Parte II: Conjunto de Instruções

Introdução

Introdução

- Instruções
  - Operações (ou comandos) dados ao hardware
  - São as palavras da **linguagem de máquina** (ling. de baixo nível)

- Objetivos
  - Como humanos passam instruções aos computadores?
  - Como computadores as interpretam e executam?
  - Conjuntos de instruções típicas?

- Linguagens de máquina são semelhantes
  - HW seguem os mesmo princípios fundamentais

- Muitas situações exigem o conhecimento de linguagem de máquina
  - Ex. Sistemas embarcados, Sistemas Operacionais, Drivers de diferentes periféricos etc.

- Quem conhece como o HW funciona e como interagimos com ele em baixo nível tem facilidade com qualquer outra linguagem de alto nível
  - É o papel do engenheiro ou cientista da computação!!

Estudo do MIPs

- Processador MIPs
  - Comum na década de 80 em plataformas Nintendo, NEC, Sony, SG etc.
  - Simples, sendo bastante didática
  - Muito semelhante aos RISC modernos
- Simulador SPIM
  - http://spimsimulator.sourceforge.net/
- O conjunto de instruções do MIPs e seu uso se dá de forma análoga a outras arquiteturas

Operandos do Hw

- Registradores
  - Memórias internas ao processador para realização das operações
  - São muito rápidas e são um bloco fundamental para a construção dos processadores (veremos depois)
  - Instruções devem possuir registradores como operandos
- Compiladores associam variáveis de linguagens de alto nível a registradores
  - Variáveis são regiões de dados da memória
  - Para operações no processador, essas dados devem ser transferidos entre a memória e os registradores
- No MIPs
  - São 32 registradores de uso geral de 32 bits cada.
  - Nomenclatura $s0 - $s31 (variáveis) ou $t0 - $t31 (temporários)
  - Porque só 32? Porque mais registradores aumentam a complexidade do processador e afetam desempenho

Operações Aritméticas

- Operações fundamentais em qualquer arquitetura
- No MIPs:

Linguagem de montagem do MIPS

Categoria	Instrução	Exemplo	Significado	Comentário
Aritmética	add	`add a , b , c`	`a = b + c`	Sempre três operandos
	subtract	`sub a , b , c`	`a = b − c`	Sempre três operandos

Exemplo

- Em linguagem de alto nível:

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

- Em assembly (ling. de montagem) do MIPs:
  - Assumindo variáveis nos regs. $s0 a $s4
  - Uso de $t0 e $t1 como regs. temporários

```
add   $t0,$s1,$s2   # registrador $t0 contém g + h
add   $t1,$s3,$s4   # registrador $t1 contém i + j
sub   $s0,$t0,$t1   # f recebe $t0 - $t1, que é (g + h) - (i + j)
```

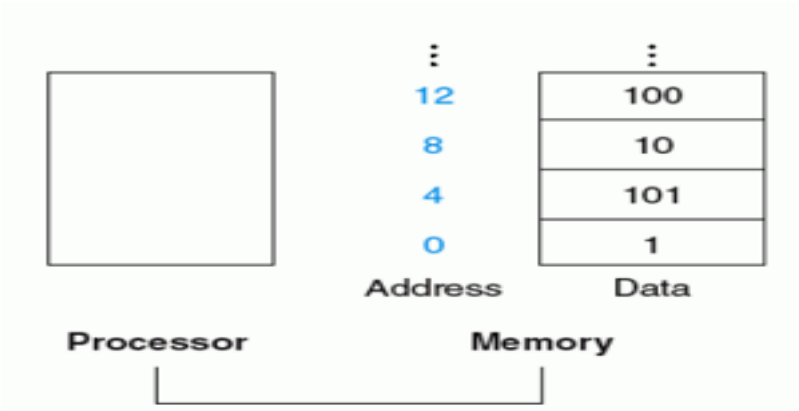
Transferência de Dados

- Programas podem usar estruturas de dados complexas e muito extensas.
  - Há poucos registradores para operações
  - Logo, muito comum o uso de instruções de transferência de dados entre a memória e os registradores
  - No MIPs: Load (lw) e Store (sw)

Transferência de dados	load word	`lw $s1, 100 ($s2)`	`$s1 = Memória[$s2 + 100]`	Dados transferidos da memória para registradores
	store word	`sw $s1, 100($s2)`	`Memória[$s2 + 100] = $s1`	Dados transferidos de registradores para a memória

Restrição de Alinhamento

- Programas constumam armazenar dados em Arrays
- A memória é endereçada em bytes (8 bits) na maioria das arquiteturas
  - Como manipular dados em 32 bits (palavra), como no MIPs?
  - Palavras sequenciais em múltiplos de 4 bytes!
  - É um ajuste necessário na maioria das arquiteturas
    - Se deve à evolução gradual das arquiteturas (8, 16, 32 e agora 64 bits)
- Em um array, para acessar a 8o. palavra da memória:
  - Índice 8 de um array de itens de 32 bits – Ex. A[8]
  - Posição da memória = 4x8 = 32



Exemplo

- Em ling. De alto nível:

Suponha que a variável h esteja associada ao registrador $s2 e que o endereço-base do array A esteja armazenado em $s3. Qual o código de montagem do MIPS para o comando de atribuição seguinte, escrito em C?

```
A[12] = h + A[8];
```

- Em assembly do MIPs:

```
lw    $t0,32 ($s3)    # Registrador temporário $t0 recebe A[8]
add   $t0,$s2, $t0    # Registrador temporário $t0 recebe h + A[8]
 sw   $t0,48($s3)   # h + A[8] é armazenado de volta na memória em A[12]
```

Exemplo 2

- Array indexado

A seguir, apresentamos um exemplo de um array indexado por uma variável.

```
g = h + A[i];
```

Suponha que `A` é um array de 100 elementos cujo endereço-base está no registrador `$s3` e que o compilador associa as variáveis `g`, `h` e `i` aos registradores `$s1`, `$s2` e `$s4`. Qual o código gerado para o MIPS, correspondente ao comando C acima?

Solução 2

- Índice do Array i, na memória : 4x$4

```
add     $t1,$s4,$s4    # Registrador temporário $t1 recebe 2 * i
add     $t1,$t1,$t1    # Registrador temporário $t1 recebe 4 * i
```

- Endereço de A[i]

```
add  $t1,$t1,$s3   # Registrador temporário $t1 recebe o endereço de A[i] (4 * i + $s3)
```

- Carregando A[i] no reg. Temporário

```
lw    $t0,0($t1)    # Registrador temporário $t0 recebe A[i]
```

- Soma

```
add  $s1,$s2,$t0   # g recebe h + A[i]
```

Instruções de Desvio

- Programas necessitam de desvios e repetições em sua execução conforme a verificação de alguma condição
  - Diferença básica entre computador e calculadora
- No MIPs (beq - branch if equal, bne – branch if not equal)
  - Se reg1 == reg2, pula se igual, p/ a instrução com label L1

```
beq registrador1, registrador2, L1
```

  - Se reg1 <> reg2, pula se não igual

```
bne registrador1, registrador2, L1
```

- No MIPs, salto incondicional (jump)

```
j L1 # jump to L1
jr $reg # jump p/ endereço em $reg
```

- Label: Endereço da memória onde se encontra a instrução
  - Montador resolve endereços de labels
  - Linguagens de alto nível normalmente abstraem labels

Exemplo

- Em alto nível:

```
        if (i == j) go to L1;
        f = g + h;
L1:     f = f - i;
```

- f, g, h e i em $s0 a $s4

```
beq    $s3,$s4,L1     # desvia para L1 se i for igual a j
add  $s0,$s1,$s2   # f = g + h (instrução não executada se i for igual a j)
L1: sub $s0,$s0,$s3   # f = f - i (sempre executado)
```

Exemplo 2

- Com as mesmas variáveis e regs. Anteriores

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

- Melhor verificar a condição negativa primeiro, que onde há necessidade de desvio

```
bne   $s3,$s4,Else  # desvia para Else se i ≠ j
add  $s0,$s1,$s2    # f = g + h (salta essa instrução se i ≠ j)
j Exit   # desvia para Exit
Else:   sub $s0, $s1, $s2 # f = g - h (salta essa instrução se i = j)
Exit:
```

Loops

- Em Alto Nível

A seguir apresentamos um loop programado na linguagem C:

```
Loop:  g = g + A[i];
       i = i + j;
       if (i != h) go to Loop;
```

Suponha que A seja um array de 100 elementos e que o compilador associe as variáveis g, h, i e j aos registradores $s1, $s2, $s3 e $s4, respectivamente. Vamos admitir também que a base do array A esteja em $s5. Qual é o código, em linguagem de montagem do MIPS, que corresponde a este loop escrito em C?

- Em MIPs

```
Loop: add   $t1,$s3,$s3   # Registrador temporário $t1 recebe 2 * i
      add   $t1,$t1,$t1   # Registrador temporário $t1 recebe 4 * i
      add   $t1,$t1,$s5   # $t1 recebe endereço de A[i]
      lw    $t0,0($t1)    # Registrador temporário $t0 recebe A[i]
     add  $s1,$s1,$t0     # g recebe = g + A[i]
     add  $s3,$s3,$s4     # i recebe = i + j
    bne  $s3,$s2, Loop    # desvia para Loop se i ≠ h
```

Exemplo

- Construção do While

```
while (save[i] == k)
  i = i + j;
```

- No MIPs:
  - i, j e k em $s3 a $s5, save em $s6

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

Verificação de menor ou maior

- No MIPs, instrução slt (set on less than)
  - -Se $s3 < $s4, $t0 =1
  - Caso contrário, $t0 = 0

```
slt  $t0, $s3, $s4
```

- No MIPs existe o registrador read only $zero, com valor zero.
  - Usado para comparação
- Exemplo

Qual o código MIPS para testar se uma variável `a`, correspondente ao registrador `$s0`, é menor que outra variável `b` (registrador `$s1`) e desviar para o label `Less` se a condição for verdadeira?

```
slt     $t0,$s0,$s1   # reg $t0 recebe 1 se $s0 < $s1 (a < b)
bne     $t0,$zero, Less   # desvia para Less se $t0 ≠ 0
                          # (isto é, se a < b)
```

Suporte a procedimentos

- Comum estruturar programas em sub-rotinas
- Para procedimentos, programas precisam:

1. Colocar os parâmetros em um lugar onde eles possam ser acessados pelo procedimento.
2. Transferir o controle para o procedimento.
3. Garantir os recursos de memória necessários à execução do procedimento.
4. Realizar a tarefa desejada.
5. Colocar o resultado em um lugar acessível ao programa que chamou o procedimento.
6. Retornar o controle para o ponto de origem.

- No MIPs
  - Registradores
    - $a0 a $a3: Argumentos do procedimentos
    - $v0 e $v1: Valores de Retorno do procedimento
    - $ra: Reg. Com valor de end. De retorno do procedimento
  - Instrução
    - Jal end_proc - Jump and link, ajustando $ra para retorno

Utilizando mais registradores

- Ao chamar procedimentos, pode ser necessário o uso de mais registradores.
  - Conteúdo original de registradores podem ter que ser salvos antes de chamar um procedimento, para que esses possam ser manipulados pelo procedimento. Na volta, os valores originais devem ser recuperados para continuidade do programa.
- Pode-se usar a memória em uma estrutura de pilha
  - LIFO (last in, first out)
  - Regs são colocados no topo da pilha, incrementando seu índice (push)
  - Regs são retirados do topo, decrementado seu índice (pop)
- No MIPs, registrador especial $sp (stack pointer)
  - Endereço do topo da pilha

Exemplo

- Chamando função
  - g, h, i e j são passados como parâmetros em $a0 a $a3
  - retorno f em $s0

```
int  leaf_example (int g, int h, int i, int j)
{
        int f;

        f = (g + h) - (i + j);
        return f;
}
```

Exemplo

- Salvando regs usados no procedimento na pilha

```
leaf_example:
sub  $sp,$sp,12   # ajusta a pilha para abrir espaço para guardar 3 itens
sw   $t1, 8($sp)  # salva o conteúdo do registrador $t1 com o objetivo de preservá-lo
sw   $t0, 4($sp)  # salva o conteúdo do registrador $t0 com o objetivo de preservá-lo
sw   $s0, 0($sp)  # salva o conteúdo do registrador $s0 com o objetivo de preservá-lo
```

- Realizando aritmética

```
add   $t0,$a0,$a1   # registrador $t0 contém g + h
add   $t1,$a2,$a3   # registrador $t1 contém i + j
sub   $s0,$t0,$t1   # f recebe $t0 – $t1, que é (g + h) – (i + j)
```

- Retorno no reg.

```
add  $v0,$s0,$zero   # retorna f ($v0 recebe $s0 + 0)
```

- Recuperando valores originais dos regs da pilha

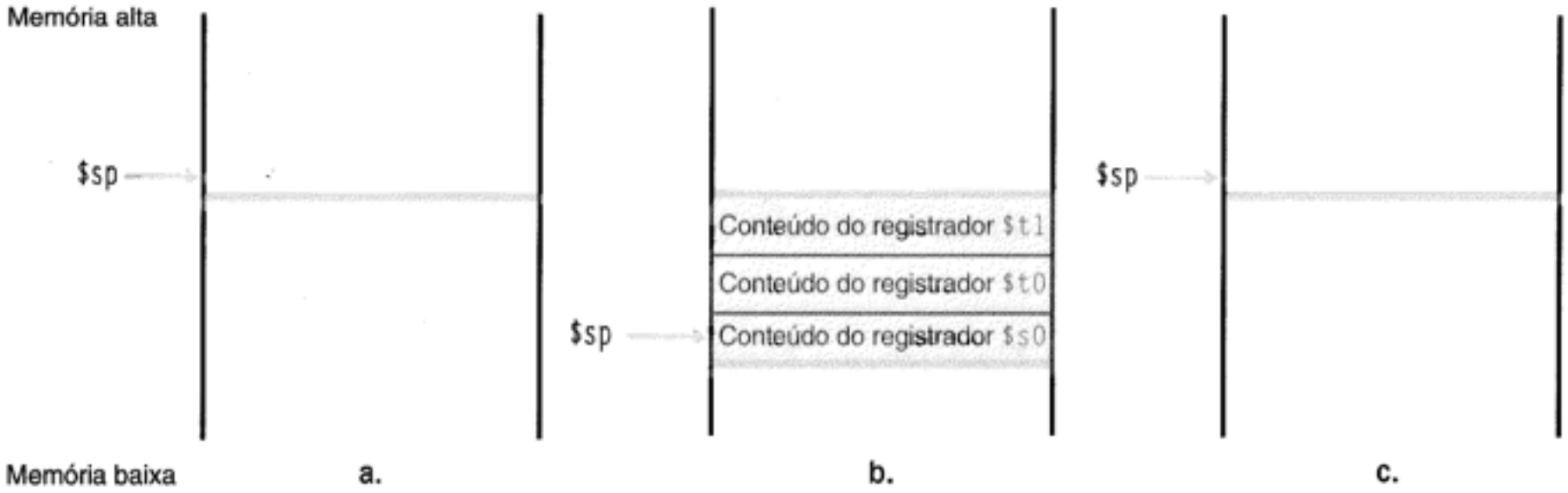
```
lw   $s0, 0($sp)    # restaura o valor de $s0 para o chamador
lw   $t0, 4($sp)    # restaura o valor de $t0 para o chamador
lw   $t1, 8($sp)    # restaura o valor de $t1 para o chamador
add  $sp,$sp,12     # ajusta a pilha de modo a remover os 3 itens
```

- Retornando do procedimento

```
jr   $ra            # desvia de volta para o programa que chamou
```

Exemplo

- Na prática, no MIPs assume-se que os regs $t0 a $t9 são temporários e não precisam ser preservados
  - De conhecimento dos desenvolvedores, de forma que os procedimentos podem usa-los livremente



Procedimentos Aninhados

- Chamada recursiva de procedimentos
  - Registradores precisam ser preservados
    - Registradores com argumentos
    - Registradores temporários usados (mesmo proc. !!)
    - Registradores não-temporários
    - Registrador de retorno
- Ex.

```
int   fact (int n)
{
      if (n < 1) return (1);
      else return (n * fact(n – 1));
}
```

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,100`	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	`lw $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1,100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load half	`lh $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Halfword memory to register
	store half	`sh $s1,100($s2)`	Memory[$s2 + 100] = $s1	Halfword register to memory
	load byte	`lb $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1,100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immed.	`lui $s1,100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,100`	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,100`	$s1 = $s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set less than immediate	`slti $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Representação de Instruções

- Instruções e registradores são mantidos e processados pelo computador como palavras binárias
  - Chamada de Linguagem de Máquina

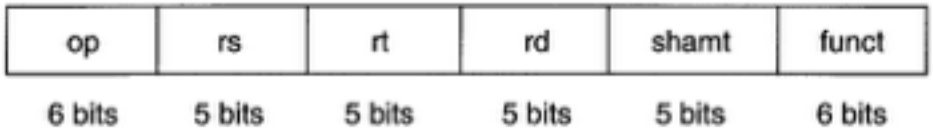
- Linguagem de Montagem (Assembly) tem uma relação direta com a linguagem de máquina

- Convenção para Registradores: Números de 0 a 31

- MIPs têm instruções de 32 bits
  - Uniformidade gera simplicidade do HW

Codificação MIPs

- Instruções tipo R (registradores como operandos)

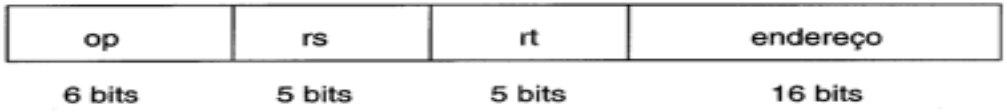


A seguir, veja o significado de cada um dos campos das instruções do MIPS:

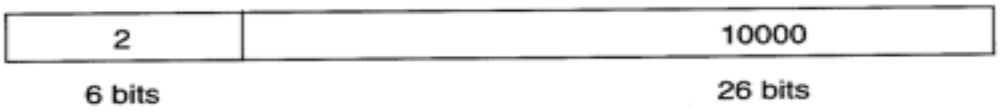
- *op*: a operação básica a ser realizada pela instrução, tradicionalmente chamada de *código de operação* (opcode)
- *rs*: o registrador contendo o primeiro operando-fonte
- *rt*: o registrador contendo o segundo operando-fonte
- *rd*: o registrador que guarda o resultado da operação, também conhecido como *registrador-destino*
- *shamt*: de quantidade de bits a serem deslocados. (Isto será explicado no Capítulo 4, quando estudarmos as instruções de deslocamento; até lá, este campo não será usado, e portanto conterá sempre zero.)
- *funct*: função. Este campo seleciona uma variação específica da operação apontada no campo op, sendo às vezes chamado de *código de função*.

Codificação MIPs

- Instruções tipo I (transf. De dados)
  - Campo endereço desloca o conteúdo de rs como base
  - Possibilita maior campo de endereçamento (mem. Grande)



- Instrução do tipo J (Jump)



Codificação MIPs

- Instruções de desvio condicional (bne e beq), precisam de 2 operandos, restando apenas 16 bits para o desvio
  - Pode ser pouco para memória grande
  - Usa então desvio relativo em relação ao PC (Contador de Programa), que o processador usa exclusivamente para endereçar o à posição atual do programa
  - End. De destino = PC + desvio

Bytes e Operandos Imediatos

- MIPs é 32 bits, mas às vezes é necessário manipular bytes
  - Instruções para manipular o byte mais à direita de um reg.

```
lb $t0,0($sp)        # Leia o byte a partir do registrador-fonte
sb $t0,0($gp)        # Escreva o byte na memória
```

- À vezes devemos manipular constantes, como um loop com determinado número de iterações
  - Instruções com operandos imediatos (não precisam ser buscados na memória)
  - Passam as constantes nas próprias instruções (16 bits), são as instruções I.

```
addi $sp,$sp,4       # $sp = $sp + 4
slti $t0,$s2,10      # reg $t0 recebe 1 se $s2 < 10
```

Registradores MIPs

Nome	Número do registrador	Utilização	Preservado na chamada?
$zero	0	valor da constante 0	n.-a.
$v0–$v1	2-3	valores para guardar resultados e para avaliar expressões	não
$a0–$a3	4-7	argumentos	sim
$t0–$t7	8-15	temporários	não
$s0–$s7	16-23	salvos	sim
$t8–$t9	24-25	mais temporários	não
$gp	28	global pointer	sim
$sp	29	stack pointer	sim
$fp	30	frame pointer	sim
$ra	31	endereço de retorno a procedimento	sim

Figura 3.13 Convenção do MIPS para utilização de registradores. O registrador 1, chamado $at, é reservado para o montador (veja Seção 3.9), e os registradores 26–27, chamados $k0-$k1, são reservados para o sistema operacional.

Codificação de Instruções

Linguagem de máquina do MIPS

Nome	Formato	Exemplo						Comentários
add	R	0	18	19	17	0	32	add $s1, $s2, $s3
sub	R	0	18	19	17	0	34	sub $s1, $s2, $s3
lw	I	35	18	17	100			lw $s1, 100($s2)
sw	I	43	18	17	100			sw $s1, 100($s2)
beq	I	4	17	18	25			beq $s1, $s2, 100
bne	I	5	17	18	25			bne $s1, $s2, 100
slt	R	0	18	19	17	0	42	slt $s1, $s2, $s3
j	J	2	2500					j 10000 **(veja Seção 3.8)**
Jr	R	0	31	0	0	0	8	jr $ra
jal	J	3	2500					**jal 10000 (veja Seção 3.8)**
Tamanho do campo		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Todas as instruções do MIPS têm 32 bits
Formato R	R	op	rs	rt	rd	shamt	funct	Formato das instruções aritméticas
Formato I	I	op	rs	rt	endereço			Formato das instruções de transferência de dados

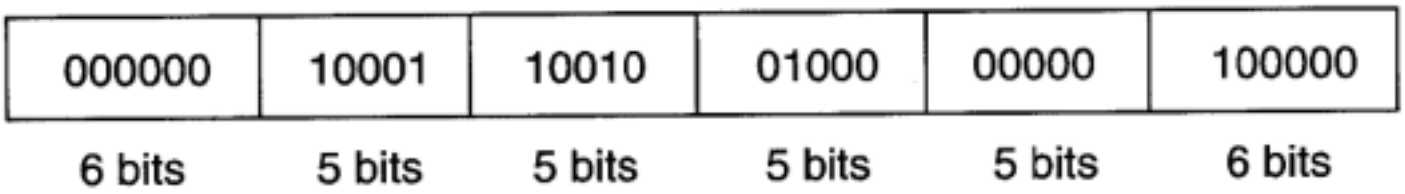
Exemplo

- Instrução

```
add  $t0, $s1, $s2
```

- código do add: 1o campo em 0, último em 32
- 1o reg. Operando: 17 = $s1
- 2o reg. Operando: 18 = $s2
- Reg. Destino: 8 = $t0
- Quinto campo sem uso: Fica com 0

- Em binário



Exemplo

- Em Assembly, assumindo início em 80000

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

- Em ling. De máq.

80000	0	19	19	9	0	32
80004	0	9	9	9	0	32
80008	0	9	22	9	0	32
80012	35	9	8	0		
80016	5	8	21	8		
80020	0	19	20	19	0	32
80024	2	80000				
80028	...					
80012	35	9	8	0		

Exemplo

- Eng. Reversa

Qual a instrução em linguagem de montagem correspondente à seguinte instrução de máquina?

(Bits:	31 28	26				5	2 0)
	0000	0000	1010	1111	1000	0000	0010	0000

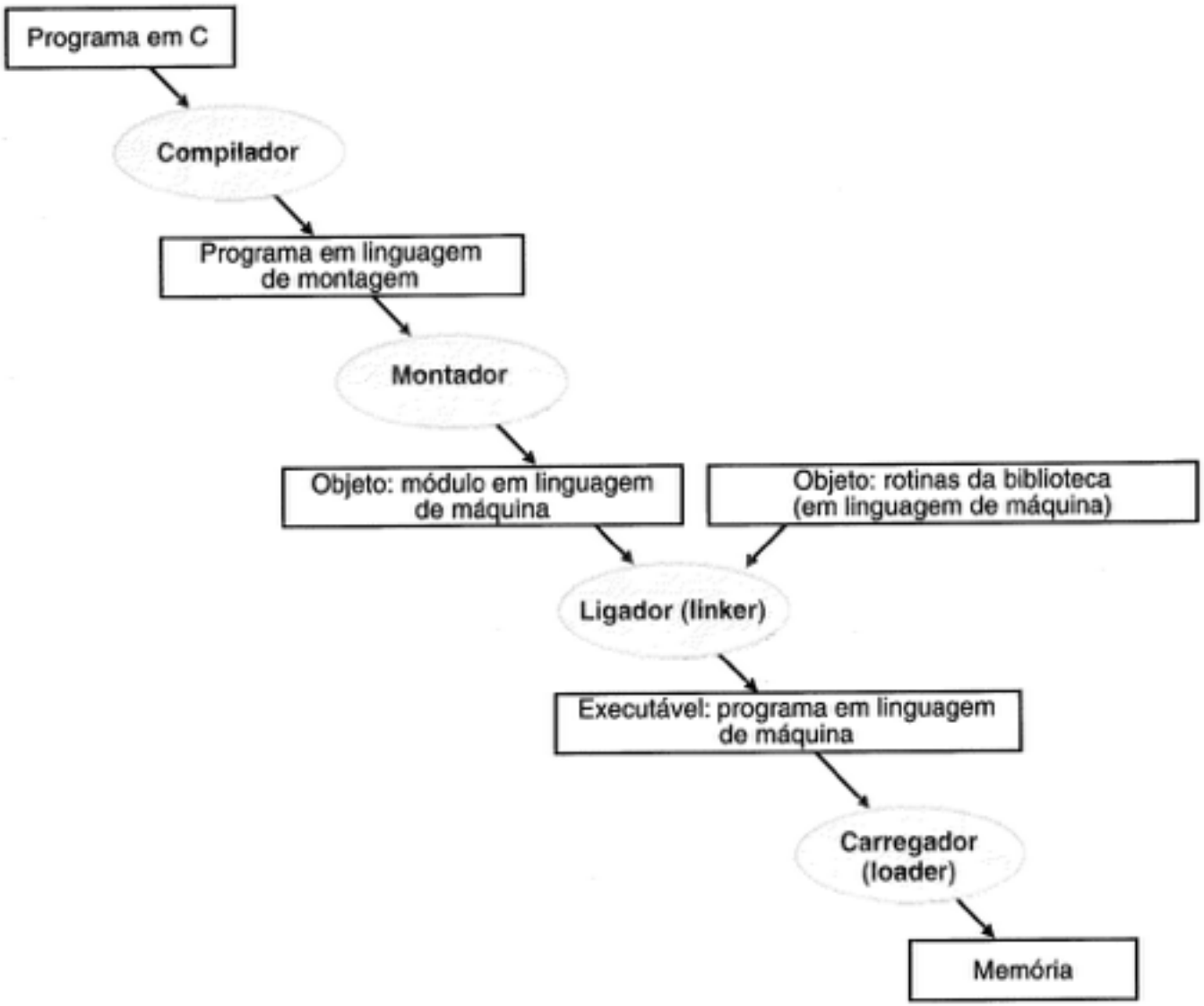
- Começa com 000000, é tipo R

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

- Funct 32, é add
- Basta decodificar os regs.

```
add  $s0,$a1,$t7
```

Executando um programa



Organização Memória do MIPS

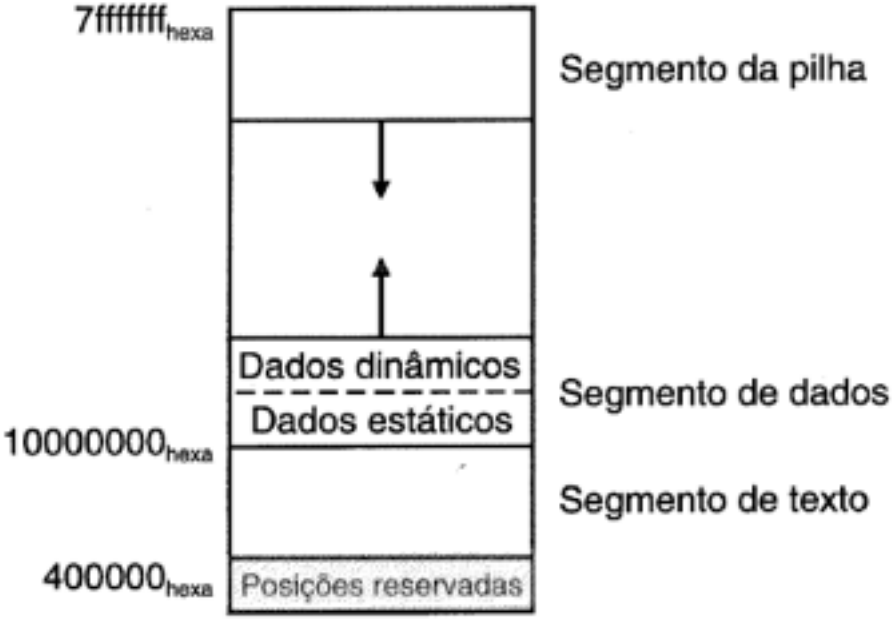
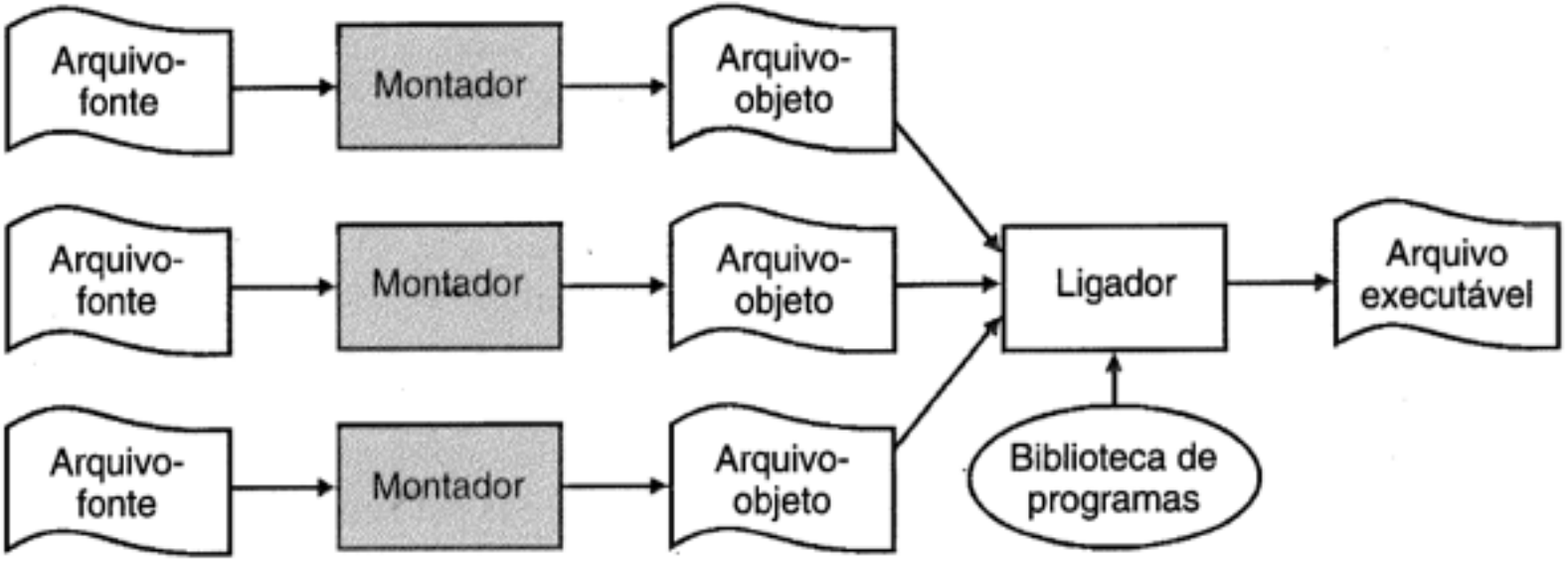


Figura A.9 Layout da memória do MIPS.

Assembly e Montador (Assembler)

- Linguagem de montagem deve trazer legibilidade e facilidades para o programador
  - Algumas diretivas são usadas para interpretação do montador e não aparecem no binário.
  - Ex. Labels são convertidos para endereços



Exemplo

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00000110000010000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

Figura A.2 Código para a rotina que calcula e imprime a soma dos quadrados dos números inteiros entre 0 e 100, expressa na linguagem de máquina do MIPS.

```
addiu   $29, $29, -32
sw      $31, 20($29)
sw      $4,  32($29)
sw      $5,  36($29)
sw      $0,  24($29)
sw      $0,  28($29)
lw      $14, 28($29)
lw      $24, 24($29)
multu   $14, $14
addiu   $8,  $14, 1
slti    $1,  $8, 101
sw      $8,  28($29)
mflo    $15
addu    $25, $24, $15
bne     $1,  $0, -9
sw      $25, 24($29)
lui     $4,  4096
lw      $5,  24($29)
jal     1048 812
addiu   $4,  $4, 1072
lw      $31, 20($29)
addiu   $29, $29, 32
jr      $31
move    $2,  $0
```

Figura A.3 A mesma rotina escrita na linguagem de montagem. O código dessa rotina não permite o uso de símbolos para expressar endereços de memória ou para identificar registradores, nem inclui comentários.

Exemplo

```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sd      $a0, 32($sp)
        sw      $0,  24($sp)
        sw      $0,  28($sp)
loop:
        lw      $t6, 28($sp)
        mul     $t7, $t6, $t6
        lw      $t8, 24($sp)
        addu    $t9, $t8, $t7
        sw      $t9, 24($sp)
        addu    $t0, $t6, 1
        sw      $t0, 28($sp)
        ble     $t0, 100, loop
        la      $a0, str
        lw      $a1, 24($sp)
        jal     printf
        move    $v0, $0
        lw      $ra, 20($sp)
        addu    $sp, $sp, 32
        j       $ra

        .data
        .align  0
str:
        .asciiz "A soma dos quadrados dos
        inteiros de 0 a 100 é %d\n"
```

Figura A.4 A mesma rotina escrita na linguagem de montagem com labels, mas sem comentários. Os comandos que começam com ponto são diretivas do montador (veja, páginas atrás, a seção Sintaxe do Montador). A diretiva `.text` indica que as linhas seguintes contêm instruções. A diretiva `.data` indica que as linhas seguintes contêm dados. A diretiva `.align n` indica que os itens nas linhas seguintes devem ser alinhados em uma fronteira de 2^n bytes. Por exemplo, `.align 2` significa que o próximo item referenciado deve estar em uma fronteira de palavra. A diretiva `.globl main` declara que `main` é um símbolo global, e portanto visível aos códigos de outros arquivos. Finalmente, a diretiva `.asciiz` representa um string de caracteres terminado com o caractere null, armazenado na memória.

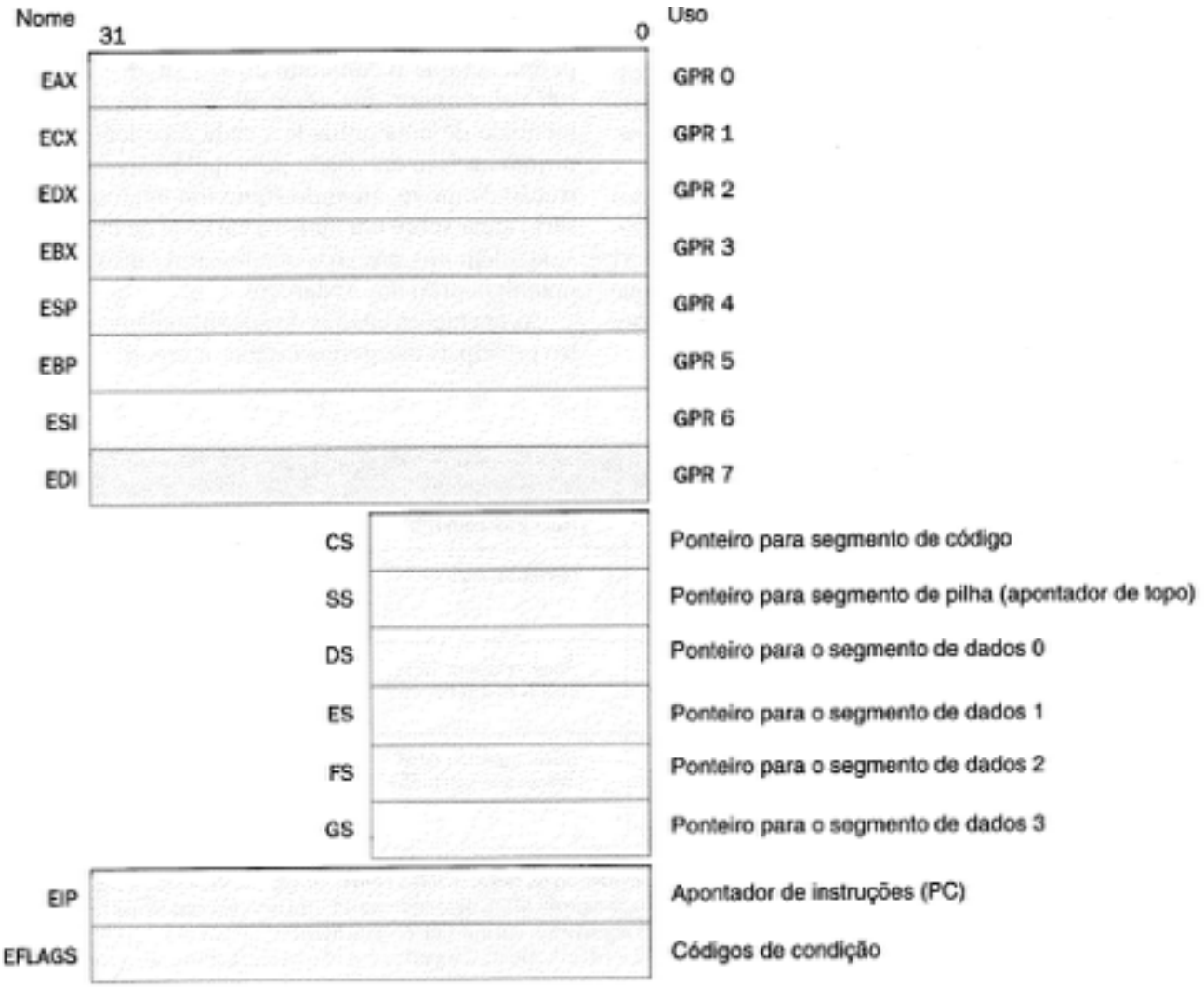
Exemplo mais simples

```
                                                                    # exemplo que faz sum = v[i], com i de 0 a 9
.data                    # define inicia da area de dados do programa
.globl v                 # define variavel global v[10]
V:                                                                  # no endereco de v
.word 2,5,6,7,8,9,8,5,4,3  # inicializa com os valores
.globl sum               # variavel para armazenar resultado
sum:
.word 0
.text                    # define inicio da area de programa da memoria
.globl main              # define proc main como global
main:                    # inicio do main
                         # inicializacoes
sub $sp, $sp,4           # ajusta pilha para guardar 1 word
sw $ra,0($sp)             # guarda end de retorno na pilha p caso de proc
                         # programa que soma elementos do vetor
add $t0,$0,$0            # $t0 sera indice do loop iniciado em 0
lw $t3,sum($0)            # $t3 sera acumulo da soma
addi $t4,$0,4            # sera usado para multiplic por 4
ori $t5,$0,10           # sera usado para limite do loop
loop:                    # inicio do loop
mul $t1,$t0,$t4          # p restricao de alinhamento do vetor v
lw $t2, v($t1)           # le v[i] em $t2
add $t3,$t3,$t2          # sum = sum + v[i]
addi $t0,$t0,1                          # i = i + 1
bne $t0,$t5, loop        # repete loop se i <> 10
sw $t3,sum($0)           # salva variavel sum na memoria (sum e variavel global, longe das instrucoes, mas montador SPIM
                         # resolve o ajuste do endereco colocando instrucoes a mais - VER!)
                         #finalizacoes
lw $ra,0($sp)            # restaura end retorno da pilha
add $sp,$sp,4            # libera pilha
jr $ra                   # retorna pra quem chamou o programa
```

O Caso do X86

- Resultado de um grupo de ideias e evoluções do 8088 (8 bits)
- Possui regs de uso específico e especial
- 386 passou a ser 32 bits, com 32 registradores
- 1997 introduziu instruções especiais MMX
- Muito popular, portanto referência

Regs. Básicos x86



Tipos de endereçamento x86

- Um operando deve atuar como fonte e destino
  - Restringe programador quanto à quantidade de regs.
- Em compensação, um operando direto da memória
  - Evita um load em relação ao MIPs

Tipo de operando-fonte/destino	Segundo operando-fonte
Registrador	Registrador
Registrador	Imediato
Registrador	Memória
Memória	Registrador
Memória	Imediato

Figura 3.30 Combinações de operandos para instruções aritméticas, lógicas e de transferência de dados. A arquitetura 80x86 permite todas as combinações mostradas. A única restrição é o modo memória-memória, não suportado pela arquitetura. As constantes imediatas podem ser de 16 ou de 32 bits. Um registrador é escolhido entre os 14 primeiros apresentados na Figura 3.29 (os registradores EIP e EFLAGS não podem ser escolhidos para armazenar operandos).

X86 vs MIPs

- Aritméricas e lógicas semelhantes
- Movimentos de dados semelhantes
- Desvios condicionais
  - Instruções x86 alteram flags, em um reg. Especial.
  - Desvio são baseados nos flags, portanto, desvios devem vir antecedidas de instruções para ajuste de flag
  - Ex. Verificar se igual: instrução de subtração seguida de instrução que verifica se flag de zero está ativo
- x86 tem instruções de manipulação de strings
  - Pouco usuais, pois não são muito eficientes
- x86 tem codificação de instruções complexa, com diferentes tamanhos e tipos
  - Ruim para desempenho da arquitetura

Instruções típicas x86

Instrução	Significado
Controle	**Desvios condicionais e incondicionais**
JNZ, JZ	Desvia se condição para EIP + deslocamento de 8 bits; JNE (para JNZ), JE (para JZ) são nomes alternativos
JMP	Desvio incondicional — deslocamento de 8 ou de 32 bits
CALL	Chamada a sub-rotina — 16 bits de deslocamento; endereço de retorno é colocado na pilha
RET	Retira o endereço de retorno da pilha e desvia para ele
LOOP	Desvio de loop: testa ECX; desvia para EIP + 8 bits de deslocamento se ECX ≠ 0
Transferência de dados	**Movimento de dados entre a memória e os registradores e entre os registradores e a memória**
MOV	Movimenta dados entre dois registradores ou entre um registrador e a memória
PUSH, POP	Coloca o operando-fonte na pilha; retira o operando do topo da pilha, colocando-o em um registrador
LES	Carrega ES e um dos GPRs a partir da memória
Aritmética, lógica	**Operações aritméticas e lógicas usando registradores e a memória**
ADD, SUB	Soma a fonte com o destino; subtrai a fonte do destino; formato registrador-memória
CMP	Compara fonte com destino; formato registrador-memória
SHL, SHR, RCR	Desloca à esquerda; deslocamento lógico à direita; rotação à direita com bit código de condição de carry preenchido
CBW	Converte byte dos bits mais à direita de EAX para 16 bits, à direita de EAX
TEST	AND lógico do operando-fonte e do operando-destino com alteração do código de condição
INC, DEC	Incrementa o destino, decrementa o destino; formato registrador-memória
OR, XOR	OR lógico; OR exclusivo: formato registrador-memória
String	**Movimenta operandos string; tamanho é dado pelo prefixo de repetição**
MOVS	Copia a fonte para o destino incrementando ESI e EDI; pode ser repetido
LODS	Carrega um byte, uma palavra ou uma palavra dupla de um string para o registrador EAX

Linguagem e desempenho

- Um importante objetivo de projetistas de computadores é definir um conjunto de instruções que sejam executadas de forma eficiente, melhorando desempenho.
- Aspectos de custo, complexidade de Hw e facilidade de utilização (programadores e compiladores) também são importantes.
- A última década mostrou que quanto mais simples o processador melhor
  - Instruções mais complexas normalmente não proporcionam melhor desempenho
  - A evolução das linguagens de alto nível e dos compiladores ajudam essa visão
  - Mais complexidade leva a mais circuitos, causando maior aquecimento do processador e obrigando o aumento dos ciclos de clock