    li $v0, 1 # Código de impressão de inteiro
    syscall # Imprime o resultado
    jr $ra # Retorna para a main

sair: li $v0, 10 # Código para finalizar o programa
    syscall # Finaliza o programa
```

---

# Arranjos N-Dimensionais

Arranjos são conjuntos de elementos de um mesmo tipo, acessados através de um índice que indica a posição do valor no arranjo. Neste capítulo, será visto como criar, acessar e manipular arranjos unidimensionais (como vetores e cadeias de caracteres) e bidimensionais (matrizes) em MIPS, através de alocação estática (em *.data*) ou dinâmica (memória *heap*).

Felizmente, a maioria dos estudantes de computação já possuem bastante prática com este tipo de estrutura, devido a sua experiência em outras linguagens de alto nível. Por isso, a abordagem terá maior foco em exemplos de tradução de código em C, como declaração ou alocação das estruturas, acesso, leitura e escrita.

## 5.1 Vetores

Em C, um vetor do tipo de inteiro com 500 posições pode ser declarado da seguinte forma:

---

```
int vetor[500];
```

---

Sabendo que cada campo do vetor de inteiros é uma *word* de 4 bytes, é necessário criar um espaço de  $500 * 4 = 2000$  bytes para obter uma estrutura com a mesma capacidade de armazenamento em MIPS. Como foi descrito na tabela 2.1, isto pode ser feito com a diretiva *.space* da seguinte forma:

---

```
.data  
vetor: .space 2000
```

---

O rótulo *vetor* irá conter o endereço em memória correspondente ao primeiro elemento do vetor, no “índice zero”. Para cada 4 bytes somados ao endereço, o índice correspondente aumenta em 1 unidade.

Também é possível declarar um vetor inicializado, cujos campos já possuem um valor definido em memória. Considerando a linha de código em C:

---

```
int vet[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
```

---

O código equivalente em MIPS seria:

---

```
.data  
vet: .word 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
```

---

Cada item listado é armazenado como um bloco de 4 bytes de forma *contígua*. Assim como no exemplo anterior, o endereço base pode ser obtido através do rótulo *vet*, já contendo o valor 10 armazenado.



O endereçamento dos campos do vetor não é diferente do que foi visto nos capítulos anteriores, principalmente na seção 1.7 de endereçamento, e nos exemplos com o registrador `$sp` na seção 4.6 sobre pilha.

Sabemos que para acessar o primeiro elemento do vetor, é necessário o endereço base somado a um *offset* de zero, ou seja, o próprio endereço base. Para acessar o último elemento no exemplo do vetor de tamanho 10, é necessário somar o endereço base com um *offset* de  $9 * 4 = 36$ , como demonstrado nas instruções a seguir, onde o endereço base é carregado em `$t0` e `vet[9]` em `$a0`:

---

```
la $t0, vet
lw $a0, 36($t0)
```

---

O valor não poderia ser acessado com um *offset* de 40, pois o endereço resultante corresponderia ao final do vetor na memória. O elemento em si encontra-se armazenado entre  $e + 36$  e  $e + 40$ . Por isso, o último item de um vetor com  $N$  elementos é acessado pelo índice  $N - 1$ .

Obviamente, não utilizamos vetores inserindo diretamente o *offset* dessa forma. Como um vetor é percorrido com estruturas de iteração como *for*, o endereço de cada campo deve ser calculado pelo programa, somando-se o endereço base ao índice atual multiplicado pelo número de bytes do tipo de dado - no caso de inteiros, 4. Entretanto, se todos os campos do vetor devem ser percorridos de forma sequencial, outra forma de computar o endereço final é somando 4 ao endereço base, a cada iteração.

Vale lembrar que, como foi explicado na seção 3.3 sobre operações lógicas, a instrução *sll* (shift à esquerda) de 2 bits pode ser utilizada como uma instrução equivalente à multiplicação por 4.

O programa 5.1 demonstra como realizar a leitura de um vetor com 5 posições e em seguida imprimir o seu conteúdo.

---

#### Programa 5.1 – Leitura e Escrita de Vetor

---

```
.data
ent: .asciiz "Insira o valor de Vet["
ent2: .asciiz "]: "
.align 2
vet: .space 20

.text
main: la $a0, vet # Endereço do vetor como parâmetro
      jal leitura # leitura(vet)
      move $a0, $v0 # Endereço do vetor retornado
      jal escrita # escrita(vet)
      li $v0, 10 # Código para finalizar o programa
      syscall # Finaliza o programa

leitura:
      move $t0, $a0 # Salva o endereço base de vet
      move $t1, $t0 # Endereço de vet[i]
      li $t2, 0 # i = 0
l: la $a0, ent # Carrega o endereço da string
   li $v0, 4 # Código de impressão de string
   syscall # Impressão da string
   move $a0, $t2 # Carrega o índice do vetor
   li $v0, 1 # Código de impressão de inteiro
   syscall # Imprime o índice i
   la $a0, ent2 # Carrega o endereço da string
```

```
li $v0, 4 # Código de impressão de string
syscall # Impressão da string
li $v0, 5 # Código de leitura de inteiro
syscall # Leitura do valor
sw $v0, ($t1) # Salva o valor lido em vet[i]
add $t1, $t1, 4 # Endereço de vet[i+1]
addi $t2, $t2, 1 # i++
blt $t2, 5, 1 # if(i < 5) goto 1
move $v0, $t0 # Endereço de vet para retorno
jr $ra # Retorna para a main
```

escrita:

```
move $t0, $a0 # Salva o endereço base de vet
move $t1, $t0 # Endereço de vet[i]
li $t2, 0 # i = 0
e: lw $a0, ($t1) # Carrega o valor de vet[i]
li $v0, 1 # Código de impressão de inteiro
syscall # Imprime vet[i]
li $a0, 32 # Código ASCII para espaço
li $v0, 11 # Código de impressão de caractere
syscall # Imprime um espaço
add $t1, $t1, 4 # Endereço de vet[i+1]
addi $t2, $t2, 1 # i++
blt $t2, 5, e # if(i < 5) goto e
move $v0, $t0 # Endereço de vet para retorno
jr $ra # Retorna para a main
```

---

Como são 5 elementos, é definido um espaço de 20 bytes para o vetor. Deve-se ressaltar que é preciso ter cuidado ao utilizar a diretiva *.space*, pois ela aloca o número de bytes *no próximo endereço disponível*. Isto significa que, se o rótulo *vet* é definido após as strings *ent* e *ent1* em um endereço não múltiplo de 4, poderia ser acusado erro em tempo de execução por *not aligned on word boundary*. Por isso, é utilizada a diretiva *.allign* com o número 2, que é o código para *word*, ali-

nhando o rótulo em um endereço válido. Para as diretivas com tipo definido, como *.word*, *.float* e *.double*, o alinhamento já é feito automaticamente. Outra solução seria definir *vet* como o primeiro rótulo no segmento de dados.

O endereço de *vet* é passado como parâmetro em \$a0 para os procedimentos de leitura e escrita. O valor do endereço de *vet[i]* é computado em \$t1 somando 4 a cada iteração, pois todos os campos do vetor devem ser percorridos de forma sequencial. Ao final dos procedimentos, o endereço base é carregado em \$v0 para retorno. No procedimento de escrita, é impresso o caractere de espaço para separar os valores do vetor. Segue um exemplo de execução para o programa 5.1:

---

```
Insira o valor de Vet[0]: 7
Insira o valor de Vet[1]: 19
Insira o valor de Vet[2]: 2
Insira o valor de Vet[3]: -5
Insira o valor de Vet[4]: 46
7 19 2 -5 46
-- program is finished running --
```

---

## 5.2 Strings

Como se sabe, cadeia de caracteres ou *string* também é um arranjo unidimensional ou vetor; a diferença é que cada campo possui o tamanho de 1 byte. Como 1 byte equivale a 8 bits, o valor possui capacidade de representação decimal de 0 a 255. Cada valor neste intervalo representa um caractere diferente, definido pelo famoso padrão *ASCII*.

Assim, percorrer strings é bastante simples, valendo a mesma lógica vista na seção anterior, mas com o incremento de apenas 1 unidade no endereço em vez de 4. Além disso, em vez de instruções como *lw* e *sw* (*load word* e *store word*), são utilizadas as instruções *lb* e *sb* (*load byte* e *store byte*).

É importante que uma string seja *null-terminated*, ou seja, possua o valor 0 (*NULL*) na sua última posição, indicando o último caractere da string. Caso contrário, podem ocorrer alguns problemas, como no código a seguir:

---

```
.data
str1: .ascii "string 1"
str2: .asciiz "string 2"

.text
main: la $a0, str1
      li $v0, 4
      syscall
      li $v0, 10
      syscall
```

---

As duas strings são armazenadas em sequência no segmento de dados, e *str2* é *null-terminated* enquanto *str1* não é. A primeira string é carregada para impressão com uma chamada de sistema. Executando o código, ele apresentará a seguinte saída:

---

```
string 1string 2
-- program is finished running --
```

---

O protocolo do código 4 da chamada de sistema especifica que uma string terminada em nulo deveria ser passada como parâmetro em *\$a0*. Ao percorrer *str1*, em vez de encon-

trar o valor 0 depois do último caractere, o sistema acessou o primeiro byte de *str2*, então ela foi tratada como mera continuação de *str1*.

A utilização do código 8 da chamada de sistema, que realiza a leitura de uma string, ainda não foi vista nesta apostila. Como pode ser conferido na tabela 2.2, o protocolo especifica que o registrador \$a0 contenha o endereço para armazenamento da string lida, enquanto \$a1 deve indicar o número máximo de caracteres para leitura.

O comprimento da string em si não pode ser maior do que  $N - 1$ , onde  $N$  é o valor carregado em \$a1, pois o caractere nulo é inserido automaticamente na última posição. Caso o tamanho seja ainda menor que  $N - 1$ , também é inserido um caractere de *newline* ( $\backslash n$ ) antes do caractere nulo.

Um ponto interessante desta chamada de sistema no ambiente do MARS, é que se o número máximo de caracteres definido for alcançado enquanto se digita a entrada, a leitura é automaticamente interrompida. Ou seja, no caso do valor 4 em \$a1, ao executar a chamada de sistema e digitar “abc”, a operação é finalizada sem a necessidade de pressionar *enter*. Entretanto, é permitida a deleção de caracteres digitados enquanto o tamanho máximo não for atingido.

Como exemplo, segue o programa 5.2 que faz a leitura de 2 strings de no máximo 100 caracteres, e realiza a intercalação entre elas armazenando a sequência resultante em uma terceira string, que consequentemente possui um tamanho máximo de 200 caracteres.

---

Programa 5.2 – Intercalação de Strings

---

```
.data
ent1: .asciiz "Insira a string 1: "
ent2: .asciiz "Insira a string 2: "
str1: .space 100
str2: .space 100
str3: .space 200

.text
main: la $a0, ent1 # Parâmetro: mensagem
      la $a1, str1 # Parâmetro: endereço da string
      jal leitura # leitura(mensagem, string)
      la $a0, ent2 # Parâmetro: mensagem
      la $a1, str2 # Parâmetro: endereço da string
      jal leitura # leitura(mensagem, string)
      la $a0, str1 # Parâmetro: endereço da string 1
      la $a1, str2 # Parâmetro: endereço da string 2
      la $a2, str3 # Parâmetro: endereço da string 3
      jal intercala # intercala(str1, str2, str3)
      move $a0, $v0 # Move o retorno da string resultante
      li $v0, 4 # Código de impressão de string
      syscall # Imprime a string intercalada
      li $v0, 10 # Código para finalizar o programa
      syscall # Finaliza o programa

leitura:
      li $v0, 4 # Código de impressão de string
      syscall # Imprime a mensagem
      move $a0, $a1 # Endereço da string para leitura
      li $a1, 100 # Número máximo de caracteres
      li $v0, 8 # Código de leitura de string
      syscall # Faz a leitura da string
      jr $ra # Retorna para a main
```

intercala:

```
    move $v0, $a2 # Salva o endereço de str3 para retorno
c1: lb $t0, ($a0) # ch1 = str1[i]
    beqz $t0, c2 # if(ch1 == NULL) goto c2
    addi $a0, $a0, 1 # str1[i++]
    beq $t0, 10, c2 # if(ch1 == '\n') goto c2
    sb $t0, ($a2) # str3[k] = ch1
    addi $a2, $a2, 1 # str3[k++]
c2: lb $t1, ($a1) # ch2 = str2[j]
    beqz $t1, c # if(ch2 == NULL) goto c
    addi $a1, $a1, 1 # str2[j++]
    beq $t1, 10, c # if(ch2 == '\n') goto c
    sb $t1, ($a2) # str3[k] = ch2
    addi $a2, $a2, 1 # str3[k++]
c:  add $t0, $t0, $t1 # ch1 = ch1 + ch2
    bnez $t0, c1 # if(ch1 != 0) goto c1
    sb $zero, ($a2) # str3[k] = NULL
    jr $ra # Retorna para a main
```

---

Dada a descrição do programa, os rótulos de *str1*, *str2* e *str3* são definidos com o número de bytes igual à quantidade máxima de caracteres. O procedimento de leitura para as strings 1 e 2 utiliza o mesmo valor como parâmetro para o número máximo de caracteres para leitura.

No procedimento *intercala*, cada iteração carrega primeiro um byte de *str1* e o armazena em *str3* se ele não for zero, o que indica o fim da string. Se o byte tiver o valor 10, ele é ignorado pois indica o caractere “\n”. O endereço de cada string é incrementado em 1. Então, o mesmo processo é realizado para *str2*. Para continuar a iteração, é verificado se a soma dos bytes carregados de *str1* e *str2* é diferente de zero. Quando a soma for zero, significa que a última posição de ambas foi alcançada.



## 5.3 Matrizes

A maneira com que tratamos matrizes em linguagens de alto nível é um pouco abstrata demais. Ela leva a pensar que realmente existe uma estrutura com linhas e colunas na memória, quando o fato é que há apenas um longo vetor, como a própria memória.

Uma matriz 3x3 é na realidade um vetor com 9 posições. Neste exemplo, quando é acessado o campo da segunda linha e segunda coluna da matriz, está sendo acessada a quinta posição do vetor. É possível fazer esta substituição com a seguinte fórmula:

$$Vet_i = (Mat_i * Mat_{ncol}) + Mat_j$$

A posição atual no vetor é igual à linha atual da matriz multiplicada pelo número de colunas por linha, somado com a coluna atual da matriz. Considera-se que o primeiro índice de cada linha ou coluna inicia em zero. Logo, a segunda posição no exemplo anterior é o índice 1, então  $(1 * 3) + 1 = 4$  (índice 4 ou quinta posição do vetor).

$$\begin{pmatrix} 0_{00} & 1_{01} & 2_{02} \\ 3_{10} & 4_{11} & 5_{12} \\ 6_{20} & 7_{21} & 8_{22} \end{pmatrix}$$

O cálculo também funciona para matrizes não quadradas, e a lógica pode ser adaptada para arranjos com qualquer número de dimensões.

Enquanto em C a conversão dos índices é feita automaticamente, para referenciar matrizes por linha e coluna em MIPS é necessário efetuar o cálculo para obter o índice do vetor que a representa. Então, como foi visto nas seções anteriores deste capítulo, o índice deve ser multiplicado pelo tamanho do dado e somado ao endereço base do vetor.

Para demonstrar os conceitos apresentados, considere a declaração de 2 matrizes em C, uma inicializada e a outra não:

---

```
double mat1[5][7];  
int mat2[3][2] = { { 12, 7 },  
                  { -5, 0 },  
                  { 1, 4 } };
```

---

A primeira matriz, *mat1*, possui 5 linhas e 7 colunas, o que é equivalente a um vetor de comprimento 35. Como a matriz é do tipo *double*, são necessários 8 bytes para cada campo da estrutura. Logo, a matriz exige um total de 280 bytes para armazenamento.

A segunda matriz *mat2*, com 3 linhas, 2 colunas e do tipo inteiro, necessita de  $3 * 2 * 4 = 24$  bytes. Porém, ela já é inicializada, então é possível listar os valores com a diretiva *.word* e o espaço necessário será alocado. A listagem deve ser feita especificamente na ordem *esquerda para direita e cima para baixo*, criando o vetor que representa a matriz. Segue a definição de *mat1* e *mat2* em MIPS:

---

```
.data  
mat1: .space 280  
mat2: .word 12, 7, -5, 0, 1, 4
```

---

O código a seguir apresenta a lógica clássica para percorrer uma matriz em C, utilizando duas estruturas de *for* aninhadas.

---

```
for(i=0; i < n; i++){  
    for(j=0; j<m; j++){  
        mat[i][j] = 1;  
    }  
}
```

---

Em MIPS, é possível executar a mesma lógica a partir de duas instruções de bifurcação: uma para verificar o valor de *j*, simulando o *loop* interno, executando antes da outra instrução para *i*, simulando o *loop* externo. Dentro do *loop*, ou seja, antes das instruções de *branch*, são colocadas as instruções referentes aos comandos do bloco. Para o caso do exemplo acima, o endereço do campo da matriz deve ser computado e armazenar o valor 1.

---

loop:

```
mul $t2, $t0, $a2 # i * ncol  
add $t2, $t2, $t1 # (i * ncol) + j  
sll $t2, $t2, 2 # [(i * ncol) + j] * 4 (inteiro)  
add $t2, $t2, $a0 # Soma o endereço base de mat  
li $t3, 1 # aux = 1  
sw $t0, ($t2) # mat[i][j] = aux  
addi $t1, $t1, 1 # j++  
blt $t1, $a2, loop # if(j < ncol) goto loop  
li $t1, 0 # j = 0  
addi $t0, $t0, 1 # i++  
blt $t0, $a1, loop # if(i < nlin) goto loop
```

---

Sempre que se for programar operações sobre matrizes, é recomendável que se implemente as instruções para cálculo do endereço de `mat[i][j]` como um procedimento. A vantagem é que em toda operação que percorre a matriz, em vez de re-

petir o código para cálculo do índice, pode-se apenas chamar o procedimento a cada iteração. Além disso, o programador pode “ignorar” a parte de baixo nível dos endereços, direcionando sua preocupação para o problema em questão que referencia os campos da matriz com “i” e “j”.

Para exemplificar, segue o programa 5.3 que faz a leitura e escrita de uma matriz inteira 3x3.

---

### Programa 5.3 – Leitura e Escrita de Matriz

---

```
.data
Mat: .space 36 # 3x3 * 4 (inteiro)
Ent1: .asciiz "    Insira o valor de Mat["
Ent2: .asciiz "]"
Ent3: .asciiz "]: "

.text
main: la $a0, Mat # Endereço base de Mat
      li $a1, 3 # Número de linhas
      li $a2, 3 # Número de colunas
      jal leitura # leitura(mat, nlin, ncol)
      move $a0, $v0 # Endereço da matriz lida
      jal escrita # escrita(mat, nlin, ncol)
      li $v0, 10 # Código para finalizar o programa
      syscall # Finaliza o programa

indice:
      mul $v0, $t0, $a2 # i * ncol
      add $v0, $v0, $t1 # (i * ncol) + j
      sll $v0, $v0, 2 # [(i * ncol) + j] * 4 (inteiro)
      add $v0, $v0, $a3 # Soma o endereço base de mat
      jr $ra # Retorna para o caller
```

leitura:

```
    subi $sp, $sp, 4 # Espaço para 1 item na pilha
    sw $ra, ($sp) # Salva o retorno para a main
    move $a3, $a0 # aux = endereço base de mat
l: la $a0, Ent1 # Carrega o endereço da string
    li $v0, 4 # Código de impressão de string
    syscall # Imprime a string
    move $a0, $t0 # Valor de i para impressão
    li $v0, 1 # Código de impressão de inteiro
    syscall # Imprime i
    la $a0, Ent2 # Carrega o endereço da string
    li $v0, 4 # Código de impressão de string
    syscall # Imprime a string
    move $a0, $t1 # Valor de j para impressão
    li $v0, 1 # Código de impressão de inteiro
    syscall # Imprime j
    la $a0, Ent3 # Carrega o endereço da string
    li $v0, 4 # Código de impressão de string
    syscall # Imprime a string
    li $v0, 5 # Código de leitura de inteiro
    syscall # Leitura do valor (retorna em $v0)
    move $t2, $v0 # aux = valor lido
    jal indice # Calcula o endereço de mat[i][j]
    sw $t2, ($v0) # mat[i][j] = aux
    addi $t1, $t1, 1 # j++
    blt $t1, $a2, l # if(j < ncol) goto l
    li $t1, 0 # j = 0
    addi $t0, $t0, 1 # i++
    blt $t0, $a1, l # if(i < nlin) goto l
    li $t0, 0 # i = 0
    lw $ra, ($sp) # Recupera o retorno para a main
    addi $sp, $sp, 4 # Libera o espaço na pilha
    move $v0, $a3 # Endereço base da matriz para retorno
    jr $ra # Retorna para a main
```

escrita:

```
    subi $sp, $sp, 4 # Espaço para 1 item na pilha
    sw $ra, ($sp) # Salva o retorno para a main
    move $a3, $a0 # aux = endereço base de mat
e: jal indice # Calcula o endereço de mat[i][j]
    lw $a0, ($v0) # Valor em mat[i][j]
    li $v0, 1 # Código de impressão de inteiro
    syscall # Imprime mat[i][j]
    la $a0, 32 # Código ASCII para espaço
    li $v0, 11 # Código de impressão de caractere
    syscall # Imprime o espaço
    addi $t1, $t1, 1 # j++
    blt $t1, $a2, e # if(j < ncol) goto e
    la $a0, 10 # Código ASCII para newline ('\n')
    syscall # Pula a linha
    li $t1, 0 # j = 0
    addi $t0, $t0, 1 # i++
    blt $t0, $a1, e # if(i < nlin) goto e
    li $t0, 0 # i = 0
    lw $ra, ($sp) # Recupera o retorno para a main
    addi $sp, $sp, 4 # Libera o espaço na pilha
    move $v0, $a3 # Endereço base da matriz para retorno
    jr $ra # Retorna para a main
```

---

## 5.4 Memória Dinâmica (*Heap*)

Foram vistos 2 dos três tipos de memória acessíveis pelo usuário: estática, cujos elementos são indicados pela diretiva *.data* e alocados no início da execução; e a memória de *stack*, explorada na seção 4.6 sobre pilha e recursividade. Nesta seção, será visto como utilizar a memória dinâmica *heap*.

Assim como em outras linguagens, a memória dinâmica permite que espaços de memória sejam alocados durante a execução. Basicamente, em MIPS isto é feito a partir de uma *syscall* (chamada de sistema), cujo nome do serviço é *sbrk* (*allocate heap memory*).

O procedimento é relativamente simples. O código a ser carregado em *\$v0* é o número 9, com a quantidade de bytes a ser alocada em *\$a0*. Após a execução da instrução *syscall*, o registrador *\$v0* conterá o endereço da memória alocada.

No caso de códigos em C que utilizam a função *malloc()*, como no exemplo a seguir:

---

```
int *n = malloc(sizeof(int));
*n = 3;
int *vet = malloc(sizeof(int) * 10);
vet[0] = 7;
vet[3] = 11;
vet[8] = 34;
char *s = malloc(sizeof(char) * 20);
scanf("%s", s);
```

---

Pode-se fazer a tradução para MIPS da seguinte forma:

---

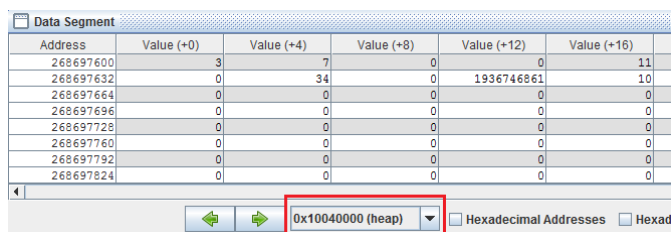
```
li $a0, 4 # 4 bytes (inteiro)
li $v0, 9 # Código de alocação dinâmica heap
syscall # Aloca 4 bytes (endereço em $v0)
move $t0, $v0 # Move para $t0
li $t1, 3 # aux = 3
sw $t1, ($t0) # *n = 3
li $a0, 40 # 40 bytes (espaço para 10 inteiros)
li $v0, 9 # Código de alocação dinâmica heap
syscall # Aloca 40 bytes
```

```

move $t1, $v0 # Move para $t1
li $t2, 7 # aux = 7
sw $t2, ($t1) # v[0] = 7
li $t2, 11 # aux = 11
sw $t2, 12($t1) # v[3] = 11
li $t2, 34 # aux = 34
sw $t2, 32($t1) # v[8] = 34
li $a0, 20 # 20 bytes (espaço para 20 char)
li $v0, 9 # Código de alocação dinâmica heap
syscall # Aloca 20 bytes
move $a0, $v0 # Endereço base da string
li $a1, 20 # Número máximo de caracteres
li $v0, 8 # Código para leitura de string
syscall # scanf("%s", s)

```

É possível visualizar os dados alocados dinamicamente no painel de execução do MARS, de forma similar ao que foi descrito na seção 2.4 sobre depuração com o segmento de dados. A diferença é que deve-se alterar o segmento visualizado de *.data* para *heap*, como ilustrado na figura 5.1.



Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)
268697600	3	7	0	0	11
268697632	0	34	0	1936746861	10
268697664	0	0	0	0	0
268697696	0	0	0	0	0
268697728	0	0	0	0	0
268697760	0	0	0	0	0
268697792	0	0	0	0	0
268697824	0	0	0	0	0

Navigation buttons: Previous, Next, Address: 0x10040000 (heap), Hexadecimal Addresses, Hexad

Figura 5.1 – Dados na memória dinâmica heap

Isto pode ser útil no caso da alocação de estruturas mais complexas, como *structs* em C que contêm tipos diferentes de dados, para saber exatamente em qual endereço cada “variá-



vel” está armazenada na estrutura e quanto deve-se atribuir de *offset* para acessá-la.

No contexto de arranjos, a alocação dinâmica é especialmente útil quando não se sabe previamente qual será o tamanho máximo de um vetor, ou o número máximo de linhas e colunas de uma matriz, por exemplo.

O mesmo cálculo que foi efetuado nas seções anteriores para determinar qual o número de bytes a ser alocado com a diretiva *.space* continua válido para a alocação dinâmica. A diferença é justamente que, como a alocação pode ser realizada em tempo de execução, o tamanho dos arranjos pode ser inserido pelo usuário.

Como exemplo, segue o programa 5.4 que a faz a leitura de uma matriz real  $N \times M$ , com o valor de  $N$  e  $M$  inseridos pelo usuário, e imprime a matriz lida. Como é uma matriz real, poderia ser utilizado ponto flutuante de precisão simples ou dupla, requeirindo um tamanho de 4 ou 8 bytes por campo da matriz, respectivamente. Será utilizado *double* para calcular com um tamanho diferente de *word*, que requer 4 bytes assim como ponto flutuante de precisão simples.

---

#### Programa 5.4 – Matriz Real Dinâmica

---

```
.data
Ent1: .asciiz "    Insira o valor de Mat["
Ent2: .asciiz "]"
Ent3: .asciiz "]: "
N: .asciiz "    Insira o número de linhas da matriz: "
M: .asciiz "    Insira o número de colunas da matriz: "
```

```
.text
main: li $s0, 8 # Número de bytes por campo: 8 (double)
      jal tamanho # Determina a ordem N x M, aloca e
      retorna o endereço da matriz
      move $a0, $v0 # Endereço base da matriz
      jal leitura # leitura da matriz
      move $a0, $v0 # Endereço da matriz lida
      jal escrita # Impressão da matriz
      li $v0, 10 # Código para finalizar o programa
      syscall # Finaliza o programa
```

tamanho:

```
la $a0, N # Carrega o endereço da string
li $v0, 4 # Código de impressão de string
syscall # Imprime a string
li $v0, 5 # Código de leitura de inteiro
syscall # Faz a leitura de N
move $a1, $v0 # Move para $a1
la $a0, M # Carrega o endereço da string
li $v0, 4 # Código de impressão de string
syscall # Imprime a string
li $v0, 5 # Código de leitura de inteiro
syscall # Faz a leitura de M
move $a2, $v0 # Move para $a2
mul $a0, $a1, $a2 # Número de linhas * Número de
colunas
mul $a0, $a0, $s0 # * Número de bytes por campo
li $v0, 9 # Código para alocação dinâmica
syscall # Aloca o espaço necessário para a matriz
(endereço base em $v0)
jr $ra
```

indice:

```
mul $v0, $t0, $a2 # i * ncol
add $v0, $v0, $t1 # (i * ncol) + j
```

```
mul $v0, $v0, $s0 # [(i * ncol) + j] * bytes por campo
add $v0, $v0, $a3 # Soma o endereço base de mat
jr $ra # Retorna para o caller
```

leitura:

```
subi $sp, $sp, 4 # Espaço para 1 item na pilha
sw $ra, ($sp) # Salva o retorno para a main
move $a3, $a0 # aux = endereço base de mat
l: la $a0, Ent1 # Carrega o endereço da string
li $v0, 4 # Código de impressão de string
syscall # Imprime a string
move $a0, $t0 # Valor de i para impressão
li $v0, 1 # Código de impressão de inteiro
syscall # Imprime i
la $a0, Ent2 # Carrega o endereço da string
li $v0, 4 # Código de impressão de string
syscall # Imprime a string
move $a0, $t1 # Valor de j para impressão
li $v0, 1 # Código de impressão de inteiro
syscall # Imprime j
la $a0, Ent3 # Carrega o endereço da string
li $v0, 4 # Código de impressão de string
syscall # Imprime a string
li $v0, 7 # Código de leitura de double
syscall # Leitura do valor (retorna em $f0)
jal indice # Calcula o endereço de mat[i][j]
s.d $f0, ($v0) # mat[i][j] = $f0
addi $t1, $t1, 1 # j++
blt $t1, $a2, l # if(j < M) goto l
li $t1, 0 # j = 0
addi $t0, $t0, 1 # i++
blt $t0, $a1, l # if(i < N) goto l
li $t0, 0 # i = 0
lw $ra, ($sp) # Recupera o retorno para a main
addi $sp, $sp, 4 # Libera o espaço na pilha
```

```
move $v0, $a3 # Endereço base da matriz para retorno
jr $ra # Retorna para a main
```

escrita:

```
subi $sp, $sp, 4 # Espaço para 1 item na pilha
sw $ra, ($sp) # Salva o retorno para a main
move $a3, $a0 # aux = endereço base de mat
e: jal indice # Calcula o endereço de mat[i][j]
l.d $f12, ($v0) # Valor em mat[i][j]
li $v0, 3 # Código de impressão de double
syscall # Imprime mat[i][j]
la $a0, 32 # Código ASCII para espaço
li $v0, 11 # Código de impressão de caractere
syscall # Imprime o espaço
addi $t1, $t1, 1 # j++
blt $t1, $a2, e # if(j < M) goto e
la $a0, 10 # Código ASCII para newline ('\n')
syscall # Pula a linha
li $t1, 0 # j = 0
addi $t0, $t0, 1 # i++
blt $t0, $a1, e # if(i < N) goto e
li $t0, 0 # i = 0
lw $ra, ($sp) # Recupera o retorno para a main
addi $sp, $sp, 4 # Libera o espaço na pilha
move $v0, $a3 # Endereço base da matriz para retorno
jr $ra # Retorna para a main
```

---

Segue um exemplo de execução para o programa 5.4:

---

```
Insira o número de linhas da matriz: 3
Insira o número de colunas da matriz: 2
Insira o valor de Mat[0][0]: 1
Insira o valor de Mat[0][1]: 2
Insira o valor de Mat[1][0]: 3
Insira o valor de Mat[1][1]: 4
Insira o valor de Mat[2][0]: 5
```

---

```
    Insira o valor de Mat[2][1]: 6
1.0 2.0
3.0 4.0
5.0 6.0

-- program is finished running --
```

---

# Manipulação de Arquivos

Neste capítulo, será visto como gerenciar a entrada e saída de dados por meio de arquivos, o que inclui abertura, fechamento, leitura e escrita de arquivo. É um conteúdo simples que envolve mais a utilização de serviços do sistema do que lógica de programação.

O MARS e outros ambientes como o SPIM oferecem o mesmo protocolo de chamadas de sistema para a manipulação de arquivos, que pode ser conferido na tabela [2.2](#) contendo as principais chamadas de sistema. Aqui, vamos analisá-las com mais detalhes.

Antes de realizar qualquer acesso ou alteração em um arquivo, é necessário abri-lo. Considerando um arquivo de nome “dados.txt”, em C isso é feito da seguinte forma:

---

```
FILE *arq = fopen("dados.txt", "r");
if(!arq)
    printf("Arquivo não encontrado!\n");
```

---

Em MIPS, visto que a chamada de sistema para a abertura de arquivo possui o seguinte protocolo:

*Abrir arquivo*

\$v0: código 13

\$a0: endereço de string terminada em nulo (*asciiiz*) contendo o nome do arquivo

\$a1: *flags*

\$a2: *mode*

Valores para *flag*:

0: somente leitura

1: somente escrita, com criação do arquivo

9: somente escrita, com criação e anexo (*append*)

Valores para *mode*: ignorado

Retorno em \$v0: *file descriptor* ou valor negativo em erro

Pode-se simular o código em C da forma a seguir. Vale destacar que nesse exemplo, o arquivo “dados.txt” deve estar no mesmo diretório do arquivo *.jar* do MARS, e não na mesma pasta do arquivo *.asm* do código; caso contrário, ele não será encontrado. Porém, também é possível especificar outros diretórios-filho.

---

*.data*

Arquivo: *.asciiiz* "dados.txt"

```
Erro: .ascii "Arquivo não encontrado!\n"
.text
main:
    li $v0, 13 # Código de abertura de arquivo
    la $a0, Arquivo # Nome do arquivo
    li $a1, 0 # Somente leitura
    syscall # Tenta abrir o arquivo
    bgez $v0, fim # if(file_descriptor >= 0) goto fim
    la $a0, Erro # else erro: carrega o endereço da
    string
    li $v0, 4 # Código de impressão de string
    syscall # Imprime o erro
fim: li $v0, 10 # Código para finalizar o programa
    syscall # Finaliza o programa
```

---

Independente do que for executado com o arquivo, é recomendável que ele seja *fechado* após terminar de utilizá-lo. Considerando ainda o mesmo exemplo, em C isso seria feito com o comando *fclose(arq)*. Para reproduzir a operação em MIPS, vale um protocolo igualmente simples:

*Fechar arquivo*

\$v0: código 16

\$a0: *file descriptor*

Retorno em \$v0: nenhum

Pode-se observar que com exceção da abertura de arquivo, todas as outras operações necessitam do *file descriptor* retornado na abertura, que é um valor que identifica o arquivo. Nas seções a seguir, será visto respectivamente como realizar a leitura de um arquivo com exemplo para utilizar os dados lidos, e como salvar ou escrever dados em um arquivo.



## 6.1 Leitura

Assumindo que um arquivo já foi aberto e o seu *file descriptor* está armazenado em um registrador, pode-se realizar a sua leitura cujo protocolo é definido da forma a seguir:

### *Leitura de arquivo*

\$v0: código 14

\$a0: *file descriptor*

\$a1: endereço do *buffer* de entrada (*input buffer*)

\$a2: número máximo de caracteres para leitura

Retorno em \$v0:

Valores positivos: número de caracteres lidos

Zero: fim do arquivo (*EOF*)

Valores negativos: erro

Percebe-se que a leitura de arquivo possui parâmetros bastante semelhantes aos utilizados na leitura de string. O tamanho do *buffer* de entrada a ser utilizado irá depender do número máximo de caracteres para leitura.

Se o objetivo é ler uma grande quantidade de texto do arquivo de uma só vez e armazená-lo em memória, o *buffer* deve ter o tamanho correspondente. Mas muitas vezes a operação de leitura deve executar lendo um caractere de cada vez, da mesma forma que *fgetc()* em C, neste caso um *buffer* com 1 byte de tamanho pode ser suficiente. Em todo caso, é possível simular qualquer função para leitura de arquivo em MIPS.

Supondo que deve-se determinar o número de caracteres contidos em um arquivo, segue o trecho de código de uma

possível tentativa para solucionar o problema, considerando que o *file descriptor* está em \$s0:

---

```
.data
buffer: .space 1000
...
.text
...
move $a0, $s0
la $a1, buffer
li $a2, 1000
syscall
move $a0, $v0
li $v0, 1
syscall
...
```

---

Se é conhecido que o arquivo nunca terá mais que 1000 caracteres, de fato o código acima irá imprimir o número correto de caracteres contidos no arquivo. Porém, para toda quantidade acima de mil caracteres, o programa irá imprimir sempre o mesmo valor 1000, pois é o máximo que ele é capaz de ler.

Uma abordagem correta que funcionaria para qualquer caso deve executar com uma lógica semelhante ao código em C a seguir:

---

```
while((ch = fgetc(arq)) != EOF)
    n++;
```

---

Como descrito no protocolo, a chamada de sistema indica o fim de arquivo quando o valor retornado em \$v0 é zero. Para realizar a leitura de apenas um caractere a cada *loop* da

mesma forma que *fgetc*, deve-se carregar o valor 1 no parâmetro \$a2 e um *buffer* com pelo menos 1 espaço (além do byte nulo) em \$a1.

O programa 6.1 é um exemplo completo de como abrir um arquivo, realizar sua leitura contando os caracteres contidos e fechar o arquivo em seguida.

---

#### Programa 6.1 – Contagem de caracteres em arquivo

---

```
.data
buffer: .asciiz " "
Arquivo: .asciiz "dados.txt"
Erro: .asciiz "Arquivo não encontrado!\n"

.text
main:
    la $a0, Arquivo # Nome do arquivo
    li $a1, 0 # Somente leitura
    jal abertura # Retorna file descriptor no sucesso
    move $s0, $v0 # Salva o file descriptor em $s0
    move $a0, $s0 # Parâmetro file descriptor
    la $a1, buffer # Buffer de entrada
    li $a2, 1 # 1 caractere por leitura
    jal contagem # Retorna em $v0 o num. de carac.
    move $a0, $v0 # Move o resultado para impressão
    li $v0, 1 # Código de impressão de inteiro
    syscall # Imprime o resultado
    li $v0, 16 # Código para fechar o arquivo
    move $a0, $s0 # Parâmetro file descriptor
    syscall # Fecha o arquivo
    li $v0, 10 # Código para finalizar o programa
    syscall # Finaliza o programa
```

contagem:

```
li $v0, 14 # Código de leitura de arquivo
syscall # Faz a leitura de 1 caractere
addi $t0, $t0, 1 # n++
bnez $v0, contagem # if(ch != EOF) goto contagem
subi $t0, $t0, 1 # Desconsidera EOF
move $v0, $t0 # Move o resultado para retorno
jr $ra # Retorna para a main
```

abertura:

```
li $v0, 13 # Código de abertura de arquivo
syscall # Tenta abrir o arquivo
bgez $v0, a # if(file_descriptor >= 0) goto a
la $a0, Erro # else erro: carrega o endereço da string
li $v0, 4 # Código de impressão de string
syscall # Imprime o erro
li $v0, 10 # Código para finalizar o programa
syscall # Finaliza o programa
a: jr $ra # Retorna para a main
```

---

Manipular os valores retornados em \$v0 é relativamente simples. Uma tarefa menos trivial é utilizar o conteúdo armazenado no *buffer* pela leitura.

Supondo que o arquivo *dados.txt* possui valores numéricos separados por linha, uma maneira de carregá-los para a memória em C seria com a função *fgets()*, que pode realizar a leitura de uma linha inteira (até o caractere “\n”). Em MIPS, sabe-se que a função deveria ser implementada manualmente, fazendo a verificação do “fim de linha”. A diferença é que são detectados 2 caracteres para o final da linha: primeiro o *carriage return* (código 13 da tabela *ASCII*) e só então o caractere de *newline* (código 10).

Além disso, não existe uma função de conversão de “string para inteiro” em MIPS, então ela também deve ser implementada. Isso pode ser feito facilmente somando o valor decimal dos caracteres numéricos, que podem ser obtidos com a subtração do seu valor *ASCII* por 48. Porém, antes de somar a próxima “casa” é necessário multiplicar o número por 10.

Exemplo: para converter a string “123” para inteiro, primeiro o número 1 é carregado, e multiplicado por 10 para se somar com o número seguinte (2), resultando em 12. Antes de somar com o terceiro número (3), o valor 12 é multiplicado por 10 para computar 120, que somado a 3 resulta em 123.

Para exemplificar, o programa 6.2 faz a leitura de um arquivo contendo um número por linha, e imprime a soma dos valores lidos.

---

#### Programa 6.2 – Leitura e soma de valores em arquivo

---

```
.data
buffer: .asciiz " "
Arquivo: .asciiz "dados.txt"
Erro: .asciiz "Arquivo não encontrado!\n"
.text
main:
    la $a0, Arquivo # Nome do arquivo
    li $a1, 0 # Somente leitura
    jal abertura # Retorna file descriptor no sucesso
    move $s0, $v0 # Salva o file descriptor em $s0
    move $a0, $s0 # Parâmetro file descriptor
    la $a1, buffer # Buffer de entrada
    li $a2, 1 # 1 caractere por leitura
    jal leitura # Retorna a soma dos números do arquivo
    move $a0, $v0 # Move o resultado para impressão
```

```
li $v0, 1 # Código de impressão de inteiro
syscall # Imprime o resultado
li $v0, 16 # Código para fechar o arquivo
move $a0, $s0 # Parâmetro file descriptor
syscall # Fecha o arquivo
li $v0, 10 # Código para finalizar o programa
syscall # Finaliza o programa
```

leitura:

```
li $v0, 14 # Código de leitura de arquivo
syscall # Faz a leitura de 1 caractere
beqz $v0, f # if(EOF) termina a leitura
lb $t0, ($a1) # Carrega o caractere lido no buffer
beq $t0, 13, leitura # if(carriage return) ignora
beq $t0, 10, l # if(newline) goto l
subi $t0, $t0, 48 # char para decimal
mul $t1, $t1, 10 # Casa decimal para a esquerda
add $t1, $t1, $t0 # Soma a unidade lida
j leitura # Continua a leitura do número
l: add $t2, $t2, $t1 # Soma o número lido
li $t1, 0 # Zera o número
j leitura # Leitura do próximo número
f: add $v0, $t2, $t1 # Resultado da soma em $v0
jr $ra # Retorna para a main
```

abertura:

```
li $v0, 13 # Código de abertura de arquivo
syscall # Tenta abrir o arquivo
bgez $v0, a # if(file_descriptor >= 0) goto a
la $a0, Erro # else erro: carrega o endereço da string
li $v0, 4 # Código de impressão de string
syscall # Imprime o erro
li $v0, 10 # Código para finalizar o programa
syscall # Finaliza o programa
a: jr $ra # Retorna para a main
```

---

## 6.2 Escrita

Para escrever dados em um arquivo, da mesma forma que na leitura, é necessário o *file descriptor* do arquivo, mas é preciso lembrar que o arquivo deve ter sido aberto para escrita, com os valores 1 ou 9 na *flag* em \$a1, como descrito anteriormente. O protocolo para a escrita é definido a seguir:

### *Escrita em arquivo*

\$v0: código 15

\$a0: *file descriptor*

\$a1: endereço do *buffer* de saída (*output buffer*)

\$a2: número máximo de caracteres para escrita

Retorno em \$v0: número de caracteres escritos

Uma das diferenças na abertura do arquivo para escrita em relação à leitura, é que não é preciso fazer a verificação se o arquivo existe e a impressão de erro; pois se ele não existir, será criado.

Assim, não são necessárias muitas linhas para gravar uma simples mensagem de texto em um arquivo, como pode ser observado no código a seguir.

---

```
.data
Arquivo: .asciiz "dados.txt"
Msg: .asciiz "Apostila de MIPS"
.text
main:
```

```
la $a0, Arquivo # Nome do arquivo
li $a1, 1 # Somente escrita
li $v0, 13 # Código de abertura de arquivo
syscall # Abre o arquivo (se não existir, será
criado)
move $a0, $v0 # Parâmetro file descriptor
li $v0, 15 # Código de escrita em arquivo
la $a1, Msg # Buffer de saída
li $a2, 16 # 16 caracteres
syscall # Escreve a mensagem no arquivo
li $v0, 16 # Código para fechar o arquivo
syscall # Fecha o arquivo
li $v0, 10 # Código para finalizar o programa
syscall # Finaliza o programa
```

---

No caso de strings, basta carregar como *output buffer* o endereço de uma string definida com a diretiva *asciiiz*.

Para valores numéricos, deve-se realizar a conversão para string, o processo inverso do que foi feito para a leitura de arquivo. Um algoritmo muito usado é dividir o número por 10 até que o quociente seja zero, convertendo cada resto de divisão para caractere e armazenando-os na ordem inversa, resultando na string contendo o número original.

Exemplo: considerando o número 123 novamente, ele é dividido por 10 resultando em 12 com resto 3. O quociente 12 é dividido novamente por 10 resultando em 1 com resto 2; o número 2 deve ser inserido antes do resto anterior (3), formando a sequência “23”. O último resultado (1) é dividido novamente por 10 resultando em 0 com resto 1, que é colocado junto dos outros restos formando a string final “123”. Quando o resultado é zero, o algoritmo é interrompido.



Em MIPS, não há uma forma simples de armazenar os restos das divisões na ordem inversa sem saber previamente qual é o tamanho do número (quantos algarismos ele contém). Por isso, uma solução adequada utiliza *pilha* para salvar os restos. Quando o quociente da divisão for zero, os valores são desempilhados e colocados em uma string na ordem correta.

O programa 6.3 realiza a leitura de valores inteiros positivos do usuário e salva os números que são ímpares em um arquivo. Se um valor negativo ou zero for lido, a leitura é interrompida, o arquivo é fechado e o programa é finalizado.

O procedimento *intstring* demonstra como implementar o algoritmo de conversão de inteiro para string com pilha. Em *escreveImpar*, o valor lido é verificado com a instrução *andi* para determinar se ele é par ou ímpar, armazenando 0 em um registrador auxiliar caso ele seja par, e 1 caso seja ímpar, como foi descrito na seção 3.3.

Assim como na leitura de arquivo, a troca de linha possui 2 caracteres: *carriage return* (\r) e *newline* (\n). Para pular linha entre os números salvos, o rótulo *newline* é escrito no arquivo a cada inserção de valor válido pelo usuário.

---

#### Programa 6.3 – Escrita de valores ímpares em arquivo

---

```
.data
Buffer: .space 20
newline: .asciiz "\r\n"
Arquivo: .asciiz "dados.txt"
.text
main:
```

```
la $a0, Arquivo # Nome do arquivo
li $a1, 1 # Somente escrita
li $v0, 13 # Código de abertura de arquivo
syscall # Abre o arquivo (se não existir, será
criado)
move $s0, $v0 # Salva o file descriptor
jal escreveImpar # Le números do usuário e escreve
no arquivo apenas os ímpares
move $a0, $s0 # Parâmetro file descriptor
li $v0, 16 # Código para fechar o arquivo
syscall # Fecha o arquivo
li $v0, 10 # Código para finalizar o programa
syscall # Finaliza o programa
```

intstring:

```
div $a0, $a0, 10 # n = n / 10
mfhi $t0 # aux = resto
subi $sp, $sp, 4 # Espaço para 1 item na pilha
sw $t0, ($sp) # Empilha o resto da divisão
addi $v0, $v0, 1 # Número de caracteres++
bnez $a0, intstring # if(n != 0) goto intstring
i: lw $t0, ($sp) # Desempilha um resto de divisão
addi $sp, $sp, 4 # Libera espaço de 1 item na pilha
add $t0, $t0, 48 # Converte a unidade (0-9) para
caractere
sb $t0, ($a1) # Armazena no buffer de saída
addi $a1, $a1, 1 # Incrementa o endereço do buffer
addi $t1, $t1, 1 # i++
bne $t1, $v0, i # if(iterações != num. carac.) goto i
sb $zero, ($a1) # Armazena NULL no buffer de saída
jr $ra # Retorna para o caller
```

escreveImpar:

```
move $t2, $ra # aux = retorno para a main
e: li $v0, 5 # Código de leitura de inteiro
```

```
syscall # Faz a leitura de um inteiro N
blez $v0, s # if(N == 0) goto s
andi $t0, $v0, 1 # Armazena 0 se N for par, 1 se ímpar
beqz $t0, e # if(N par) goto e
move $a0, $v0 # Parâmetro N para intstring
la $a1, Buffer # Buffer de saída
li $v0, 0 # Zera o contador
li $t1, 0 # Zera o contador
jal intstring # Converte N para string em $a1
move $a0, $s0 # Parâmetro file descriptor
la $a1, Buffer # Buffer de saída
move $a2, $v0 # Número de caracteres para escrita
li $v0, 15 # Código de escrita em arquivo
syscall # Escreve o número no arquivo
la $a1, newline # Endereço da string "\r\n"
li $a2, 2 # 2 caracteres
li $v0, 15 # Código de escrita em arquivo
syscall # Pula linha no arquivo
j e # goto e
s: jr $t2 # Retorna para a main
```

---

---

# Referências

Kann, Charles W., "Introduction To MIPS Assembly Language Programming" (2015). Gettysburg College Open Educational Resources. Book 2. <http://cupola.gettysburg.edu/oer/2>.

Britton, R.: MIPS Assembly Language Programming. Prentice-Hall, Englewood Cliffs (2003).

UWM College of Engineering & Applied Science, Computer Science Course, Integer Multiplication and Division. Disponível em: <<http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/mult.html>>.

Programming Tutorials and Lecture Notes, MIPS Floating Point. Disponível em: <[https://chortle.ccsu.edu/AssemblyTutorial/Chapter-31/ass31\\_2.html](https://chortle.ccsu.edu/AssemblyTutorial/Chapter-31/ass31_2.html)>.

WikiBooks, MIPS Assembly/MIPS Details. Disponível em: <[https://en.wikibooks.org/wiki/MIPS\\_Assembly/MIPS\\_Details](https://en.wikibooks.org/wiki/MIPS_Assembly/MIPS_Details)>.

Missouri State University, Using MARS through its Integrated Development Environment (IDE). Disponível em: <<http://courses.missouristate.edu/KenVollmar/mars/Help/MarsHelpIDE.html>>.

Missouri State University, MARS Interactive Debugging Features. Disponível em: <<http://courses.missouristate.edu/KenVollmar/mars/Help/MarsHelpDebugging.html>>.

University of Illinois CS course, MIPS Architecture and Assembly Language Overview. Disponível em: <<http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm>>.

Howard Huang, MIPS floating-point arithmetic. Disponível em: <<http://howardhuang.us/teaching/cs232/10-MIPS-floating-point-arithmetic.pdf>>.

The University of Iowa, MIPS Instruction formats. Disponível em: <<http://homepage.cs.uiowa.edu/~ghosh/1-24-06.pdf>>.

The University of Texas at Dallas, The Program Counter. Disponível em: <<https://www.utdallas.edu/~dodge/EE2310/lec13.pdf>>.

University of Pittsburgh, Subroutines/Functions in MIPS. Disponível em: <<http://people.cs.pitt.edu/~xujie/cs447/Mips/sub.html>>.

EECS Instructional & Electronics Support, Machine Structures. Disponível em: <<http://www-inst.eecs.berkeley.edu/cs61c/fa07/proj/Project2Background.html>>.