



Departamento de Electrónica, Telecomunicações e Informática

Arquitectura de Computadores I

2º Ano, 1º semestre

Introdução à Arquitectura MIPS

Texto de apoio às aulas práticas da disciplina de Arquitectura de Computadores I

Índice

	Pág.
Descrição da Arquitectura.....	4
Registos do <i>CPU</i>	4
Sintaxe da Linguagem Assembly	6
Modos de Endereçamento da memória externa.....	8
Chamadas ao Sistema (<i>System Calls</i>)	8
Passagem dos parâmetros da linha de comando	9
Coprocessador de Vírgula Flutuante (CP1)	9
Registos do Coprocessador CP0	11
Input e Output.....	12
Reportório de Instruções	14
Instruções de Transferência Memória-Registo (<i>Load</i>).....	14
Instruções de Transferência Registo-Memória (<i>Store</i>).....	15
Instruções de Transferência Registo-Registo (<i>Move</i>).....	16
Instruções de Manipulação de Constantes (<i>Load Immediate</i>).....	17
Instruções de Cálculo sobre Inteiros: Operações Aritméticas	17
Instruções de Cálculo sobre Inteiros: Operações Lógicas Bit-a-Bit (<i>Bitwise</i>).....	18
Instruções de Cálculo sobre Inteiros: Operações de Deslocamento (<i>Shift</i>) e de Rotação (<i>Rotate</i>).....	19
Instruções de Comparação	19
Instruções de Salto Relativo (<i>Branch</i>) e Salto Absoluto (<i>Jump</i>).....	20
Instruções de Cálculo em Vírgula Flutuante	22
Instruções para Manipulação de Excepções e <i>Traps</i>	23
Organização da Memória	23

Descrição da Arquitectura

Um processador MIPS é constituído por uma unidade de processamento de inteiros (o *CPU*) e vários coprocessadores que realizam tarefas subsidiárias ou operam sobre outros tipos de dados como sejam números em vírgula flutuante (Figura 1). O coprocessador de controlo de sistema (CP0) é responsável pela gestão de *traps*, excepções e pelo sistema de gestão da memória virtual. O coprocessador de vírgula flutuante (CP1), por sua vez, realiza operações aritméticas sobre números reais representados em vírgula flutuante no formato IEEE-754. O MARS ¹simula o funcionamento do CPU, do CP1 e das *traps* e excepções do CP0.

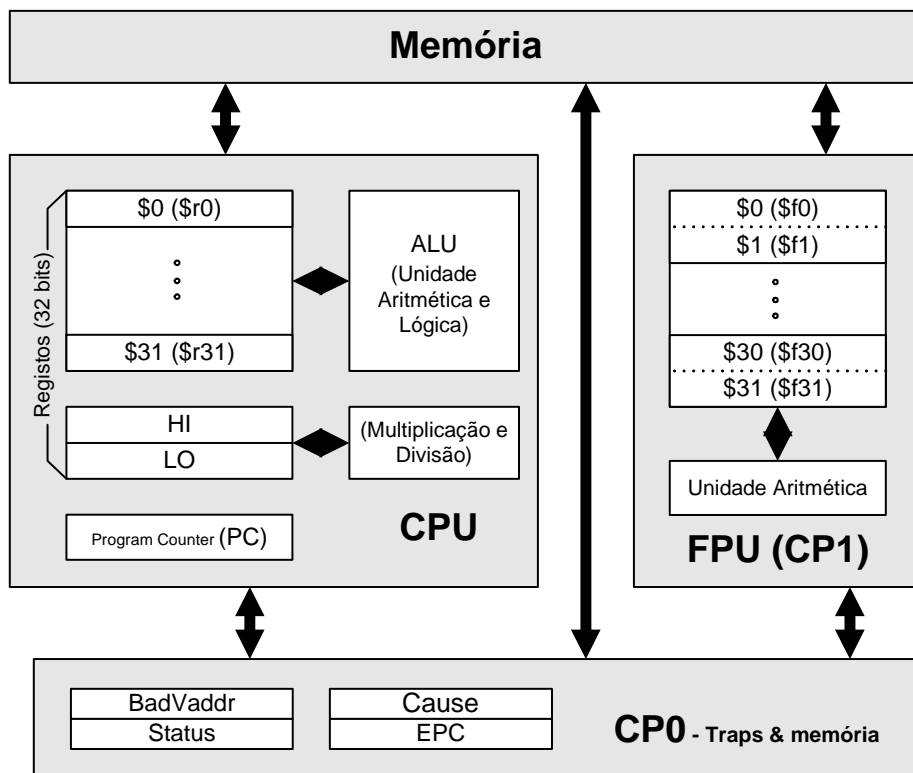


Figura 1 - Arquitectura simplificada do MIPS

Registos do CPU

A unidade de processamento central (*CPU*) do MIPS possui um *program counter* de 32 bits (**PC**), dois registos de 32 bits para armazenar os resultados de multiplicações e divisões inteiras (**HI** e **LO**) e 32 registos de uso geral de 32 bits, numerados de 0 a 31 (**\$0** a **\$31**). Dois destes registos, embora de uso geral, apresentam particularidades específicas: o registo **\$0** contém sempre o valor 0 (*hardwired*) e o registo **\$31** (*link register*) é usado pelas instruções de *Jump and Link* para armazenar o endereço de retorno na evocação de subrotinas. O MARS simula a existência e uso de todos estes registos.

¹ MARS - MIPS Assembler and Runtime Simulator (<http://courses.missouristate.edu/KenVollmar/MARS/>)

Relativamente à utilização, por parte dos programas *Assembly*, dos registos de uso geral, o MIPS estabeleceu um conjunto de regras ou convenções que devem ser seguidas pelos programadores. Embora estas convenções não correspondam a imposições do *hardware*, a sua adopção é fortemente recomendada, de forma a garantir que o código gerado funcione correctamente em outras plataformas. Com vista a facilitar o seguimento da convenção estabelecida pela MIPS, a cada um dos registos foi atribuído um nome lógico que evidencia o seu tipo de uso. Na Tabela I são identificados, para cada registo de uso geral, a correspondente designação lógica e o tipo de uso que lhe está destinado pela convenção.

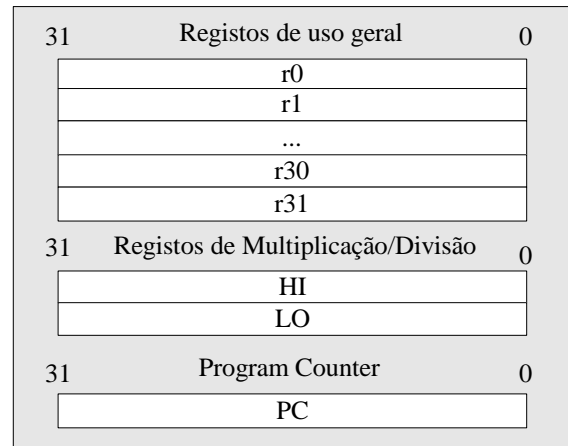


Figura 2 - Os registos do CPU

Nome Lógico	Nome Real	Uso
\$zero	\$0	Constante 0
\$at	\$1	Reservado pelo assembler
\$v0..\$v1	\$2..\$3	Cálculo de expressões e valor de retorno das funções.
\$a0..\$a3	\$4..\$7	Primeiros 4 parâmetros das funções
\$t0..\$t7	\$8..\$15	Geral (pode não ser preservado pelas funções)
\$s0..\$s7	\$16..\$23	Geral (deve ser preservado pelas funções)
\$t8..\$t9	\$24..\$25	Geral (pode não ser preservado pelas funções)
\$k0..\$k1	\$26..\$27	Reservado pelo kernel do S.O.
\$gp	\$28	Ponteiro para área global (<i>Global Pointer</i>)
\$sp	\$29	<i>Stack Pointer</i>
\$fp	\$30	<i>Frame Pointer</i>
\$ra	\$31	Endereço de retornos das funções (<i>Return Address</i>)

Tabela I - Registos do MIPS e convenção de uso

Os registos **\$at** (\$1) e **\$k0-\$k1** (\$26-\$27) estão reservados para uso pelo assembler e sistema operativo.

Os registos **\$a0-\$a3** (\$4-\$7) são usados para passar às subrotinas² os seus quatro primeiros parâmetros. Se for necessário passar mais do que quatro parâmetros os remanescentes deverão ser passados na *stack*.

Os registos **\$v0-\$v1** (\$2-\$3) são usados para armazenar temporariamente o resultado de expressões aritméticas ou lógicas ou para o retorno de valores das funções.

² Alguns compiladores, por forma a generalizarem a geração de código, optam por passar todos os parâmetros na *stack*.

Os registos **\$t0-\$t9** (**\$8-\$15**, **\$24**, **\$25**) são usados para armazenar informação temporária que não precise de ser preservada entre chamadas a subrotinas; estes registos, cujo conteúdo pode ser livremente alterado pelas subrotinas, são designados por registos “*caller-saved*”, i.e., cabe ao programa em execução preservar os seus conteúdos no caso de necessitar do seu valor após uma chamada a uma subrotina.

Os registos **\$s0-\$s7** (**\$16-\$23**) são usados para armazenar valores com um tempo de vida longo que precisam de ser preservados entre chamadas a subrotinas; são designados por registos “*callee-saved*”, i.e., cabe à subrotina em execução preservar os seus valores no caso que necessitar de recorrer à sua utilização temporária.

O registo **\$sp** (**\$29**), ou *stack-pointer*, aponta para a última posição ocupada na *stack*. Embora o MIPS não tenha instruções próprias para a manipulação de *stacks*, convencionou-se que aquela cresce no sentido decrescente dos endereços e que o conteúdo de **\$sp** é um ponteiro para a última posição ocupada da mesma.

O registo **\$fp** (**\$30**), ou *frame-pointer*, é usado por alguns compiladores para acesso à informação na *stack*.

O registo **\$gp** (**\$28**) é um apontador global (*global pointer*) que aponta para o meio de um bloco de memória de 64 Kbytes definido na memória *heap* e usado para definir constantes e variáveis globais.

O registo **\$ra** (**\$31**), finalmente, é usado pela instrução **jal** para armazenar o endereço de retorno das subrotinas.

Sintaxe da Linguagem Assembly

Um programa em linguagem *Assembly* é um texto a 4 colunas, em que a primeira representa de uma forma simbólica endereços de memória (*labels*), a segunda instruções, a terceira operandos das instruções e a última comentários. Não existe nenhuma imposição quanto ao posicionamento das colunas no texto a não ser obviamente que a sua ordem deve ser respeitada, embora isso seja feito frequentemente para melhorar a legibilidade do texto. Não é possível ter mais do que uma instrução por linha.

Exemplo:

```
Var_1:    .data
          .word 1
          .text
          .globl main
main:     li      $v0, 4      #Comentário
          lw      $a0, Var_1
          ...
```

Os endereços são representados por identificadores -- etiquetas (*labels*) -- e devem ser terminados pelo carácter “:”. É considerado como comentário tudo o que estiver entre o carácter “#” e o fim da linha, sendo consequentemente ignorado pelo assembler.

Os identificadores são sequências de caracteres alfanuméricos, barras horizontais, “_”, e pontos, “.”, que não comecem por um carácter numérico. As mnemónicas (*opcodes*) das instruções são palavras reservadas, não podendo consequentemente ser usadas como identificadores. Um programa *assembly* pode estar distribuído por mais que um ficheiro; os identificadores podem ser locais, sendo assim apenas válidos no contexto desse ficheiro, ou globais, sendo assim válidos em todos os ficheiros que constituem o programa.

As *strings* são sequências de caracteres delimitadas por aspas (“). A utilização de caracteres especiais segue a convenção da linguagem de programação C. Os caracteres válidos são:

“\t” => tab; “\n” => new line; “\” => quote

O campo instrução corresponde a uma instrução (nativa ou virtual) do MIPS ou a uma directiva. As directivas são mecanismos que permitem controlar a forma como a assemblagem (processo de geração de código máquina) é levada a cabo, não resultando assim em código executável. São constituídas por um identificador (cujo primeiro carácter é sempre o símbolo “.”) e em alguns casos por um ou mais parâmetros. O MARS suporta o seguinte subconjunto das directivas definidas para o assembler de MIPS:

- Para controlo dos segmentos:

.data <address>

Os valores definidos nas linhas subsequentes devem ser colocados no segmento de dados do utilizador, opcionalmente a partir do endereço <address>.

.text <address>

Os valores definidos nas linhas subsequentes devem ser colocados no segmento de texto do utilizador, opcionalmente a partir do endereço <address>. Todos os valores devem medir 32 bits, ou seja, serem instruções ou palavras (words).

.kdata <address>

Os valores definidos nas linhas subsequentes devem ser colocados no segmento de dados do kernel, opcionalmente a partir do endereço <address>.

.ktext <address>

Os valores definidos nas linhas subsequentes devem ser colocados no segmento de texto do kernel, opcionalmente a partir do endereço <address>.

- Para criação de constantes e variáveis em memória:

.ascii str

armazena uma *string* em memória sem lhe acrescentar o terminador NUL.

.asciiz str

armazena uma *string* em memória acrescentando-lhe o terminador NUL.

.byte b₁, ..., b_n

armazena as grandezas de 8 bits **b₁, ..., b_n** em sucessivos bytes de memória.

.half h₁, ..., h_n

armazena as grandezas de 16 bits **h₁, ..., h_n** em sucessivas meias-palavras de memória.

.word w₁, ..., w_n

armazena as grandezas de 32 bits **w₁, ..., w_n** em sucessivas palavras de memória.

.float f₁, ..., f_n

armazena os números em vírgula flutuante com precisão simples (32 bits) **f₁, ..., f_n** em posições de memória sucessivas.

.double d₁, ..., d_n

armazena os números em vírgula flutuante com precisão dupla (64 bits) **d₁, ..., d_n** em posições de memória sucessivas.

.space n

reserva **n** bytes (o MARS só deixa usar esta directiva no segmento **.data**).

Para controlo do alinhamento:

.align n

alinha o próximo item num endereço múltiplo de 2ⁿ. Por exemplo **.align 2** seguido de **.word xpto** garante que a palavra **xpto** é armazenada num endereço múltiplo de 4.

.align 0

desliga o alinhamento automático das directivas **.half**, **.word**, **.float**, e **.double** até à próxima directiva **.data** ou **.kdata**.

Para referências externas:

.globl sym

declara que o símbolo **sym** é global e pode ser referenciado a partir de outros ficheiros.

.extern sym size

declara que o item associado a **sym** ocupa **size** bytes e é um símbolo global. Esta directiva permite ao assembler armazenar o item numa porção do segmento de dados que seja eficientemente acedido através do registo \$gp.

Modos de Endereçamento da memória externa

Os processadores MIPS seguem uma arquitectura do tipo “load/store”, o que significa que apenas as instruções de *load* e de *store* podem aceder à memória. As instruções de cálculo operam apenas sobre o conteúdo de registos. Além disso o MIPS possui apenas um modo de endereçamento da memória, representado por **imm(reg)**, em que o endereço de memória é determinado pela soma do conteúdo do registo **reg** (qualquer dos registos gerais do CPU) com a constante **imm** (limitada a uma quantidade de 16 bits em complemento para dois).

Chamadas ao Sistema (System Calls)

O MARS disponibiliza um conjunto de funções típicas de um sistema operativo (ver Tabela II). A sua utilização é feita através da instrução de chamada ao sistema **syscall**. Para invocar uma determinada função o programa deve carregar o código respectivo no registo **\$v0** e os argumentos nos registos **\$a0** a **\$a1** (ou **\$f12** para valores em vírgula flutuante). O valor de retorno (caso exista), é devolvido no registo **\$v0** (ou **\$f0** no caso de ser em vírgula flutuante).

Exemplo de utilização da *System Call* “print_string”:

```
        .data
str:     .asciiz "Uma string!"
        .text
        .globl main
main:    li      $v0, 4
        la      $a0, str
        syscall          # print_string ("Uma string");
        ...
```

As funções “*print*” imprimem um valor (**int** – inteiro, **float** – real com precisão simples, **double** – real com precisão dupla, **string** – sequência de caracteres terminada por um NUL e **char** – carácter) na janela *Console*. As funções “*print_bin*” e “*print_hex*” imprimem um valor numérico, respectivamente, em binário e hexadecimal.

As funções “*read*” lêem valores a partir do teclado. O valor lido deve ser terminado pela tecla “Enter”. Nas funções de leitura de valores numéricos, são ignorados todos os caracteres lidos a partir (e incluindo) o

primeiro carácter não numérico. A função “*read_string*” lê **n-1** caracteres para o *buffer* de entrada e termina a *string* com um carácter NUL. A função termina quando o número de caracteres lido for igual a **n-1** ou quando receber um *new line* (tecla “*Enter*”).

A função “*sbrk*” devolve um ponteiro para um bloco de memória contendo *n* bytes adicionais.

A função “*exit*” termina a execução de um programa.

Protótipo equivalente em C	\$v0	Parâmetros de entrada	Retorno
void print_int(int value)	1	\$a0 = value	
void print_float(float value)	2	\$f12 = value	
void print_double(double value)	3	\$f12 = value	
void print_string(char *str)	4	\$a0 = str	
int read_int(void)	5		\$v0
float read_float(void)	6		\$f0
double read_double(void)	7		\$f0
void read_string(char *buf, int length)	8	\$a0 = buf, \$a1 = length	
void *Sbrk(int amount)	9	\$a0 = amount	\$v0
void Exit(void)	10		
void print_char(int character)	11	\$a0 = character	
char read_char(void)	12		\$v0
void print_hex(int value)	34	\$a0 = value	
void print_bin(int value)	35	\$a0 = value	

Tabela II - *System calls* do MARS

Passagem dos parâmetros da linha de comando

O MARS permite que o programa Assembly recupere da *stack* os parâmetros que lhe sejam passados a partir da linha de comando. Para isso, os primeiros 32 bytes do segmento de texto são automaticamente preenchidos pelo MARS com uma porção de código que inicializa os ponteiros para os locais próprios. O valor de **argc**, variável que em linguagem C contém o número de parâmetros da linha de comando, é passado ao programa através do registo **\$a0**. Por sua vez, **\$a1** contém o valor de **argv**, variável que contém um ponteiro para uma lista de ponteiros. Cada um dos ponteiros dessa lista (num total de **argc** ponteiros) aponta para uma string que contém o conteúdo de um dos parâmetros. Finalmente, **\$a2** contém o ponteiro para a lista de variáveis do *environment*. Note que, ao contrário do que acontece em C, a lista de parâmetros não inclui o nome do programa. Consequentemente, **argv[0]** aponta directamente para o primeiro parâmetro da lista.

Coprocessador de Vírgula Flutuante (CP1)

O MIPS possui um coprocessador de vírgula flutuante (o coprocessador CP1) que manipula números reais representados em vírgula flutuante com precisão simples (32 bits) e precisão dupla (64 bits) de acordo com a norma IEEE-754. Os operandos para as operações de cálculo disponibilizadas pelo coprocessador pertencem a um conjunto específico de registos designados por **\$f0-\$f31**. Estes registos desempenham um papel duplo.

Por um lado podem ser vistos como um conjunto de 32 registos independentes com capacidade para armazenar quantidades em vírgula flutuante de precisão simples.

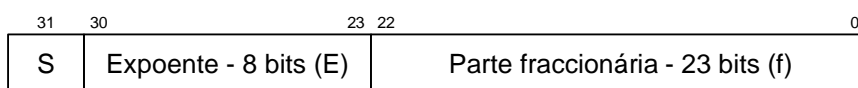
Por outro lado, cada registo par pode ser agrupado ao seguinte (ímpar) para formar um registo com capacidade para armazenar quantidades em vírgula flutuante com precisão dupla. De forma a simplificar a sua utilização, as operações de cálculo, sejam de precisão simples ou dupla, usam apenas os registos de índice par.

Este coprocessador possui ainda dois registos de controlo de 32 bits que são usados para controlo do modo de arredondamento, tratamento de excepções, e para salvaguarda do estado. Uma *flag* interna ao coprocessador, à frente designada por *fp_flag* ou *cl_flag*, serve de suporte às instruções de comparação e salto condicional relativas a entidades em vírgula flutuante.

Apenas a título de resumo, a norma IEEE 754 estabelece as seguintes características para os formatos de representação numérica em vírgula flutuante:

Precisão simples

Os números reais representados em vírgula flutuante com precisão simples são representados por palavras de 32 bits organizados de acordo com a figura seguinte.



O valor do número em vírgula flutuante é dado pela expressão:

$$F = (-1)^S 1.f 2^{E-127}$$

sendo S é o valor do sinal representado pelo bit b_{31} ('0' representa um número positivo; '1' representa um número negativo), f (bits $b_{22}...b_0$) o valor binário da parte fraccionária da quantidade representada, no qual se pressupõe a existência de um *hidden* bit de valor 1 à esquerda da vírgula (representação normalizada), e E (bits $b_{30}...b_{23}$) o valor do expoente codificado em excesso de 127. Existem um conjunto de valores particulares que representam excepções à regra e que se encontram sumariados na Tabela III.

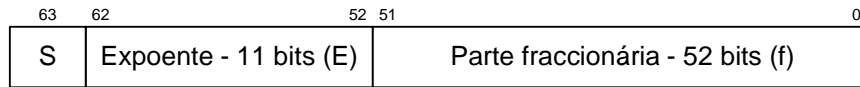
E	f	Significado
0	0	Representa o valor numérico 0 (zero)
0	$\neq 0$	Representação desnormalizada de acordo com a equação $F = (-1)^S 0.f 2^{-126}$
255	0	Representa $\pm\infty$
255	$\neq 0$	Representa NAN (Not A Number)

Tabela III - Excepções no formato IEEE-754

Refira-se em particular que, na representação não normalizada, o *hidden* bit não existe e o expoente é fixo e igual a -126. Esta representação permite diminuir o intervalo entre a menor representação possível e zero, minimizando, consequentemente, o potencial de *underflow*.

Precisão dupla

Os números reais representados em vírgula flutuante com precisão dupla são representados por palavras de 64 bits organizados de acordo com a figura seguinte.



O valor do número em vírgula flutuante é dado pela expressão:

$$F = (-1)^S 1.f 2^{E-1023}$$

sendo S o valor do sinal representado pelo bit b_{63} ('0' representa um número positivo; '1' representa um número negativo), f (bits $b_{51} \dots b_0$) o valor binário da mantissa no qual se pressupõe a existência de um *hidden* bit de valor 1 à esquerda da vírgula, e E (bits $b_{62} \dots b_{52}$) o valor do expoente codificado em excesso de 1023.

A Tabela IV apresenta um resumo das principais características destes formatos em precisão simples e dupla respectivamente.

	Precisão simples	Precisão dupla
Comprimento de palavra	32 bits	64 bits
Mantissa (<i>Hidden</i> bit + parte fraccionária)	1 + 23 bits	1 + 52 bits
Expoente	8 bits	11 bits
<i>Offset</i> do expoente	127	1023
Gama de representação aproximada	$2^{128} \approx 3.8 \times 10^{38}$	$2^{1024} \approx 9 \times 10^{307}$
Menor número normalizado	$2^{-126} \approx 10^{-38}$	$2^{-1022} \approx 10^{-308}$
Precisão aproximada	$2^{-23} \approx 10^{-7}$	$2^{-52} \approx 10^{-15}$

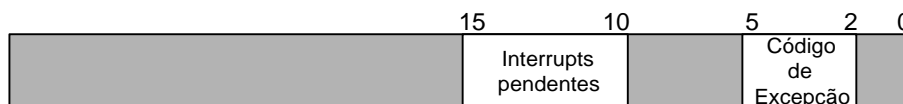
Tabela IV - Mapa comparativo do formato IEEE 754 em precisão simples e dupla

Registos do Coprocessador CP0

O coprocessador CP0 contém registos necessários para a gestão das exceções, do sistema de memória virtual e das *caches* de dados e instruções. Contém ainda um registo com a indicação da versão do processador. O MARS simula apenas 4 destes registos:

BadVAddr contém o endereço de memória onde ocorreu a exceção de memória

Cause indica o tipo de exceção ocorrida. Apenas os bits 2 a 5 e 10 a 15 têm significado, conforme se pode observar na figura seguinte.

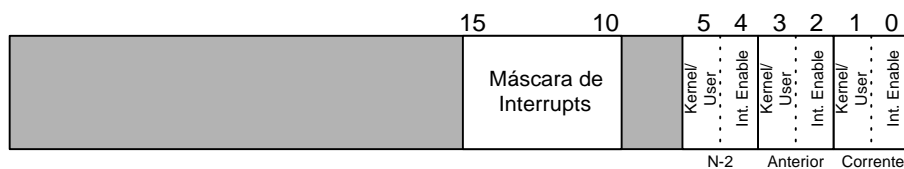


Os cinco bits que identificam os *interrupts* pendentes ($b_{15} \dots b_{10}$) correspondem aos cinco níveis de interrupção possíveis. Um bit com o valor lógico '1' neste campo indica a ocorrência de um *interrupt* a esse nível que ainda não foi servido. O código de exceção, correspondente aos bits $b_5 \dots b_2$, contém um código de identificação de interrupção de acordo com a Tabela V.

Número	Nome	Descrição
0	INT	<i>Interrupt</i> externo
4	ADDRL	Erro de endereço (em instrução de <i>load</i> ou na operação de <i>fetch</i>)
5	ADDRS	Erro de endereço (em instruções de <i>store</i>)
6	IBUS	Erro de <i>Bus</i> em operação de <i>fetch</i> de instrução
7	DBUS	Erro de <i>Bus</i> em operação de <i>Load</i> ou <i>Store</i> de dados
8	SYSCALL	Excepção de chamada ao sistema
9	BKPT	<i>Break Point</i>
10	RI	Excepção de instrução reservada
12	OVF	Excepção provocada por <i>overflow</i> aritmético

Tabela V - Códigos de excepção

Status contém um conjunto de bits de *status* e controlo dos quais são implementados pelo MARS os indicados na figura seguinte.



A máscara de *interrupts* ($b_{15}..b_{10}$) contém um bit para cada um dos cinco níveis de interrupção. Quando um desses bits toma o valor lógico ‘1’, o nível de interrupção correspondente encontra-se activo. Os seis bits menos significativos deste registo implementam uma *stack* de três níveis para os bits de identificação do *kernel/user* e *interrupt enable*. Um valor lógico ‘0’ no bit *kernel/user* indica que o programa estava a correr parte do *kernel* no momento em que ocorreu a interrupção. Um valor lógico ‘1’ indica que a execução pertencia ao utilizador.

EPC guarda o endereço da instrução que causou a excepção.

Input e Output

Para além de simular as operações básicas do *CPU* do MIPS e do respectivo sistema operativo, o MARS também simula operações de entrada/saída (*Input/Output* ou *I/O*) relativamente a uma ligação a um terminal virtual mapeado em memória. Quando um programa Assembly se encontra em execução, o simulador “liga” o seu próprio terminal virtual ao processador simulado. O programa pode assim ler caracteres directamente do teclado à medida que estes vão sendo digitados pelo operador. Da mesma forma, é possível executar instruções para escrever caracteres directamente no ecrã da consola virtual. A única excepção a esta regra diz respeito à associação de teclas [Control+C]. O resultado não é passado para o programa mas, alternativamente, este é colocado em modo “*break*” devolvendo o comando ao MARS e actualizando o conteúdo das várias janelas. Para poder usar este modo especial de *I/O*, é necessário que o campo “*Mapped I/O*” da janela de diálogo iniciada pelo comando “*Simulator | Settings*” esteja activado.

O dispositivo terminal correspondente à consola virtual consiste em duas unidades independentes: um receptor e um emissor. O receptor lê caracteres a partir do teclado à medida que estes vão sendo digitados. O emissor, por sua vez, escreve caracteres no ecrã da consola virtual. As duas unidades são completamente

independentes. Isso significa, por exemplo, que não é executado o eco automático dos caracteres digitados no teclado. Pelo contrário, o processador deve ler um carácter de entrada a partir do receptor e retransmiti-lo para o emissor para obter o seu eco visual.

O programa acede ao terminal virtual usando quatro registos especiais mapeados em memória (Figura 3). Mapeados em memória significa que cada um destes registos surge associado a um endereço de memória especial e dedicado a esse fim. O “Registo de Controlo de Recepção” localiza-se no endereço **0xffff0000**. Apenas dois dos seus 32 bits são usados de facto. O bit **b₀**, designado por “*Ready*”, indica, quando o seu valor é ‘1’, que um novo carácter foi recebido a partir do teclado, e que este ainda não foi lido do “Registo de Dados do Receptor”. Este bit é alterado apenas pelo simulador, não podendo consequentemente, ser alterado pelo programa; muda automaticamente de ‘0’ para ‘1’ quando um carácter é recebido do teclado, e regressa automaticamente a ‘0’ quando o carácter correspondente é lido do “Registo de Dados do Receptor”.

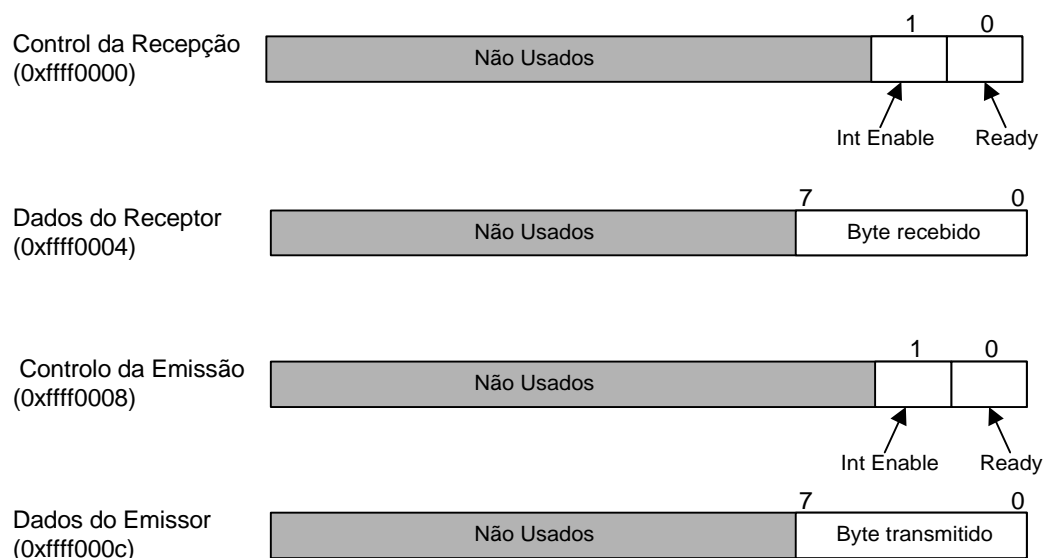


Figura 3 - Os registos para acesso I/O ao terminal virtual

O bit **b₁** do “Registo de Controlo de Recepção” é o “*Interrupt Enable*”. Este bit pode ser lido ou escrito pelo programa. Inicialmente, o seu valor é colocado a ‘0’. Se o programa o forçar a ‘1’, é gerada uma interrupção de nível zero sempre que o “*Ready*” bit passar de ‘0’ para ‘1’. Para que esta interrupção possa ser recebida pelo processador, os *interrupts* devem estar activos através do registo de **Status** do coprocessador C0.

O segundo registo do dispositivo terminal é o “Registo de Dados do Receptor”, o qual existe no endereço **0xffff0004**. Os oito bits menos significativos deste registo contêm o último carácter digitado no teclado. Todos os restantes 24 bits têm o valor ‘0’. Este registo é do tipo “*read-only*”. Qualquer tentativa para escrever no mesmo é ignorada pelo simulador. Ler o conteúdo deste registo provoca o “*reset*” do bit “*Ready*” do registo anterior.

O terceiro registo do dispositivo terminal é o “Registo de Controlo de Emissão” localizado no endereço **0xffff0008**. Apenas os dois bits menos significativos têm significado, e este comportam-se de forma muito semelhante aos bits do “Registo de Controlo de Recepção”. O bit **b₀**, quando ‘1’, indica que o emissor

está pronto o receber mais um carácter. Se o seu valor for ‘0’, então o emissor encontra-se ainda ocupado a transmitir o carácter anterior. O bit **b₁**, por sua vez, é o “*Interrupt Enable*”. Um ‘1’ neste bit desencadeia uma interrupção de nível um sempre que o “*Ready*” bit passar de ‘0’ para ‘1’

O último registo do dispositivo terminal é o “Registo de Dados do Emissor”, o qual existe no endereço **0xffff000c**. Quando escrito pelo programa, os seus oito bits menos significativos são interpretados como sendo um carácter ASCII, o qual é enviado e afixado no écran da consola. O registo deverá ser escrito apenas quando o bit “*Ready*” do “Registo de Controlo de Emissão” se encontrar no seu estado activo (‘1’). Qualquer operação de escrita efectuada sem ser nessas condições não terá qualquer efeito prático, embora o programa em execução não tenha conhecimento do facto.

Reportório de Instruções

O conjunto de instruções disponibilizadas pelo simulador está enriquecido em relação ao conjunto de instruções da máquina real, não apenas pela existência do maior número de modos de endereçamento mas também pela existência de novas instruções. Muitas destas novas instruções desdobram-se na máquina real em duas ou mais instruções; no entanto, algumas há que representam apenas sintaxes mais apropriadas para operações que na máquina real podem ser realizadas por uma única instrução. Considere-se, a título de exemplo, a instrução virtual **move \$s0,\$s1**, e as instruções nativas **add \$s0,\$s1,\$0** ou **or \$s0,\$s1,\$0**: embora as três instruções façam rigorosamente o mesmo a instrução virtual é a que retrata de uma forma mais adequada uma operação do tipo “**a** toma o valor de **b**”. O uso de instruções virtuais pode, por isso, contribuir para tornar o programa *assembly* mais curto e mais legível.

No texto que se segue faz-se uma apresentação pormenorizada do reportório de instruções dividido por categorias funcionais. As instruções apresentadas podem ser completamente nativas, completamente virtuais ou possuírem um código de operação (*opcode*) nativo mas modos de endereçamento enriquecidos. Por forma a distinguir facilmente as instruções pertencentes a cada caso optou-se por escrever as pertencentes ao primeiro em letra normal, as pertencentes ao segundo em **negrito** (*bold*) e as pertencentes ao último em letra normal com as partes não nativas em **negrito**. Em todas as instruções que caem nesta última categoria apenas um dos operandos é virtual e apresenta uma das formas seguintes:

Imm representa um valor constante com 16 ou 32 bits; o primeiro caso corresponde à instrução nativa

Src representa um registo do *CPU* ou uma constante (imediato); o primeiro caso corresponde à instrução nativa

Imm(Reg) **Reg** é um dos registos internos do CPU e **Imm** é uma quantidade de 16 bits em complemento para dois.

Instruções de Transferência Memória-Registo (*Load*)

Incluem-se nesta categoria instruções que realizam uma operação algorítmica do tipo “**var** = **value**” (**var** toma o valor de **value**), em que **var** representa um registo e **value** um valor armazenado em memória (na descrição da instrução, *addr* representa o conteúdo do registo *Reg* adicionado à Constante *Imm*).

la Rdst, address	Load Address	Rdst = address
-------------------------	--------------	----------------

Calcula o endereço addr e coloca o resultado no registo geral Rdst.

lb Rdst, Imm(Reg)	Load Byte	Rdst = *(byte)addr
--------------------------	-----------	--------------------

Lê da memória o byte cujo endereço é addr, estende-o, com sinal, para 32 bits e coloca o resultado no registo Rdst.

lbu Rdst, Imm(Reg)	Load Byte Unsigned	Rdst = *(unsigned byte)addr
---------------------------	--------------------	-----------------------------

Lê da memória o byte cujo endereço é addr, estende-o, com zeros, para 32 bits e coloca o resultado no registo Rdst.

lw Rdst, Imm(Reg)	Load Word	Rdst = *(word)addr
--------------------------	-----------	--------------------

Lê da memória a palavra cujo endereço é addr e coloca-a no registo Rdst.

As instruções seguintes permitem transferir informação da memória para registos dos coprocessadores.

lwc³ CReg, Imm(Reg)	Load Word Coprocessor z	CReg = *(word)addr
---------------------------------------	-------------------------	--------------------

Carrega o registo CReg do coprocessador Cz com o valor de 32 bits armazenado no endereço addr.

l.d FPR _{Reg} , Imm(Reg)	Load Double	FPR _{Reg} = *(double)addr
--	-------------	------------------------------------

Carrega o registo FPR_{Reg} do coprocessador de vírgula flutuante com o valor real duplo armazenado no endereço addr.

l.s FPR _{Reg} , Imm(Reg)	Load Single	FPR _{Reg} = *(single)addr
--	-------------	------------------------------------

Carrega o registo FPR_{Reg} do coprocessador de vírgula flutuante com o valor real simples armazenado no endereço addr.⁴

Instruções de Transferência Registo-Memória (Store)

Incluem-se nesta categoria instruções que realizam uma operação algorítmica do tipo “**var** = **value**” (**var** toma o valor de **value**), em que **var** representa uma variável em memória e **value** um valor armazenado em registo (na descrição da instrução, *addr* representa o conteúdo do registo *Reg* adicionado à Constante *Imm*).

sb Rdst, Imm(Reg)	Store Byte	*addr = (byte)Rdst
--------------------------	------------	--------------------

Armazena no endereço addr o byte menos significativo do registo Rdst.

sw Rdst, Imm(Reg)	Store Word	*addr = (word)Rdst
--------------------------	------------	--------------------

Armazena no endereço addr a palavra do registo Rdst.

As instruções seguintes permitem transferir informação dos registos dos coprocessadores para memória.

swcz CReg, Imm(Reg)	Store Word Coprocessor z	*(word)addr = CReg
----------------------------	--------------------------	--------------------

Armazena no endereço addr o valor de 32 bits contido no registo CReg do coprocessador Cz.

s.d FPR _{Reg} , Imm(Reg)	Store Double	*(double)addr = FPR _{Reg}
--	--------------	------------------------------------

Armazena no endereço addr o número real duplo do registo FPR_{Reg} de coprocessador de vírgula flutuante.

s.s FPR _{Reg} , Imm(Reg)	Store Single	*(single)addr = FPR _{Reg}
--	--------------	------------------------------------

Armazena no endereço addr o número real simples do registo FPR_{Reg} do coprocessador de vírgula flutuante.⁵

³ z deverá ser substituído pelo número do coprocessador pretendido.

⁴ Esta instrução é equivalente à instrução `lwc1 FPRReg, Imm(Reg)`.

Instruções de Transferência Registo-Registo (*Move*)

Incluem-se nesta categoria instruções que realizam uma operação algorítmica do tipo “**var** = **value**” (**var** toma o valor de **value**), em que **var** e **value** são registos.

move Rdst,Rsrc	Move	Rdst = Rsrc
------------------------------	------	-------------

Copia o valor do registo Rsrc para o registo Rdst.

O resultado das operações de multiplicação e divisão inteiras é colocado nos registos **HI** e **LO**, ambos de 32 bits. As instruções seguintes permitem transferir informação entre aqueles registos e os registos gerais.

mfhi Rdst	Move From HI	Rdst = HI
-------------------------	--------------	-----------

Copia o valor do registo HI para o registo Rdst.

mflo Rdst	Move From LO	Rdst = LO
-------------------------	--------------	-----------

Copia o valor do registo LO para o registo Rdst.

mtli Rsrc	Move To HI	HI = Rsrc
-------------------------	------------	-----------

Copia o valor do registo Rsrc para o registo HI.

mtlo Rsrc	Move To LO	LO = Rsrc
-------------------------	------------	-----------

Copia o valor do registo Rsrc para o registo LO.

As instruções seguintes permitem transferir informação entre os registos dos coprocessadores e os registos gerais do CPU.

mfcz Rdst,CReg	Move From Coprocessor Cz	Rdst = CReg
------------------------------	--------------------------	-------------

Copia o valor do registo CReg do coprocessador Cz para o registo geral Rdst.

mtcz Rsrc,CReg	Move To Coprocessor Cz	CReg = Rsrc
------------------------------	------------------------	-------------

Copia o valor do registo geral Rsrc para o registo CReg do coprocessador Cz.

Copia o valor real duplo (64 bits) do registo FPReg do coprocessador de vírgula flutuante para os registos gerais Rdst e Rdst+1.

As instruções seguintes permitem transferir valores entre registos do coprocessador de vírgula flutuante.

mov.d \$fd,\$fs	Move Double	\$fd = \$fs
-------------------------------	-------------	-------------

Copia o valor real duplo do registo \$fs para o registo \$fd.

mov.s \$fd,\$fs	Move Single	\$fd = \$fs
-------------------------------	-------------	-------------

Copia o valor real simples do registo \$fs para o registo \$fd.

⁵ Esta instrução é equivalente à instrução `swc1 FPReg, Imm(Reg)`.

Instruções de Manipulação de Constantes (*Load Immediate*)

Incluem-se nesta categoria instruções que realizam uma operação algorítmica do tipo “**var** = **value**” (**var** toma o valor de **value**), em que **var** representa um registo e **value** uma constante (imediato).

li Rdst, Imm	Load Immediate	Rdst = Imm
---------------------	----------------	------------

Copia a constante Imm para o registo Rdst.

lui Rdst, Imm	Load Upper Immediate	Rdst = Imm << 16
----------------------	----------------------	------------------

Copia a constante Imm (16 bits) para os 16 bits mais significativos do registo Rdst, colocando os menos significativos a zero.

li.d \$fd, double	Load Immediate Double	\$fd = double
------------------------------	----------------------------------	--------------------------

Copia a constante real double para os registos do coprocessador de vírgula flutuante \$fd e \$fd+1.

li.s \$fd, float	Load Immediate Single	\$fd = float
-----------------------------	----------------------------------	-------------------------

Copia a constante real float para o registo do coprocessador de vírgula flutuante \$fd.

Instruções de Cálculo sobre Inteiros: Operações Aritméticas

Incluem-se nesta categoria instruções que realizam operações aritméticas sobre quantidades inteiras.

Calcula o valor absoluto do valor do registo Rsrc e coloca o resultado no registo Rdst.

add Rdst, Rsrc, Src	Addition	Rdst = Rsrc + Src
----------------------------	----------	-------------------

Calcula a soma do registo Rsrc e da quantidade Src (registo ou constante) e coloca o resultado no registo Rdst. Gera uma excepção em caso de *overflow*.

addi Rdst, Rsrc, Imm	Addition Immediate	Rdst = Rsrc + Imm
-----------------------------	--------------------	-------------------

Calcula a soma do registo Rsrc e da constante Imm e coloca o resultado no registo Rdst. Gera uma excepção em caso de *overflow*.

addiu Rdst, Rsrc, Imm	Addition Immediate Unsigned	Rdst = Rsrc + Imm
------------------------------	-----------------------------	-------------------

Calcula a soma do registo Rsrc e da constante Imm e coloca o resultado no registo Rdst. Ignora *overflow*.

addu Rdst, Rsrc, Src	Addition Unsigned	Rdst = Rsrc + Src
-----------------------------	-------------------	-------------------

Calcula a soma do registo Rsrc e da quantidade Src (registo ou constante) e coloca o resultado no registo Rdst. Ignora *overflow*.

div Rsrc, Rsrc2	Division	HI = Rsrc % Rsrc2 LO = Rsrc / Rsrc2
------------------------	----------	--

Calcula a divisão inteira do registo Rsrc pelo registo Rsrc2. O resultado, resto e quociente, é colocado nos registos HI e LO. Gera uma excepção em caso de *overflow*.

divu Rsrc, Rsrc2	Division Unsigned	HI = Rsrc % Rsrc2 LO = Rsrc / Rsrc2
-------------------------	-------------------	--

Calcula a divisão inteira do registo Rsrc pelo registo Rsrc2. O resultado, resto e quociente, é colocado nos registos HI e LO. Ignora *overflow*.

div Rdst, Rsrc, Src	Division	Rdst = Rsrc / Src
----------------------------	----------	-------------------

Calcula o quociente da divisão inteira entre o registo Rsrc e a quantidade (registo ou constante) Src e coloca-o no registo Rdst. Gera uma excepção em caso de *overflow*.

divu Rdst, Rsrc, Src	Division Unsigned	Rdst = Rsrc / Src
-----------------------------	-------------------	-------------------

Calcula o quociente da divisão inteira entre o registo Rsrc e a quantidade (registo ou constante) Src e coloca-o no registo Rdst. Ignora *overflow*.

mul Rdst, Rsrc, Src	Multiply	Rdst = Rsrc * Src
----------------------------	----------	-------------------

Calcula o produto do registo Rsrc pela quantidade Src (registo ou constante) e coloca o resultado no registo Rdst. Ignora *overflow*.

mulo Rdst,Rsrc,Src	Multiply	$Rdst = Rsrc * Src$
---------------------------	----------	---------------------

Calcula o produto do registo Rsrc pela quantidade Src (registo ou constante) e coloca o resultado no registo Rdst. Gera uma exceção em caso de *overflow*.

mulou Rdst,Rsrc,Src	Multiply Unsigned	$Rdst = Rsrc * Src$
----------------------------	-------------------	---------------------

Calcula o produto do registo Rsrc pela quantidade Src (registo ou constante) e coloca o resultado no registo Rdst. Gera uma exceção em caso de *overflow*.

mult Rsrc1,Rsrc2	Multiply	$HI,LO = Rsrc1 * Rsrc2$
-------------------------	----------	-------------------------

Calcula o produto dos registos Rsrc1 e Rsrc2 e coloca o resultado nos registos HI e LO.

multu Rsrc1,Rsrc2	Multiply Unsigned	$HI,LO = Rsrc1 * Rsrc2$
--------------------------	-------------------	-------------------------

Calcula o produto dos registos Rsrc1 e Rsrc2 e coloca o resultado nos registos HI e LO. Interpreta os operandos como valores sem sinal..

neg Rdst,Rsrc	Negate	$Rdst = - Rsrc$
----------------------	--------	-----------------

Calcula a negação (complemento para 2) do registo Rsrc e coloca o resultado no registo Rdst. Gera uma exceção em caso de *overflow*.

negu Rdst,Rsrc	Negate	$Rdst = 0 - Rsrc$
-----------------------	--------	-------------------

Calcula a negação (complemento para 2) do registo Rsrc e coloca o resultado no registo Rdst. Ignora *overflow*.

rem Rdst,Rsrc,Src	Remainder	$Rdst = Rsrc \% Src$
--------------------------	-----------	----------------------

Calcula o resto da divisão inteira entre o registo Rsrc e a quantidade (registo ou constante) Src e coloca o resultado no registo Rdst.

remu Rdst,Rsrc,Src	Remainder Unsigned	$Rdst = Rsrc \% Src$
---------------------------	--------------------	----------------------

Calcula o resto da divisão inteira entre o registo Rsrc e a quantidade (registo ou constante) Src e coloca o resultado no registo Rdst. Interpreta os operandos como valores sem sinal.

sub Rdst,Rsrc,Src	Subtract	$Rdst = Rsrc - Src$
--------------------------	----------	---------------------

Calcula a subtração do registo Rsrc pela quantidade Src (registo ou constante) e coloca o resultado no registo Rdst. Gera uma exceção em caso de *overflow*.

subu Rdst,Rsrc,Src	Subtract Unsigned	$Rdst = Rsrc - Src$
---------------------------	-------------------	---------------------

Calcula a subtração do registo Rsrc pela quantidade Src (registo ou constante) e coloca o resultado no registo Rdst. Ignora *overflow*.

Instruções de Cálculo sobre Inteiros: Operações Lógicas Bit-a-Bit (*Bitwise*)

Incluem-se nesta categoria instruções que realizam operações lógicas bit-a-bit (*bitwise*) sobre quantidades inteiras.

and Rdst,Rsrc,Src	And (Bitwise)	$Rdst = Rsrc \& Src$
--------------------------	---------------	----------------------

Calcula o produto lógico bit-a-bit (*bitwise and*) do registo Rsrc e da quantidade Src (registo ou constante) e coloca o resultado no registo Rdst.

andi Rdst,Rsrc,Imm	And Immediate (Bitwise)	$Rdst = Rsrc \& Imm$
---------------------------	-------------------------	----------------------

Calcula o produto lógico bit-a-bit (*bitwise and*) do registo Rsrc e da constante Imm e coloca o resultado no registo Rdst.

nor Rdst,Rsrc,Src	Nor (Bitwise)	$Rdst = Rsrc \! Src$
--------------------------	---------------	-----------------------

Calcula o não-ou bit-a-bit (*bitwise nor*) do registo Rsrc e da quantidade Src (registo ou constante) e coloca o resultado no registo Rdst.

not Rdst,Rsrc	Not (Bitwise)	
----------------------	---------------	--

Calcula a negação bit-a-bit (*bitwise not*) do registo Rsrc e coloca o resultado no registo Rdst.

or Rdst,Rsrc,Src	Or (Bitwise)	$Rdst = Rsrc Src$
-------------------------	--------------	---------------------

Calcula o ou bit-a-bit (*bitwise or*) do registo Rsrc e da quantidade Src (registo ou constante) e coloca o resultado no registo Rdst.

ori Rdst,Rsrc,Imm	Or Immediate (Bitwise)	$Rdst = Rsrc Imm$
--------------------------	------------------------	---------------------

Calcula o ou bit-a-bit (*bitwise or*) do registo Rsrc e da constante Imm e coloca o resultado no registo Rdst.

xor Rdst, Rsrc, Src	XOR	$Rdst = Rsrc \wedge Src$
----------------------------	-----	--------------------------

Calcula o ou-exclusivo (XOR) entre registo Rsrc e a quantidade Src (registo ou constante) e coloca o resultado no registo Rdst.

xori Rdst, Rsrc, Imm	XOR Immediate	$Rdst = Rsrc \wedge Imm$
-----------------------------	---------------	--------------------------

Calcula o ou-exclusivo (XOR) entre registo Rsrc e a constante Imm e coloca o resultado no registo Rdst.

Instruções de Cálculo sobre Inteiros: Operações de Deslocamento (*Shift*) e de Rotação (*Rotate*)

Incluem-se nesta categoria instruções que realizam operações de deslocamento (*shift*) e de rotação (*rotate*) sobre quantidades inteiras.

rol Rdst, Rsrc, Src	Rotate Left	$Rdst = Rsrc \ll Src$
----------------------------	-------------	-----------------------

Coloca o resultado da rotação para a esquerda do registo Rsrc pela quantidade (registo ou constante) Src no registo Rdst.

ror Rdst, Rsrc, Src	Rotate Right	$Rdst = Rsrc \gg Src$
----------------------------	--------------	-----------------------

Coloca o resultado da rotação para a direita do registo Rsrc pela quantidade (registo ou constante) Src no registo Rdst.

sll Rdst, Rsrc, Src	Shift Left Logical	$Rdst = Rsrc \ll Src$
----------------------------	--------------------	-----------------------

Desloca (shift) Src bits para a esquerda o conteúdo do registo Rsrc e coloca o resultado em Rdst.

sllv Rdst, Rsrc, Rsrc2	Shift Left Logical Variable	$Rdst = Rsrc \ll Rsrc2$
-------------------------------	-----------------------------	-------------------------

Desloca (shift) Rsrc2 bits para a esquerda o conteúdo do registo Rsrc e coloca o resultado em Rdst.

sra Rdst, Rsrc, Src	Shift Right Arithmetic	$Rdst = Rsrc \gg Src$
----------------------------	------------------------	-----------------------

Desloca (shift) Src bits para a direita o conteúdo do registo Rsrc e coloca o resultado em Rdst.

srav Rdst, Rsrc, Rsrc2	Shift Right Arithmetic Variable	$Rdst = Rsrc \gg Rsrc2$
-------------------------------	---------------------------------	-------------------------

Desloca (shift) Rsrc2 bits para a direita o conteúdo do registo Rsrc e coloca o resultado em Rdst.

srl Rdst, Rsrc, Src	Shift Right Logical	$Rdst = Rsrc \gg Src$
----------------------------	---------------------	-----------------------

Desloca (shift) Src bits para a direita o conteúdo do registo Rsrc e coloca o resultado em Rdst. Operação sem-sinal (*unsigned*).

srlv Rdst, Rsrc, Rsrc2	Shift Right Logical Variable	$Rdst = Rsrc \gg Rsrc2$
-------------------------------	------------------------------	-------------------------

Desloca (shift) Rsrc2 bits para a direita o conteúdo do registo Rsrc e coloca o resultado em Rdst. Operação sem-sinal (*unsigned*).

Instruções de Comparação

Todas as instruções nesta categoria fazem uma comparação (operação relacional) entre duas quantidades e afectam um registo com o resultado (verdadeiro ou falso) da comparação.

seq Rdst, Rsrc, Src	Set on Equal	$Rdst = (Rsrc == Src) ? 1 : 0$
----------------------------	--------------	--------------------------------

Coloca Rdst a 1 se Rsrc é igual a Src e a 0 caso contrário.

sge Rdst, Rsrc, Src	Set on Greater Than or Equal	$Rdst = (Rsrc \geq Src) ? 1 : 0$
----------------------------	------------------------------	----------------------------------

Coloca Rdst a 1 se Rsrc é maior ou igual a Src e a 0 caso contrário.

sgeu Rdst, Rsrc, Src	Set on Greater Than or Equal Unsigned	$Rdst = (Rsrc \geq Src) ? 1 : 0$
-----------------------------	---------------------------------------	----------------------------------

Coloca Rdst a 1 se Rsrc é maior ou igual a Src e a 0 caso contrário. Operação sem-sinal (*unsigned*).

sgt Rdst, Rsrc, Src	Set on Greater Than	$Rdst = (Rsrc > Src) ? 1 : 0$
----------------------------	---------------------	-------------------------------

Coloca Rdst a 1 se Rsrc é maior que Src e a 0 caso contrário.

sgtu Rdst, Rsrc, Src	Set on Greater Than Unsigned	$Rdst = (Rsrc > Src) ? 1 : 0$
-----------------------------	------------------------------	-------------------------------

Coloca Rdst a 1 se Rsrc é maior que Src e a 0 caso contrário. Operação sem-sinal (*unsigned*).

sle Rdst,Rsrc,Src	Set on Less Than or Equal	$Rdst = (Rsrc \leq Src) ? 1 : 0$
--------------------------	---------------------------	----------------------------------

Coloca Rdst a 1 se Rsrc é menor ou igual a Src e a 0 caso contrário.

sleu Rdst,Rsrc,Src	Set on Less Than or Equal Unsigned	$Rdst = (Rsrc \leq Src) ? 1 : 0$
---------------------------	------------------------------------	----------------------------------

Coloca Rdst a 1 se Rsrc é menor ou igual a Src e a 0 caso contrário. Operação sem-sinal (*unsigned*).

slt Rdst,Rsrc,Src	Set on Less Than	$Rdst = (Rsrc < Src) ? 1 : 0$
--------------------------	------------------	-------------------------------

Coloca Rdst a 1 se Rsrc é menor que Src e a 0 caso contrário.

sltu Rdst,Rsrc,Src	Set on Less Than Unsigned	$Rdst = (Rsrc < Src) ? 1 : 0$
---------------------------	---------------------------	-------------------------------

Coloca Rdst a 1 se Rsrc é menor que Src e a 0 caso contrário. Operação sem-sinal (*unsigned*).

slti Rdst,Rsrc,Imm	Set on Less Than Immediate	$Rdst = (Rsrc < Imm) ? 1 : 0$
---------------------------	----------------------------	-------------------------------

Coloca Rdst a 1 se Rsrc é menor que Imm e a 0 caso contrário.

sltiu Rdst,Rsrc,Imm	Set on Less Than Immediate Unsigned	$Rdst = (Rsrc < Imm) ? 1 : 0$
----------------------------	-------------------------------------	-------------------------------

Coloca Rdst a 1 se Rsrc é menor que Imm e a 0 caso contrário. Operação sem-sinal (*unsigned*).

sne Rdst,Rsrc,Src	Set on Not Equal	$Rdst = (Rsrc \neq Src) ? 1 : 0$
--------------------------	------------------	----------------------------------

Coloca Rdst a 1 se Rsrc é diferente de Src e a 0 caso contrário.

Instruções de Salto Relativo (*Branch*) e Salto Absoluto (*Jump*)

Nas instruções de salto relativo (*branch*) o salto é definido por uma quantidade com sinal de 16 bits que representa a distância em número de instruções (não de bytes) para onde se pretende saltar; ou seja, o salto está limitado a $2^{15}-1$ instruções para a frente ou 2^{15} instruções para trás. No entanto, ao nível da linguagem *assembly* especifica-se o endereço para onde se pretende saltar, deixando ao assembler o papel de transformar esse valor num deslocamento (*offset*). Nas instruções de salto absoluto (*jump*) é especificado directamente o endereço para onde se pretende saltar, através de um registo ou de uma quantidade de 26 bits que multiplicada por 4 (número de bytes de cada instrução) define o endereço alvo.

b Label	Branch	goto label
----------------	--------	------------

Salta para o endereço Label.

bczf Label	Branch Coprocessor z FALSE	if (cz_flag == FALSE) goto label
-------------------	----------------------------	----------------------------------

Salta para o endereço Label se a *flag* de condição do coprocessador z é FALSE.

bczt Label	Branch Coprocessor z TRUE	if (cz_flag == TRUE) goto label
-------------------	---------------------------	---------------------------------

Salta para o endereço Label se a *flag* de condição do coprocessador z é TRUE.

beq Rsrc1,Src2,Label	Branch on Equal	if (Rsrc1 == Src2) goto label
-----------------------------	-----------------	-------------------------------

Salta para o endereço Label se o registo Rsrc1 é igual à quantidade Src2 (registo ou constante).

beqz Rsrc,Label	Branch on Equal Zero	if (Rsrc == 0) goto label
------------------------	----------------------	---------------------------

Salta para o endereço Label se o registo Rsrc1 é igual a zero.

bge Rsrc1,Src2,Label	Branch on Greater Than or Equal	if (Rsrc1 >= Src2) goto label
-----------------------------	---------------------------------	-------------------------------

Salta para o endereço Label se o registo Rsrc1 é maior ou igual à quantidade Src2 (registo ou constante).

bgeu Rsrc1,Src2,Label	Branch on Greater Than or Equal Unsigned	if (Rsrc1 >= Src2) goto label
------------------------------	--	-------------------------------

Salta para o endereço Label se o registo Rsrc1 é maior ou igual à quantidade Src2 (registo ou constante). Operação sem sinal (*unsigned*).

bgez Rsrc,Label	Branch on Greater Than or Equal Zero	if (Rsrc >= 0) goto label
------------------------	--------------------------------------	---------------------------

Salta para o endereço Label se o registo Rsrc1 é maior ou igual a 0.

bgezal Rsrc, Label	Branch on Greater Than or Equal Zero and Link	if (Rsrc1 >= 0) { \$31=PC; goto label }
---------------------------	---	--

Salta para o endereço Label se o registo Rsrc1 é maior ou igual que 0. Antes de saltar guarda o endereço da próxima instrução no registo \$31.

bgt Rsrc1, Src2, Label	Branch on Greater Than	if (Rsrc1 > Src2) goto label
-------------------------------	------------------------	------------------------------

Salta para o endereço Label se o conteúdo do registo Rsrc1 é maior que quantidade Src2 (registo ou constante).

bgtu Rsrc1, Src2, Label	Branch on Greater Than Unsigned	if (Rsrc1 > Src2) goto label
--------------------------------	---------------------------------	------------------------------

Salta para o endereço Label se o conteúdo do registo Rsrc1 é maior que quantidade Src2 (registo ou constante). Operação sem sinal (*unsigned*).

bgtz Rsrc, Label	Branch on Greater Than Zero	if (Rsrc1 > 0) goto label
-------------------------	-----------------------------	---------------------------

Salta para o endereço Label se o conteúdo do registo Rsrc1 é maior que 0.

ble Rsrc1, Src2, Label	Branch on Less Than or Equal	if (Rsrc1 <= Src2) goto label
-------------------------------	------------------------------	-------------------------------

Salta para o endereço Label se o conteúdo do registo Rsrc1 é menor ou igual que quantidade Src2 (registo ou constante).

bleu Rsrc1, Src2, Label	Branch on Less Than or Equal Unsigned	if (Rsrc1 <= Src2) goto label
--------------------------------	---------------------------------------	-------------------------------

Salta para o endereço Label se o conteúdo do registo Rsrc1 é menor ou igual que a quantidade Src2 (registo ou constante). Operação sem sinal (*unsigned*).

blez Rsrc, Label	Branch on Less Than or Equal Zero	if (Rsrc1 <= 0) goto label
-------------------------	-----------------------------------	----------------------------

Salta para o endereço Label se o conteúdo do registo Rsrc1 é menor ou igual que 0.

blt Rsrc1, Src2, Label	Branch on Less Than	if (Rsrc1 < Src2) goto label
-------------------------------	---------------------	------------------------------

Salta para o endereço Label se o conteúdo do registo Rsrc1 é menor que quantidade Src2 (registo ou constante).

bltu Rsrc1, Src2, Label	Branch on Less Than Unsigned	if (Rsrc1 < Src2) goto label
--------------------------------	------------------------------	------------------------------

Salta para o endereço Label se o conteúdo do registo Rsrc1 é menor que a quantidade Src2 (registo ou constante). Operação sem sinal (*unsigned*).

bltz Rsrc, Label	Branch on Less Than Zero	if (Rsrc1 < 0) goto label
-------------------------	--------------------------	---------------------------

Salta para o endereço Label se o conteúdo do registo Rsrc1 é menor que 0.

bltzal Rsrc, Label	Branch on Less Than Zero and Link	if (Rsrc1 < 0) { \$31=PC; goto label }
---------------------------	-----------------------------------	---

Salta para o endereço Label se o registo Rsrc1 é menor que 0. Antes de saltar guarda o endereço da próxima instrução no registo \$31.

bne Rsrc1, Src2, Label	Branch on Not Equal	if (Rsrc1 != Src2) goto label
-------------------------------	---------------------	-------------------------------

Salta para o endereço Label se o conteúdo do registo Rsrc1 é diferente à quantidade Src2 (registo ou constante).

bnez Rsrc, Label	Branch on Not Equal Zero	if (Rsrc != 0) goto label
-------------------------	--------------------------	---------------------------

Salta para o endereço Label se o conteúdo do registo Rsrc1 é diferente a zero.

j Label	Jump	goto label
----------------	------	------------

Salta para o endereço Label.

jal Label	Jump and Link	\$31=PC; goto label
------------------	---------------	---------------------

Salta para o endereço Label, armazenando o endereço da próxima instrução no registo \$31.

jalr Rsrc	Jump and Link Register	\$31=PC; goto Rsrc
------------------	------------------------	--------------------

Salta para o endereço dado pelo conteúdo do registo Rsrc, armazenando o endereço da próxima instrução no registo \$31.

jr Rsrc	Jump Register	goto Rsrc
----------------	---------------	-----------

Salta para o endereço dado pelo conteúdo do registo Rsrc.

Instruções de Cálculo em Vírgula Flutuante

Todas as instruções nesta categoria realizam operações de cálculo sobre quantidades reais representadas em vírgula flutuante (formato IEEE-754). Estas operações estão disponíveis tanto para quantidades em precisão simples como em precisão dupla. As mnemónicas usadas num caso e noutro são praticamente iguais apenas diferindo num sufixo: **.s** para as operações em precisão simples, e **.d** para precisão dupla. Como forma de simplificar a apresentação das instruções seguintes usa-se um sufixo **.p** para representar genericamente os dois tipos de precisão, devendo este ser substituído, aquando da utilização da instrução, pelo sufixo adequado (**.s** para precisão simples e **.d** para precisão dupla).

<code>abs.p \$fd,\$fs</code>	Absolute Value	$\$fd = \$fs $
------------------------------	----------------	-------------------

Calcula o valor absoluto do número real (simples ou duplo) no registo \$fs e coloca o resultado no registo \$fd.

<code>add.p \$fd,\$fs,\$ft</code>	Addition	$\$fd = \$fs + \$ft$
-----------------------------------	----------	----------------------

Calcula a soma dos números reais (simples ou duplos) nos registos \$fs e \$ft e coloca o resultado no registo \$fd.

<code>c.eq.p \$fs,\$ft</code>	Compare Equal	$fp_flag = (\$fs == \$ft) ? true : false$
-------------------------------	---------------	--

Compara os números reais (simples ou duplos) nos registos \$fs e \$ft e coloca a *flag* de condição a TRUE se forem iguais.

<code>c.le.p \$fs,\$ft</code>	Compare Less Than or Equal	$fp_flag = (\$fs \leq \$ft) ? true : false$
-------------------------------	----------------------------	--

Compara os números reais (simples ou duplos) nos registos \$fs e \$ft e coloca a *flag* de condição a TRUE se o primeiro for menor ou igual que o segundo.

<code>c.lt.p \$fs,\$ft</code>	Compare Less Than	$fp_flag = (\$fs < \$ft) ? true : false$
-------------------------------	-------------------	---

Compara os números reais (simples ou duplos) nos registos \$fs e \$ft e coloca a *flag* de condição a TRUE se o primeiro for menor que o segundo.

<code>cvt.d.s \$fd,\$fs</code>	Convert Single to Double	$\$fd = (double)\fs
--------------------------------	--------------------------	-----------------------

Converte o número real simples no registo \$fs para duplo e coloca o resultado no registo \$fd.

<code>cvt.d.w \$fd,\$fs</code>	Convert Integer to Double	$\$fd = (double)\fs
--------------------------------	---------------------------	-----------------------

Converte o número inteiro (*word*) no registo \$fs para real duplo e coloca o resultado no registo \$fd.

<code>cvt.s.d \$fd,\$fs</code>	Convert Double to Single	$\$fd = (float)\fs
--------------------------------	--------------------------	----------------------

Converte o número real duplo no registo \$fs para simples e coloca o resultado no registo \$fd.

<code>cvt.s.w \$fd,\$fs</code>	Convert Integer to Single	$\$fd = (float)\fs
--------------------------------	---------------------------	----------------------

Converte o número inteiro (*word*) no registo \$fs para real simples e coloca o resultado no registo \$fd.

<code>cvt.w.d \$fd,\$fs</code>	Convert Double to Integer	$\$fd = (int)\fs
--------------------------------	---------------------------	--------------------

Converte o número real duplo no registo \$fs para inteiro (*word*) e coloca o resultado no registo \$fd.

<code>cvt.w.s \$fd,\$fs</code>	Convert Single to Integer	$\$fd = (int)\fs
--------------------------------	---------------------------	--------------------

Converte o número real simples no registo \$fs para inteiro (*word*) e coloca o resultado no registo \$fd.

<code>div.p \$fd,\$fs,\$ft</code>	Divide	$\$fd = \$fs / \$ft$
-----------------------------------	--------	----------------------

Calcula a divisão dos números reais (simples ou duplos) nos registos \$fs e \$ft e coloca o resultado no registo \$fd.

<code>mul.p \$fd,\$fs,\$ft</code>	Multiply	$\$fd = \$fs * \$ft$
-----------------------------------	----------	----------------------

Calcula o produto dos números reais (simples ou duplos) nos registos \$fs e \$ft e coloca o resultado no registo \$fd.

<code>neg.p \$fd,\$fs</code>	Negate	$\$fd = 0 - \fs
------------------------------	--------	-------------------

Faz a negação do número real (simples ou duplo) no registo \$fs e coloca o resultado no registo \$fd.

<code>sub.p \$fd,\$fs,\$ft</code>	Subtract	$\$fd = \$fs - \$ft$
-----------------------------------	----------	----------------------

Calcula a subtração dos números reais (simples ou duplos) nos registos \$fs e \$ft e coloca o resultado no registo \$fd.

Instruções para Manipulação de Excepções e *Traps*

<code>break n</code>	Break	
----------------------	-------	--

Provoca a excepção n. A excepção 1 é reservada para o *debugger*.

<code>nop</code>	No Operation	
------------------	--------------	--

Não faz nada.

<code>rfe</code>	Return From Exception	
------------------	-----------------------	--

Restaura o conteúdo do registo Status.

<code>syscall</code>	System Call	
----------------------	-------------	--

Chamada ao sistema operativo. Ver secção “Chamadas ao Sistema (System Calls)”.

Organização da Memória

À semelhança do que tipicamente acontece nos sistemas baseados no processador MIPS o MARS divide a memória em três segmentos: texto, dados e *stack*. O segmento texto (**`.text`**) começa no endereço **0x400000** (o código de utilizador é carregado, por omissão, a partir do endereço **0x400020**) e é usado para alojar as instruções do programa. O segmento de dados (**`.data`**) começa no endereço **0x10010000** e é usado para armazenar os

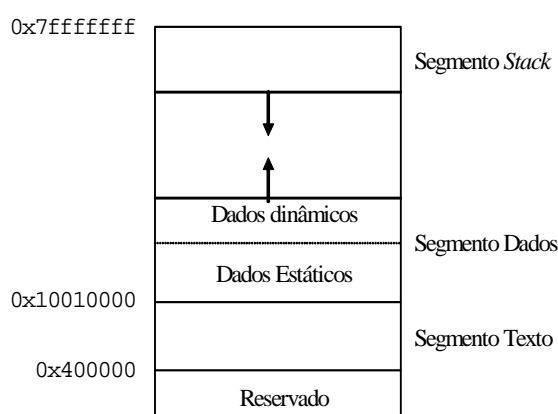


Figura 4 - A organização da memória

dados do programa. Este segmento está por sua vez subdividido em duas partes. A primeira (a partir do endereço **0x10010000**) é usada para alocar objectos cujos comprimento e endereço são conhecidos aquando da assemblagem. Imediatamente a seguir são colocados os objectos alocados dinamicamente usando a chamada ao sistema `sbrk`⁶. Por fim o segmento *stack* (**`.stack`**) começa no topo da memória virtual (**0x7fffffff**) e cresce no sentido contrário ao dos endereços.

Esta divisão da memória em três segmentos não é a única possível. Contudo, ela possui duas características importantes: os dois segmentos dinâmicos estão o mais afastados possível um do outro e podem crescer no sentido de ocuparem a totalidade do espaço de endereçamento da memória.

Além dos três segmentos descritos existem ainda mais dois segmentos próprios do *kernel* do sistema. Eles são os segmentos de texto (**`.ktext`**) e de dados (**`.kdata`**) que começam, respectivamente, nos endereços **0x80000180** e **0x90000000**.

⁶ Se pensarmos em termos de um programa em C esta é a área de memória onde são alocadas as variáveis definidas usando a função `malloc`.