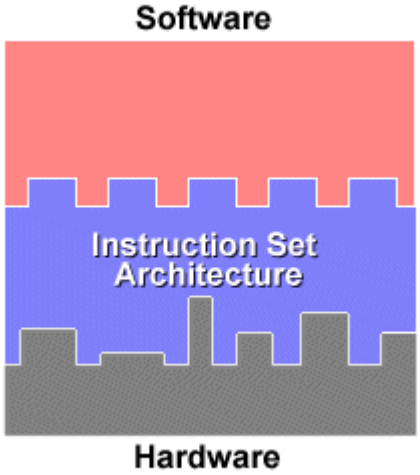
CAPÍTULO 3
CONJUNTO DE INSTRUÇÕES

- Introdução
- Classificação de Arquiteturas de Conjuntos de Instruções
- Exemplo de uma arquitetura com acumulador
- Combinações Típicas de Operandos
- Endereçamento de Memória
- Modos de Endereçamento
- Modos de endereçamento do Intel 8051
- Harvard vs Von Neumann
- Tipos e Tamanhos de Operandos
- Instruções para Fluxo de Controle
- Operações no Conjunto de Instruções
- Codificação de Um Conjunto de Instruções
- Exemplo: Intel 80x86
- A arquitetura MIPS
- Codificando Instruções MIPS
- Exemplos
- Assembly e Linguagem de Máquina
- Montador e Link-editor
- Carga de um Programa
- Uso da memória
- Chamada de Procedimento e convenções
- Exemplos de procedimentos recursivos

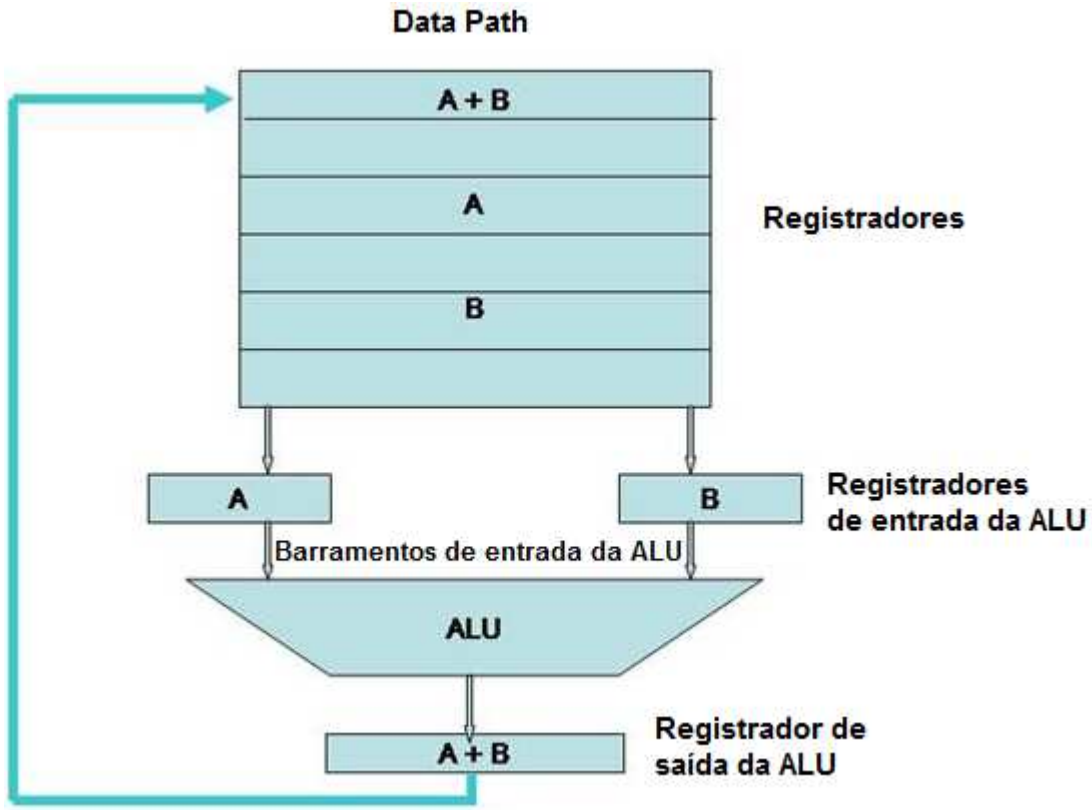
Introdução



- O Conjunto de Instruções é a parte do computador visível para o programador ou para o criador de compiladores.
- Define a fronteira entre o Software e o Hardware
- Um exemplo de arquitetura bem sucedida é a 80x86, que é CISC
- Com a enorme integração de transistores atual, os conjuntos RISC conseguem ser mais eficientes.
- Alguns processadores Intel são RISC internamente mas com uma tradução do 80x86 por hardware (compatibilidade).

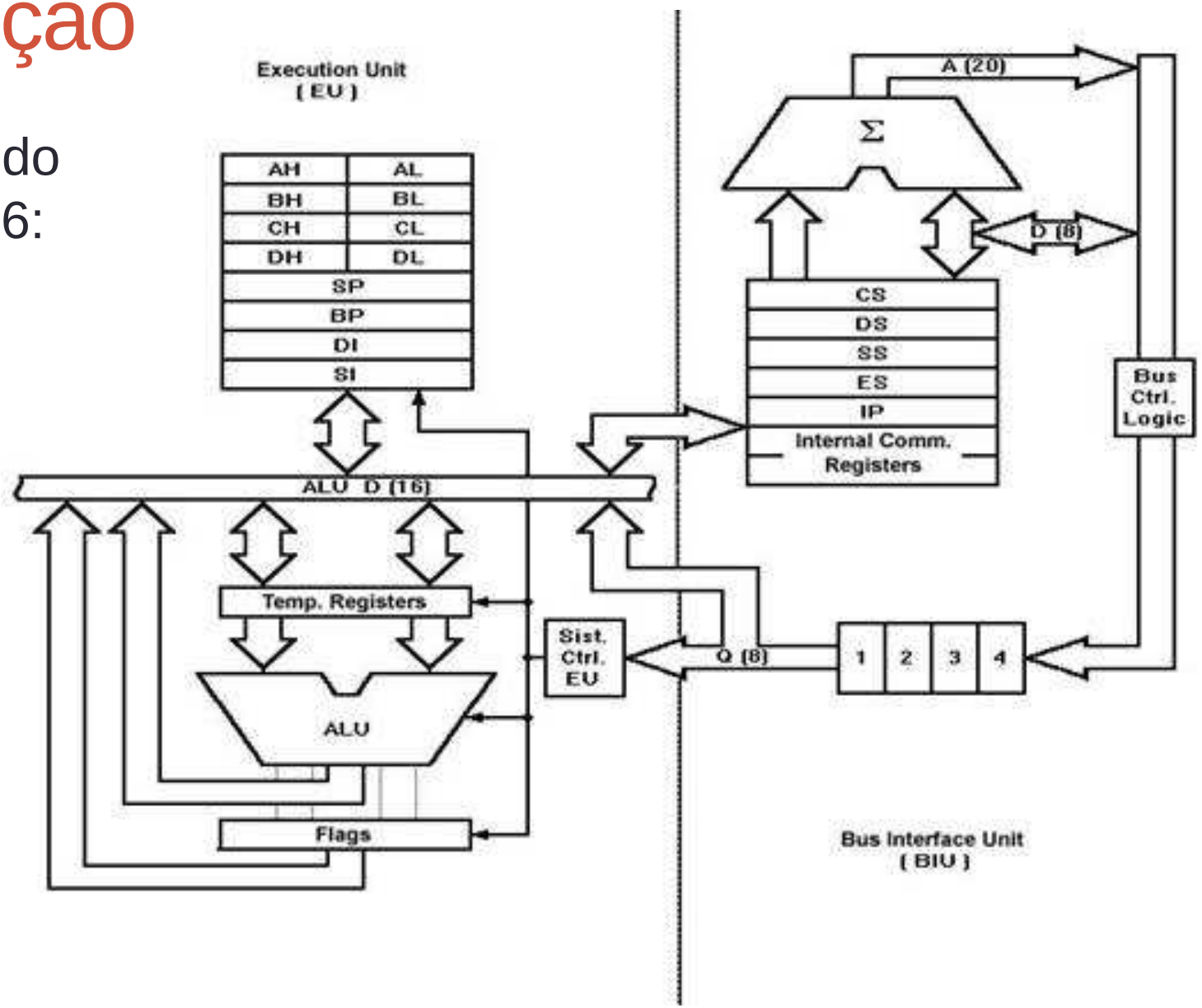
Introdução

- O conjunto de instruções é o que permite a movimentação de dados e execução de operações no processador, pelo Data Path:



Introdução

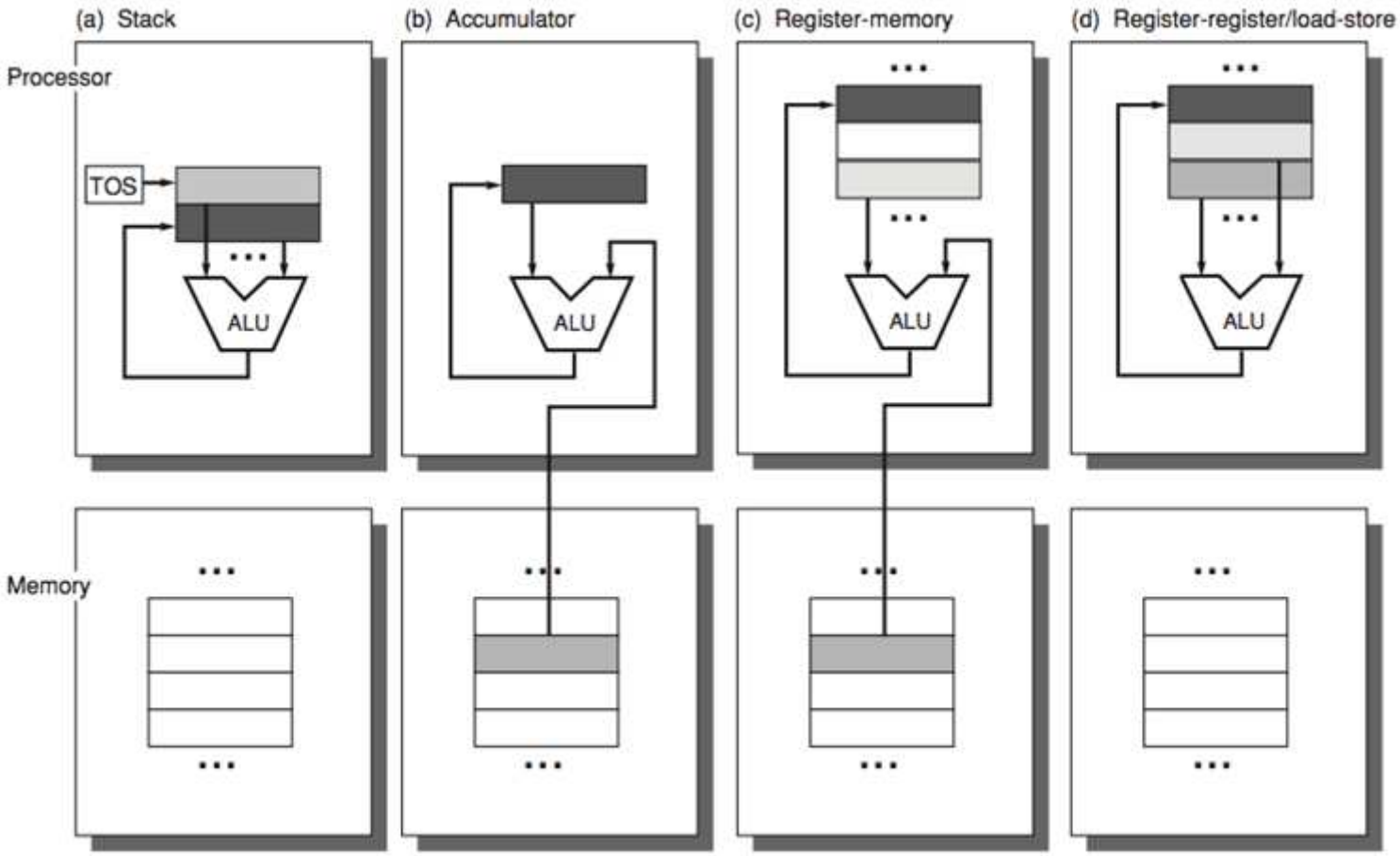
- Datapath do Intel 80x86:



Classificação de Arquiteturas de Conjuntos de Instruções

- O que diferencia os conjuntos de instruções é o tipo de armazenamento interno
- Tipos de ISA:
  - Pilha
  - Acumulador
  - Registrador-memória
  - Registrador-registrador (*load-store*)
- Os operandos podem ser nomeados de forma implícita ou explícita
- Arquiteturas com registradores usam a forma explícita
- Arquiteturas com pilha ou acumulador usam a forma implícita

Classificação de Arquiteturas de Conjuntos de Instruções



Classificação de Arquiteturas de Conjuntos de Instruções

- Compilação da operação C=A+B nas diferentes classes de arquitetura de conjunto de instruções:
- Inicialmente (antes de 1980) eram mais usados os conjuntos tipo pilha e acumuladores. Atualmente a arquitetura *load-store* é preferida

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

Exemplo: Motorola MC6800 (1974)

- Tipo acumulador
- Exemplo de código:

```
; memcpy --
; Copy a block of memory from one location to another.
;
; Entry parameters
;      cnt - Number of bytes to copy
;      src - Address of source data block
;      dst - Address of target data block

cnt         dw      $0000
src         dw      $0000
dst         dw      $0000

memcpy      public
            ldab    cnt+1       ;Set B = cnt.L
            beq     check       ;If cnt.L=0, goto check
loop        ldx     src         ;Set IX = src
            ldaa    ix          ;Load A from (src)
            inx                 ;Set src = src+1
            stx     src
            ldx     dst         ;Set IX = dst
            staa    ix          ;Store A to (dst)
            inx                 ;Set dst = dst+1
            stx     dst
            decb                ;Decr B
            bne     loop        ;Repeat the loop
            stab    cnt+1       ;Set cnt.L = 0
check       tst     cnt+0       ;If cnt.H=0,
            beq     done        ;Then quit
            dec     cnt+0       ;Decr cnt.H
            decb                ;Decr B
            bra     loop        ;Repeat the loop
done        rts                 ;Return
```

Combinações Típicas de Operandos

Número de endereços de memória	Número máximo de operandos permitidos	Tipo de arquitetura	Exemplos
0	3	registrador-registrador	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, Trimedia TM5200
1	2	registrador-memória	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	memória-memória	VAX (também tem formatos de 3 operandos)
3	3	memória-memória	VAX (também tem formatos de 2 operandos)

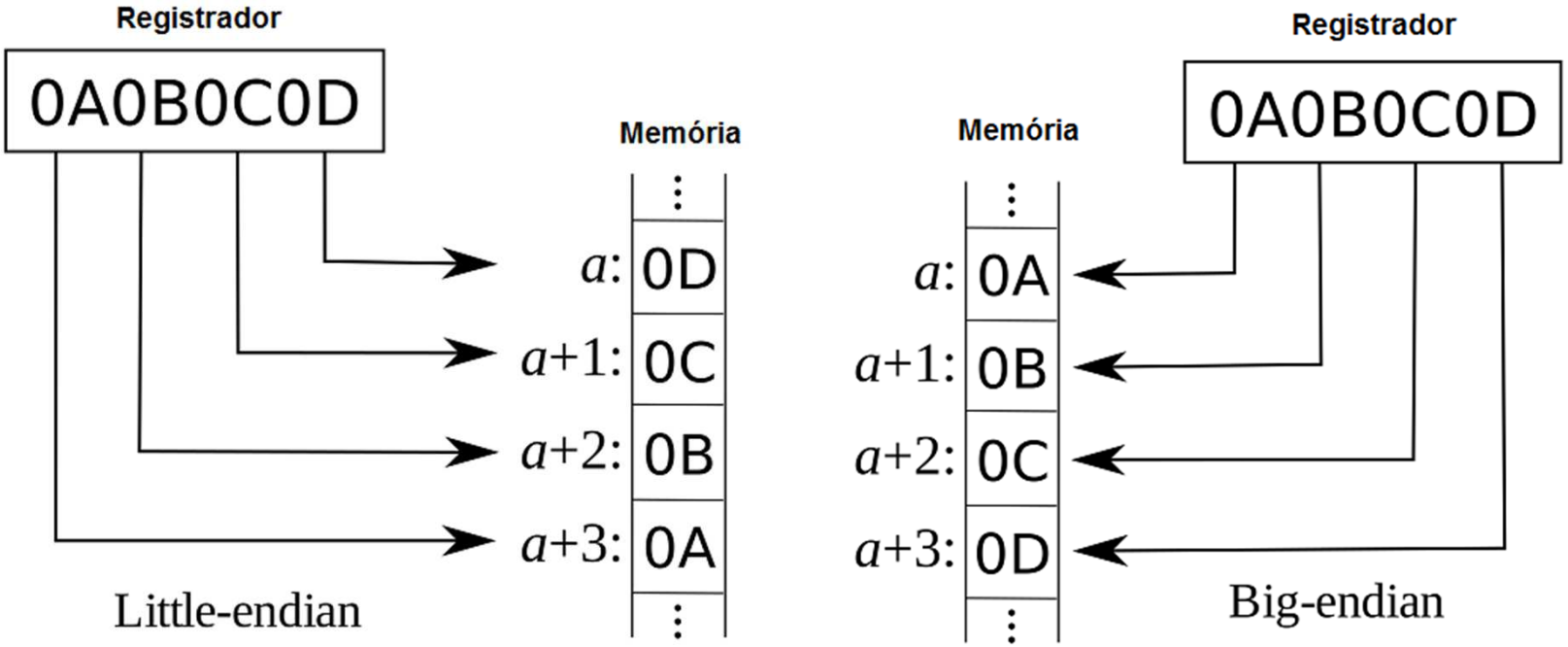
Vantagens/Desvantagens

Tipo	Vantagens	Desvantagens
registrador-registrador (0,3)	Instruções simples de comprimento fixo. Gera códigos simples. CPI não varia muito	Contagem de instruções elevada em relação ao registrador-memória. Gera programas maiores
registrador-memória (1,2)	Não requer instruções load/store. O formato das instruções tende a ser fácil de codificar e resulta em uma boa densidade.	Os operandos não são equivalentes, pois um dos operandos fonte é destruído numa operação binária. A codificação de um número de registrador e um endereço de memória pode restringir o número de registradores. O CPI pode variar de acordo com a posição do operando.
memória-memória (2,2) ou (3,3)	Mais compacto. Não desperdiça registradores com itens temporários	Variação no tamanho das instruções. Grande variação no trabalho por instrução. Acessos a memória criam gargalo.

- (m,n) = m operandos de memória e n operandos totais
- Estas vantagens e desvantagens não são absolutas. O impacto real depende do compilador

Endereçamento de Memória

- Interpretação de endereços de memória
  - A arquitetura do conjunto de instruções deve definir como um endereço de memória é interpretado: endereçamento por byte é o mais comum, mas palavras maiores podem ser acessadas
  - Colocação do dado na memória. **Little Endian** vs **Big Endian**
  - Problema do alinhamento de dados



Modos de Endereçamento

- Como os endereços são especificados nas instruções?

Modo de endereça-mento	Exemplo de instrução	Significado	Quando é usado
Registrador	$Add\ R4, R3$	$R4 \leftarrow R4 + R3$	Quando um valor está em um registrador
Imediato	$Add\ R4, \#3$	$R4 \leftarrow R4 + 3$	Para constantes
Registrador indireto	$Add\ R4, (R1)$	$R4 \leftarrow R4 + Mem[R1]$	No acesso com uso de um ponteiro ou endereço calculado
Direto ou absoluto	$Add\ R4, (1001)$	$R4 \leftarrow R4 + Mem[1001]$	Acesso a dados estáticos, porém talvez a constante de endereço precise ser grande
Deslocamento	$Add\ R4,100(R1)$	$R4 \leftarrow R4 + Mem[R1 + 100]$	Acesso a variáveis locais, além de simular registrador indireto e direto
Indexado	$Add\ R4, (R1 + R2)$	$R4 \leftarrow R4 + Mem[R1 + R2]$	Útil no endereçamento de vetores: R1 é a base do vetor e R2 o índice do elemento
Memória Indireto	$Add\ R4, @(R1)$	$R4 \leftarrow R4 + Mem[Mem[R1]]$	Se R1 é o endereço de um ponteiro p, então este modo resulta em *p.

Modos de Endereçamento (cont.)

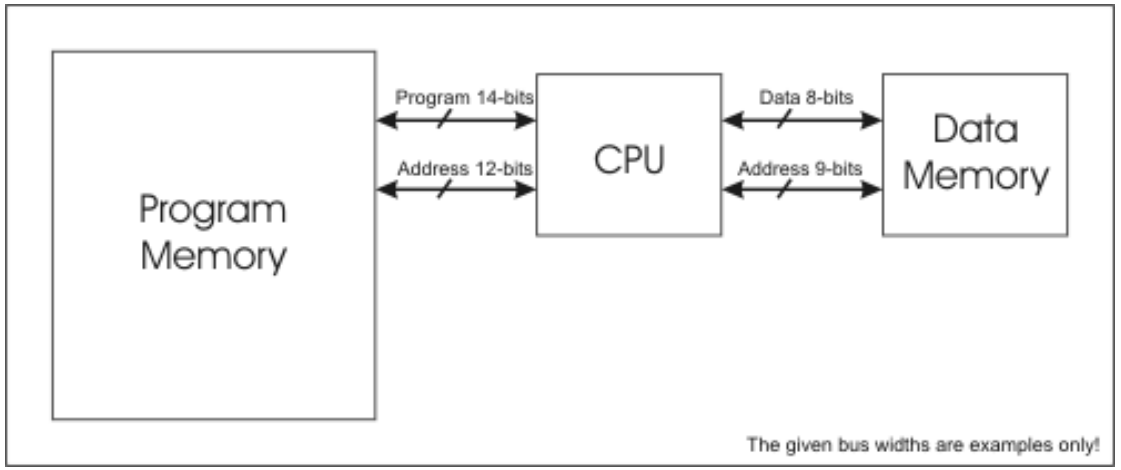
Modo de endereça-mento	Exemplo de instrução	Significado	Quando é usado
Auto- -incremento	$Add\ R1,(R2)+$	$R1 \leftarrow R1 + Mem[R2]$ $R2 \leftarrow R2 + d$	Útil para percorrer um vetor dentro de um loop. R2 aponta para o início do vetor. Cada referência incrementa R2 pelo tamanho de um elemento, d.
Auto-decremento	$Add\ R1,-(R2)$	$R2 \leftarrow R2 - d$ $R1 \leftarrow R1 + Mem[R2]$	Mesmo uso do auto-incremento. Os dois modos podem ser usados juntos para se implementar uma pilha (operações *push* e *pop*)
Escalonado	$Add\ R1,100(R2)[R3]$	$R1 \leftarrow R1 + Mem\begin{bmatrix}100 + R2 + \\ R3 * d\end{bmatrix}$	Usado também para indexar vetores.

- Pontos importantes de projeto: escolha do tamanho de deslocamento e intervalo de valores para imediatos e diretos, pois afetam o tamanho da instrução
- Normalmente, em um programa, o endereçamento imediato e de deslocamento dominam a utilização de modos de endereçamento.

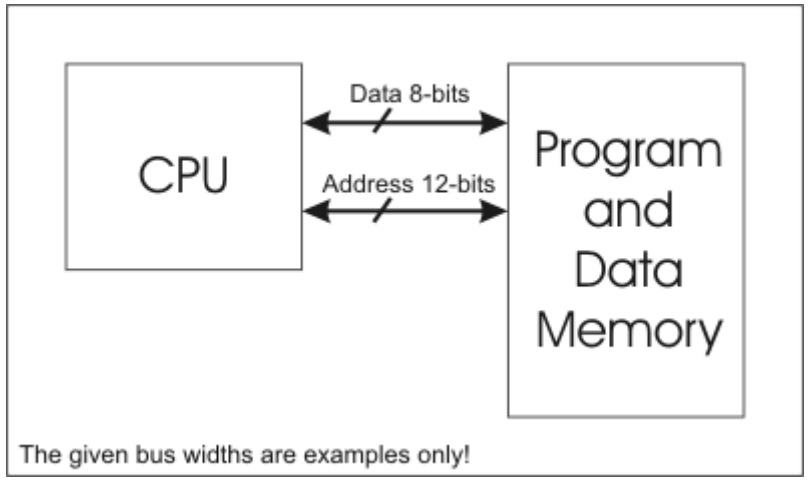
Exemplo: Intel 8051

- Microcontrolador com arquitetura harvard e conjunto de instruções CISC com acumulador.
- Versão original lançada em 1980.
- Algumas variações do original são usadas até hoje, por outros fabricantes
- Algumas empresas oferecem o 8051 como bloco IP (Intellectual Property Block) para uso em projetos FPGA e ASICs.
- Podem custar apenas US$0,25
- Performance 35 MIPS
- Mais de 2 bilhões de chips vendidos
- O 8051 ainda é utilizado
- Possui os modos de endereçamento: imediato, direto, registrador indireto, registrador direto (modo registrador) e indexado

Harvard vs Von Neumann



Harvard



Von Newmann

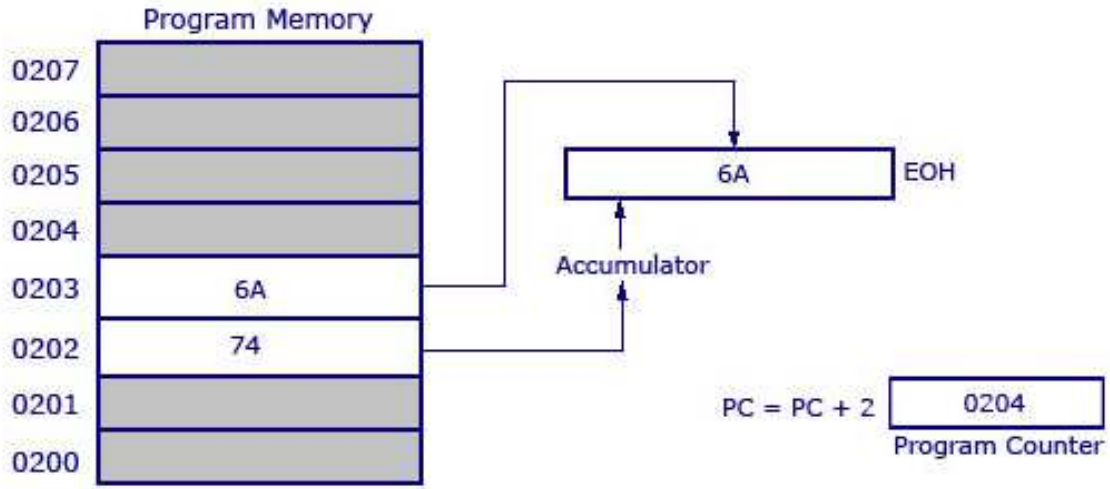
Exemplo Intel 8051

fonte: 8051 Addressing modes
http://www.circuitstoday.com/8051-addressing-modes

Modo de Endereçamento Imediato

- Transfere 8 bits de dados codificados na instrução para o acumulador

Instruction	Opcode	Bytes	Cycles
MOV A, #6AH	74H	2	1

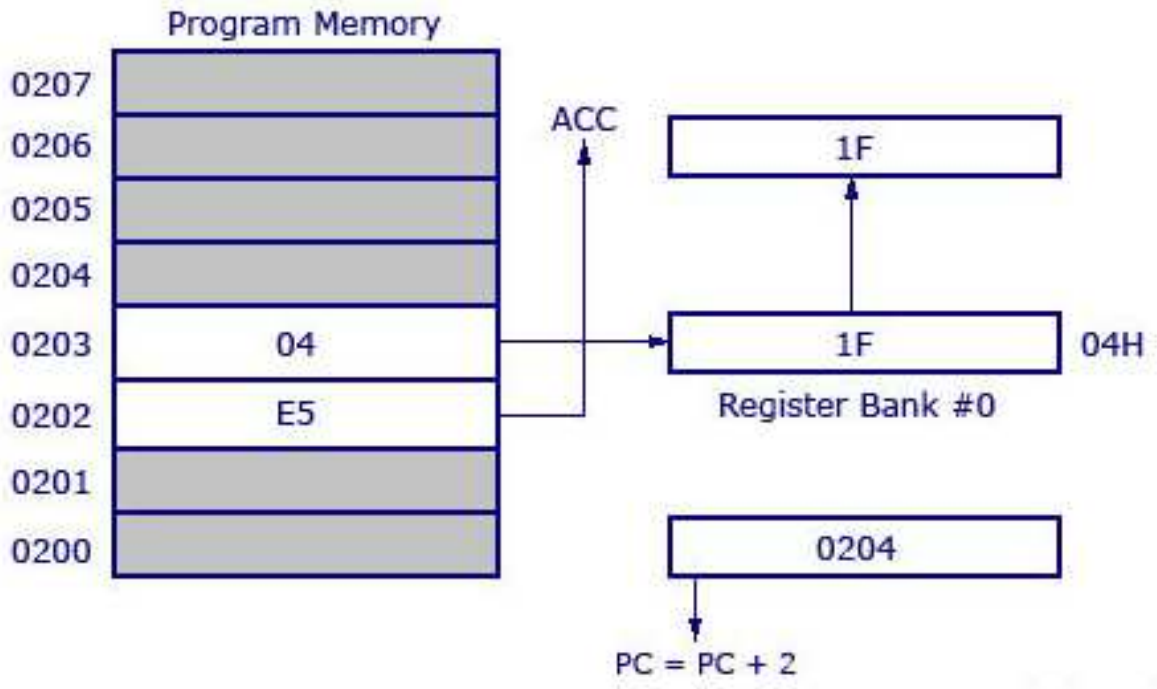


Exemplo Intel 8051: Modo de Endereçamento Direto

- O 8051 possui 32 registradores divididos em 4 bancos, mapeados na memória

Instruction	Opcode	Bytes	Cycles
MOV A, #04H	E5	2	1

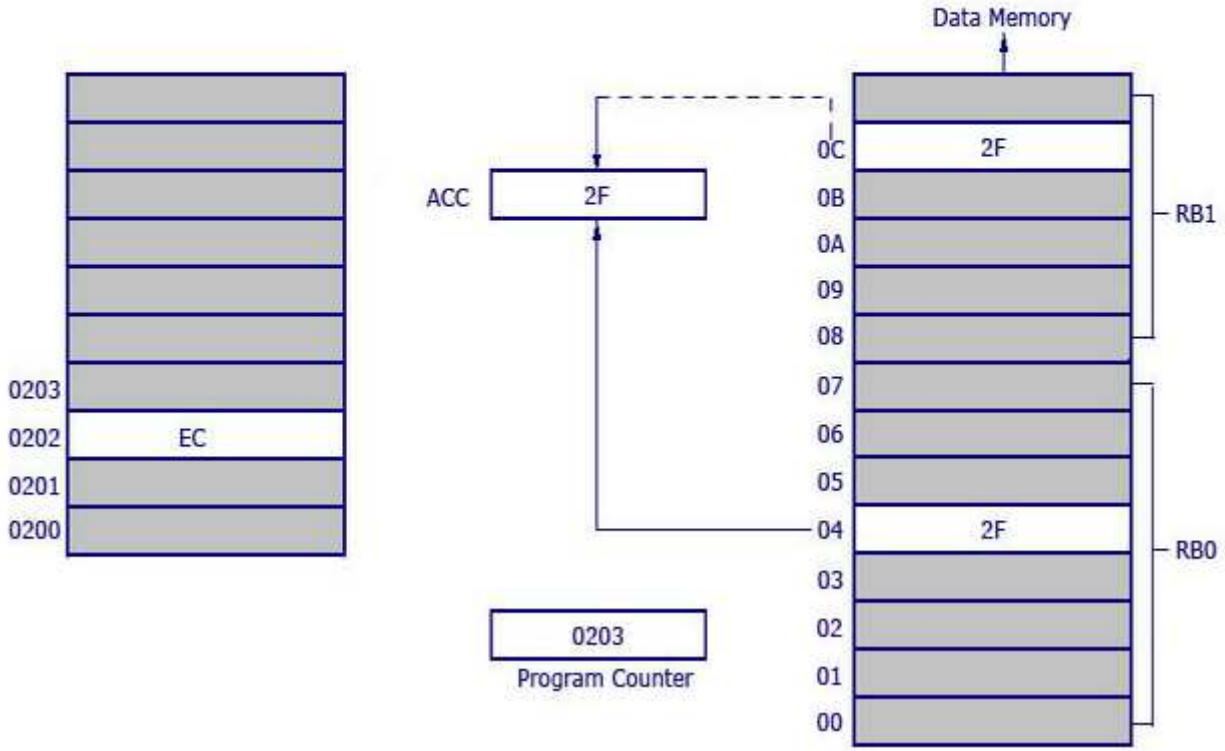
- 04H é o endereço do registrador 4 no banco 0. Esta instrução move o dado que estiver no registrador 04h para o acumulador. No exemplo o dado 1FH está no registrador 04H.



Exemplo Intel 8051: Modo de Endereçamento Registrador Direto

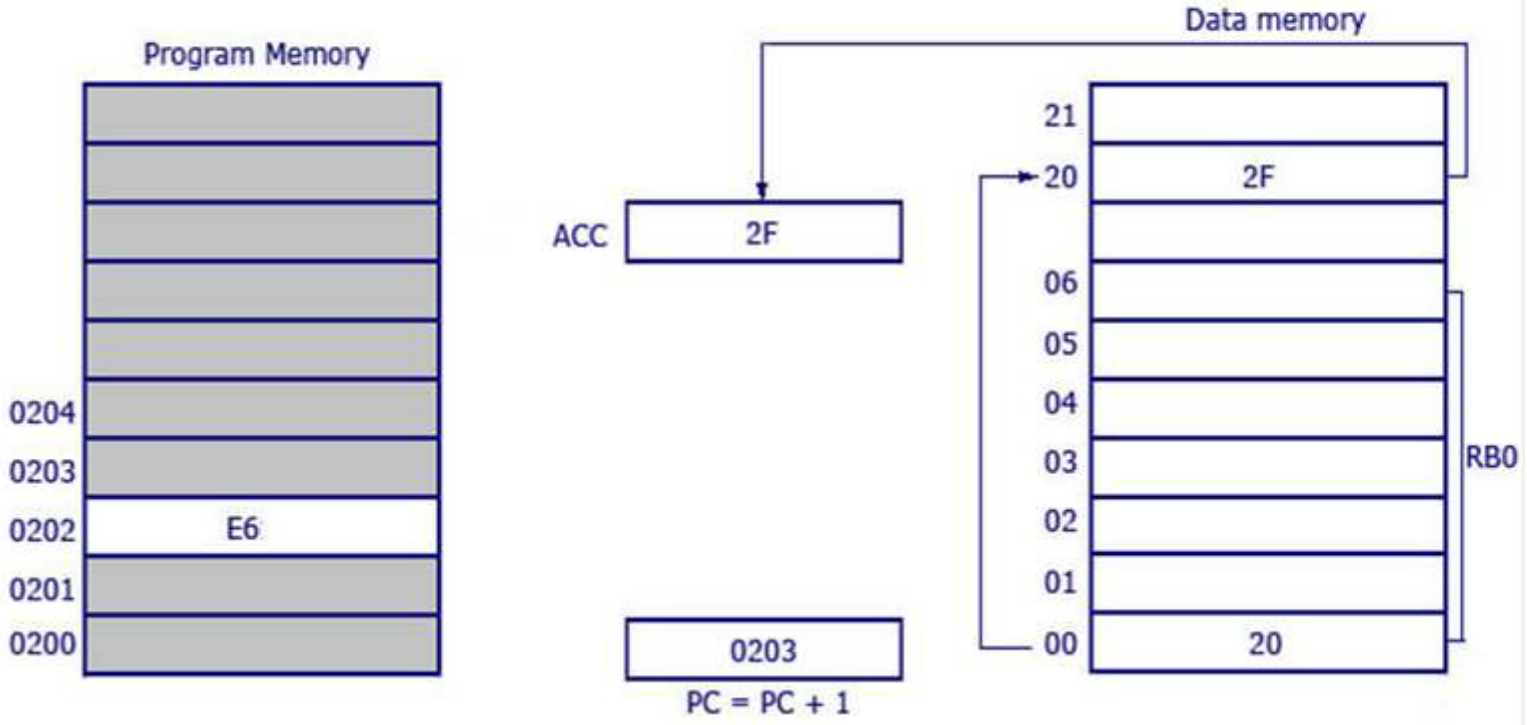
- Este modo se enquadra no modo Registrador, pois transfere um dado de um registrador para o acumulador
- Necessita de 2 bits da palavra de status para programar um dos 4 bancos de registradores
- O opcode é 11101nnn para Rn

Instruction	Opcode	Bytes	Cycles
MOV A, R4	ECH	1	1



Exemplo Intel 8051: Modo de Endereçamento Registrador Indireto

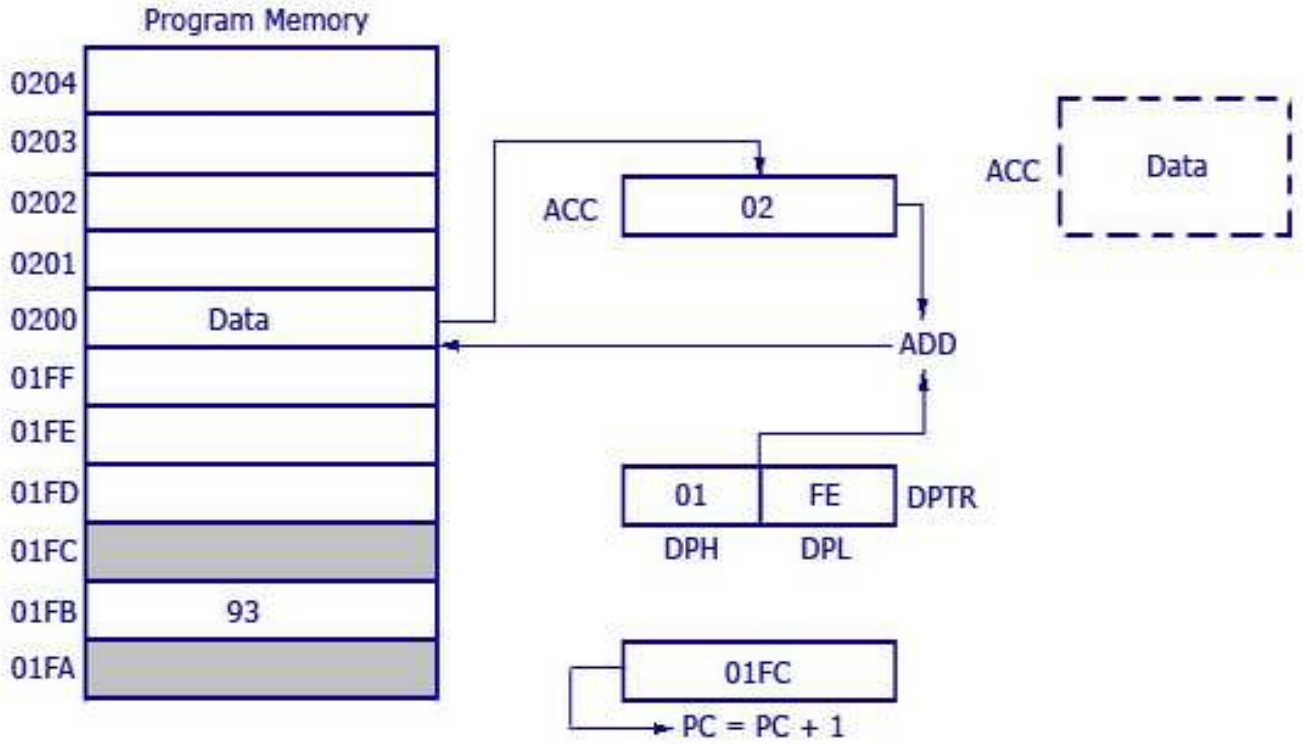
Instruction	Opcode	Bytes	Cycles
MOV A, @ R0	E6H	1	1



Exemplo Intel 8051: Modo de Endereçamento Indexado

- DPTR (*data pointer*) e PC (*program counter*) são dois registradores de 16 bits.
- O valor 01FE do data pointer será somado com 02H do acumulador obtendo-se o valor 0200 que será o endereço do dado a ser transferido para o acumulador

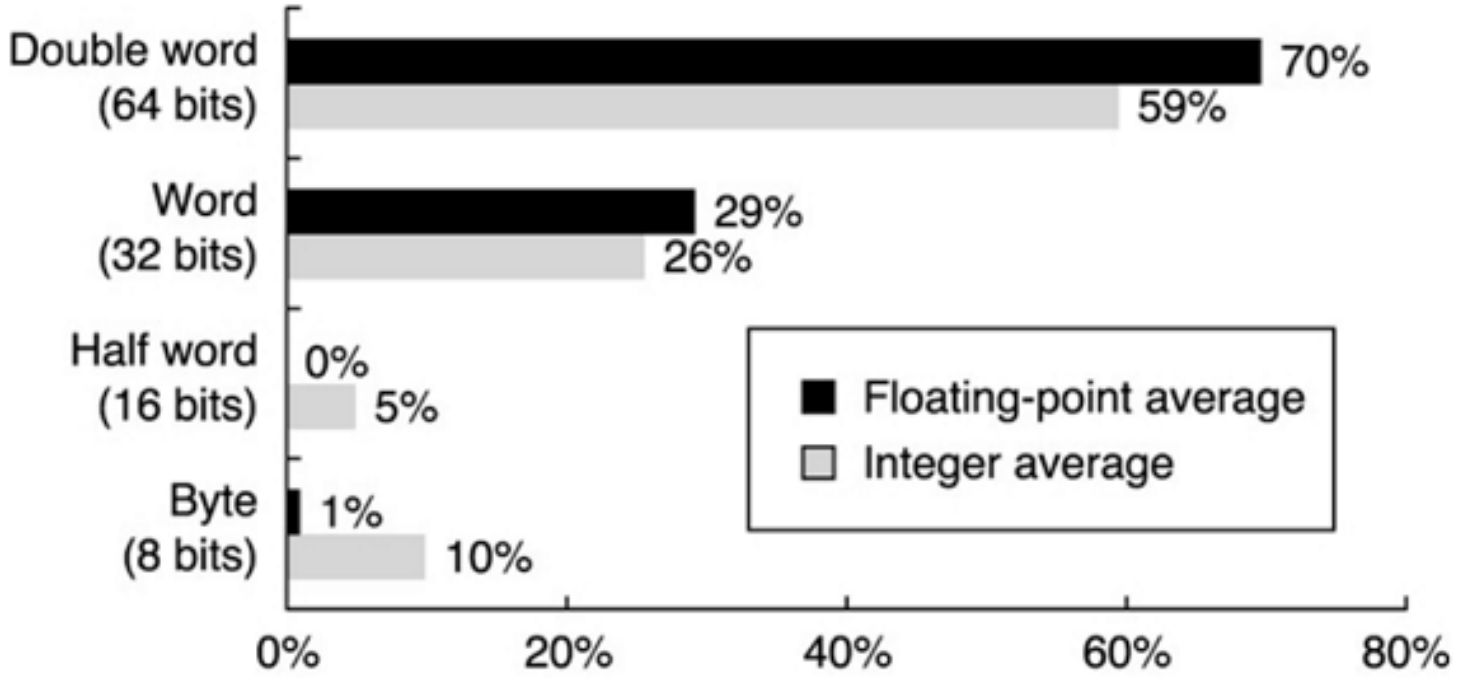
Instruction	Opcode	Bytes	Cycles
MOVC A,@A +DPTR	93H	1	2



Tipos e Tamanhos de Operandos

- O tipo do operando é codificado no opcode, isto é, existe uma instrução para cada tipo de dado
- O tamanho do dado influencia na CPI da instrução
- Os tipos comuns são: caracteres (8 bits), meia palavra (16 bits), palavra (32 bits) ponto flutuante precisão simples (32 bits) e ponto flutuante de precisão dupla (64 bits).
- Os inteiros são codificados em binário complemento de dois, os caracteres em ASCII ou Unicode de 16 bits (usado em Java) e IEEE 754 para ponto flutuante
- Existem arquiteturas com instruções para strings, como comparações e movimentações
- Algumas arquiteturas adotam o BCD (Decimal Codificado em Binário) visando aplicações de negócios
- Uma vantagem do BCD é evitar dízimas periódicas em binário.

Tipos e Tamanhos de Operandos

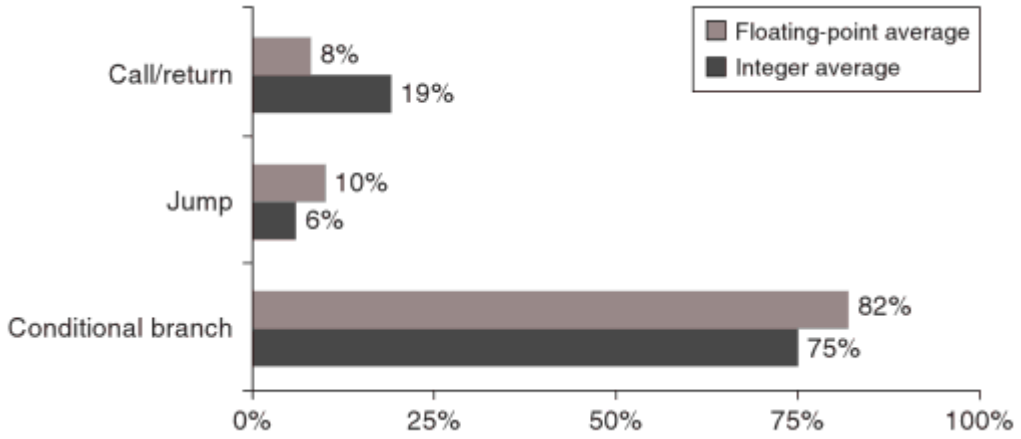


- Resultados do benchmark SPEC usando as referências de memória para examinar os tipos de dados que estão sendo acessados
- Esse tipo de análise ajuda a decidir quais tipos são mais importantes para se oferecer acesso mais eficiente (Lei de Amdahl)

Instruções para Fluxo de Controle

- Existem 4 tipos de instruções que mudam o fluxo de controle:
  - Desvios condicionais
  - Saltos
  - Chamadas de Procedimentos
  - Retorno de procedimentos
- O endereço de destino é especificado diretamente na instrução (endereçamento imediato)
- O deslocamento é relativo ao PC.

Frequência das instruções tipo branch com SPECInt e SPECFP



Operações no Conjunto de Instruções

Tipo de Instrução	Exemplos
Aritmética e Lógica	Operações aritméticos de inteiros e operações lógicas: adição, subtração, multiplicação, divisão, e lógico, ou lógico, deslocamento.
Transferência de dados	Instruções *load/store* para busca e salvamento de dados na memória
Controle	Desvio, salto, chamada e retorno de procedimento, armadilhas (interrupções)
Sistema	Chamadas de sistema operacional, instruções de gerenciamento de memória virtual
Ponto flutuante	Adição, subtração, multiplicação, divisão para ponto flutuante
Decimal	Instruções específicas para decimais (BCD)
Strings	Movimentação, comparação e pesquisa de strings
Gráficos	Operações de pixel e vértices, compactação e descompactação gráficas

Operações no Conjunto de Instruções

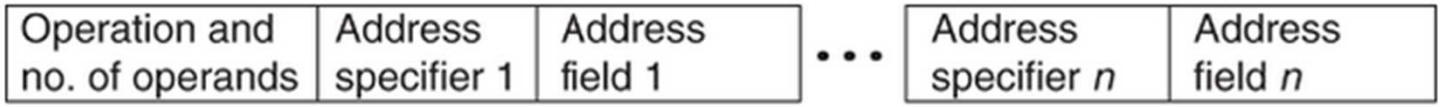
- Num programa geralmente a maior parte do programa consiste de instruções mais simples. Um exemplo para o 80x86:

Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		**96%**

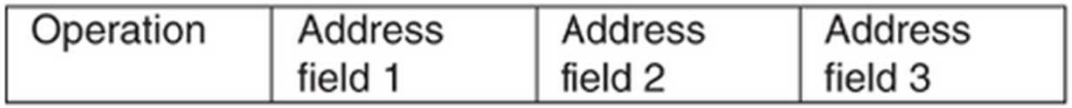
Codificação de Um Conjunto de Instruções

- Campo opcode
- A arquitetura do conjunto de instruções define como serão implementadas as instruções
- O número de registradores e o número de modos de endereçamento têm um impacto significativo.
- O que influencia o projeto:
  - Desejo de aumentar o número de registradores e modos de endereçamento
  - Impacto do tamanho dos campos
  - Preocupação com o pipeline
  - Preferência por instruções de tamanho fixo

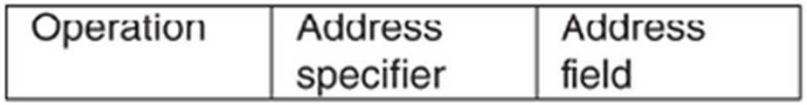
Codificação de Um Conjunto de Instruções

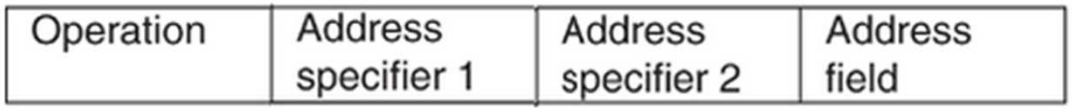


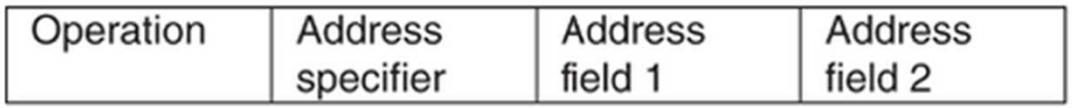
(a) Variável (ex.: Intel 80x86, VAX)



(b) Fixo (ex.: Alpha, ARM, MIPS, PowerPC, SPARC, SuperH



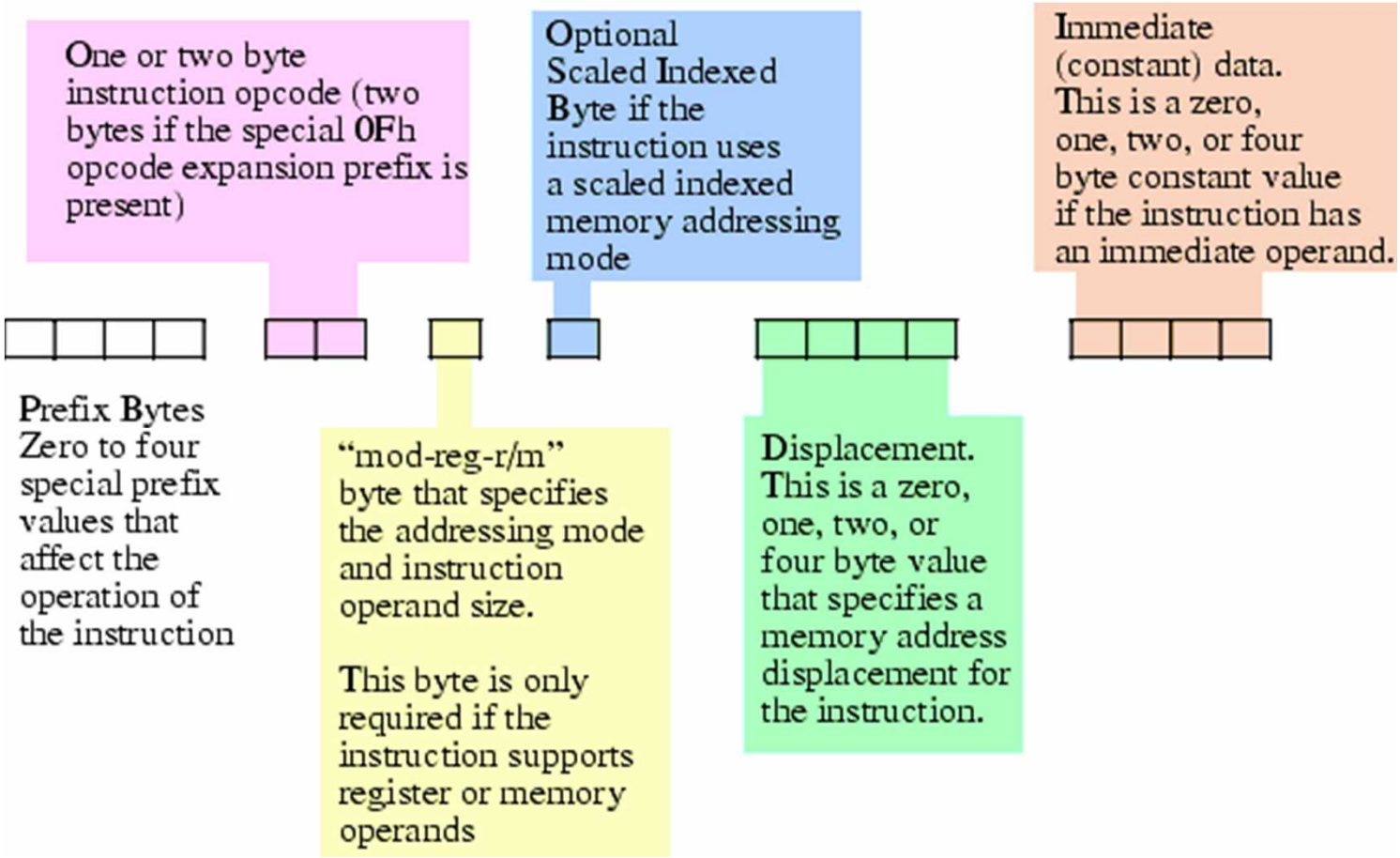




(c) Híbrido (ex.: IBM 360/370, MIPS16, Thumb, TI TMS320C54x)

Exemplo: 80x86

- Instruções de até 15 bytes (embora o diagrama mostre até 16)

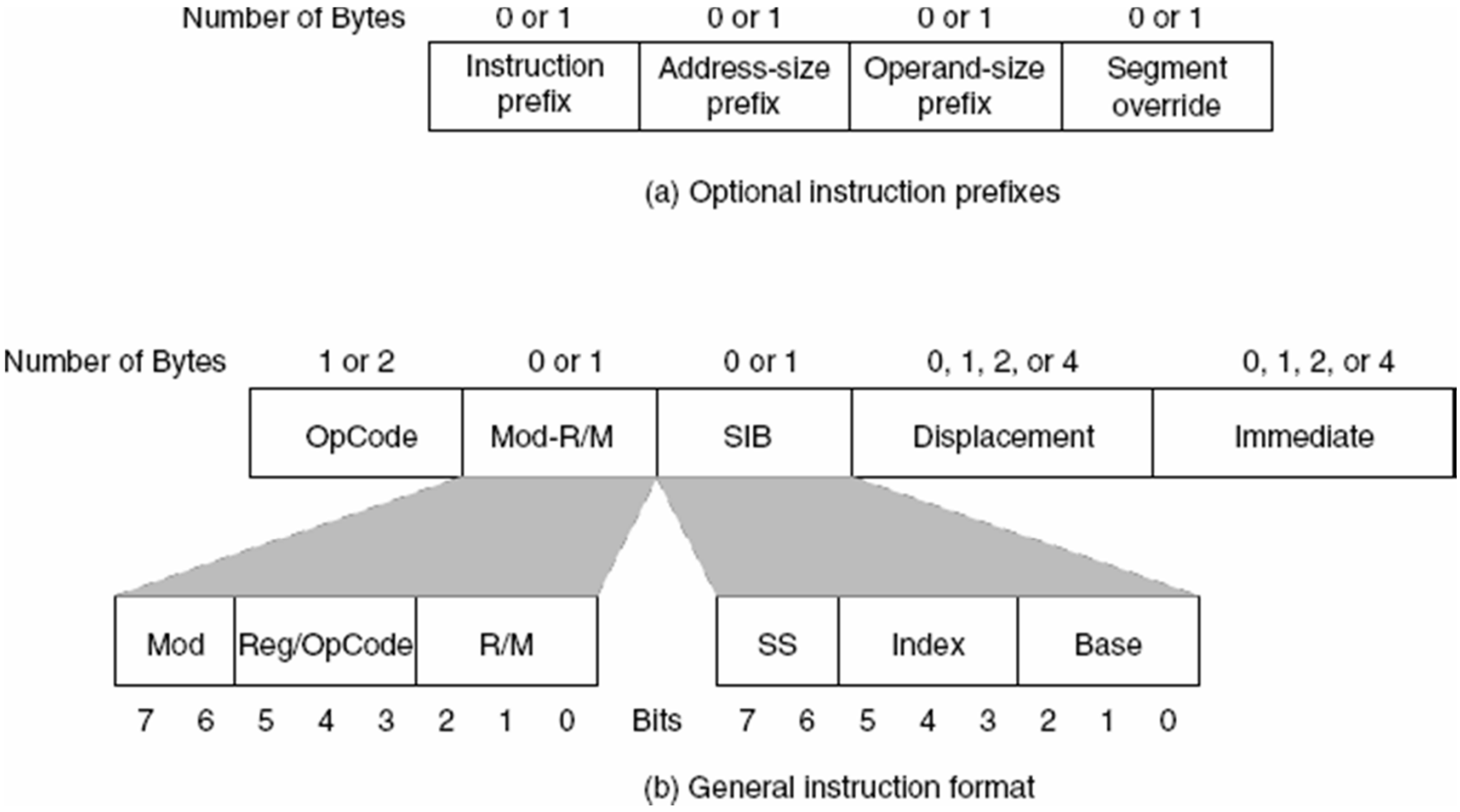


Exemplo: 80x86

- O opcode possui 1 ou 2 bytes. Para usar 2 bytes, o primeiro deve ser 0FH, expandindo para até 512 instruções
- As instruções usam uma combinação dos campos, não necessariamente todos:
  - instruction prefix – opçoes que afetam a operação da instrução
  - opcode - especifica a operação
  - Mod R/M - especifica o modo de endereçamento
  - SIB (scale index base) - usado em índice de arrays
  - address displacement - usado para endereçar a memória
  - immediate value - contém o valor de um operando constante

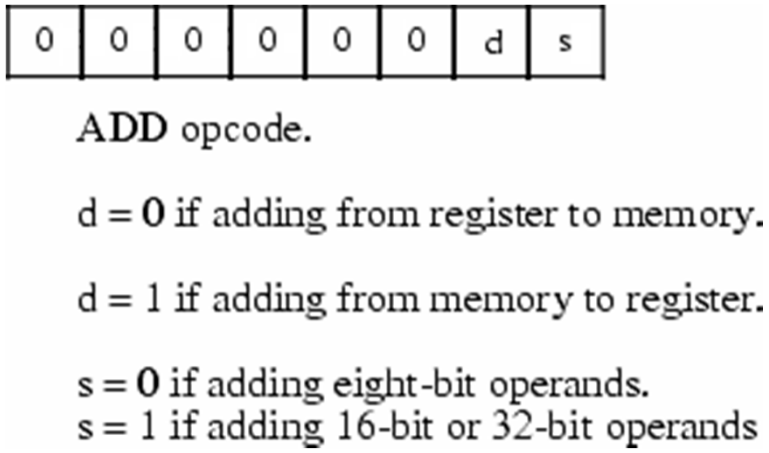
Exemplo: 80x86

- O formato visto de outra forma:



Exemplo: 80x86

- Opcode de uma instrução ADD:



- O bit **d**, especifica a **direção** da transferência de dados:
  - Se **d = 0** o operando destinato é uma localização de memória. Por exemplo:

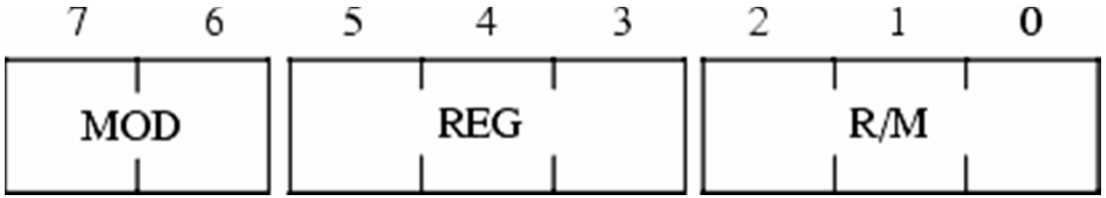
```
add [ebx], al
```

  - Se **d = 1** o operando destinato é um registrador. Por exemplo:

```
add al, [ebx]
```

Exemplo: 80x86

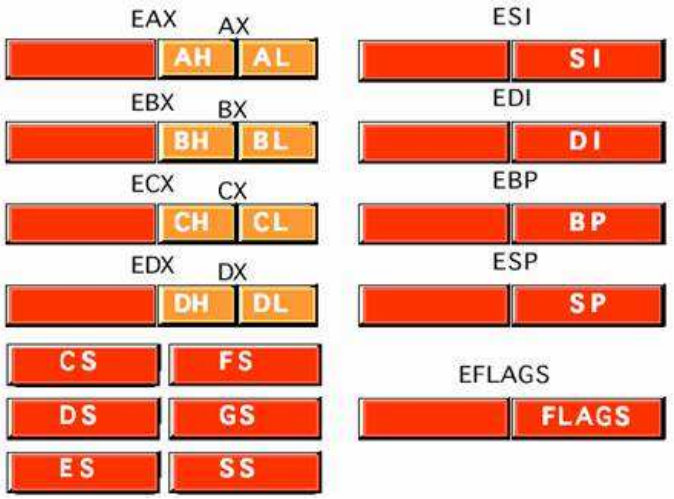
- Byte MOD-REG-R/M:



MOD	Meaning
00	Register indirect addressing mode or SIB with no displacement (when R/M = 100) or Displacement only addressing mode (when R/M = 101).
01	One-byte signed displacement follows addressing mode byte(s).
10	Four-byte signed displacement follows addressing mode byte(s).
11	Register addressing mode.

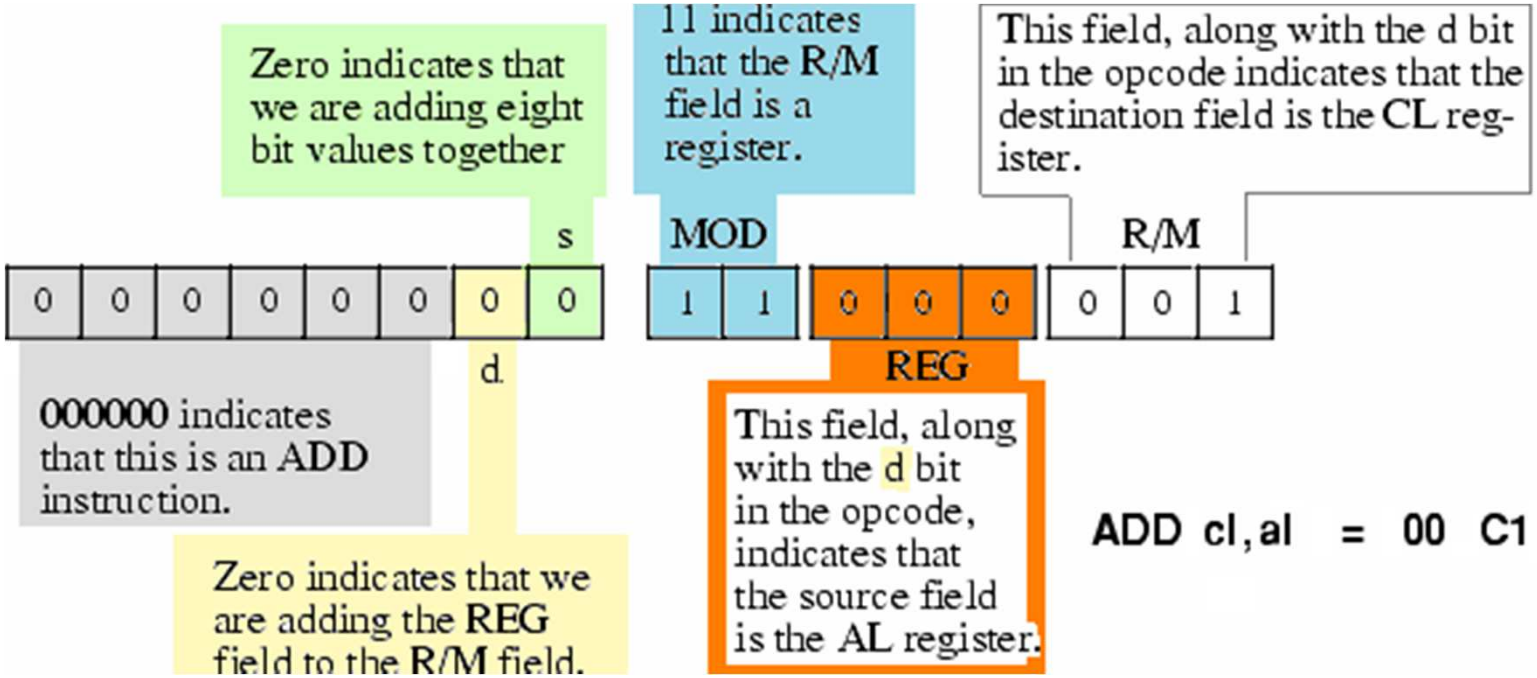
Exemplo: 80x86

- Registradores:



REG Value	Register if data size is eight bits	Register if data size is 16-bits	Register if data size is 32 bits
000	al	ax	eax
001	cl	cx	ecx
010	dl	dx	edx
011	bl	bx	ebx
100	ah	sp	esp
101	ch	bp	ebp
110	dh	si	esi
111	bh	di	edi

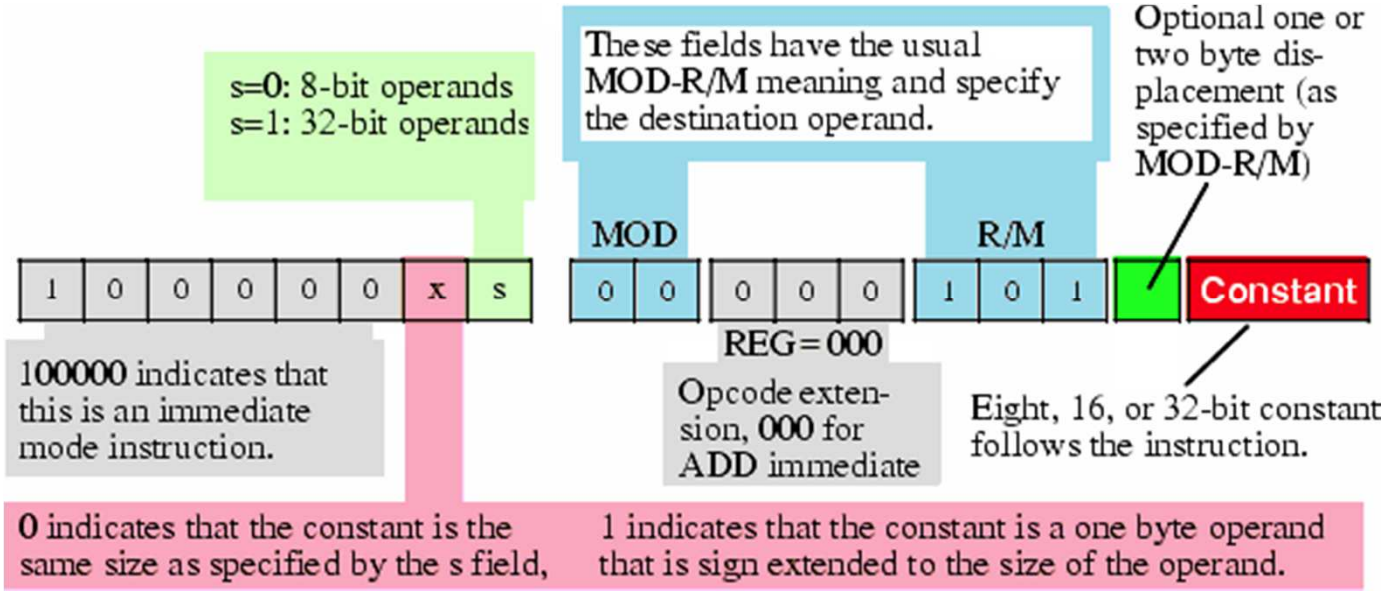
Exemplo 80x86



- Devido à flexibilidade do esquema MOD-REG-R/M, algumas instruções podem ter duas codificações e ambas legais.
- Esta instrução poderia ser também 02C8 se d=1.

Exemplo 80x86

- Codificação de um ADD imediato:
- Não tem *direction bit*, MOD-R/M codifica sempre o destino
- Se o operando for de 8 bits, o bit x será ignorado. Para operandos de 16 ou 32 bits, o bit x especifica o tamanho da constante. Se x=0, a constante será do mesmo tamanho do operando. Se x=1 a constante é sinalizada de 8 bits e seu sinal será estendido. Bom para soma de valores pequenos que é muito comum.

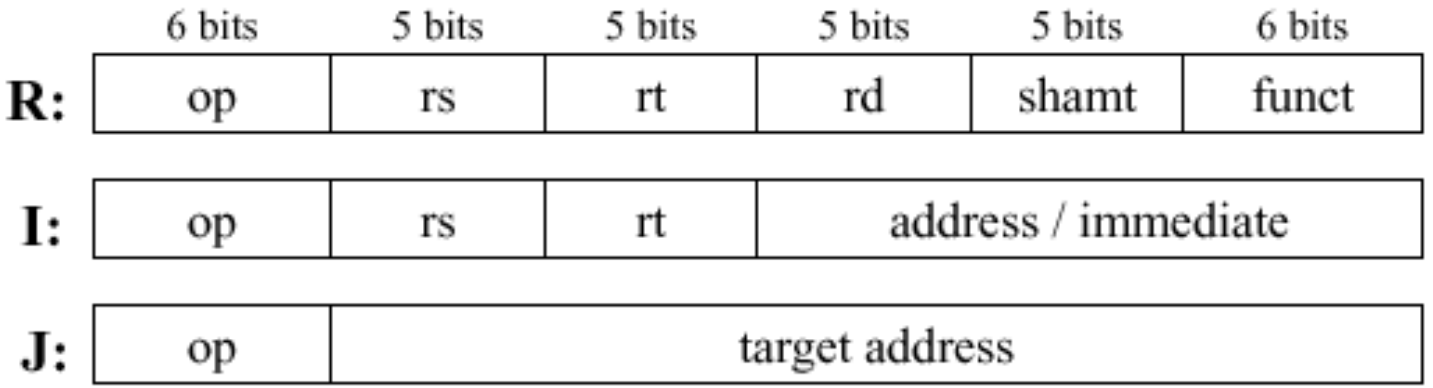


A arquitetura MIPS

- O MIPS é uma arquitetura simples tipo load-store. Existem várias versões.
- Algumas características do MIPS de 64 bits são:
  - 32 registradores de uso geral
  - Tipos de dados de 8, 16, 32 e 64 bits
  - Atuação sobre inteiros de 64 bits (MIPS64)
  - Memória endereçável por byte com endereços de 64 bits
  - Modos de endereçamento Imediato, registrador e Deslocamento

A arquitetura MIPS

- Operações MIPS
- Existem três tipos de operações. Tipo R, I e J:



op: basic operation of the instruction (opcode)
rs: first source operand register
rt: second source operand register
rd: destination operand register
shamt: shift amount
funct: selects the specific variant of the opcode (function code)
address: offset for load/store instructions (+/-2^{15})
immediate: constants for immediate instructions

A arquitetura MIPS

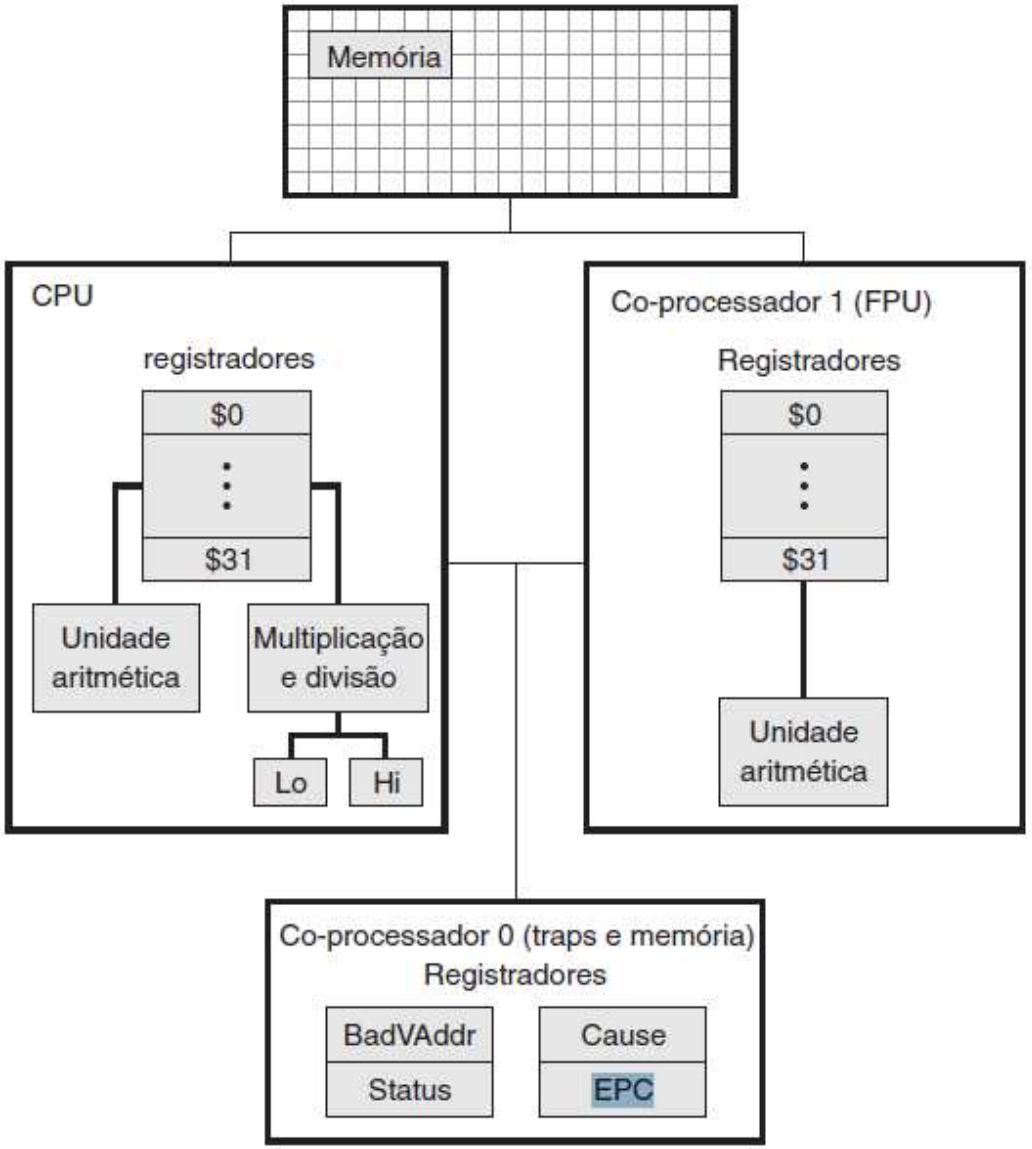
Example instruction	Instruction name	Meaning
LD R1,30(R2)	Load double word	Regs[R1] $\leftarrow_{64}$ Mem[30+Regs[R2]]
LD R1,1000(R0)	Load double word	Regs[R1] $\leftarrow_{64}$ Mem[1000+0]
LW R1,60(R2)	Load word	Regs[R1] $\leftarrow_{64}$ (Mem[60+Regs[R2]]$_0$)^{32} ## Mem[60+Regs[R2]]
LB R1,40(R3)	Load byte	Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{56} ## Mem[40+Regs[R3]]
LBU R1,40(R3)	Load byte unsigned	Regs[R1] $\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[R3]]
LH R1,40(R3)	Load half word	Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{48} ## Mem[40+Regs[R3]] ## Mem[41+Regs[R3]]
L.S F0,50(R3)	Load FP single	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R3]] ## 0^{32}
L.D F0,50(R2)	Load FP double	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R2]]
SD R3,500(R4)	Store double word	Mem[500+Regs[R4]] $\leftarrow_{64}$ Regs[R3]
SW R3,500(R4)	Store word	Mem[500+Regs[R4]] $\leftarrow_{32}$ Regs[R3]$_{32..63}$
S.S F0,40(R3)	Store FP single	Mem[40+Regs[R3]] $\leftarrow_{32}$ Regs[F0]$_{0..31}$
S.D F0,40(R3)	Store FP double	Mem[40+Regs[R3]] $\leftarrow_{64}$ Regs[F0]
SH R3,502(R2)	Store half	Mem[502+Regs[R2]] $\leftarrow_{16}$ Regs[R3]$_{48..63}$
SB R2,41(R3)	Store byte	Mem[41+Regs[R3]] $\leftarrow_{8}$ Regs[R2]$_{56..63}$

A arquitetura MIPS

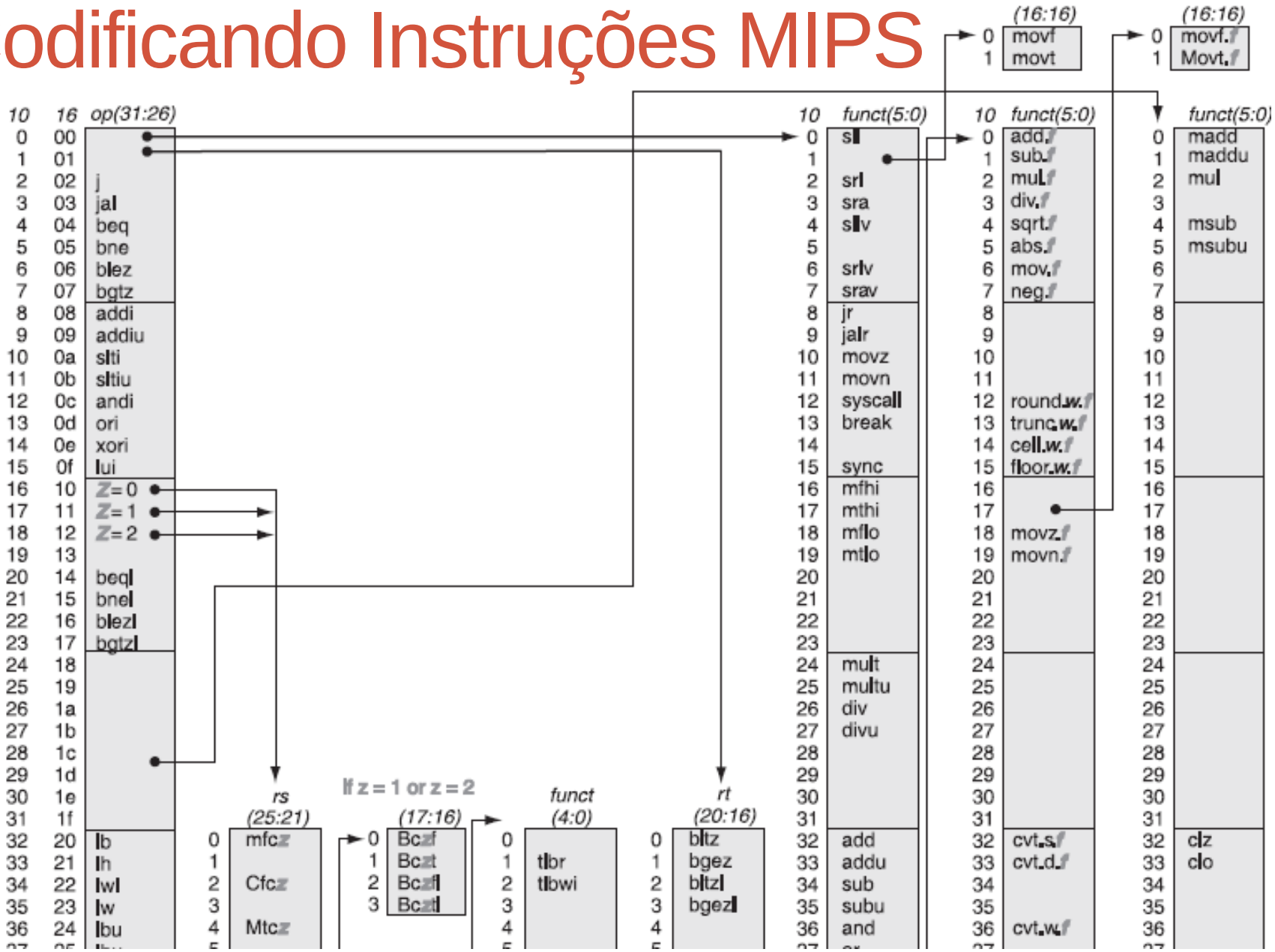
Example instruction	Instruction name	Meaning
DADDU R1,R2,R3	Add unsigned	Regs[R1]← Regs[R2]+Regs[R3]
DADDIU R1,R2,#3	Add immediate unsigned	Regs[R1]← Regs[R2]+3
LUI R1,#42	Load upper immediate	Regs[R1]← 0^{32}##42##0^{16}
DSLL R1,R2,#5	Shift left logical	Regs[R1]← Regs[R2]<<5
SLT R1,R2,R3	Set less than	if (Regs[R2]<Regs[R3]) Regs[R1] ←1 else Regs[R1]←0

Example instruction	Instruction name	Meaning
J name	Jump	$PC_{36..63}$←name
JAL name	Jump and link	Regs[R31]←PC+8; $PC_{36..63}$←name; ((PC + 4)−2^{27}) ≤ name < ((PC+4)+2^{27})
JALR R2	Jump and link register	Regs[R31]←PC+8; PC←Regs[R2]
JR R3	Jump register	PC←Regs[R3]
BEQZ R4,name	Branch equal zero	if (Regs[R4]==0) PC←name; ((PC+4)−2^{17}) ≤ name < ((PC+4)+2^{17})
BNE R3,R4,name	Branch not equal zero	if (Regs[R3]!= Regs[R4]) PC←name; ((PC+4)−2^{17}) ≤ name < ((PC+4)+2^{17})
MOVZ R1,R2,R3	Conditional move if zero	if (Regs[R3]==0) Regs[R1]←Regs[R2]

A arquitetura MIPS



Codificando Instruções MIPS



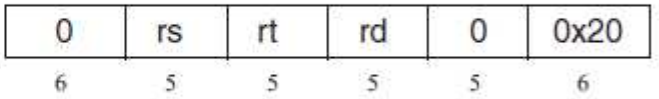
Codificando Instruções MIPS

Nome do registrador	Número	Uso
$zero	0	constante 0
$at	1	reservado para o montador
$v0	2	avaliação de expressão e resultados de uma função
$v1	3	avaliação de expressão e resultados de uma função
$a0	4	argumento 1
$a1	5	argumento 2
$a2	6	argumento 3
$a3	7	argumento 4
$t0	8	temporário (não preservado pela chamada)
$t1	9	temporário (não preservado pela chamada)
$t2	10	temporário (não preservado pela chamada)
$t3	11	temporário (não preservado pela chamada)
$t4	12	temporário (não preservado pela chamada)
$t5	13	temporário (não preservado pela chamada)
$t6	14	temporário (não preservado pela chamada)
$t7	15	temporário (não preservado pela chamada)
$s0	16	temporário salvo (preservado pela chamada)
$s1	17	temporário salvo (preservado pela chamada)
$s2	18	temporário salvo (preservado pela chamada)
$s3	19	temporário salvo (preservado pela chamada)
$s4	20	temporário salvo (preservado pela chamada)
$s5	21	temporário salvo (preservado pela chamada)
$s6	22	temporário salvo (preservado pela chamada)
$s7	23	temporário salvo (preservado pela chamada)
$t8	24	temporário (não preservado pela chamada)
$t9	25	temporário (não preservado pela chamada)
$k0	26	reservado para o kernel do sistema operacional
$k1	27	reservado para o kernel do sistema operacional
$gp	28	ponteiro para área global
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	endereço de retorno (usado por chamada de função)

Codificando Instruções MIPS

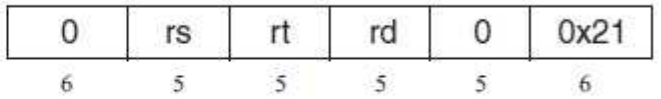
Adição (com overflow)

`add rd, rs, rt`



Adição (sem overflow)

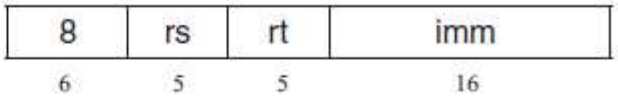
`addu rd, rs, rt`



Coloca a soma dos registradores `rs` e `rt` no registrador `rd`.

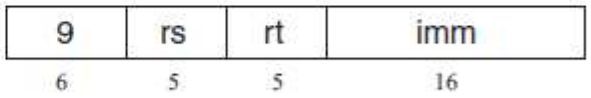
Adição imediato (com overflow)

`addi rt, rs, imm`



Adição imediato (sem overflow)

`addiu rt, rs, imm`



Coloca a soma do registrador rs e o imediato com sinal estendido no registrador `rt`.

Codificando Instruções MIPS

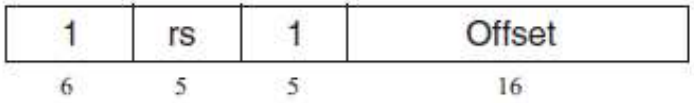
Branch se for igual

```
beq rs, rt, label
```

Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador `rs` for igual a `rt`.

Branch se for maior ou igual a zero

```
bgez rs, label
```



Desvia condicionalmente pelo número de instruções especificado pelo offset se o registrador `rs` for maior ou igual a 0.

Codificando Instruções MIPS

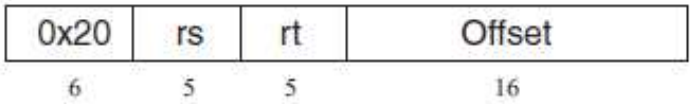
Load endereço

`la rdest, address` *pseudo-instrução*

Carrega o *endereço* calculado – não o conteúdo do local – para o registrador `rdest`.

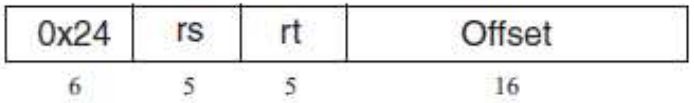
Load byte

`lb rt, address`



Load byte sem sinal

`lbu rt, address`



Carrega o byte no *endereço* para o registrador `rt`. O byte tem sinal estendido por `lb`, mas não por `lbu`.

Exemplo:

- Qual será o código binário executável (linguagem de máquina) que seria gerado para o programa de alto nível abaixo:

```
A[300] = h + A[300];
```

- Suponha que o endereço do início do vetor está no registrador **$t1** e a variável h no registrador **$s2**.

Exemplo:

Código “C”:

```
A[300] = h + A[300]
```

Código assembly:

```
lw   $t0, 1200($t1)
add  $t0, $s2, $t0
sw   $t0, 1200($t1)
```

op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Exemplo:

op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

O resultado do que vai na memória é:

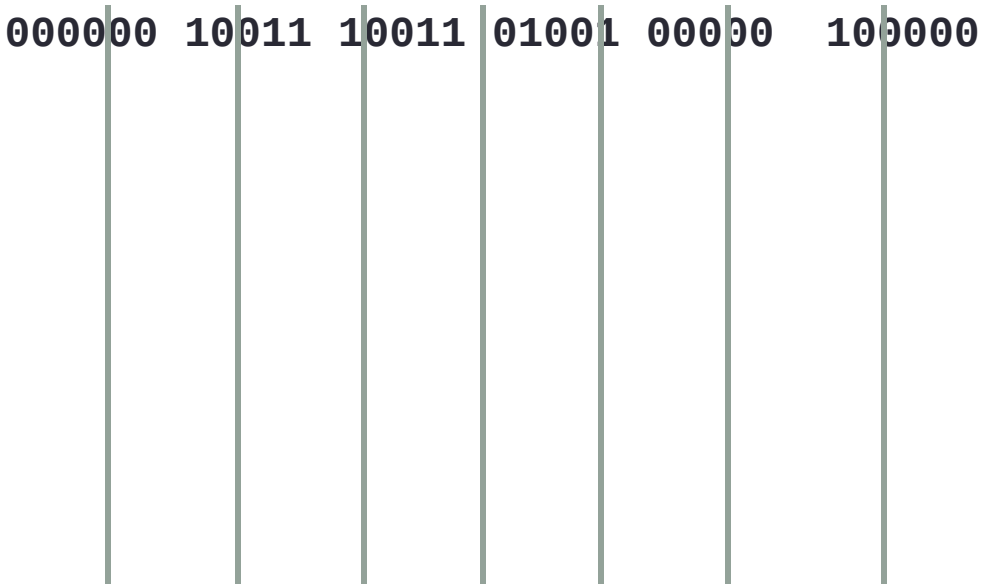
0x00400000	0x8d2804b0	lw	$t0, 1200($t1)
0x00400004	0x02484020	add	$t0, $s2, $t0
0x00400008	0xad2804b0	sw	$t0, 1200($t1)

Mais um exemplo:

```
Loop:   add  $t1, $s3, $s3   # starts from 80000
        add  $t1, $t1, $t1
        add  $t1, $t1, $s6
        lw   $t0, 0($t1)
        bne  $t0, $s5, Exit
        add  $s3, $s3, $s4
        j    Loop
Exit:
```

	6	5	5	5	5	6	
80000	0	19	19	9	0	32	R-type
80004	0	9	9	9	0	32	R-type
80008	0	9	22	9	0	32	R-type
80012	35	9	8	0			I-type
80016	5	8	21	2			I-type
80020	0	19	20	19	0	32	R-type
80024	2	20000					J-type
80028							

	6	5	5	5	5	6	
80000	0	19	19	9	0	32	R-type
80004	0	9	9	9	0	32	R-type
80008	0	9	22	9	0	32	R-type
80012	35	9	8	0			I-type
80016	5	8	21	2			I-type
80020	0	19	20	19	0	32	R-type
80024	2	20000					J-type
80028							

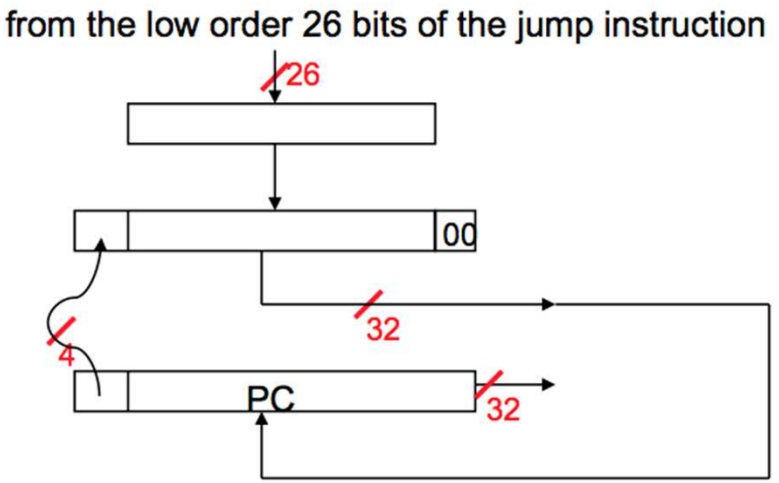



```
0x00800000  0x02734820      Loop: add  $t1, $s3, $s3
0x00800004  0x01294820      add $t1, $t1, $t1
0x00800008  0x01364820      add $t1, $t1, $s6
0x0080000c  0x8d280000      lw $t0, 0($t1)
0x00800010  0x15150002      bne $t0, $s5, Exit
0x00800014  0x02749820      add $s3, $s3, $s4
0x00800018  0x08004E20      j Loop
```

Instrução Jump

```
addi $a0, $0, 1
j next
next:
j skip1
add $a0, $a0, $a0
skip1:
j skip2:
add $a0, $a0, $a0
add $a0, $a0, $a0
skip2:
j skip3
loop:
add $a0, $a0, $a0
add $a0, $a0, $a0
add $a0, $a0, $a0
skip3:
j loop
```

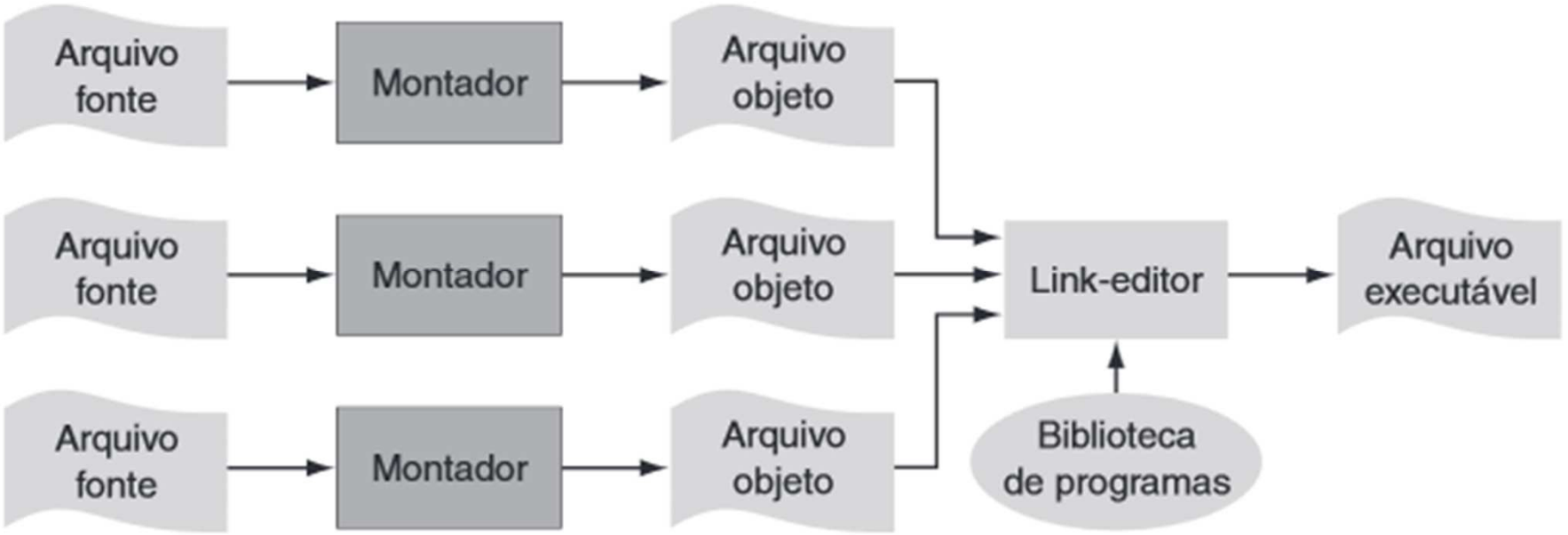
```
[0x000000]  0x20040001  # addi $a0, $zero, 1 ($a0 = 1)
[0x000004]  0x08000002  # j 0x0002 (jump to addr 0x0008)
[0x000008]  0x08000004  # j 0x0004 (jump to addr 0x0010)
[0x00000C]  0x00842020  # add $a0, $a0, $a0 ($a0 = $a0 + $a0)
[0x000010]  0x08000007  # j 0x0007 (jump to addr 0x001C)
[0x000014]  0x00842020  # add $a0, $a0, $a0 ($a0 = $a0 + $a0)
[0x000018]  0x00842020  # add $a0, $a0, $a0 ($a0 = $a0 + $a0)
[0x00001C]  0x0800000B  # j 0x000B (jump to addr 0x002C)
[0x000020]  0x00842020  # add $a0, $a0, $a0 ($a0 = $a0 + $a0)
[0x000024]  0x00842020  # add $a0, $a0, $a0 ($a0 = $a0 + $a0)
[0x000028]  0x00842020  # add $a0, $a0, $a0 ($a0 = $a0 + $a0)
[0x00002C]  0x08000008  # j 0x0008 (jump to addr 0x0020)
```



Assembly e Linguagem de Máquina

- Linguagem de máquina é a sequencia de bits de cada instrução definida pela arquitetura do conjunto de instruções
- Assembly é a representação simbólica da linguagem de máquina
- É mais legível porque utiliza símbolos no lugar de bits
- Os símbolos podem representar registradores, locais de memória para dados ou código, etc.
- Uma ferramenta chamada **montador (ou assembler)** traduz da linguagem assembly para a linguagem de máquina.

Assembly e Linguagem de Máquina



Assembly e Linguagem de Máquina

```
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       1048812
addiu     $4,  $4, 1072
lw        $31, 20($29)
addiu     $29, $29, 32
jr        $31
move      $2,  $0
```

Assembly

```
00100111101111101111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

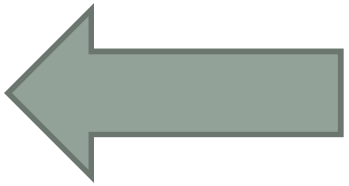
Linguagem de Máquina

Assembly e Linguagem de Máquina

```
        .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
        jr $ra

        .data
        .align 0
str:
        .asciiz "The sum from 0 ..100 is %d \n"
```

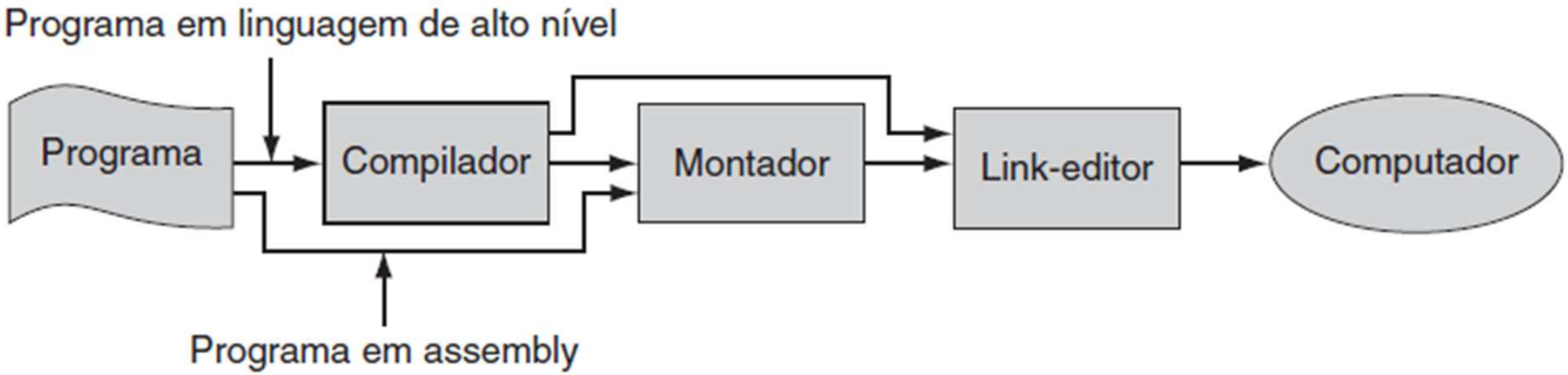
```
#include <stdio.h>

int main (int argc, char *argv[])
{
    int i;
    int sum =0;
    for (i =0;i <=100;i =i +1)
        sum =sum +i *i;
    printf ("The sum from 0 ..100 is %d \n",sum);
}
```

Código em Linguagem C



O mesmo código assembly porém com rótulos e sem comentários

Assembly e Linguagem de Máquina



- O Assembly pode ser uma linguagem de saída do compilador ou a linguagem adotada diretamente pelo programador
- O compilador pode gerar o binário diretamente.

Montador e Link-editor

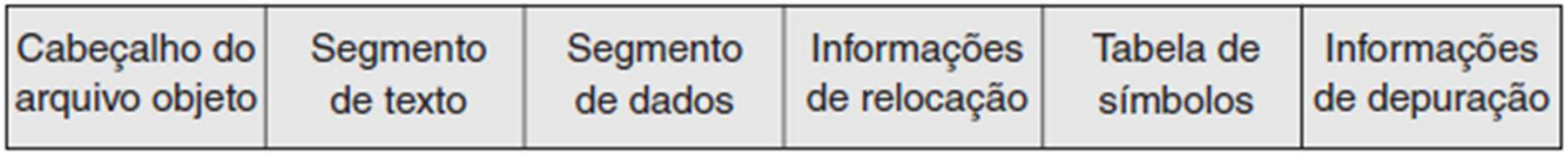
- Um montador traduz um arquivo de instruções em assembly para um arquivo de instruções de máquina binárias e dados binários.
- Funciona em duas etapas:
  - A primeira etapa é encontrar locais de memória com rótulos, de modo que o relacionamento entre os nomes simbólicos e endereços é conhecido quando as instruções são traduzidas.
  - A segunda etapa é traduzir cada instrução assembly combinando os equivalentes numéricos dos opcodes, especificadores de registradores e rótulos em uma instrução válida.
- Um **rótulo** é **externo** (também chamado **global**) se o objeto rotulado puder ser referenciado a partir de arquivos diferentes de onde está definido.
- **Rótulos locais** ocultam nomes que não devem ser visíveis a outros módulos

Montador e Link-editor

- O montador depende de outra ferramenta, o link-editor, para combinar uma coleção de arquivos-objeto e bibliotecas em um arquivo executável, resolvendo os rótulos externos.
- O montador auxilia o link-editor, oferecendo listas de rótulos e referências não resolvidas.
- Se uma linha começa com um rótulo, o montador registra em sua **tabela de símbolos** o nome do rótulo e o endereço da word de memória que a instrução ocupa.
- Quando o montador atinge o final de um arquivo assembly, a tabela de símbolos registra o local de cada rótulo definido no arquivo.
- Um montador não reclama sobre referências não resolvidas porque o rótulo correspondente provavelmente estará definido em outro arquivo.

Formato do Arquivo Objeto

- Os montadores produzem arquivos objeto, dividido em seções:
- O **cabeçalho do arquivo objeto** descreve o tamanho e a posição das outras partes do arquivo.
- O **segmento de texto** contém o código em linguagem de máquina para rotinas no arquivo de origem. Essas rotinas podem ser não-executáveis devido a referências não resolvidas.
- O **segmento de dados** contém uma representação binária dos dados no arquivo de origem. Os dados também podem estar incompletos devido a referências não resolvidas a rótulos em outros arquivos.
- As **informações de relocação** identificam instruções e words de dados que dependem de **endereços absolutos**.
- A **tabela de símbolos** associa endereços a rótulos externos no arquivo de origem e lista referências não resolvidas.
- As **informações de depuração** contêm uma descrição do modo como o programa foi compilado.



Montador e Link-editor

- Os montadores oferecem diversos recursos convenientes que ajudam a tornar os programas em assembly mais curtos e mais fáceis de escrever.
- Por exemplo, para armazenar caracteres da string na memória:

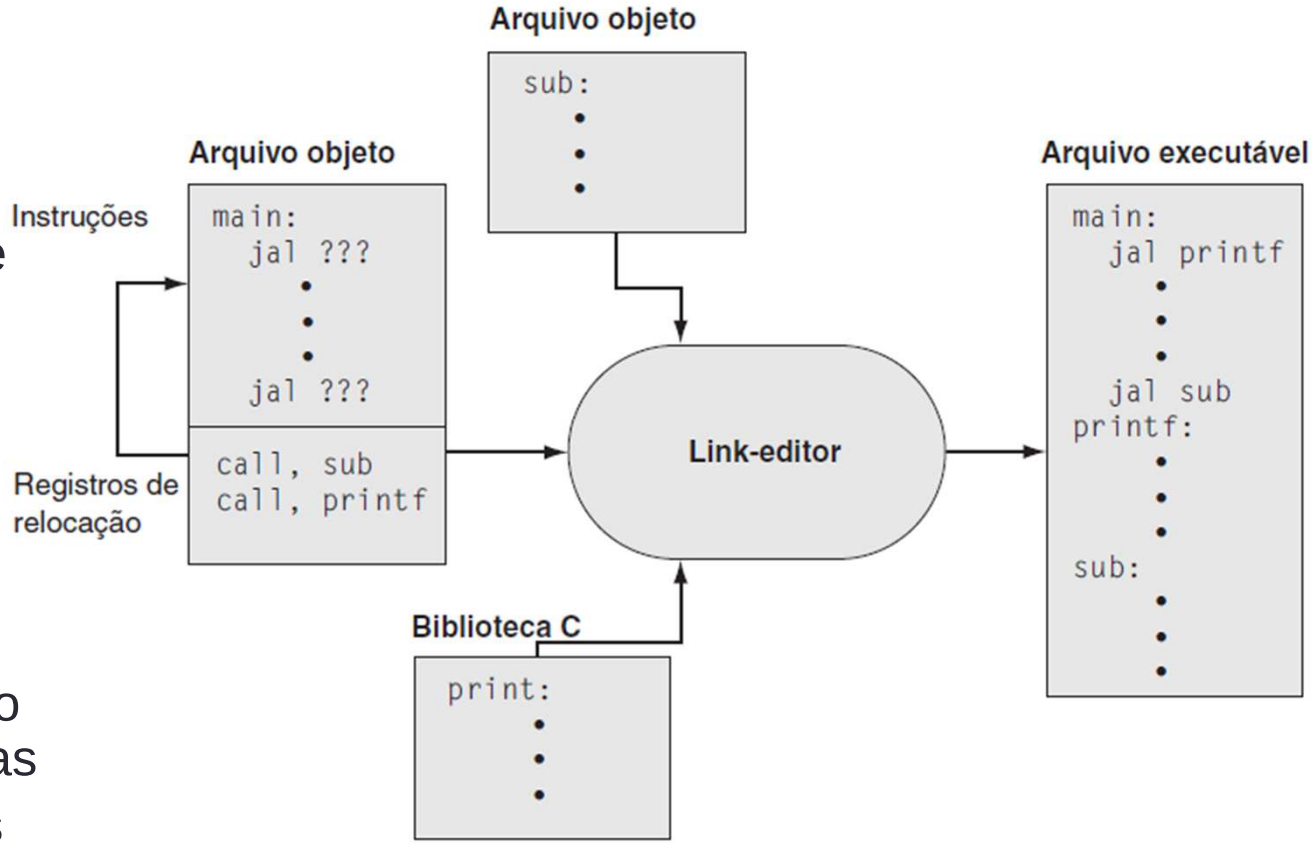
```
.asciiz “The sum from 0 .. 100 is %d\n”
```

- Que é equivalente a:

```
.byte 84, 104, 101, 32, 115, 117, 109, 32
.byte 102, 114, 111, 109, 32, 48, 32, 46
.byte 46, 32, 49, 48, 48, 32, 105, 115
.byte 32, 37, 100, 10, 0
```

Montador e Link-editor

- O Link-editor realiza três tarefas:
  - Pesquisa as bibliotecas de programa para encontrar rotinas de biblioteca usadas pelo programa
  - Determina os locais da memória que o código de cada módulo ocupará e reloca suas instruções ajustando referências absolutas
  - Resolve referências entre os arquivos



ASSEMBLY

MIPS/SPIM

QtSpim

- http://spimsimulator.sourceforge.net/
- QtSpim - Publisher's description

  “QtSpim is a self-contained simulator that runs MIPS32 programs. QtSpim also provides a simple debugger and minimal set of operating system services. QtSpim implements almost the entire MIPS32 assembler-extended instruction set.It reads and executes assembly language programs written for this processor.”
- Pseudo-instrução: expande-se para várias instruções de máquina.
- O SPIM não roda programas de qualquer variante de processador MIPS (MIPS64 por exemplo)

Ordem de Bytes

- Os processadores MIPS podem operar com a ordem de bytes *big-endian* ou *little-endian*.
- A ordem de bytes do SPIM é a mesma ordem de bytes da máquina utilizada para executar o simulador.
- Como determinar qual a ordem de bytes da sua máquina?

```c
#include <stdio.h>

union teste{
    long a;
    char b[4];
}
main()
{
    union teste t;
    t.a=0x12345678;
    printf("a = %xhex\n",t.a);
    printf("b[0] = %x\n",t.b[0]);
    printf("b[1] = %x\n",t.b[1]);
    printf("b[2] = %x\n",t.b[2]);
    printf("b[3] = %x\n",t.b[3]);
}
```

Modos de endereçamento e Pseudo-instruções

- O assembler implementa uma máquina virtual com mais modos de endereçamentos que a máquina real:

Formato	Cálculo de endereço
(registrador)	conteúdo do registrador
imm	imediato
imm (registrador)	imediato + conteúdo do registrador
rótulo	endereço do rótulo
rótulo ± imediato	endereço do rótulo + ou – imediato
rótulo ± imediato (registrador)	endereço do rótulo + ou – (imediato + conteúdo do registrador)

- Por exemplo, suponha que o rótulo table referenciasse o local de memória 0×10000004 e um programa tivesse a instrução:

```
ld $a0, table + 4($a1)
```

- O montador traduziria essa instrução para as instruções

```
lui $at, 4096
addu $at, $at, $a1
lw $a0, 8($at)
```

Pseudo-instruções

- Por exemplo, as instruções:

```
move $s0,$s1
la $s1,0x12345678
```

São traduzidas pelo montador para:

```
addu $16, $0, $17
lui $1, 4660
ori $17, $1, 22136
```

- A instrução load imediato (para fazer $s0=1 por exemplo) também é uma pseudo-instrução:

```
li $s0,1  ⇒  ori $16, $0, 1
```

Guia de referência (baixar o pdf no site)

MIPS32® Instruction Set Quick Reference

R_D – Destination register
R_S, R_T – Source operand registers
R_A – Return address register (R31)
PC – Program counter
Acc – 64-bit accumulator
Lo, Hi – Accumulator low ($Acc_{31:0}$) and high ($Acc_{63:32}$) parts
± – Signed operand or sign extension
∅ – Unsigned operand or zero extension
:: – Concatenation of bit fields
R2 – MIPS32 Release 2 instruction
DOTTED – Assembler pseudo-instruction

Please refer to *"MIPS32 Architecture For Programmers Volume II: The MIPS32 Instruction Set"* for complete instruction set information.

Arithmetic Operations			
ADD	R_D, R_S, R_T	$R_D = R_S + R_T$	(overflow trap)
ADDI	R_D, R_S, const16	$R_D = R_S + const16^{\pm}$	(overflow trap)
ADDIU	R_D, R_S, const16	$R_D = R_S + const16^{\pm}$	
ADDU	R_D, R_S, R_T	$R_D = R_S + R_T$	
CLO	R_D, R_S	R_D = CountLeadingOnes(R_S)	
CLZ	R_D, R_S	R_D = CountLeadingZeros(R_S)	
LA	R_D, label	R_D = Address(label)	
LI	R_D, imm32	R_D = imm32	
LUI	R_D, const16	R_D = const16 << 16	
MOVE	R_D, R_S	$R_D = R_S$	
NEGU	R_D, R_S	$R_D = -R_S$	
SEB[R2]	R_D, R_S	$R_D = R_{S_{7:0}}^{\pm}$	
SEH[R2]	R_D, R_S	$R_D = R_{S_{15:0}}^{\pm}$	
SUB	R_D, R_S, R_T	$R_D = R_S - R_T$	(overflow trap)
SUBU	R_D, R_S, R_T	$R_D = R_S - R_T$	

Logical and Bit-Field Operations		
AND	R_D, R_S, R_T	$R_D = R_S \& R_T$
ANDI	R_D, R_S, const16	$R_D = R_S \& const16^{\varnothing}$
EXT[R2]	R_D, R_S, P, S	$R_S = R_{S_{P+S-1:P}}^{\varnothing}$
INS[R2]	R_D, R_S, P, S	$R_{D_{P+S-1:P}} = R_{S_{S-1:0}}$
NOP		No-op
NOR	R_D, R_S, R_T	$R_D = \sim(R_S \mid R_T)$
NOT	R_D, R_S	$R_D = \sim R_S$
OR	R_D, R_S, R_T	$R_D = R_S \mid R_T$
ORI	R_D, R_S, const16	$R_D = R_S \mid const16^{\varnothing}$
WSBH[R2]	R_D, R_S	$R_D = R_{S_{23:16}} :: R_{S_{31:24}} :: R_{S_{7:0}} :: R_{S_{15:8}}$
XOR	R_D, R_S, R_T	$R_D = R_S \oplus R_T$
XORI	R_D, R_S, const16	$R_D = R_S \oplus const16^{\varnothing}$

Condition Testing and Conditional Move Operations		
MOVN	R_D, R_S, R_T	if $R_T \neq 0$, $R_D = R_S$
MOVZ	R_D, R_S, R_T	if $R_T = 0$, $R_D = R_S$
SLT	R_D, R_S, R_T	$R_D = (R_S^{\pm} < R_T^{\pm})$? 1 : 0
SLTI	R_D, R_S, const16	$R_D = (R_S^{\pm} < const16^{\pm})$? 1 : 0
SLTIU	R_D, R_S, const16	$R_D = (R_S^{\varnothing} < const16^{\varnothing})$? 1 : 0
SLTU	R_D, R_S, R_T	$R_D = (R_S^{\varnothing} < R_T^{\varnothing})$? 1 : 0

Multiply and Divide Operations		
DIV	R_S, R_T	$Lo = R_S^{\pm} / R_T^{\pm}$; $Hi = R_S^{\pm}$ mod $R_T^{\pm}$
DIVU	R_S, R_T	$Lo = R_S^{\varnothing} / R_T^{\varnothing}$; $Hi = R_S^{\varnothing}$ mod $R_T^{\varnothing}$
MADD	R_S, R_T	$Acc \mathrel{+}= R_S^{\pm} \times R_T^{\pm}$
MADDU	R_S, R_T	$Acc \mathrel{+}= R_S^{\varnothing} \times R_T^{\varnothing}$
MSUB	R_S, R_T	$Acc \mathrel{-}= R_S^{\pm} \times R_T^{\pm}$
MSUBU	R_S, R_T	$Acc \mathrel{-}= R_S^{\varnothing} \times R_T^{\varnothing}$

Sintaxe do Montador

- Os comentários nos arquivos do montador começam com um sinal #.
- Rótulos são declarados por sua colocação no início de uma linha e seguidos por dois-pontos, por exemplo:

```
            .data
item:       .word         1
            .text
            .globl main             # Precisa ser global
main:       lw $t0, item
```

- O SPIM admite um subconjunto das diretivas do montador do MIPS. Alguns exemplos:
  - **.asciiz str** Armazena a string *str* na memória e a termina com nulo.
  - **.byte b1,..., bn** Armazena os *n* valores em bytes sucessivos da memória.
  - **.data** Itens subseqüentes são armazenados no segmento de dados. Se o argumento
  - **.text** Itens subseqüentes são colocados no segmento de texto do usuário.

Chamadas de sistema

- O SPIM oferece um pequeno conjunto de serviços semelhantes aos oferecidos pelo sistema operacional, por meio da instrução de chamada ao sistema (syscall).
- Por exemplo, o código a seguir imprime “the answer = 5”:

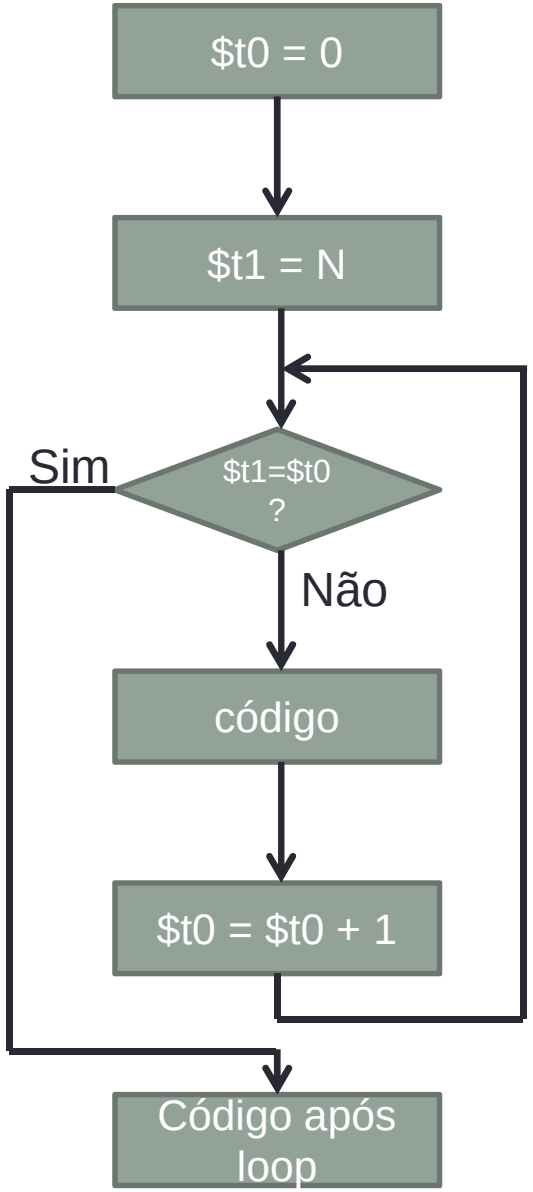
```
        .data
str:    .asciiz “Resposta = ”
        .text
        li $v0, 4    # chamada de sistema print_str
        la $a0, str  # endereço da string a imprimir
        syscall      # imprime a string

        li $v0, 1    # chamada de sistema para print_int
        li $a0, 5    # inteiro a imprimir
        syscall      # imprime o inteiro
```

Chamadas de Sistema

Serviço	Código de chamada do sistema	Argumentos	Resultado
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (em $v0)
read_float	6		float (em $f0)
read_double	7		double (em $f0)
read_string	8	$a0 = buffer, $a1 = tamanho	
sbrk	9	$a0 = valor	endereço (em $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (em $a0)
open	13	$a0 = nome de arquivo (string), $a1 = flags, $a2 = modo	descritor de arquivo (em $a0)
read	14	$a0 = descritor de arquivo, $a1 = buffer, $a2 = tamanho	número de caracteres lidos (em $a0)
write	15	$a0 = descritor de arquivo, $a1 = buffer, $a2 = tamanho	número de caracteres escritos (em $a0)
close	16	$a0 = descritor de arquivo	
exit2	17	$a0 = resultado	

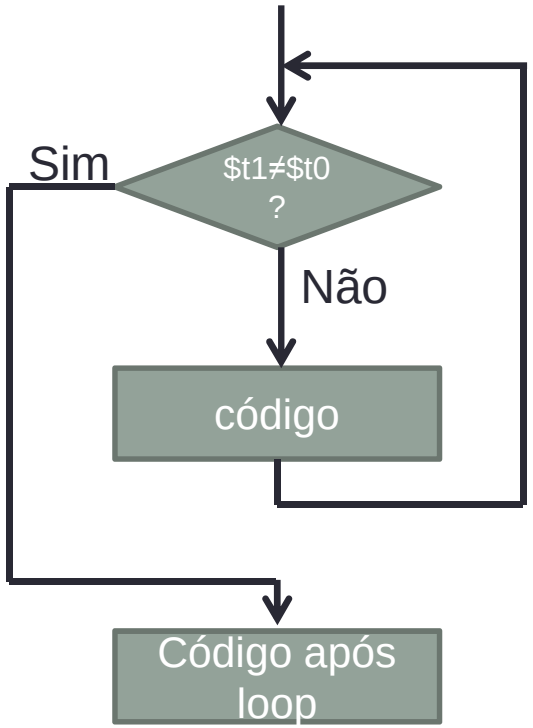
Comando FOR crescente
`for(i=0;i<N;i++){ }`

```
        .text
        .globl main
main:   li $t0,0
        li $t1,10
Loop:   beq $t1,$t0,Exit
        li $v0,1            # syscall
        add $a0,$t0,$zero   # imprime
        syscall             # int $t0

        addi $t0,$t0,1
        j Loop
Exit:   jr $ra
```



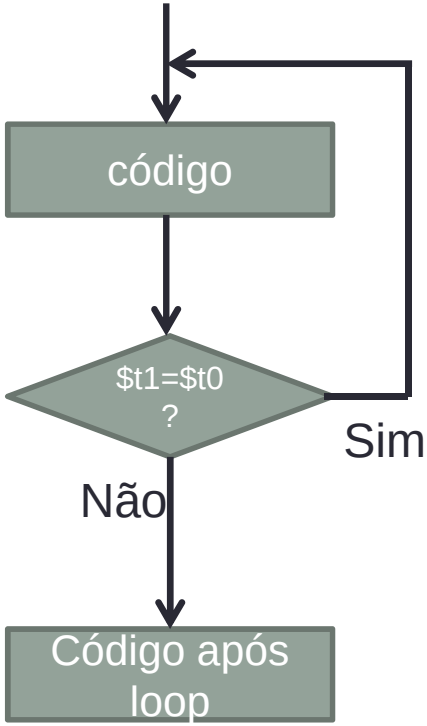
Comando while
while(i==j){ }

```
        .data
str:    .asciiz "FIM\n"
        .text
        .globl main
main:   li $t0,1
        li $t1,1
Loop:   bne $t0,$t1,Exit
        li $v0,1            # syscall
        add $a0,$t0,$zero   # imprime
        syscall             # int $t0
        li $v0,5            # syscall
        syscall             # leitura
        add $t1,$v0,$zero   # int $t1
        j Loop
Exit:   li $v0, 4           # syscall
        la $a0, str         # imprime
        syscall             # FIM
        jr $ra
```



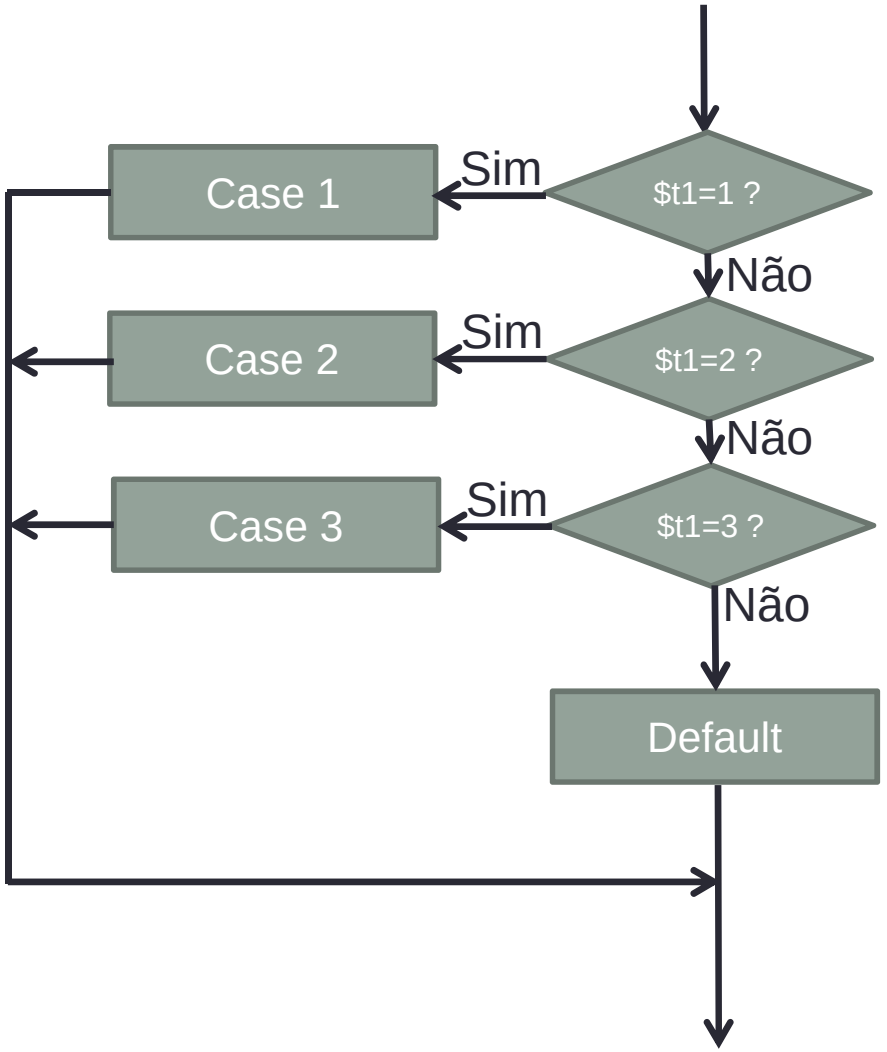
Comando while
do{ }while(i==j);

```
        .data
str:    .asciiz "FIM\n"
        .text
        .globl main
main:   li $t0,1
Loop:   li $v0,1            # syscall
        add $a0,$t0,$zero   # imprime
        syscall             # int $t0
        li $v0,5            # syscall
        syscall             # leitura
        add $t1,$v0,$zero   # int $t1
        beq $t0,$t1,Loop
        li $v0, 4           # syscall
        la $a0, str         # imprime
        syscall             # FIM
        jr $ra
```



Comando Switch-Case

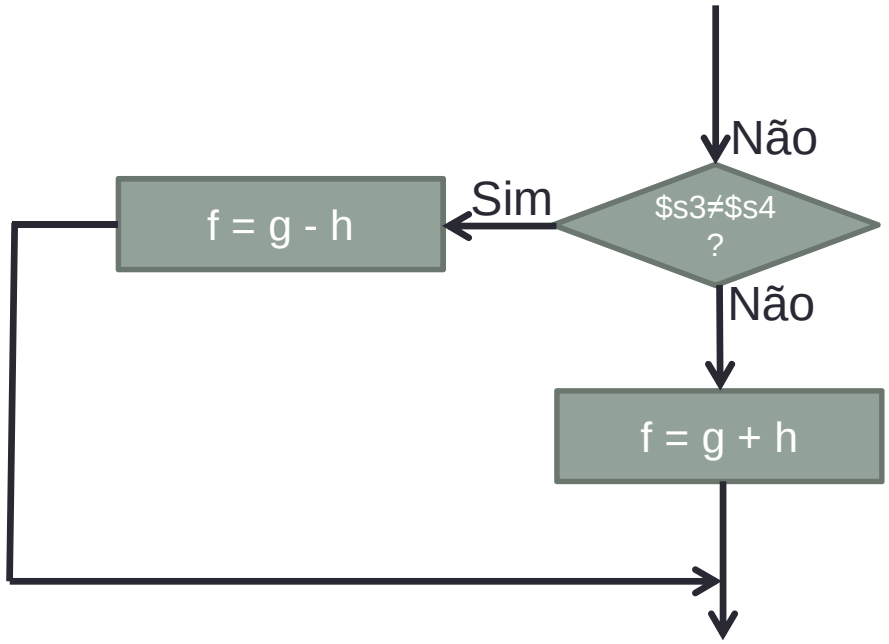
```
            .data
str1:       .asciiz "Case 1\n"
str2:       .asciiz "Case 2\n"
str3:       .asciiz "Case 3\n"
str4:       .asciiz "Default\n"
str5:       .asciiz "FIM\n"
            .text
            .globl main
main:       li $v0,5            # syscall
            syscall             # leitura
            add $t1,$v0,$zero   # int $t1
            li $t0,1
            beq $t0,$t1,case1
            li $t0,2
            beq $t0,$t1,case2
            li $t0,3
            beq $t0,$t1,case3
            j default
case1:      li $v0, 4           # syscall
            la $a0, str1        # imprime
            syscall             # str1
            j Exit
case2:      li $v0, 4           # syscall
            la $a0, str2        # imprime
            syscall             # str2
            j Exit
case3:      li $v0, 4           # syscall
            la $a0, str3        # imprime
            syscall             # str3
            j Exit
default:    li $v0, 4           # syscall
            la $a0, str4        # imprime
            syscall             # str4
Exit:       jr $ra
```



Comando if / then / else

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

- Variáveis:
  - i em $s3
  - j em $s4
  - f em $s0
  - g em $s1
  - h em $s2




```
        .text
        .globl main
main:   bne $s3,$s4,Else # Bloco else se i!=j
        add $s0,$s1,$s2  # f = g + h
        j Exit           # Sai do bloco if
Else:   sub $s0,$s1,$s2  # f = g - h
Exit:   jr $ra
```

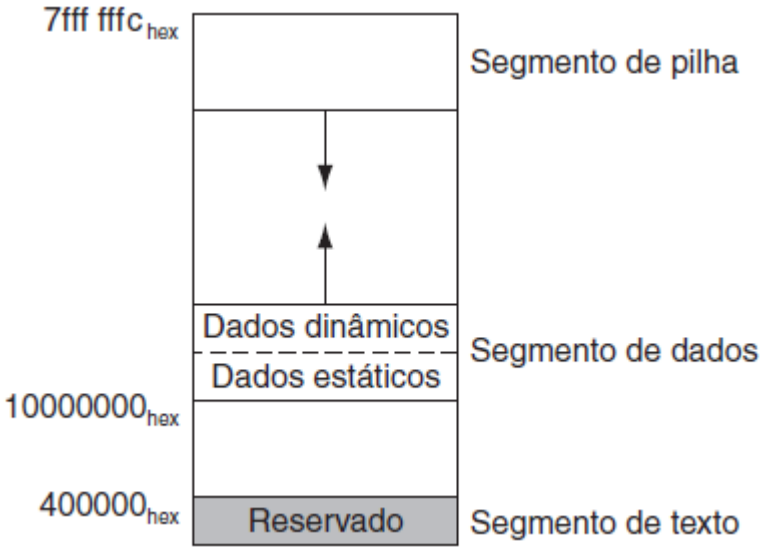
Carga de um Programa

- Etapas para iniciar um programa:

1. Lê o cabeçalho do arquivo executável para determinar o tamanho dos segmentos de texto e de dados.
2. Cria um novo espaço de endereçamento para o programa. Esse espaço de endereçamento é grande o suficiente para manter os segmentos de texto e de dados, junto com um segmento de pilha.
3. Copia instruções e dados do arquivo executável para o novo espaço de endereçamento.
4. Copia argumentos passados ao programa para a pilha.
5. Inicializa os registradores da máquina. Em geral, a maioria dos registradores é apagada, mas o stack pointer precisa receber o endereço do primeiro local da pilha livre.
6. Desvia para a rotina de partida, que copia os argumentos do programa da pilha para os registradores e chama a rotina main do programa. Se a rotina main retornar, a rotina de partida termina o programa com a chamada do sistema exit.

Uso da memória

- Normalmente são feitas convenções de uso do hardware. Uma delas é a divisão da memória de um programa:



- Para carregar a word no segmento de dados no endereço 10010020hexa para o registrador $v0, são necessárias duas instruções:

```
lui $s0, 0×1001 # 0x1001 significa 1001 base 16
lw $v0, 0x0020($s0) # 0x10010000 + 0x0020 = 0x10010020
```

Uso da memória

- Para evitar repetir a instrução lui em cada load e store, os sistemas MIPS normalmente dedicam um registrador ($gp) como um *ponteiro global* para o segmento de dados estático.
- Esse registrador contém o endereço 10008000hexa (32K acima do início do segmento de dados) , de modo que as instruções load e store podem usar seus campos de 16 bits com sinal para acessar os primeiros 64KB do segmento de dados estático.
- Com esse ponteiro global, podemos reescrever o exemplo como uma única instrução:

```
lw $v0, 0x8020($gp)
```

Convenção para chamadas de procedimento

- Os registradores $at (1), $k0 (26) e $k1 (27) são reservados para o montador e o sistema operacional e não devem ser usados por programas do usuário ou compiladores.
- Os registradores $a0-$a3 (4-7) são usados para passar os quatro primeiros argumentos às rotinas (os argumentos restantes são passados na pilha). Os registradores $v0 e $v1 (2, 3) são usados para retornar valores das funções.
- Os registradores $t0-$t9 (8-15, 24, 25) são registradores salvos pela rotina que chama, que são usados para manter quantidades temporárias que não precisam ser preservadas entre as chamadas.
- Os registradores $s0-$s7 (16-23) são registradores salvos pela rotina sendo chamada, que mantêm valores de longa duração, que devem ser preservados entre as chamadas.
- O registrador $gp (28) é um ponteiro global que aponta para o meio de um bloco de 64K de memória no segmento de dados estático.
- O registrador $sp (29) é o stack pointer, que aponta para o último local na pilha.
- O registrador $fp (30) é o frame pointer. A instrução jal escreve no registrador $ra (31), o endereço de retorno de uma chamada de procedimento.

Frame de Pilha

- A chamada de procedimento utiliza um bloco de memória chamado **frame de chamada de procedimento**. Essa memória é usada para diversas finalidades:
  - Para manter valores passados a um procedimento como argumentos
  - Para salvar registradores que um procedimento pode modificar, mas que o *caller* não deseja que sejam alterados
  - Para oferecer espaço para variáveis locais a um procedimento
- O frame consiste na memória entre o frame pointer ($fp), que aponta para a primeira word do frame, e o stack pointer ($sp), que aponta para a última word do frame
- O uso do $fp simplifica o código, mas é opcional. O montador do gcc usa, mas o do MIPS não. Neste caso $fp é mais um registrador de uso geral $s8.

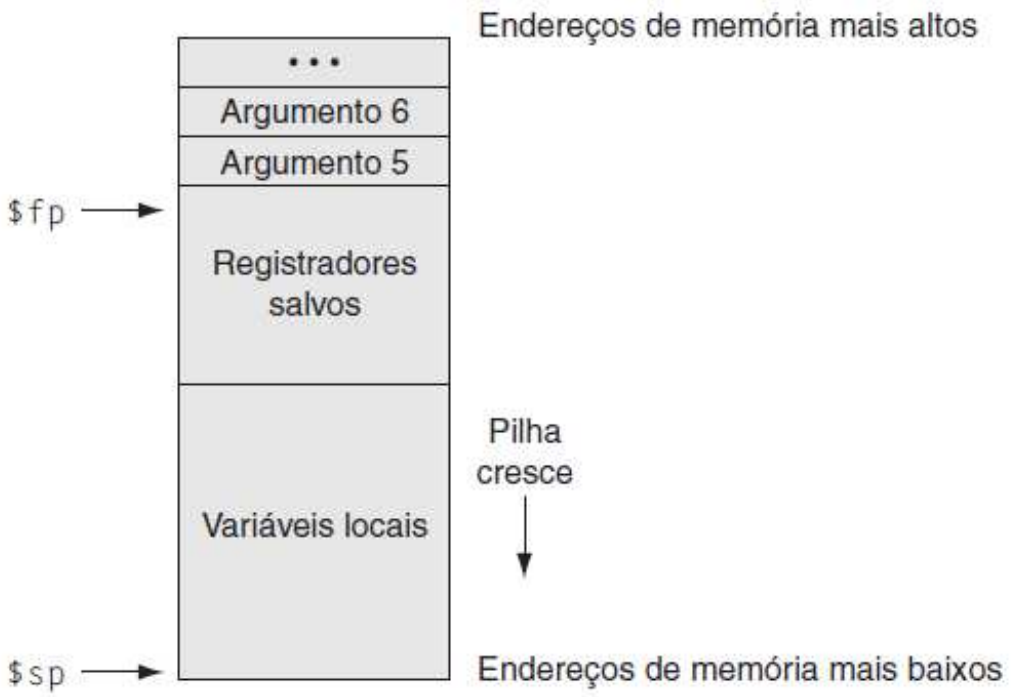
Chamada de Procedimento

O frame de pilha também é chamado frame de chamada de procedimento.

- O procedimento que está executando utiliza o frame pointer para acessar rapidamente os valores em seu frame de pilha.

- Por exemplo, um argumento no frame de pilha pode ser lido para o registrador **$v0** com a instrução:

lw $v0, 0($fp)



Chamada de Procedimento

- O que a rotina que chama deve fazer:

1. Passar argumentos. Por convenção, os quatro primeiros argumentos são passados nos registradores $a0-$a3. Quaisquer argumentos restantes são colocados na pilha e aparecem no início do frame de pilha do procedimento chamado.
2. Salvar registradores salvos pelo *caller*. O procedimento chamado pode usar esses registradores ($a0-$a3 e $t0-$t9) sem primeiro salvar seu valor. Se o *caller* espera utilizar um desses registradores após uma chamada, ele deverá salvar seu valor antes da chamada.
3. Executar uma instrução jal, que desvia para a primeira instrução da subrotina e salva o endereço de retorno no registrador $ra.

Chamada de Procedimento

- Antes que uma rotina chamada comece a executar, ela precisa realizar as seguintes etapas para configurar seu frame de pilha:

1. Alocar memória para o frame, subtraindo o tamanho do frame do stack pointer.
2. Salvar os registradores salvos pela subrotina no frame. Uma subrotina precisa salvar os valores desses registradores ($s0-$s7, $fp e $ra) antes de alterá-los, pois o *caller* espera encontrar esses registradores inalterados após a chamada. O registrador $fp é salvo para cada procedimento que aloca um novo frame de pilha. No entanto, o registrador $ra só precisa ser salvo se a subrotina fizer uma chamada. Os outros registradores salvos pela subrotina, que são utilizados, também precisam ser salvos.
3. Estabelecer o frame pointer somando o tamanho do frame de pilha menos 4 a $sp e armazenando a soma no registrador $fp.

Chamada de Procedimento

- Finalmente, a subrotina retorna ao *caller* executando as seguintes etapas:

1. Se a subrotina for uma função que retorna um valor, coloque o valor retornado no registrador $v0.
2. Restaure todos os registradores salvos pela subrotina que foram salvos na entrada do procedimento.
3. Remova o frame de pilha somando o tamanho do frame a $sp.
4. Retorne desviando para o endereço no registrador $ra.

Exemplo de chamada de procedimento

```
main ( )
{
        printf (“The factorial of 10 is %d\n”, fact (10));
}

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

Exemplo de chamada de procedimento

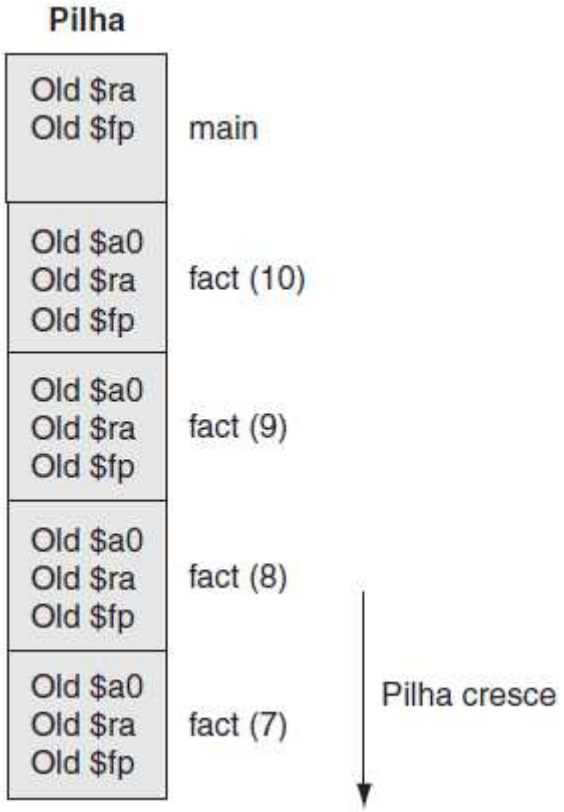
```
        .text
        .globl main
main:
        subu    $sp,$sp,32      # Frame de pilha tem 32 bytes
        sw      $ra,20($sp)     # Salva endereço de retorno
        sw      $fp,16($sp)     # Salva frame pointer antigo
        addiu   $fp,$sp,28      # Prepara frame pointer

        li      $a0,10          # Coloca argumento (10) em $a0
        jal     fact            # Chama função fatorial
        la      $a0,$LC         # Coloca string de formato em $a0
        move    $a1,$v0         # Move resultado de fact para $a1
        jal     printf          # Chama a função print

        lw      $ra,20($sp)     # Restaura endereço de retorno
        lw      $fp,16($sp)     # Restaura frame pointer
        addiu   $sp,$sp,32      # Remove frame de pilha
        jr      $ra             # Retorna a quem chamou

        .rdata
$LC:
        .ascii “The factorial of 10 is %d\n\000”
```

Exemplo de chamada de procedimento

```
        .text
fact:
        subu    $sp,$sp,32          # Frame de pilha tem 32 bytes
        sw      $ra,20($sp)         # Salva endereço de retorno
        sw      $fp,16($sp)         # Salva frame pointer
        addiu   $fp,$sp,28          # Prepara frame pointer
        sw      $a0,0($fp)          # Salva argumento (n)

        lw      $v0,0($fp)          # Carrega n
        bgtz    $v0,$L2             # Desvia se n > 0
        li      $v0,1               # Retorna 1
        j       $L1                 # Desvia para o código de retorno
$L2:
        lw      $v1,0($fp)          # Carrega n
        subu    $v0,$v1,1           # Calcula n – 1
        move    $a0,$v0             # Move valor para $a0
        jal     fact                # Chama função de fatorial
        lw      $v1,0($fp)          # Carrega n
        mul     $v0,$v0,$v1         # Calcula fact(n-1) * n

$L1:                                # Resultado está em $v0
        lw      $ra, 20($sp)        # Restaura $ra
        lw      $fp, 16($sp)        # Restaura $fp
        addiu   $sp, $sp, 32        # Remove o frame de pilha
        jr      $ra                 # Retorna a quem chamou
```

Pilha em procedimentos recursivos

- Vejamos a pilha na chamada fact(7).
- main executa primeiro, de modo que seu frame está mais abaixo na pilha.
- main chama fact(10), cujo frame de pilha vem em seguida na pilha.
- Cada invocação chama fact recursivamente para calcular o próximo fatorial mais inferior.



Chamada de Procedimento “folha”

- Se o procedimento não for chamar nenhuma outra subrotina, a chamada pode ser simplificada. Por ex.:

```
        # Call a function to add that number together with 13
        addi $a0, $zero, 13             # Set up first argument to function  (13)
        move $a1, $v0                   # Set up second argument (the number entered by user)
        jal doAdd                       # do add, result now in $v0
        move $s0, $v0                   # save result for later (b/c v0 clobbered below)

        # Print string announcing the result
        la      $a0, str_2              # 'load address' of string to print
        li      $v0, 4
        syscall

        # Print actual result
        li      $v0, 1                  # syscall #1 = print integer
        move    $a0, $s0                # tell syscall what # to print
        syscall

        # terminate the program
        li $v0, 10
        syscall

        jr $ra          # return to caller

# Define the (very simple) function we use
# Notice that this goes OUTSIDE of main
doAdd:
            add $v0, $a0, $a1               # add two arguments
            jr $ra                          # return
```

Outro exemplo

- Considere a seguinte rotina que calcula a função tak, que é um benchmark bastante utilizado, criado por Ikuo Takeuchi.

```
int tak (int x, int y, int z)
{
        if (y < x)
                return 1 + tak (tak (x - 1, y, z),
                                tak (y - 1, z, x),
                                tak (z - 1, x, y));
        else
                return z;
}

int main ( )
{
        tak(18, 12, 6);
}
```

Tak em Assembly:

```
        .data
cont:   .word 0
space:      .asciiz " "
nline:      .asciiz "\n"

        .text
tak:    subu $sp, $sp, 40
        sw $ra, 32($sp)
        sw $s0, 16($sp)          # x
        sw $s1, 20($sp)          # y
        sw $s2, 24($sp)          # z
        sw $s3, 28($sp)          # temporario
        move $s0, $a0
        move $s1, $a1
        move $s2, $a2
        bge $s1, $s0, L1         # if (y < x)
```

Exemplo: tak

```
addiu $a0, $s0, -1
move $a1, $s1
move $a2, $s2
jal tak              # tak (x - 1, y, z)
move $s3, $v0
addiu $a0, $s1, -1
move $a1, $s2
move $a2, $s0
jal tak              # tak (y - 1, z, x)
addiu $a0, $s2, -1
move $a1, $s0
move $a2, $s1
move $s0, $v0
jal tak              # tak (z - 1, x, y)
```

Exemplo: tak

```
        move $a0, $s3
        move $a1, $s0
        move $a2, $v0
        jal tak      # tak (tak(..), tak(..), tak(..))
        addiu $v0, $v0, 1
        j L2
L1:     move $v0, $s2
L2:     lw $ra, 32($sp)
        lw $s0, 16($sp)
        lw $s1, 20($sp)
        lw $s2, 24($sp)
        lw $s3, 28($sp)
        addiu $sp, $sp, 40
        jr $ra
```

Exemplo: tak

```
        .globl main
main:
        subu $sp, $sp, 24
        sw $ra, 16($sp)
        li $a0, 18
        li $a1, 12
        li $a2, 6
        jal tak             # tak(18, 12, 6)
        move $a0, $v0
        li $v0, 1           # syscall print_int
        syscall
        lw $ra, 16($sp)
        addiu $sp, $sp, 24
        jr $ra
```

Exercícios

Implemente em Assembly do MIPS os seguintes programas:

1. Vetores

```
main()
{
      int i, vet[100];
      for(i=0;i<100;i++) vet[i]=i;
}
```

2. Faça uma função para somar os elementos do vetor, passando como argumentos o vetor e o seu tamanho e utilize essa função no programa anterior. O programa main deverá imprimir o resultado.

Exercícios

3. Torre de Hanoi

```
void hanoi(char o, char d, char a, int n)
{
    static int i=0;
    if(n>1)hanoi(o,a,d,n-1);
    i++;
    printf("%d %c->%c\n", i, o, d);
    if(n>1)hanoi(a,d,o,n-1);
}

main()
{
    hanoi('A','B','C',7);
}
```