3.18 – Processamento de texto

Os computadores também foram projetados para processamento de texto.
O código ASCII utiliza 1 byte para representar caracteres:

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

As instruções `lw` e `sw` em conjunção com outras instruções podem ser usadas para extrair um byte de uma palavra.
Mas existem instruções especiais para a manipulação de um byte:

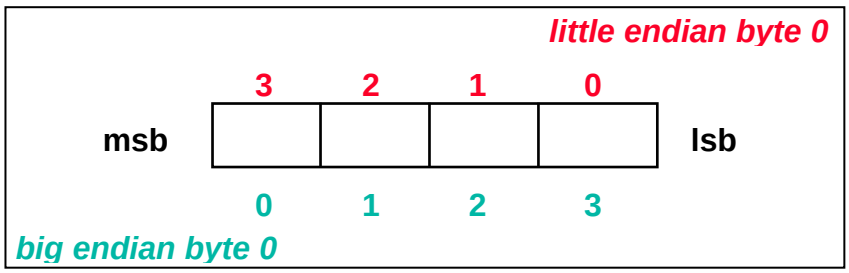
`lb` (load byte) – carrega um byte da memória colocando seu conteúdo nos 8 bits mais a direita do registrador indicado.
`sw` (store byte) – pega um byte dos oito últimos bits a direita do registrador e armazena na memória.

A combinação de caracteres forma os strings.
Os strings podem ser representados de 3 formas:

- a primeira posição do string armazena seu tamanho.
- uma variável associada ao string informa seu tamanho.
- a última posição do string é usada por um caractere que marca seu final.

Big Endian: endereço do byte mais significativo = endereço da palavra (xx00 = Big End of word).
Ex: IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

Little Endian: endereço do byte menos significativo = endereço da palavra (xx00 = Little End of word).
Ex: Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

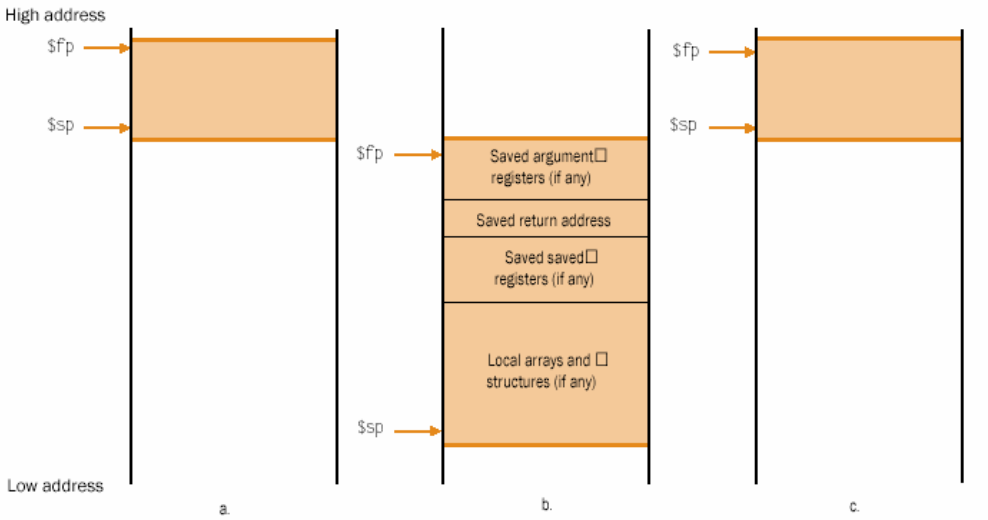


A linguagem C usa a terceira opção, terminando seus strings com um byte igual à zero (NULL).

Exemplo: Luiz ➔caracteres 76, 117,105, 122, 0.

3.19 – Alocação de Espaço para novos dados

A pilha também pode ser usada para armazenar variáveis locais dos procedimentos já que o número de registradores é pequeno.
O segmento da pilha que contém os registradores e suas variáveis locais é chamado de *quadro de procedimento* ou *registro de ativação*.
O MIPS usa um registrador de apontador de quadro de um procedimento: `$fp` (frame pointer).
O : `$fp` é usado como um registrador base para as variáveis locais de um procedimento ao referenciar a memória.



(a) antes da chamada a procedimento → o $fp aponta para a primeira palavra do frame, geralmente um registrador-argumento salvo na pilha. O $fp aponta para o topo da pilha.

(b) Durante o procedimento: a pilha é ajustada para colocar todos os registradores a serem salvos e as variáveis locais.
Como o $sp pode mudar durante a execução do programa é mais fácil referenciar as variáveis locais pelo $fp (estável).
Caso não exista variável locais o compilador irá ganhar tempo ao não inicializar ou restaurar o $fp.
(c) Depois da chamada: são restaurados o $fp e o $sp.

Nome	Número do registrador	Utilização
$zero	0	Valor da constante 0
$v0-$v1	2-3	Valores para guardar resultados e para avaliar expressões
$a0-$a3	4-7	Argumentos
$t0-$t7	8-15	Temporários
$s0-$s7	16-23	Salvos
$t8-$t9	24-25	Mais temporários
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Endereço de retorno a procedimento

3.20 - Operando Imediatos ou Constantes

O uso de constantes ocorre com muita freqüência em diversas operações: incremento de índice em vetores, contagem de iterações em um loop, ajuste do SP em chamadas a procedimentos.
Ao usarmos constantes em um programa é necessário busca-las na memória e carregá-la em registrador (`lw CONST, endereço`), além de o programa armazená-la na memória quando o programa é carregado.
Como alternativa para evitar acessos à memória , outras instruções aritméticas são incluídas no ISA.
Essas instruções possuem um operando que é uma constante mantida dentro da própria instrução, conhecido como IMEDIATO.

O formato da instrução será do Tipo I:

Op	Rs	Rt	Imediato
6 bits	5 bits	5 bits	16 bits
31 26	25 21	20 16	15 0

Exemplos:

`addi:` a instrução `addi` (add imediate).

```
add $sp, $sp, 4 # $sp ← $sp + 4
```

8	29	29	4
addi	$sp	$sp	Imediato
31 26	25 21	20 16	15 0

Em binário:

001000	11101	11101	0000.0000.0000.0100
addi	$sp	$sp	Imediato
31 26	25 21	20 16	15 0

`slti:` a instrução `slti` (set less than imediate).

```
slti $t0, $s2, 10 # se $s2 < 10 → $t0 = 1
```

8	29	29	4
addi	$sp	$sp	Imediato
31 26	25 21	20 16	15 0

Em binário:

001000	01000	10010	0000.0000.0000.1010
(10) slti	(8) $t0	(18) $s2	Imediato
31 26	25 21	20 16	15 0

`lui: Load Upper Imediate` – carrega a constante imediata (meia palavra) para os 16 bits mais significativos do registrador imediato preenchendo o restante com zero.

```
lui $t0, 255
```

0xf	0	8	255
lui	0	$t0	Imediato
31 26	25 21	20 16	15 0

Em binário:

001111	00000	01000	1111.1111.1111.1111
(0xf) lui	0	(8) $s2	(0xffff) Imediato
31 26	25 21	20 16	15 0

Conteúdo do registrador $t0:

1111.1111.1111.1111	0000.0000.0000.0000

Carregando uma constante de 32 bits:

Exemplo: Carregar a constante mostrada a seguir em $so

0000.0000.0011.1101	0000.1001.0000.0000
$(61)_{10}$	$(2304)_{10}$

```
lui $s0, 61    # 61 na parte superior de $s0.
```

$s0

0000.0000.0011.1101	0000.1001.0000.0000

```
addi $s0, $s0, 2304    # $s0 ← $s0 + 2304
```

$s0

0000.0000.0011.1101	0000.1001.0000.0000

3.21 – Endereçamento bis Desvios Condicionais e Incondicionais

Desvios Incondicionais: empregam um formato de instrução do tipo J:

Op (2)	Endereço
6 bits	26 bits
31 26	25 0

2^{26} = 67.103.864 instruções.

`j` –desvio incondicional para um endereço determinado.

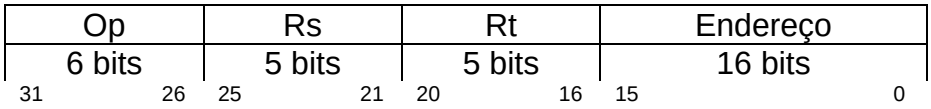
Exemplo:

```
j 10000    #desvia para o endereço 10.000
```

000010	00.0000.0000.0010.0111.0001.0000
6 bits	26 bits
31 26	25 0

Desvios condicionais:

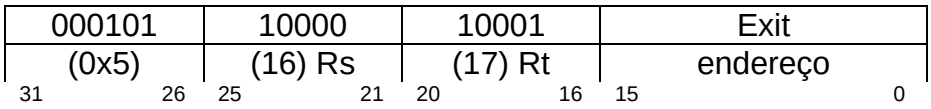
Formato:



Exemplo:

`bne` - desvia se não igual

```
bne $s0, $s1, Exit    # se $s0 • $s1 desvia para Exit
```



Como o campo de endereço de um desvio condicional tem somente 16 bits, existe uma **limitação de desvio para somente 2^{16} bytes de código!** Como contornar esse problema?

A solução é especificar um registrador e somar seu conteúdo ao campo de 16 bits da instrução de desvio: $Reg. + Endereço.

O resultado da soma deve ser armazenado no PC sempre que o predicado da comparação for verdadeiro: PC ← $Reg. + Endereço.
Isso possibilita ao programa ter **2^{32}** bytes de código.

É importante observar que os desvios condicionais são encontrados em loops e em comandos `if`, de modo que tendem a desviar para a próxima instrução.

Exemplo:

```
Loop:add $t1, $s3, $s3   # t1 ← 2* i
  add $t1, $t1, $t1   #t1← 4 *i
  add $t1, $t1, $s6   #t1← endereço de save[ i ]
  lw  $t0, 0( $t1)    #t0← save[ i ]
  bne $t0, $s5, Exit  #desvia para Exit se save[ i ] • k
  add $s3, $s3, $s4   # i ← i + j
    j Loop              # desvia para Loop
Exit:
```

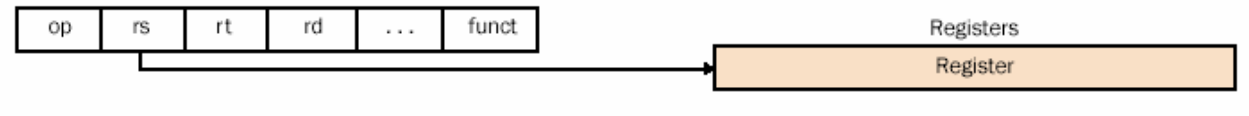
Se admitirmos que o endereço de loop seja 80.000 da memória, qual o código de máquina para esse loop?

80.000	0	19	19	9	0	32	add
80.004	0	9	9	9	0	32	add
80.008	0	9	22	9	0	32	add
80.012	35	9	8	0			lw
80.016	5	8	21	8			bne
80.020	0	19	20	19	0	32	add
80.024	2	80000					j
80.028	...						
80.012	35	9	8	0			lw

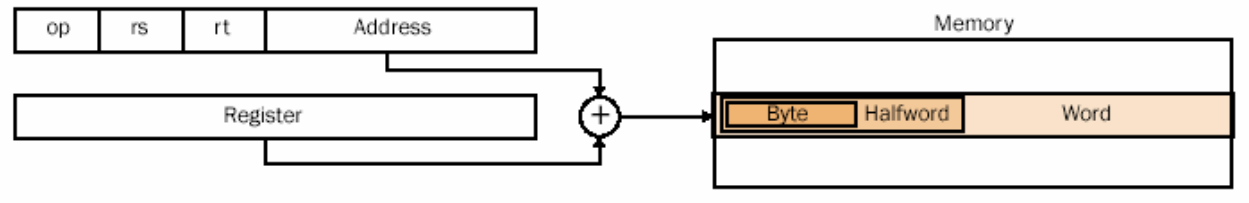
O desvio condicional do `bne` deve referenciar duas instruções posteriores, dessa forma 8.

3.22 – Modos de endereçamento no MIPS:

1. **Endereçamento a registrador**: o operando está em um registrador.

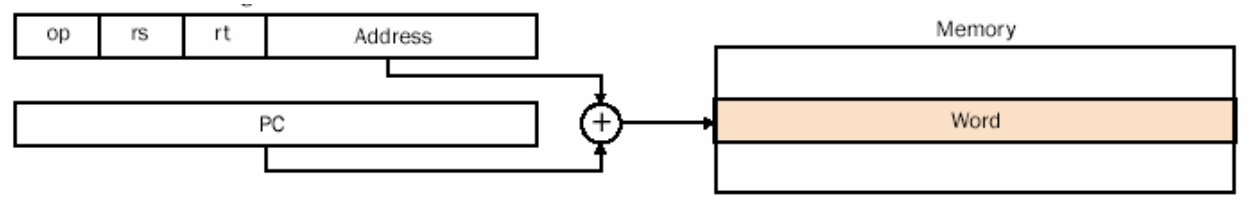


2. **Endereçamento via registrador-base ou via deslocamento**: operando está no endereço de memória obtida com a soma do conteúdo do registrador-base com a constante armazenada na instrução (Const ($reg)).

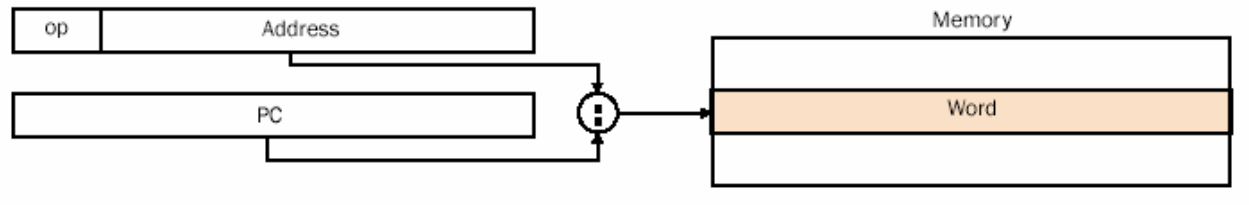


3. **Endereçamento imediato**: o operando é uma constante dentro da própria instrução.

4. **Endereçamento relativo ao PC**: operando está no endereço formado pela soma do conteúdo do PC com a constante obtida na própria instrução.



5. **Endereçamento pseudo-indireto**: o endereço de desvio é formado pela concatenação dos 26 bits obtido da instrução com os bits mais significativos do PC.



3.23 – Pseudo-instrução

São instruções que não existem em linguagem de máquina, mas que o montador aceita e a converte da linguagem de montagem em instruções de máquina.

Exemplo: A pseudo-instrução MOVE.

```
move $t0, $t1 # $t0 ← $t1 copia o conteúdo de t1 em t0.
```

O montador a converte em:

```
add $t0, $zero, $t1 # $t0 ← 0 + $t1
```

Outros exemplos:

```
blt – branch if less than → slt/bne
bgt - branch if great than
bge - branch if great or equal than
ble - branch if less or equal than
```