

# MC542

## Organização de Computadores Teoria e Prática

2007

Prof. Paulo Cesar Centoducatte

[ducatte@ic.unicamp.br](mailto:ducatte@ic.unicamp.br)

[www.ic.unicamp.br/~ducatte](http://www.ic.unicamp.br/~ducatte)

# MC542

## Arquitetura de Computadores

### Introdução; Conjunto de Instruções

“DDCA” - (Capítulo 6)

“COD” - (Capítulo )

# Arquitetura de Computadores

## Sumário

- **Introdução**
  - O que é arquitetura de computadores
  - Tendências
    - » Lei de Moore
    - » Capacidade Microprocessadores
    - » Desempenho dos processadores
    - » Capacidade e Velocidade das Memórias
- **Conjuntos de Instruções**
  - Introdução

# O que é Arquitetura de Computadores?

- 1950s a 1960s: Cursos de AC  
Aritmética Computacional
- 1970s a meados dos anos 1980s: Cursos de AC  
Projeto do Conjunto de Instruções (ISA),  
especialmente voltado para compiladores
- 1990s a 2000s: Cursos de AC  
Projeto de CPU, Sistemas de Memórias, Sistemas de  
I/O, Multiprocessadores.

# Tendências

- Gordon Moore (fundador da Intel), em 1965 observou que o número de transistores em um chip dobrava a cada ano (Lei de Moore)

Continua válida até os dias de hoje  
(porém está encontrando a barreira térmica)

- O desempenho dos processadores, medidos por diversos benchmarks, também tem crescido de forma acelerada.
- A capacidade das memórias tem aumentado significativamente nos últimos 20 anos  
(E o custo reduzido)

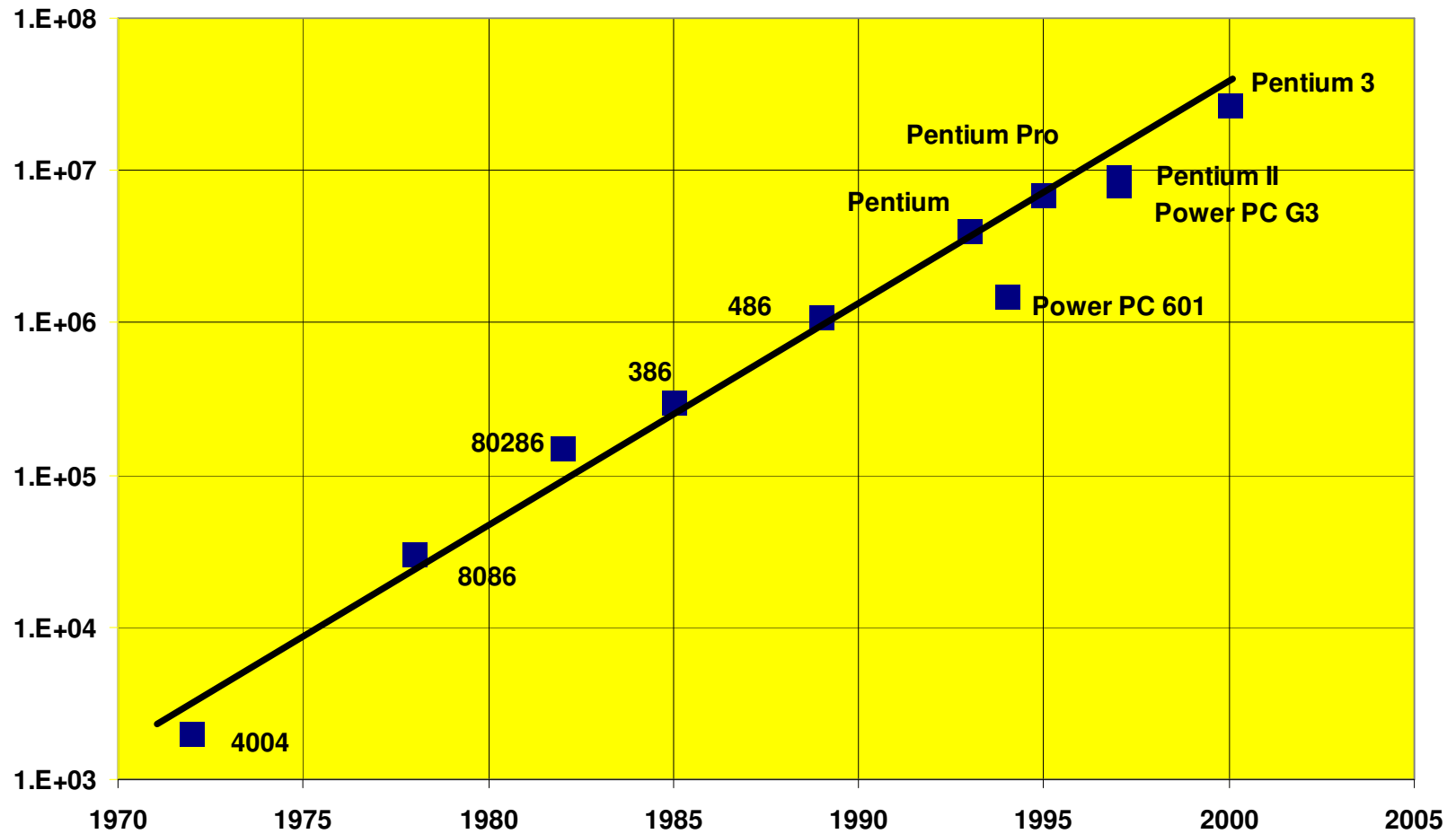
-

# Qual a Razão Desta Evolução nos Últimos Anos?

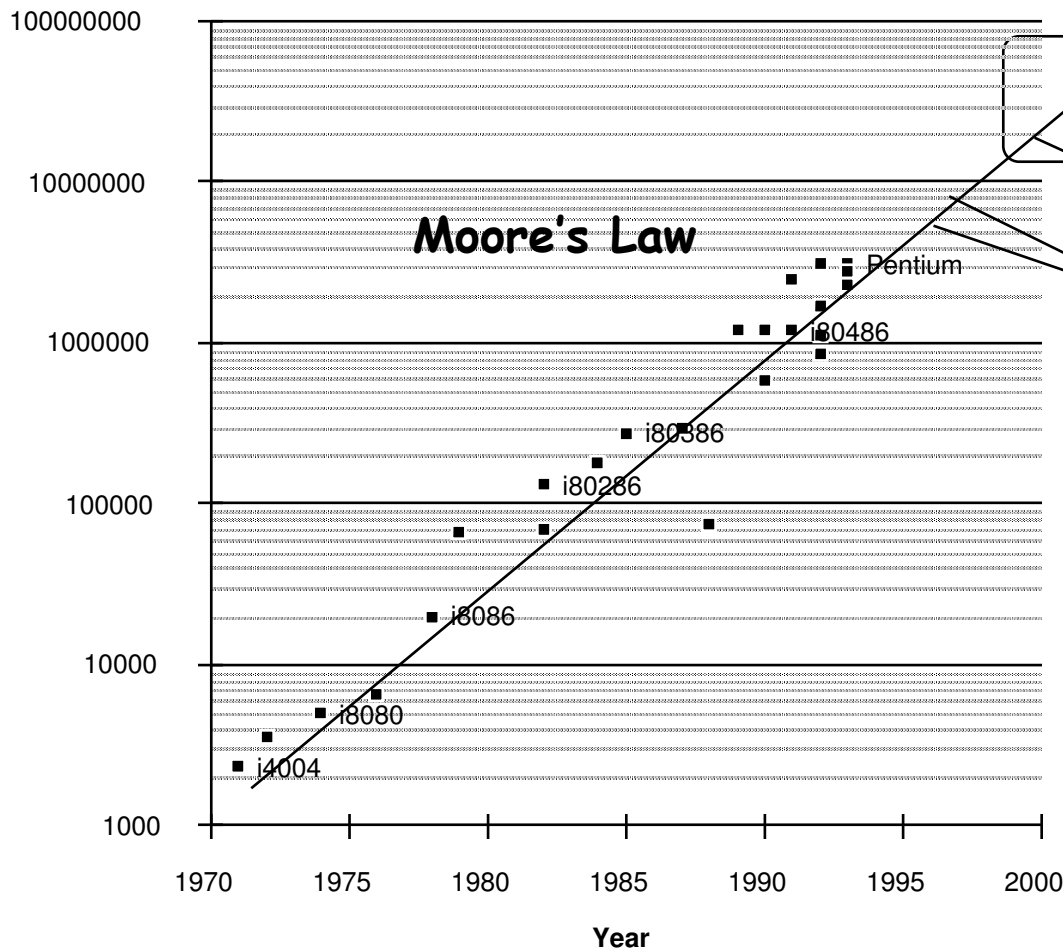
- **Desempenho**
  - Avanços tecnológicos
    - » Domínio de CMOS sobre as tecnologias mais antigas (TTL, ECL) em custo e desempenho
  - Avanços nas arquiteturas
    - » RISC, superscalar, VLIW, RAID, ...
- **Preço: Baixo custo devido**
  - Desenvolvimento mais simples
    - » CMOS VLSI: sistemas menores, menos componentes
  - Alto volume (escala)
- .....

# Tendências: Lei de Moore

## Transistors Per Chip



# Tendência Tecnológica: Capacidade Microprocessadores



Alpha 21264: 15 million  
Pentium Pro: 5.5 million  
PowerPC 620: 6.9 million  
Alpha 21164: 9.3 million  
Sparc Ultra: 5.2 million

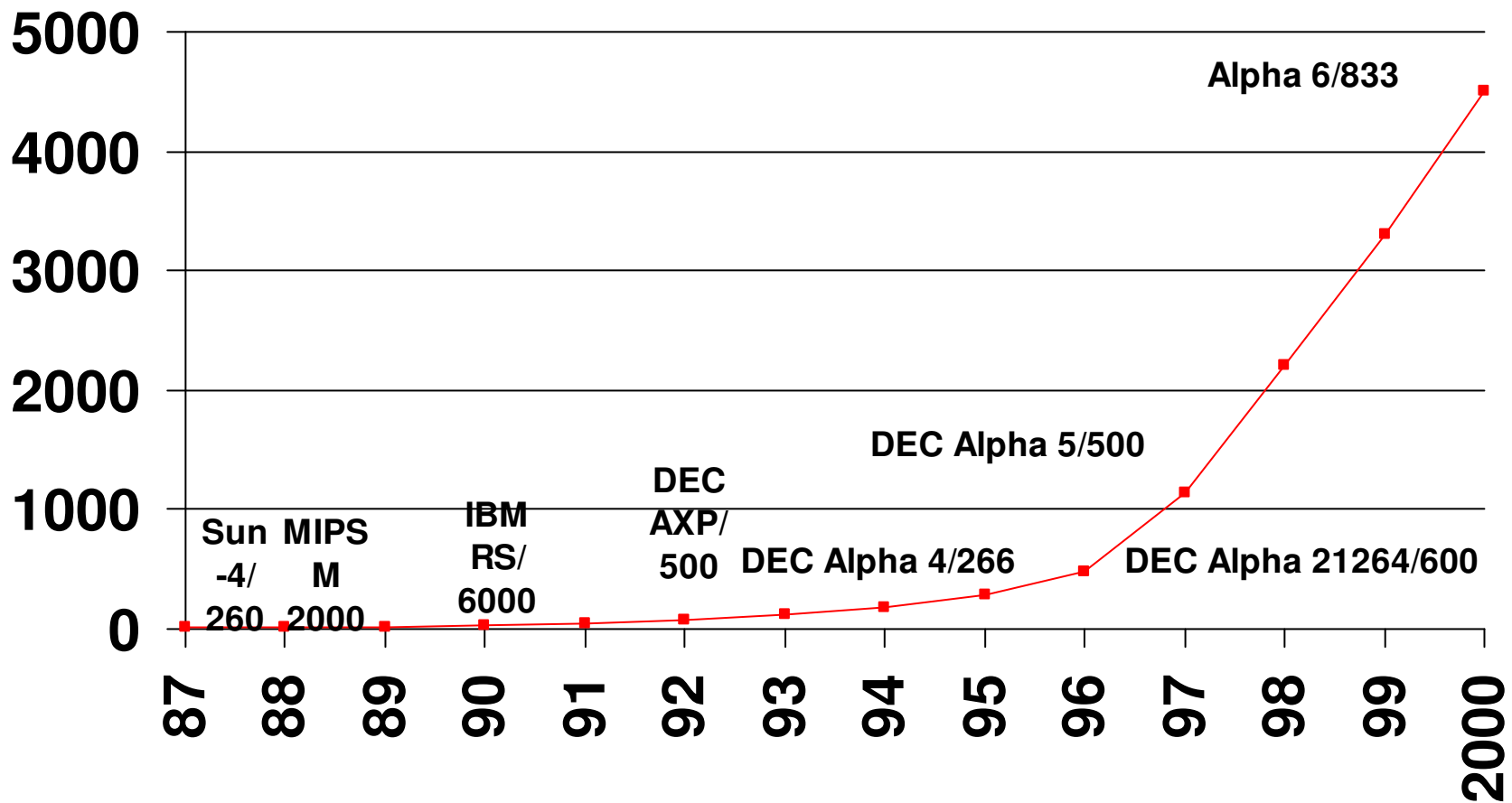
**CMOS:**

- Die size: 2X a cada 3 anos

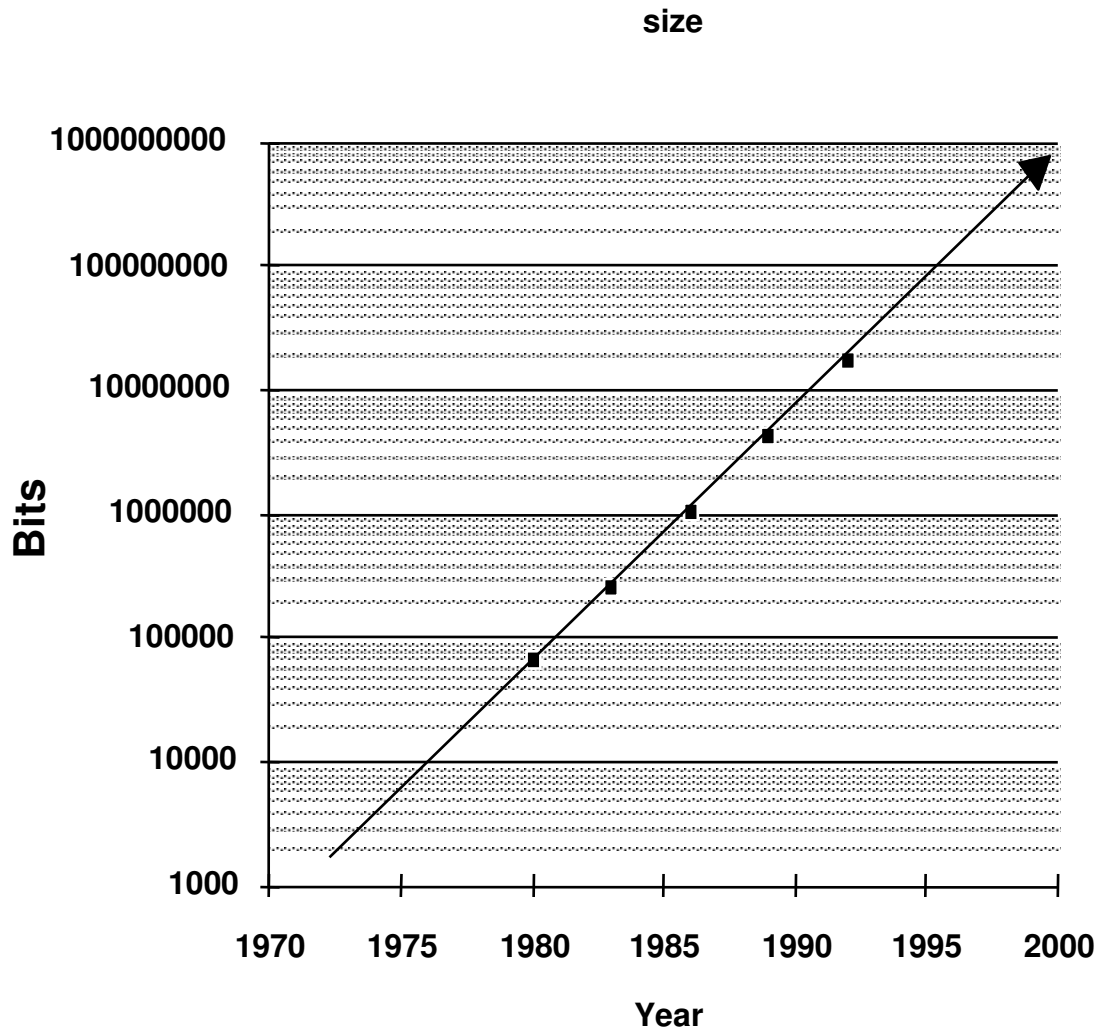


# Tendências

## Desempenho dos processadores



# Tendências Capacidade das Memórias



ano	Mbyte	cycle time
1980	0.0625	250 ns
1983	0.25	220 ns
1986	1	190 ns
1989	4	165 ns
1992	16	145 ns
1996	64	120 ns
2000	256	100 ns

# Tendências Velocidade

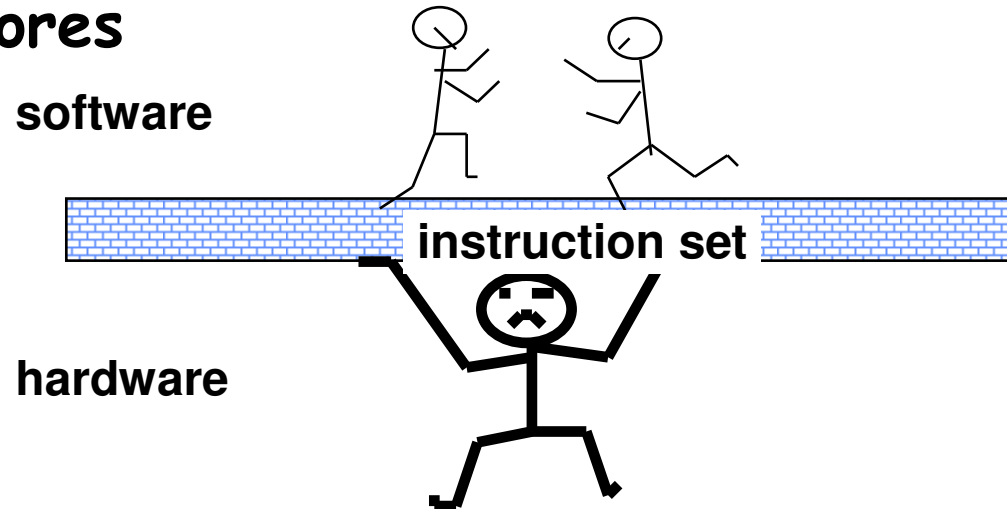
- Para a CPU o crescimento da velocidade tem sido muito acelerado
- Para Memória e disco o crescimento da velocidade tem sido modesto

Isto tem levado a mudanças significativas nas arquiteturas, SO e mesmo nas práticas de programação.

	<u>Capacidade</u>	<u>Speed (latency)</u>
Lógica	2x em 3 anos	2x em 3 anos
DRAM	4x em 3 anos	2x em 10 anos
Disco	4x em 3 anos	2x em 10 anos

# Conjunto de Instruções

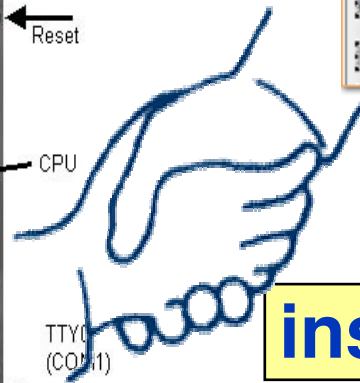
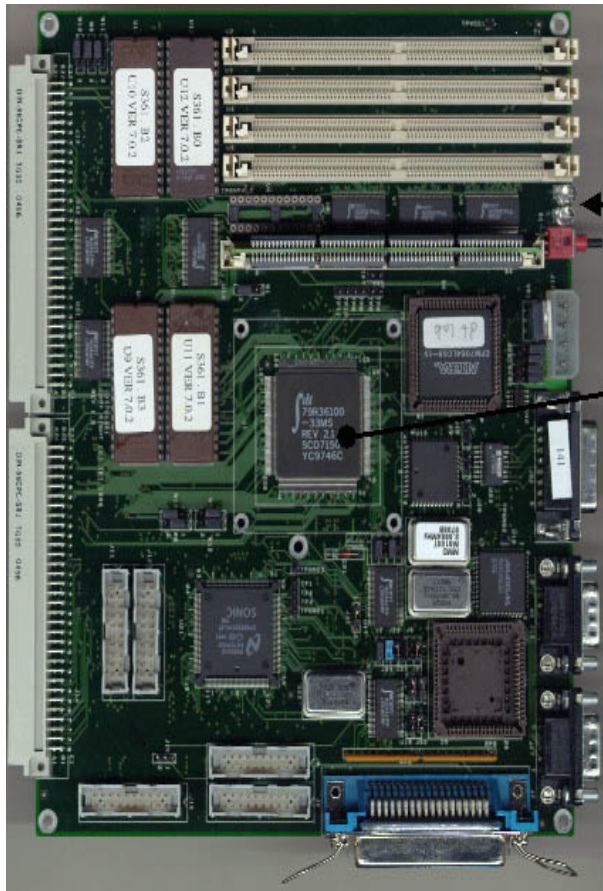
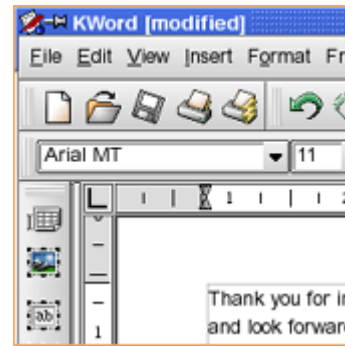
O ISA é a porção da máquina visível ao programador (nível de montagem) ou aos projetistas de compiladores



1. Quais as vantagens e desvantagens das diversas alternativas de ISA.
2. Como as linguagens e compiladores afetam (ou são afetados) o ISA.
3. Arquitetura MIPS como exemplo de arquitetura RISC.

# Introdução - ISA

software



**instruction set**

Interface entre o Hardware e o Usuário

hardware

# Evolução dos ISAs

- As maiores vantagens em uma arquitetura, em geral, são associadas com as mudanças do ISA
  - Ex: Stack vs General Purpose Registers (GPR)
- Decisões de projeto que devem ser levadas em consideração:
  - tecnologia
  - organização
  - linguagens de programação
  - tecnologia em compiladores
  - sistemas operacionais

# Projeto de um ISA

## 5 aspectos principais

- Número de operandos (explícitos) (0,1,2,3)
- Armazenamento do Operando. *Aonde ele está?*
- Endereço Efetivo. *Como é especificado?*
- Tipo & Tamanho dos operandos. *byte, int, float, ...  
como eles são especificados?*
- Operações *add, sub, mul, ...  
como são especificadas?*

# Projeto de um ISA

## Outros aspectos

- Sucessor **Como é especificado?**
- Condições **Como são determinadas?**
- Codificação **Fixa ou Variável? Tamanho?**
- Paralelismo



# Classes básicas de ISA

## Accumulator:

1 address

add A

$acc \leftarrow acc + mem[A]$

1+x address

addx A

$acc \leftarrow acc + mem[A + x]$

## Stack:

0 address

add

$tos \leftarrow tos + next$

## General Purpose Register:

2 address

add A B

$EA(A) \leftarrow EA(A) + EA(B)$

3 address

add A B C

$EA(A) \leftarrow EA(B) + EA(C)$

## Load/Store:

0 Memory

load R1, Mem1

load R2, Mem2

add R1, R2

1 Memory

add R1, Mem2

Instruções da ALU podem ter dois ou três operandos.

Instruções da ALU podem ter 0, 1, 2, 3 operandos.

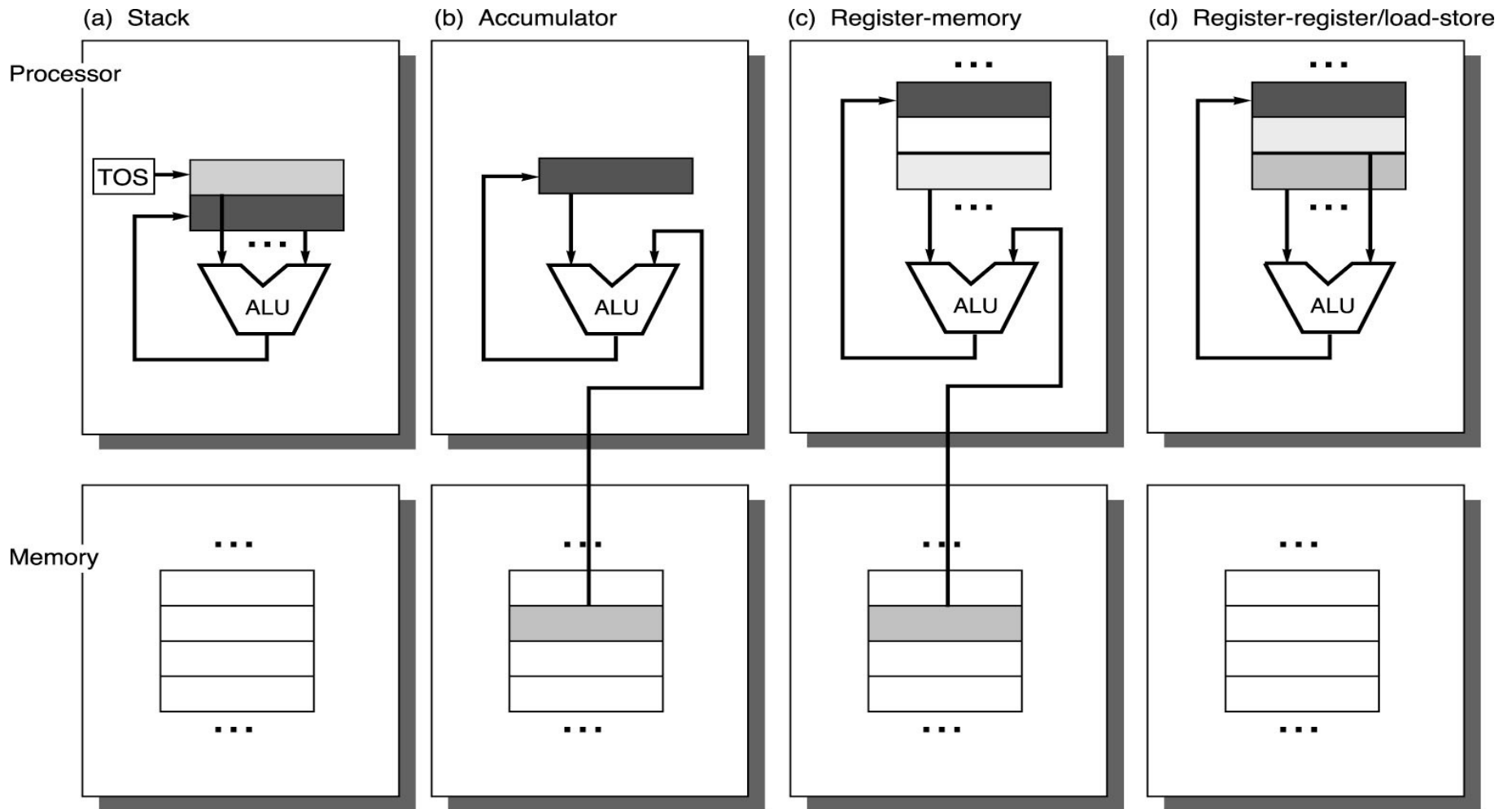
# Classes básicas de ISA

Código nas diferentes classes de endereçamento para:

$$C = A + B.$$

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

# Tipos de Máquinas



# Exemplos de ISAs

Machine	Number of general-purpose registers	Architectural style	Year
EDSAC	1	Accumulator	1949
IBM 701	1	Accumulator	1953
CDC 6600	8	Load-store	1963
IBM 360	16	Register-memory	1964
DEC PDP-8	1	Accumulator	1965
DEC PDP-11	8	Register-memory	1970
Intel 8008	1	Accumulator	1972
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-memory, memory-memory	1977
Intel 8086	1	Extended accumulator	1978
Motorola 68000	16	Register-memory	1980
Intel 80386	8	Register-memory	1985
MIPS	32	Load-store	1985
HP PA-RISC	32	Load-store	1986
SPARC	32	Load-store	1987
PowerPC	32	Load-store	1992
DEC Alpha	32	Load-store	1992

# Modos de Endereçamento

## Interpretando endereços de memória

Qual objeto é acessado em função do endereço e qual o seu tamanho?

Objetos endereçados a byte - um endereço refere-se ao número de bytes contados do início da memória.

**Little Endian** - o byte cujo endereço é `xx00` é o byte menos significativo da palavra.

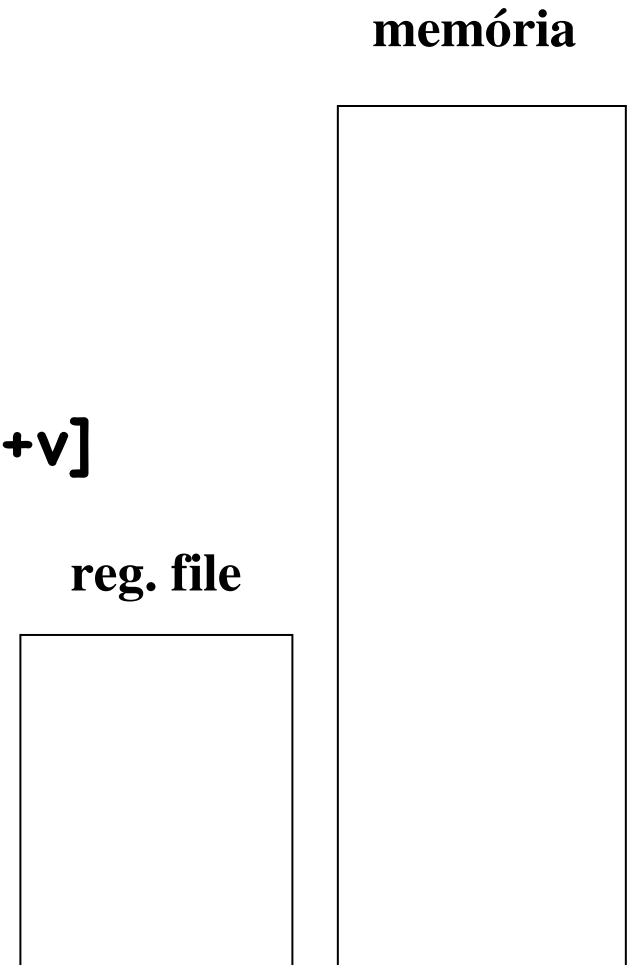
**Big Endian** - o byte cujo endereço é `xx00` é o mais significativo da palavra.

**Alinhamento** - o dado deve ser alinhado em fronteiras iguais a seu tamanho.

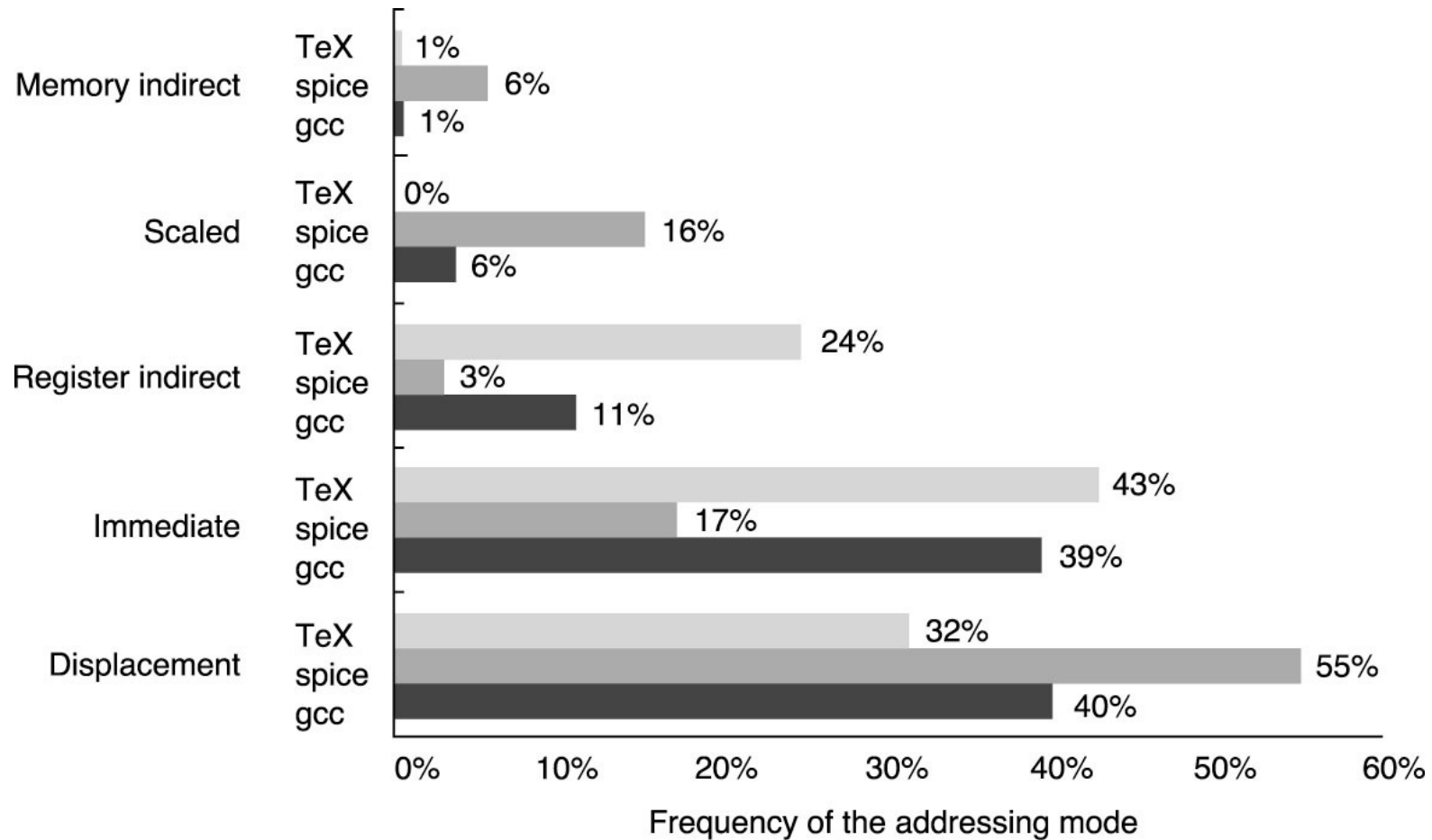
- $\text{address} / \text{sizeof}(\text{datatype}) == 0$
- bytes pode ser alinhado em qualquer endereço
- inteiros de 4 bytes são alinhados em endereços múltiplos de 4

# Modos de Endereçamento

- Register direct  $R_i$
- Immediate (literal)  $v$
- Direct (absolute)  $M[v]$
- Register indirect  $M[R_i]$
- Base+Displacement  $M[R_i + v]$
- Base+Index  $M[R_i + R_j]$
- Scaled Index  $M[R_i + R_j * d + v]$
- Autoincrement  $M[R_i++]$
- Autodecrement  $M[R_i--]$
- Memory indirect  $M[ M[R_i] ]$



# Modos de Endereçamento



# Operações em um ISA

## Tipo

Arithmetic and logical -

Data transfer

Control

System

Floating point

Decimal

String

Multimedia

## Exemplo

and, add

- move, load

- branch, jump, call

- system call, traps

- add, mul, div, sqrt

- add, convert

- move, compare

- 2D, 3D? e.g., Intel MMX and Sun VIS



# Uso de Operações em um ISA

<b>Rank</b>	<b>80x86 instruction</b>	<b>Integer average (% total executed)</b>
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%
<b>Total</b>		<b>96%</b>

FIGURE 2.11 The top 10 instructions for the 80x86.

# Instruções de Controle

(20% das instruções são desvios condicionais)

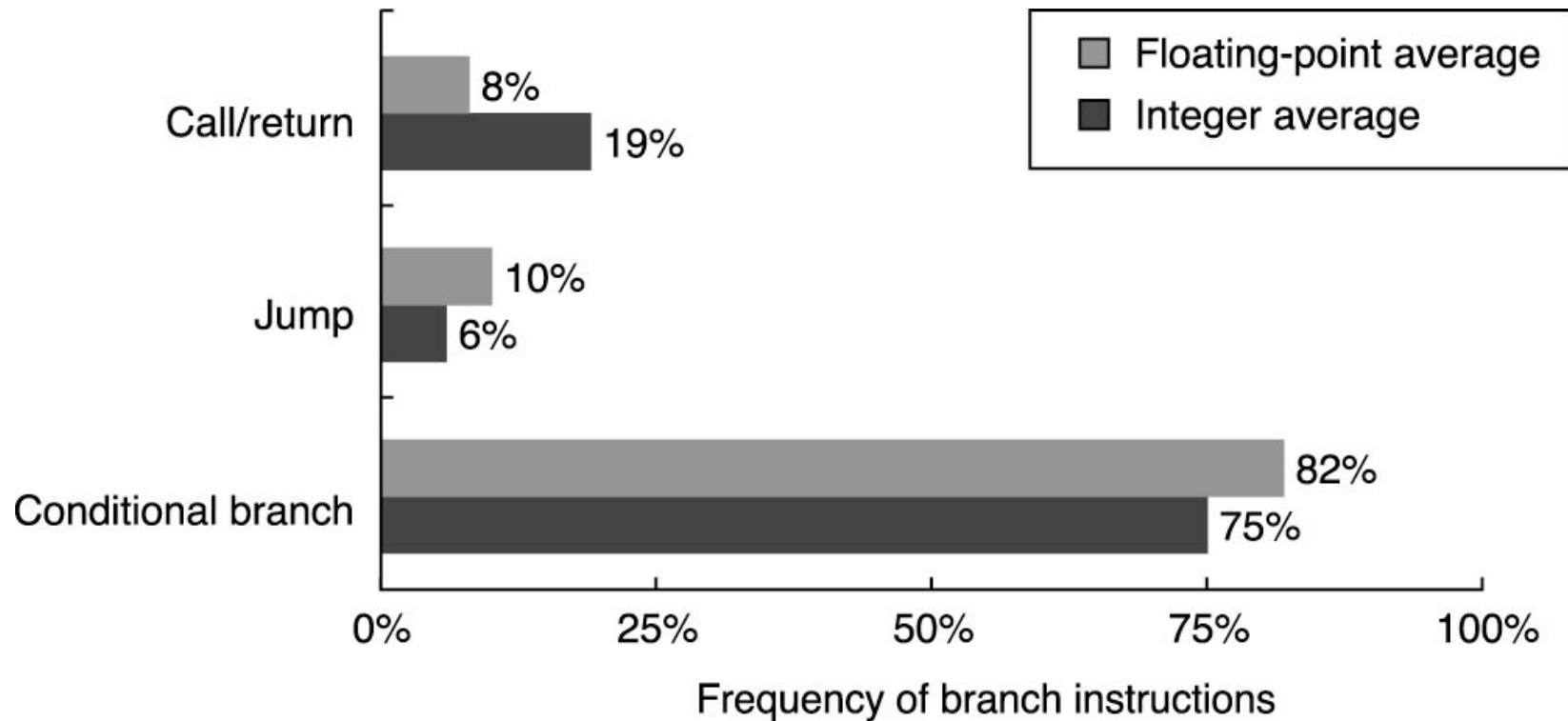
## Control Instructions:

- tomar ou não
- aonde é o alvo
- link return address
- salvar ou restaurar

## Instruções que alteram o PC:

- (condicional) branches, (incondicional) jumps
- chamadas de funções, retorno de funções
- system calls, system returns

# Instruções de Desvio



© 2003 Elsevier Science (USA). All rights reserved.

# Tipos e Tamanhos dos Operandos

O tipo do operando, em geral, é codificado no opcode - LDW significa "loading of a word".

Tamanhos típicos são:

- Character (1 byte)
- Half word (16 bits)
- Word (32 bits)
- Single Precision Floating Point (1 Word)
- Double Precision Floating Point (2 Words)

Inteiros são representados em complemento de dois.

Floating point, em geral, usa o padrão IEEE 754.

Algumas linguagens (como COBOL) usam packed decimal.

# RISC vs CISC

**RISC** = Reduced Instruction Set Computer

- Conjunto de Instruções pequeno
- Instruções de tamanho fixo
- Operações executadas somente em registradores
- Chip simples, em geral, executam com velocidade de clock elevada.

**CISC** = Complex Instruction Set Computer

- Conjunto de Instruções grande
- Instruções Complexas e de tamanho variável
- Operações Memória-Memória

## Projeto $\Rightarrow$ CISC premissas

- Conjunto de Instruções farto pode simplificar o compilador.
- Conjunto de Instruções farto pode aliviar o software.
- Conjunto de Instruções farto pode dar qualidade a arquitetura.
  - Se o tempo de execução for proporcional ao tamanho do programa, técnicas de arquitetura que levem a programas menores também levam a computadores mais rápidos.

## Projeto $\Rightarrow$ RISC premissas

- As funções devem ser simples, a menos que haja uma razão muito forte em contrário.
- Decodificação simples e execução pipelined são mais importantes que o tamanho do programa.
- Tecnologias de compiladores podem ser usadas para simplificar as instruções ao invés de produzirem instruções complexas.

# Codificação do conjunto de Instruções

## codificação de um RISC típico

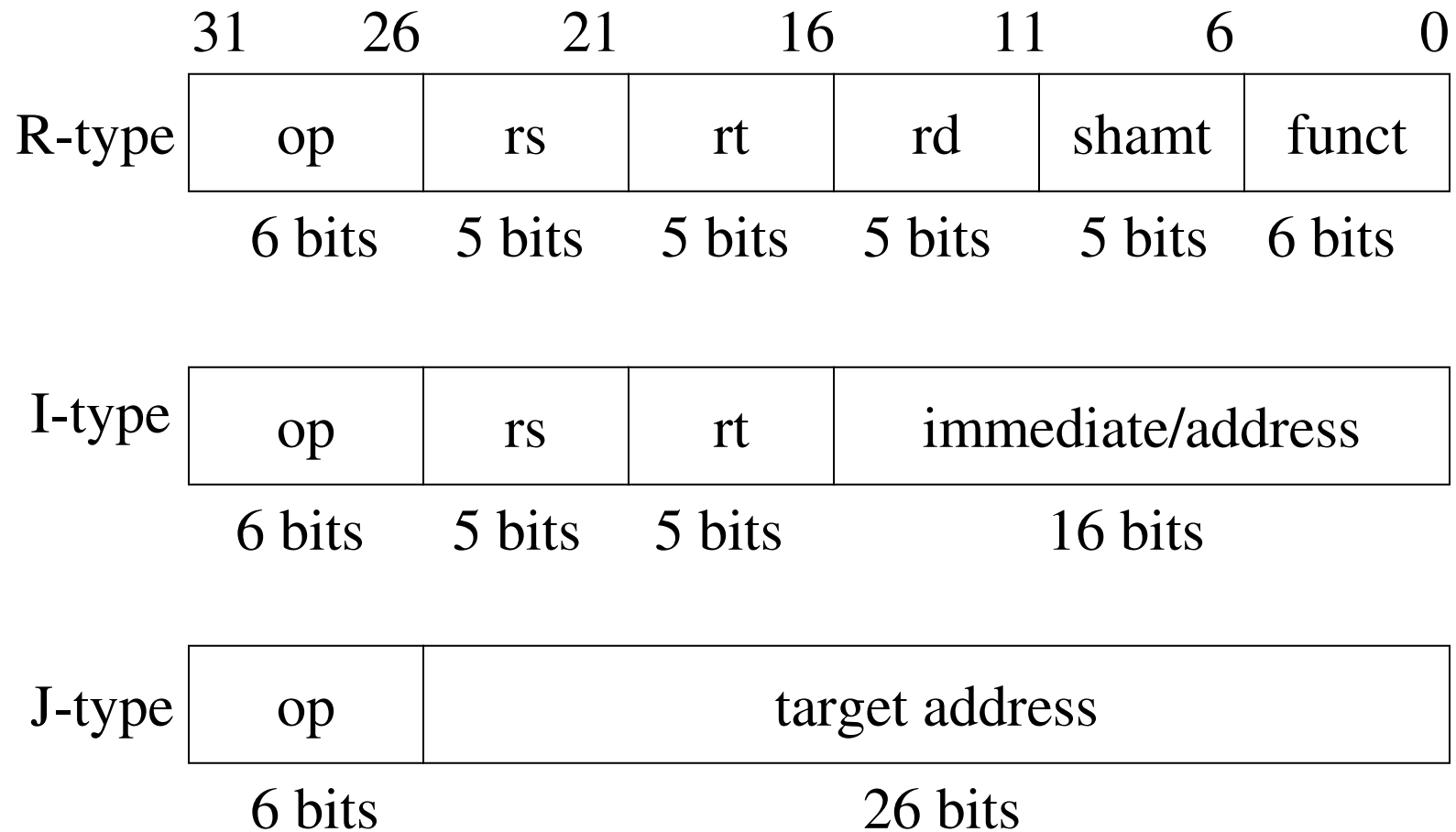
- instruções de tamanho fixo (32-bit) (3 formatos)
- 32 32-bit general-purpose registers (R0 contains zero, números de precisão dupla usam dois registradores)
- Modo de endereçamento simples para load/store:
  - base + displacement (sem indireção)
- Desvios condicionais simples
- Delayed branch para evitar penalidade no pipeline
- Exemplos: DLX, SPARC, MIPS, HP PA-RISC, DEC Alpha, IBM/Motorola PowerPC, Motorola M88000



# Codificação do conjunto de Instruções

## codificação de um RISC típico

### 3 formatos - MIPS



# Arquitetura MIPS

## Organização

Acesso à memória alinhado a:

- Byte - dados

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data

...

- Word - instruções

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

# Arquitetura MIPS

## Organização

- Palavras de 32 bits
- 3 formatos de instruções

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

# Arquitetura MIPS

## Organização

Código C:  $A[300] = h + A[300];$

Código MIPS: lw \$t0, 1200(\$t1)  
add \$t0, \$s2, \$t0  
sw \$t0, 1200(\$t1)

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

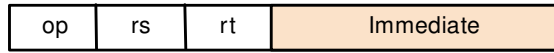
# Conjunto de Registradores MIPS

Name	Register Number	Usage
\$0	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2-3	procedure return values
\$a0-\$a3	4-7	procedure arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved variables
\$t8-\$t9	24-25	more temporaries
\$k0-\$k1	26-27	OS temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	procedure return address

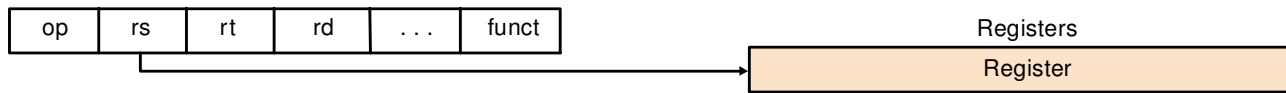
# Arquitetura MIPS

## Organização

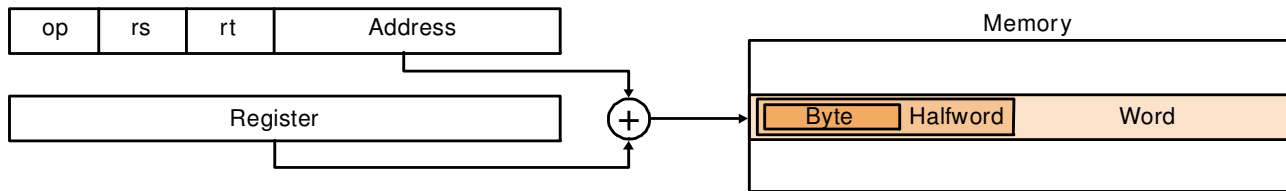
### 1. Immediate addressing



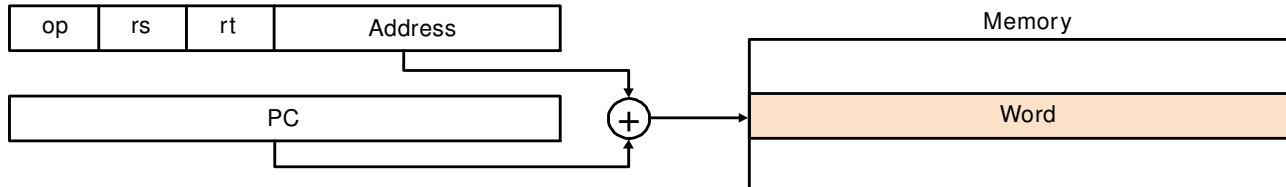
### 2. Register addressing



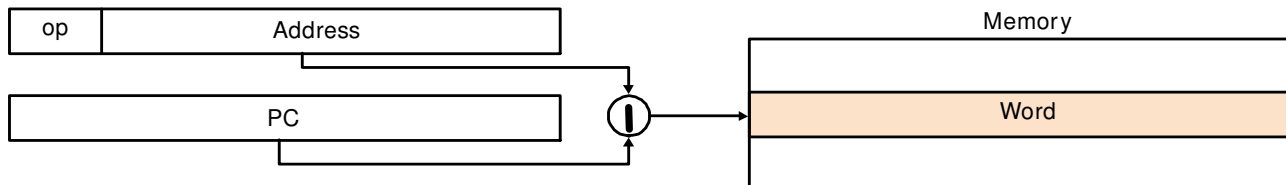
### 3. Base addressing



### 4. PC-relative addressing



### 5. Pseudodirect addressing



# Instruções MIPS

- Soma

## High-level code

```
a = b + c;
```

## MIPS assembly code

```
add a, b, c
```

**add:** mneumônico, indica qual a operação a ser executada

**b, c:** operandos fonte

**a :** operando destino, aonde será armazenado o resultado

# Instruções MIPS

- Subtração

High-level code

$a = b - c;$

MIPS assembly code

sub a, b, c

**sub** : mneumônico, indica qual a operação a ser executada

**b, c**: operandos fonte

**a** : operando destino, aonde será armazenado o resultado



# Instruções MIPS

## Código mais complexo:

### High-level code

```
a = b + c - d;  
// single line comment  
/* multiple line  
comment */
```

### MIPS assembly code

```
add t, b, c # t = b + c  
sub a, t, d # a = t - d
```

# Instruções MIPS

## Operandos

- Um computador necessita de localizações físicas de onde buscar os operandos binários.
- Um computer busca operandos de:
  - Registradores
  - Memória
  - Constantes (também denominados de *imediatos*)

# Instruções MIPS

## Operandos

- Memória é lenta.
- Muitas arquiteturas possuem um conjunto pequeno de registradores (rápidos).
- MIPS tem trinta e dois registradores de 32-bit.
- MIPS é chamado de arquitetura de 32-bit devido seus operandos serem dados de 32-bit.

(Uma versão MIPS de 64-bit também existe.)

# Conjunto de registradores MIPS

Name	Register Number	Usage
\$0	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2-3	procedure return values
\$a0-\$a3	4-7	procedure arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved variables
\$t8-\$t9	24-25	more temporaries
\$k0-\$k1	26-27	OS temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	procedure return address

# Instruções MIPS

## Com os Registradores

### High-level code

```
a = b + c;
```

### MIPS assembly code

```
# $s0 = a, $s1 = b, $s2 = c  
add $s0, $s1, $s2
```

# Instruções MIPS

- Operandos em Memória
  - word-addressable memory

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

# Instruções MIPS

- Lendo uma word-addressable memory

## Assembly code

```
lw $s3, 1($0) # read memory word 1 into $s3
               # Load Word
```

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

# Instruções MIPS

- Escrevendo uma word-addressable memory

## Assembly code

```
Sw $t4, 0x7($0) # write $t4 to memory word 7
                # Store Word
```

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

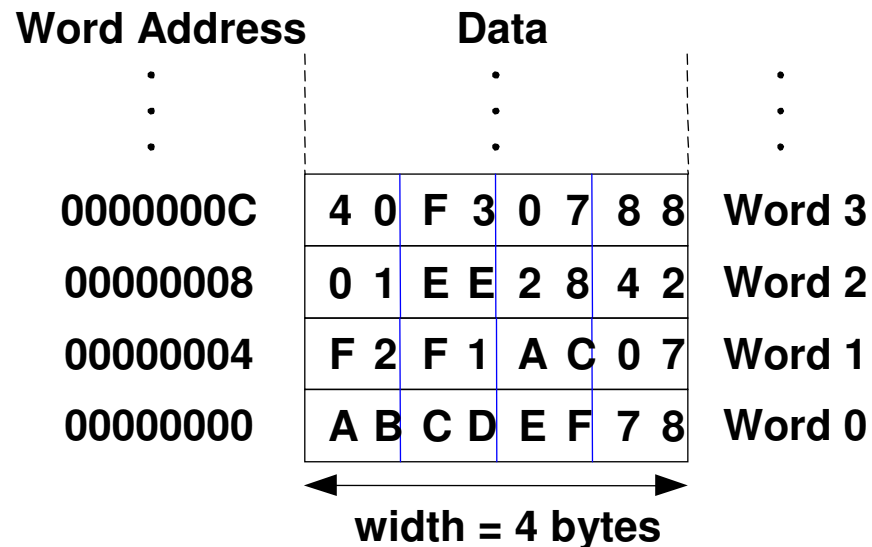


# Instruções MIPS

- Operandos em Memória

- byte-addressable memory

- » Load e store um único bytes: load byte (lb) e store byte (sb)
    - » Cada word de 32-bit tem 4 bytes, assim o endereço deve ser incrementado de 4

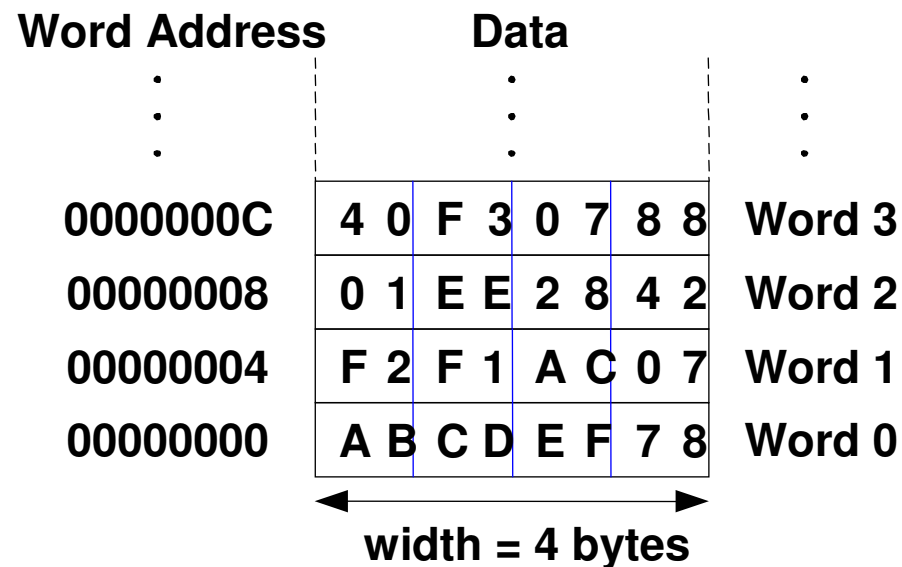


# Instruções MIPS

- Lendo uma byte-addressable memory

## MIPS assembly code

```
lw $s3, 4($0) # read memory word 1 into $s3
```

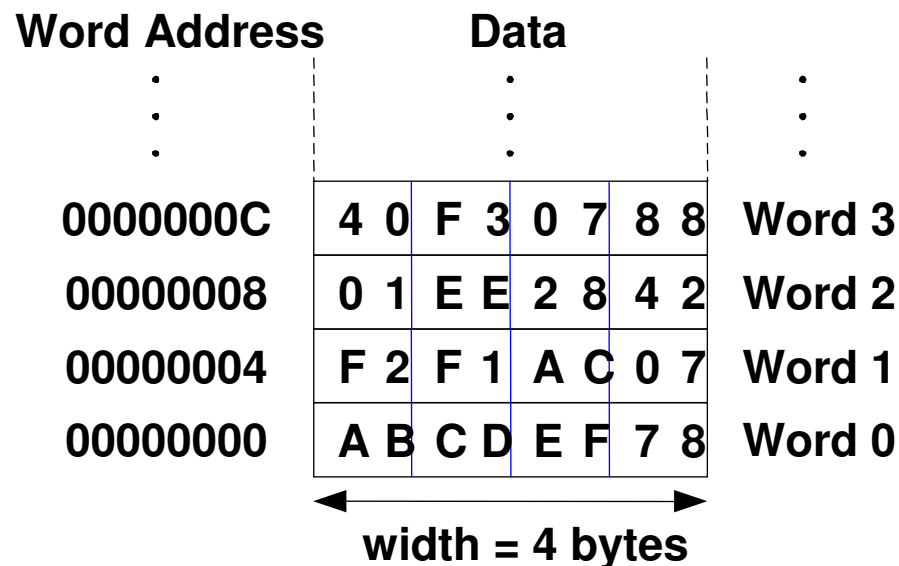


# Instruções MIPS

- Escrevendo uma byte-addressable memory

## MIPS assembly code

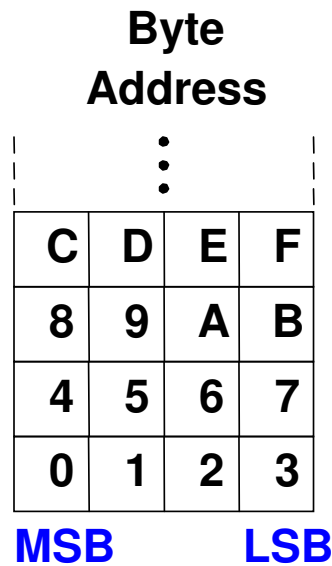
```
sw $t7, 44($0) # write $t7 into memory word 11
```



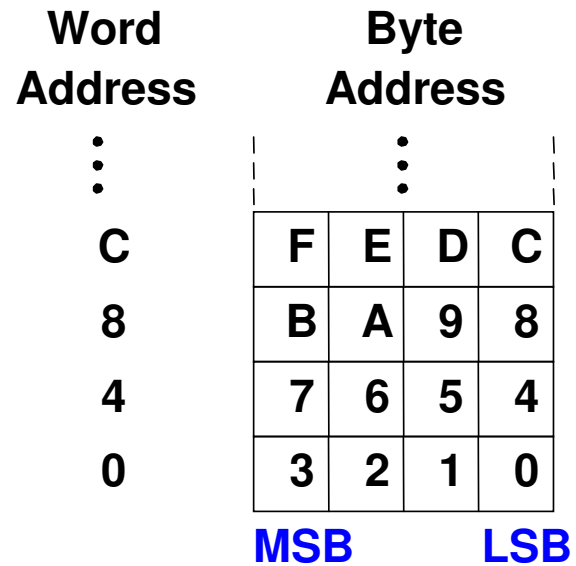
# Instruções MIPS

- Big-Endian e Little-Endian
  - Como são numerados os bytes na word

## Big-Endian



## Little-Endian



# Instruções MIPS

- Big- e Little-Endian Exemplos:
- Suponha que inicialmente `$t0` contém `0x23456789`. Após o seguinte trecho de programa ser executado em um sistema big-endian, qual o valor de `$s0`. E em um sistema little-endian?

```
sw $t0, 0($0)
```

```
lb $s0, 1($0)
```

# Instruções MIPS

- **Big- e Little-Endian Exemplos:**

- Suponha que inicialmente  $\$t0$  contém  $0x23456789$ . Após o seguinte trecho de programa ser executado em um sistema big-endian, qual o valor de  $\$s0$ . E em um sistema little-endian?

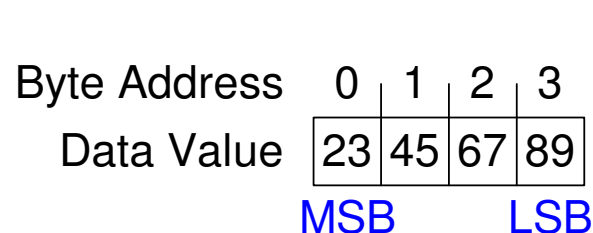
```
sw $t0, 0($0)
```

```
lb $s0, 1($0)
```

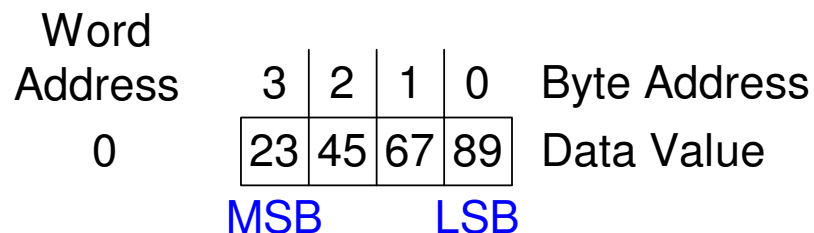
- **Big-endian:**  $0x00000045$

- **Little-endian:**  $0x00000067$

## Big-Endian



## Little-Endian



# Instruções MIPS

- Operandos: Constantes/Imediatos
  - Um imediato é um número de 16-bit em complemento de dois.

## High-level code

```
a = a + 4;  
b = a - 12;
```

## MIPS assembly code

```
# $s0 = a, $s1 = b  
addi $s0, $s0, 4  
addi $s1, $s0, -12
```

# Instruções MIPS

- Linguagem de Máquina
  - Computadores só “conhecem” 1's e 0's
  - Linguagem de Máquina: representação binária das instruções
  - Instruções de 32-bit
    - » Simplicidade em favor da regularidade: dados e instruções de 32-bit
  - Três formatos de instruções :
    - » **R-Type**: register operands
    - » **I-Type**: immediate operand
    - » **J-Type**: para jump



# Instruções MIPS

- **R-type: *Register-type***

- 3 operandos registradores:

- » rs, rt: source registers

- » rd: destination register

- Outros campos:

- » op: código da *operação ou opcode*

- » funct: função

juntos, o opcode e a função informam a operação a ser executada

- » shamt: a quantidade de *shift para instruções de deslocamento*

## R-Type



# Instruções MIPS

## Assembly Code

```
add $s0, $s1, $s2
```

```
sub $t0, $t3, $t5
```

## Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34

6 bits      5 bits      5 bits      5 bits      5 bits      6 bits

## Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)

6 bits      5 bits      5 bits      5 bits      5 bits      6 bits

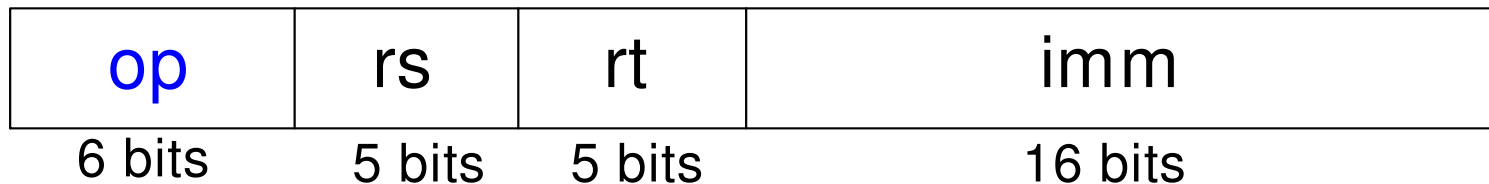
**Nota:** a ordem dos registradores no código assembly:

```
add rd, rs, rt
```

# Instruções MIPS

- **I-Type: Immediate-Type**
  - 3 operands:
    - » **rs, rt:** register operands
    - » **imm:** 16-bit em complemento de dois immediate
  - **Outros campos:**
    - » **op:** opcode

## I-Type



# Instruções MIPS

- Exemplo I-Type:

## Assembly Code

```
addi $s0, $s1, 5
addi $t0, $s3, -12
lw    $t2, 32($0)
sw    $s1, 4($t1)
```

## Field Values

	op	rs	rt	imm
	8	17	16	5
	8	19	8	-12
	35	0	10	32
	43	9	17	4

6 bits    5 bits    5 bits    16 bits

**Nota:** a ordem dos registradores no código assembly:

```
addi rt, rs, imm
lw    rt, imm(rs)
sw    rt, imm(rs)
```

## Machine Code

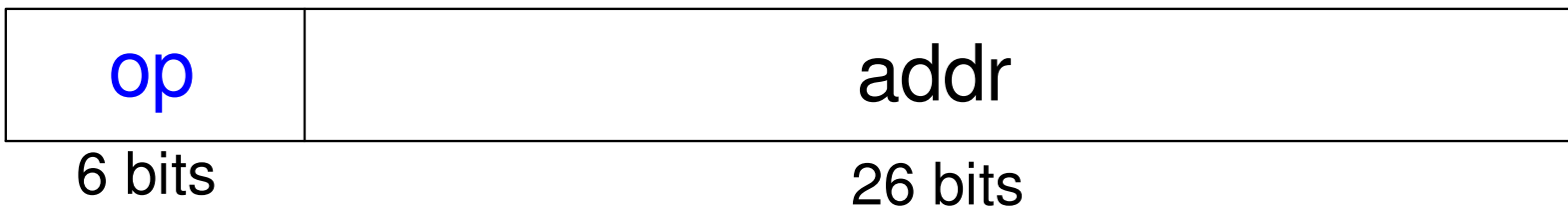
op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)

6 bits    5 bits    5 bits    16 bits

# Instruções MIPS

- J-Type: Jump-Type
  - 26-bit address operand (addr)
  - Usado nas instruções jump (j)

## J-Type



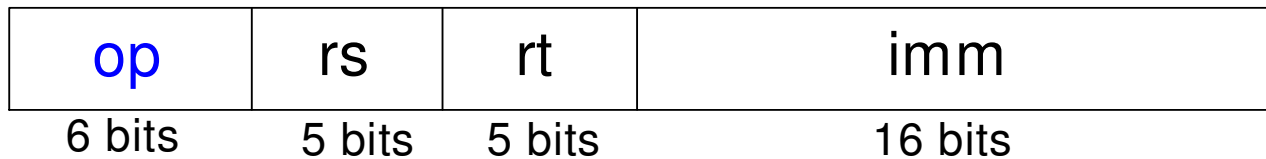
# Instruções MIPS

- Formatos das Instruções

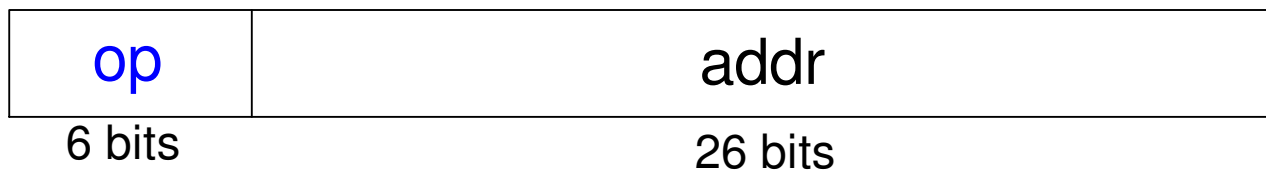
## R-Type



## I-Type



## J-Type



# Programa Armazenado

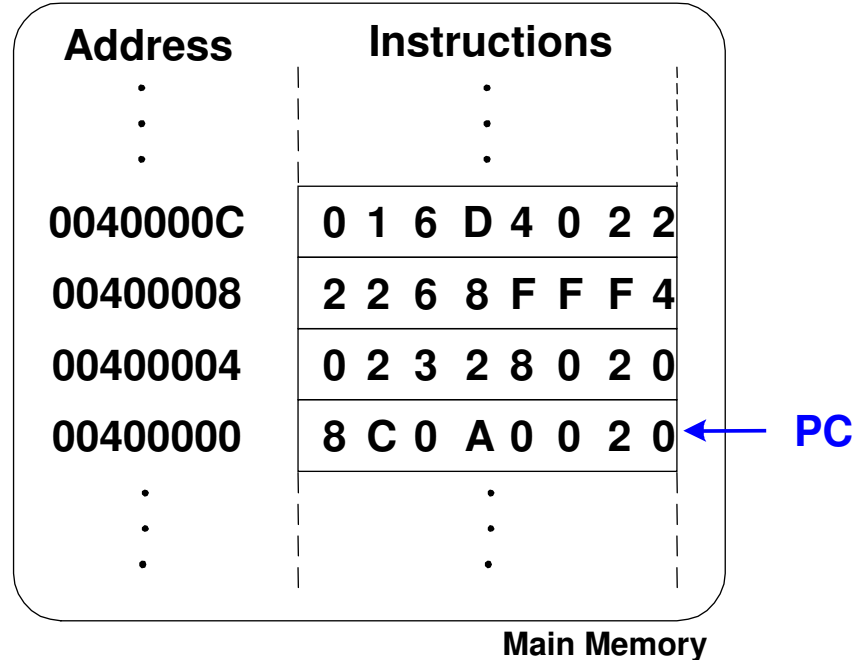
- Instruções e dados de 32-bit armazenados na memória
- Seqüência de instruções: é a única diferença entre dois programas
- Execução de um novo programa:
  - Simplesmente armazene o novo programa na memória
- Execução do programa pelo hardware do processador:
  - *fetches* (reads) as instruções da memória em seqüência
  - Executa a operação especificada
- Um *program counter* (PC) indica a instrução corrente (ou a próxima instrução).
- no MIPS, programas tipicamente iniciam no endereço de memória 0x00400000.

# Programa Armazenado

- Exemplo:

Assembly Code	Machine Code
lw \$t2, 32(\$0)	0x8C0A0020
add \$s0, \$s1, \$s2	0x02328020
addi \$t0, \$s3, -12	0x2268FFF4
sub \$t0, \$t3, \$t5	0x016D4022

## Stored Program





# Interpretando o código de Máquina

Inicia com o opcode

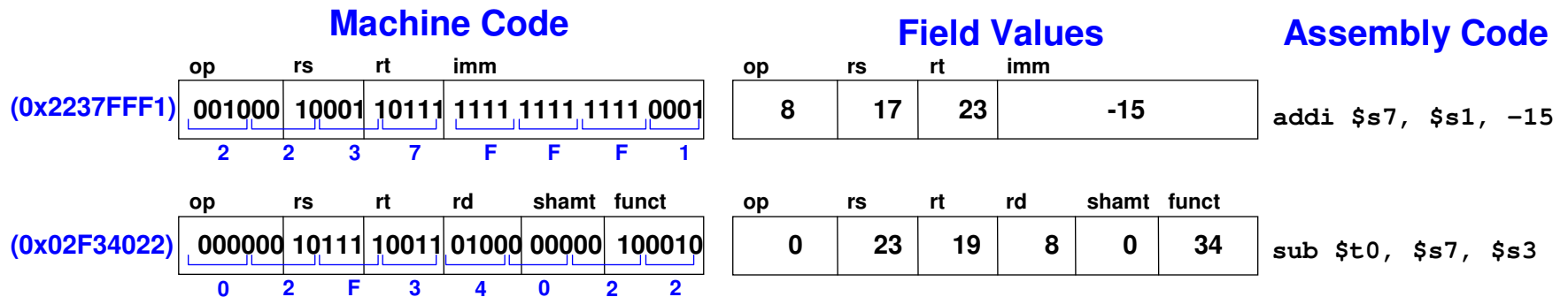
Opcode informa como fazer o parse dos bits remanecentes  
se opcode é todo 0's

R-type instruction

Function bits informa qual instrução é

Caso contrário

opcode informa qual é a instrução



# Instruções Lógicas

- `and`, `or`, `xor`, `nor`

- `and`: útil para mascarar bits

- » Estraindo o byte menos significativo de uma word:

$$0xF234012F \text{ AND } 0xFF = 0x0000002F$$

- `or`: útil para combinar bits

- » Combinar `0xF2340000` com `0x000012BC`:

$$0xF2340000 \text{ OR } 0x000012BC = 0xF23412BC$$

- `nor`: útil para inverter bits:

- »  $A \text{ NOR } \$0 = \text{NOT } A$

- `andi`, `ori`, `xori`

- O imediato de 16-bit é zero-extended (*não sign-extended*)

# Instruções Lógicas

## Source Registers

\$s1	1111	1111	1111	1111	0000	0000	0000	0000
\$s2	0100	0110	1010	0001	1111	0000	1011	0111

## Assembly Code

```
and $s3, $s1, $s2  
or $s4, $s1, $s2  
xor $s5, $s1, $s2  
nor $s6, $s1, $s2
```

## Result

\$s3	0100	0110	1010	0001	0000	0000	0000	0000
\$s4	1111	1111	1111	1111	1111	0000	1011	0111
\$s5	1011	1001	0101	1110	1111	0000	1011	0111
\$s6	0000	0000	0000	0000	0000	1111	0100	1000

# Instruções Lógicas

## Source Values

\$s1	0000	0000	0000	0000	0000	0000	1111	1111
------	------	------	------	------	------	------	------	------

imm	0000	0000	0000	0000	1111	1010	0011	0100
-----	------	------	------	------	------	------	------	------

← zero-extended →

## Assembly Code

andi \$s2, \$s1, 0xFA34

ori \$s3, \$s1, 0xFA34

xori \$s4, \$s1, 0xFA34

## Result

\$s2	0000	0000	0000	0000	0000	0000	0011	0100
------	------	------	------	------	------	------	------	------

\$s3	0000	0000	0000	0000	1111	1010	1111	1111
------	------	------	------	------	------	------	------	------

\$s4	0000	0000	0000	0000	1111	1010	1100	1011
------	------	------	------	------	------	------	------	------

# Instruções Shift

- **sll: shift left logical**
  - Exemplo: `sll $t0, $t1, 5 # $t0 <= $t1 << 5`
- **srl: shift right logical**
  - Exemplo: `srl $t0, $t1, 5 # $t0 <= $t1 >> 5`
- **sra: shift right arithmetic**
  - Exemplo: `sra $t0, $t1, 5 # $t0 <= $t1 >>> 5`

## Variable shift instructions:

- **sllv: shift left logical variable**
  - Exemplo: `sll $t0, $t1, $t2 # $t0 <= $t1 << $t2`
- **srlv: shift right logical variable**
  - Exemplo: `srl $t0, $t1, $t2 # $t0 <= $t1 >> $t2`
- **srav: shift right arithmetic variable**
  - Exemplo: `sra $t0, $t1, $t2 # $t0 <= $t1 >>> $t2`

# Instruções Shift

## Assembly Code

## Field Values

	op	rs	rt	rd	shamt	funct
sll \$t0, \$s1, 2	0	0	17	8	2	0
srl \$s2, \$s1, 2	0	0	17	18	2	2
sra \$s3, \$s1, 2	0	0	17	19	2	3

6 bits      5 bits      5 bits      5 bits      5 bits      6 bits

## Machine Code

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)

6 bits      5 bits      5 bits      5 bits      5 bits      6 bits

# Gerando Constantes

- Constantes de 16-bit usando `addi`:

## High-level code

```
// int is a 32-bit signed word  
int a = 0x4f3c;
```

## MIPS assembly code

```
# $s0 = a  
addi $s0, $0, 0x4f3c
```

- Constantes de 32-bit usando *load upper immediate* (`lui`) e `ori`:

(`lui` loads o imediato de 16-bit na metade mais significativa do registrador seta a menos significativa com 0.)

## High-level code

```
int a = 0xFEDC8765;
```

## MIPS assembly code

```
# $s0 = a  
lui $s0, 0xFEDC  
ori $s0, $s0, 0x8765
```

# Multiplicação e Divisão

- Registradores especiais: `lo`, `hi`
- Multiplicação 32 × 32 bit, resultado de 64 bit
  - `mult $s0, $s1`
  - Resultado em `hi`, `lo`
- Divisão 32-bit, quociente de 32-bit, resto de 32-bit
  - `div $s0, $s1`
  - Quociente em `lo`
  - Resto em `hi`



# Desvios

- Todo programa executa instruções for a da seqüência.
- Tipos de desvios (branches):
  - Conditional branches:
    - » branch if equal (beq)
    - » branch if not equal (bne)
  - Unconditional branches:
    - » jump (j)
    - » jump register (jr)
    - » jump and link (jal)

# Beq: exemplo

```
# MIPS assembly
```

```
addi $s0, $0, 4      # $s0 = 0 + 4 = 4
addi $s1, $0, 1      # $s1 = 0 + 1 = 1
sll  $s1, $s1, 2      # $s1 = 1 << 2 = 4
beq  $s0, $s1, target # branch is taken
addi $s1, $s1, 1      # not executed
sub  $s1, $s1, $s0     # not executed

target:               # label
add  $s1, $s1, $s0     # $s1 = 4 + 4 = 8
```

# Bne: exemplo

# MIPS assembly

```
addi    $s0, $0, 4           # $s0 = 0 + 4 = 4
addi    $s1, $0, 1           # $s1 = 0 + 1 = 1
sll     $s1, $s1, 2          # $s1 = 1 << 2 = 4
bne     $s0, $s1, target     # branch not taken
addi    $s1, $s1, 1          # $s1 = 4 + 1 = 5
sub     $s1, $s1, $s0        # $s1 = 5 - 4 = 1
```

target:

```
add     $s1, $s1, $s0        # $s1 = 1 + 4 = 5
```

# Desvio incondicional (j)

# MIPS assembly

```
addi $s0, $0, 4           # $s0 = 4
    addi $s1, $0, 1       # $s1 = 1
    j    target          # jump to target
    sra  $s1, $s1, 2      # not executed
    addi $s1, $s1, 1      # not executed
    sub  $s1, $s1, $s0    # not executed

target:
    add  $s1, $s1, $s0    # $s1 = 1 + 4 = 5
```

# Desvio incondicional (jr)

# MIPS assembly

```
0x00002000      addi $s0, $0, 0x2010
0x00002004      jr          $s0
0x00002008      addi $s1, $0, 1
0x0000200C      sra   $s1, $s1, 2
0x00002010      lw    $s3, 44($s1)
```

# Construções de Alto Nível

- **if statements**
- **if/else statements**
- **while loops**
- **for loops**

# If Statement

## High-level code

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

```
f = f - i;
```

## MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
```

```
# $s3 = i, $s4 = j
```

```
    bne $s3, $s4, L1
```

```
    add $s0, $s1, $s2
```

```
L1: sub $s0, $s0, $s3
```

Note que em assembly o teste é o oposto ( $i \neq j$ ) do teste em alto nível ( $i == j$ ).

# If / Else Statement

## High-level code

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

## MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
        bne $s3, $s4, L1
        add $s0, $s1, $s2
        j   done
L1:     sub $s0, $s0, $s3
done:
```



# While Loops

## High-level code

```
// determines the power
// of x such that 2x = 128
int pow = 1;
int x    = 0;

while (pow != 128) {
    pow = pow * 2;
    x   = x + 1;
}
```

## MIPS assembly code

```
# $s0 = pow, $s1 = x

addi $s0, $0, 1
add  $s1, $0, $0
addi $t0, $0, 128

while: beq  $s0, $t0, done
      sll  $s0, $s0, 1
      addi $s1, $s1, 1
      j   while

done:
```

# For Loops

A forma geral de um for loop é:

```
for (inicialização; condição; loop)
    corpo do loop
```

- **inicialização**: executado antes do loop
- **condição**: testada no início de cada iteração
- **loop**: executa no fim de cada iteração
- **Corpo do loop**: executado para cada vez que a condição é satisfeita

# For Loops

## High-level code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}
```

## MIPS assembly code

```
# $s0 = i, $s1 = sum
addi $s1, $0, 0
add  $s0, $0, $0
addi $t0, $0, 10
for:  beq  $s0, $t0, done
      add  $s1, $s1, $s0
      addi $s0, $s0, 1
      j   for
done:
```

# For Loops: Using `slt`

## High-level code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
    sum = sum + i;
}
```

## MIPS assembly code

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101

loop:   slt  $t1, $s0, $t0
        beq  $t1, $0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    loop

done:
```

`$t1 = 1 if  $i < 101$ .`

# Arrays

- Utilizado para acesso a uma grande quantidade de dados similares
- Elemento do Array: acesso por meio de um índice
- Tamanho do Array: número de elementos no array

# Array: exemplo

- Array com 5 elementos
- Endereço base = 0x12348000 (endereço do primeiro elemento, array[0])
- Primeiro passo para acesso a um array: carregar o endereço base em um registrador

0x12340010	array[4]
0x1234800C	array[3]
0x12348008	array[2]
0x12348004	array[1]
0x12348000	array[0]

# Array

```
// high-level code
```

```
int array[5];  
array[0] = array[0] * 2;  
array[1] = array[1] * 2;
```

```
# MIPS assembly code
```

```
# array base address = $s0
```

```
lui $s0, 0x1234          # put 0x1234 in upper half of $s0  
ori $s0, $s0, 0x8000    # put 0x8000 in lower half of $s0  
  
lw $t1, 0($s0)          # $t1 = array[0]  
sll $t1, $t1, 1         # $t1 = $t1 * 2  
sw $t1, 0($s0)          # array[0] = $t1  
  
lw $t1, 4($s0)          # $t1 = array[1]  
sll $t1, $t1, 1         # $t1 = $t1 * 2  
sw $t1, 4($s0)          # array[1] = $t1
```

# Array Usando For

```
// high-level code
```

```
int array[1000];
```

```
int i;
```

```
for (i=0; i < 1000; i = i + 1)
```

```
    array[i] = array[i] * 8;
```



# Array Usando For

# MIPS assembly code

# \$s0 = array base address, \$s1 = i

# initialization code

```
lui    $s0, 0x23B8          # $s0 = 0x23B80000
```

```
ori    $s0, $s0, 0xF000    # $s0 = 0x23B8F000
```

```
addi   $s1, $0, 0          # i = 0
```

```
addi   $t2, $0, 1000       # $t2 = 1000
```

loop:

```
slt    $t0, $s1, $t2       # i < 1000?
```

```
beq    $t0, $0, done       # if not then done
```

```
sll    $t0, $s1, 2         # $t0 = i * 4 (byte offset)
```

```
add    $t0, $t0, $s0       # address of array[i]
```

```
lw     $t1, 0($t0)         # $t1 = array[i]
```

```
sll    $t1, $t1, 3         # $t1 = array[i] * 8
```

```
sw     $t1, 0($t0)         # array[i] = array[i] * 8
```

```
addi   $s1, $s1, 1        # i = i + 1
```

```
j      loop                # repeat
```

done:

# Chamada de Procedimento

## High-level code

```
void main()  
{  
    int y;  
        y = sum(42, 7);  
        ...  
}  
  
int sum(int a, int b)  
{  
    return (a + b);  
}
```

# Chamada de Procedimento

## Chamada de Procedimento - convenções:

- Chamada:
  - Passa **argumentos** para o procedimento.
- Procedimento:
  - Não deve sobre-escrever os registradores nem a memória usados por quem chama
  - Retorna ao ponto de chamada
  - Retorna o resultado para quem chama

## Convenções MIPS:

- Chamada de procedimento: jump e link (jal)
- Retorno de procedimento: jump register (jr)
- Argumentos: \$a0 - \$a3
- Retorno do valor calculado: \$v0

# Chamada de Procedimento

## High-level code

```
int main() {  
    simple();  
    a = b + c;  
}
```

```
void simple() {  
    return;  
}
```

## MIPS assembly code

```
0x00400200 main: jal  simple  
0x00400204          add  $s0, $s1, $s2  
...
```

```
0x00401020 simple: jr  $ra
```

# Chamada de Procedimento

## High-level code

```
int main() {  
    simple();  
    a = b + c;  
}
```

```
void simple() {  
    return;  
}
```

## MIPS assembly code

```
0x00400200 main: jal  simple  
0x00400204          add  $s0, $s1, $s2  
...  
  
0x00401020 simple: jr  $ra
```

**jal:** salta para `simple` e salva `PC+4` no registrador de endereço de retorno (`$ra`), neste caso, `$ra = 0x00400204` após `jal` ser executado.

**jr \$ra:** salta para o endereço em `$ra`, neste caso `0x00400204`.

# Argumentos e Retorno de Valores

## Convenção MIPS c:

- Argumentos:  $\$a0$  -  $\$a3$
- Retorno:  $\$v0$

# Argumentos e Retorno de Valores

## High-level code

```
int main()
{
    int y;
    ...
    y = diffosums(2, 3, 4, 5); // 4 arguments
    ...
}

int diffosums(int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    return result;           // return value
}
```

# Argumentos e Retorno de Valores

## Código MIPS (assembly)

```
# $s0 = y
```

```
main:
```

```
...
```

```
addi $a0, $0, 2 # argument 0 = 2
```

```
addi $a1, $0, 3 # argument 1 = 3
```

```
addi $a2, $0, 4 # argument 2 = 4
```

```
addi $a3, $0, 5 # argument 3 = 5
```

```
jal diffofsums # call procedure
```

```
add $s0, $v0, $0 # y = returned value
```

```
...
```

```
# $s0 = result
```

```
diffofsums:
```

```
add $t0, $a0, $a1 # $t0 = f + g
```

```
add $t1, $a2, $a3 # $t1 = h + i
```

```
sub $s0, $t0, $t1 # result = (f + g) - (h + i)
```

```
add $v0, $s0, $0 # put return value in $v0
```

```
jr $ra # return to caller
```



# Argumentos e Retorno de Valores

## Código MIPS (assembly)

```
# $s0 = result
```

```
diffofsums:
```

```
add $t0, $a0, $a1 # $t0 = f + g
```

```
add $t1, $a2, $a3 # $t1 = h + i
```

```
sub $s0, $t0, $t1 # result = (f + g) - (h + i)
```

```
add $v0, $s0, $0 # put return value in $v0
```

```
jr $ra # return to caller
```

- diffofsums sobre-escreve 3 registradores: \$t0, \$t1, e \$s0
- diffofsums pode usar a *pilha* para armazenar temporariamente os registradores

# Pilha

- Cresce para baixo (dos endereços maiores para os menores)
- Stack pointer: `$sp`, aponta para o topo da pilha

Address	Data
7FFFFFFC	12345678
7FFFFFF8	
7FFFFFF4	
7FFFFFF0	
⋮	⋮
⋮	⋮
⋮	⋮

← `$sp`

Address	Data
7FFFFFFC	12345678
7FFFFFF8	AABBCCDD
7FFFFFF4	11223344
7FFFFFF0	
⋮	⋮
⋮	⋮
⋮	⋮

← `$sp`

# Chamada de Procedimentos Usando a Pilha

- O procedimento chamado não deve provocar nenhum efeito colateral.
- Más `diffosums` sobre-escreve 3 registradores:  
`$t0, $t1, $s0`

```
# MIPS assembly
```

```
# $s0 = result
```

```
diffosums:
```

```
    add $t0, $a0, $a1    # $t0 = f + g
```

```
    add $t1, $a2, $a3    # $t1 = h + i
```

```
    sub $s0, $t0, $t1    # result = (f + g) - (h + i)
```

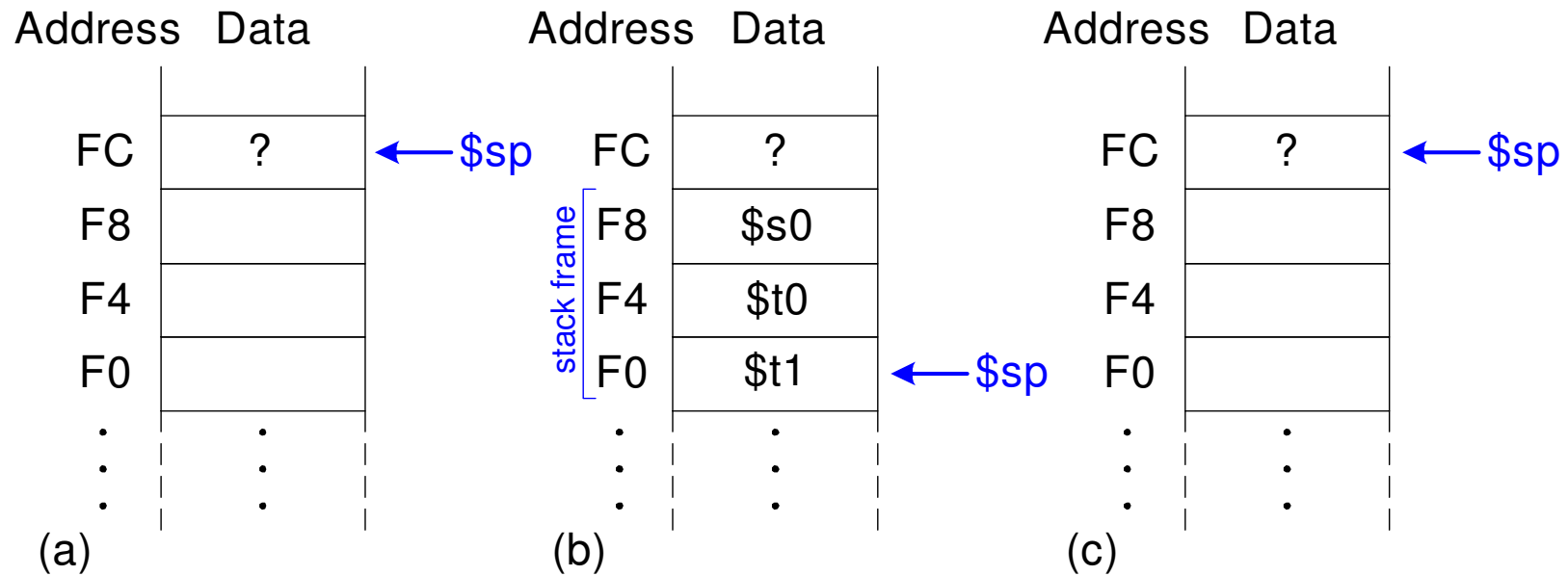
```
    add $v0, $s0, $0     # put return value in $v0
```

```
    jr  $ra             # return to caller
```

# Chamada de Procedimentos Usando a Pilha

```
# $s0 = result
diffosums:
    addi $sp, $sp, -12 # make space on stack
                        # to store 3 registers
    sw   $s0, 8($sp)   # save $s0 on stack
    sw   $t0, 4($sp)   # save $t0 on stack
    sw   $t1, 0($sp)   # save $t1 on stack
    add  $t0, $a0, $a1 # $t0 = f + g
    add  $t1, $a2, $a3 # $t1 = h + i
    sub  $s0, $t0, $t1 # result = (f + g) - (h + i)
    add  $v0, $s0, $0  # put return value in $v0
    lw   $t1, 0($sp)   # restore $t1 from stack
    lw   $t0, 4($sp)   # restore $t0 from stack
    lw   $s0, 8($sp)   # restore $s0 from stack
    addi $sp, $sp, 12  # deallocate stack space
    jr   $ra           # return to caller
```

# A Pilha durante a Chamada de `diffosums`



# Registadores

<b>Preserved</b> <i>Callee-Saved</i>	<b>Nonpreserved</b> <i>Caller-Saved</i>
<code>\$s0 - \$s7</code>	<code>\$t0 - \$t9</code>
<code>\$ra</code>	<code>\$a0 - \$a3</code>
<code>\$sp</code>	<code>\$v0 - \$v1</code>
stack above <code>\$sp</code>	stack below <code>\$sp</code>

# Chamadas Múltiplas de Procedimentos

proc1:

```
addi $sp, $sp, -4    # make space on stack
sw   $ra, 0($sp)    # save $ra on stack
jal  proc2
...
lw   $ra, 0($sp)    # restore $r0 from stack
addi $sp, $sp, 4    # deallocate stack space
jr   $ra            # return to caller
```

# Armazenando Registradores na Pilha

```
# $s0 = result
```

```
diffosums:
```

```
    addi $sp, $sp, -4    # make space on stack to
                        # store one register
    sw   $s0, 0($sp)    # save $s0 on stack
    add  $t0, $a0, $a1  # $t0 = f + g
    add  $t1, $a2, $a3  # $t1 = h + i
    sub  $s0, $t0, $t1  # result = (f + g) - (h + i)
    add  $v0, $s0, $0   # put return value in $v0
    lw   $s0, 0($sp)    # restore $s0 from stack
    addi $sp, $sp, 4    # deallocate stack space
    jr   $ra           # return to caller
```



# Chamada Recursiva de Procedimentos

## High-level code

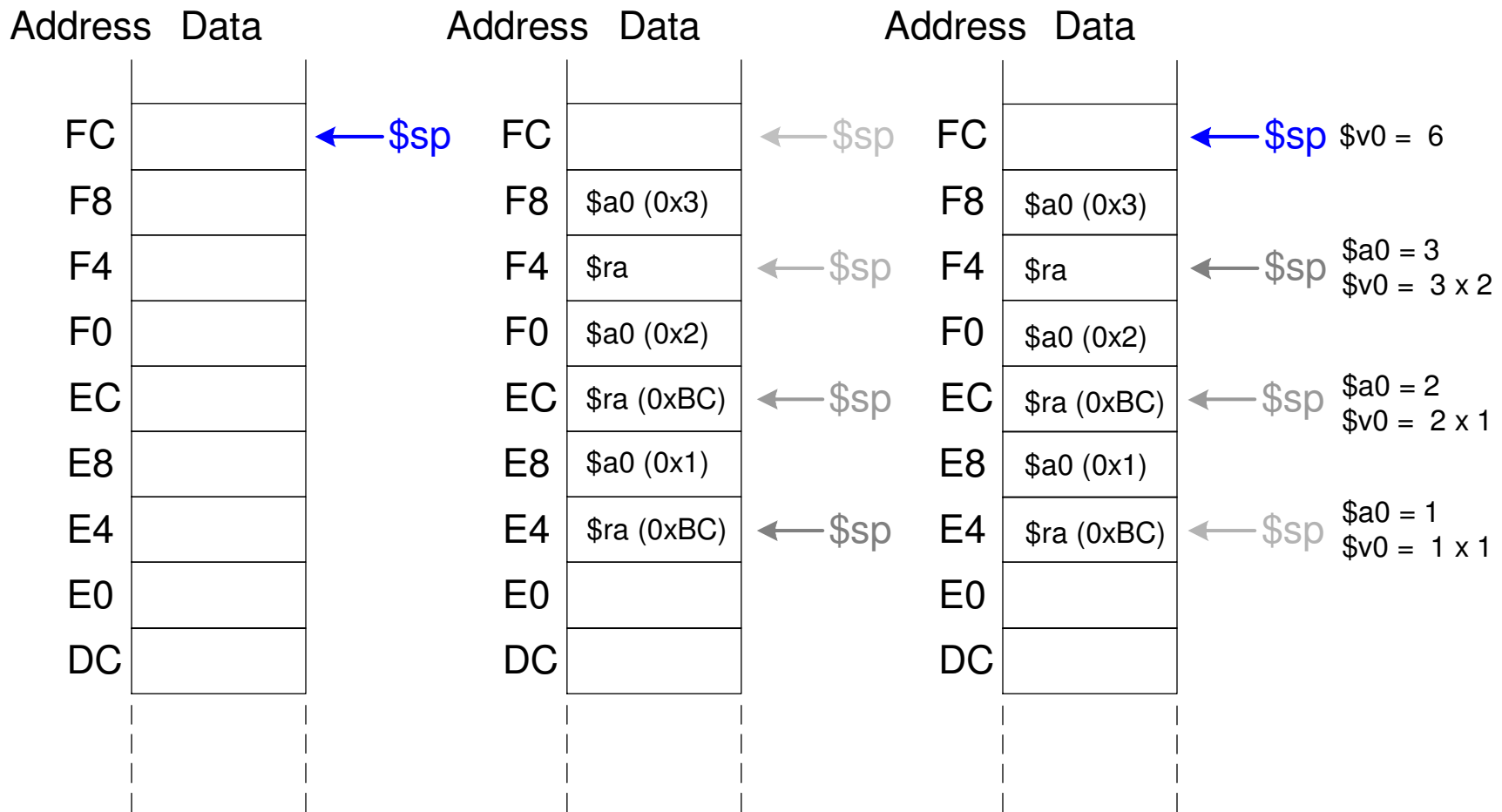
```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n * factorial(n-1));  
}
```

# Chamada Recursiva de Procedimentos

## MIPS assembly code

```
0x90 factorial: addi $sp, $sp, -8    # make room
0x94            sw  $a0, 4($sp)    # store $a0
0x98            sw  $ra, 0($sp)    # store $ra
0x9C            addi $t0, $0, 2
0xA0            slt $t0, $a0, $t0  # a <= 1 ?
0xA4            beq $t0, $0, else   # no: go to else
0xA8            addi $v0, $0, 1     # yes: return 1
0xAC            addi $sp, $sp, 8    # restore $sp
0xB0            jr  $ra            # return
0xB4            else: addi $a0, $a0, -1 # n = n - 1
0xB8            jal factorial      # recursive call
0xBC            lw  $ra, 0($sp)    # restore $ra
0xC0            lw  $a0, 4($sp)    # restore $a0
0xC4            addi $sp, $sp, 8    # restore $sp
0xC8            mul $v0, $a0, $v0  # n * factorial(n-1)
0xCC            jr  $ra            # return
```

# A Pilha Durante a Chamada Recursiva



# Modos de Endereçamento

Como endereçamos os operandos?

- Register
- Immediate
- Base Addressing
- PC-Relative
- Pseudo Direct

# Modos de Endereçamento

## Register

- Os Operandos estão somente em Registradores
  - Exemplo: `add $s0, $t2, $t3`
  - Exemplo: `sub $t8, $s1, $0`

## Immediate Addressing

- Imediato de 16-bit é usado como operando
  - Exemplo: `addi $s4, $t5, -73`
  - Exemplo: `ori $t3, $t7, 0xFF`

# Modos de Endereçamento

## Base Addressing

- O endereço do operando é:

base address + sign-extended immediate

- Exemplo: `lw $s4, 72($0)`

» Address =  $\$0 + 72$

- Exemplo: `sw $t2, -25($t1)`

» Address =  $\$t1 - 25$

# Modos de Endereçamento

## PC-Relative Addressing

```
0x10          beq    $t0, $0, else
0x14          addi   $v0, $0, 1
0x18          addi   $sp, $sp, i
0x1C          jr     $ra
0x20          else:  addi   $a0, $a0, -1
0x24          jal    factorial
```

### Assembly Code

### Field Values

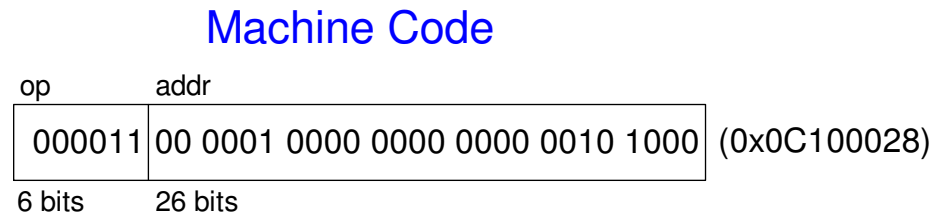
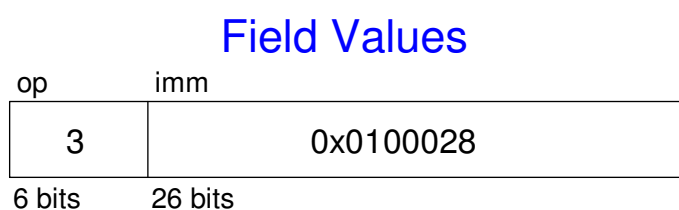
	op	rs	rt	imm
beq \$t0, \$0, else	4	8	0	3
(beq \$t0, \$0, 3)	6 bits	5 bits	5 bits	5 bits 6 bits

# Modos de Endereçamento

## Pseudo-direct Addressing

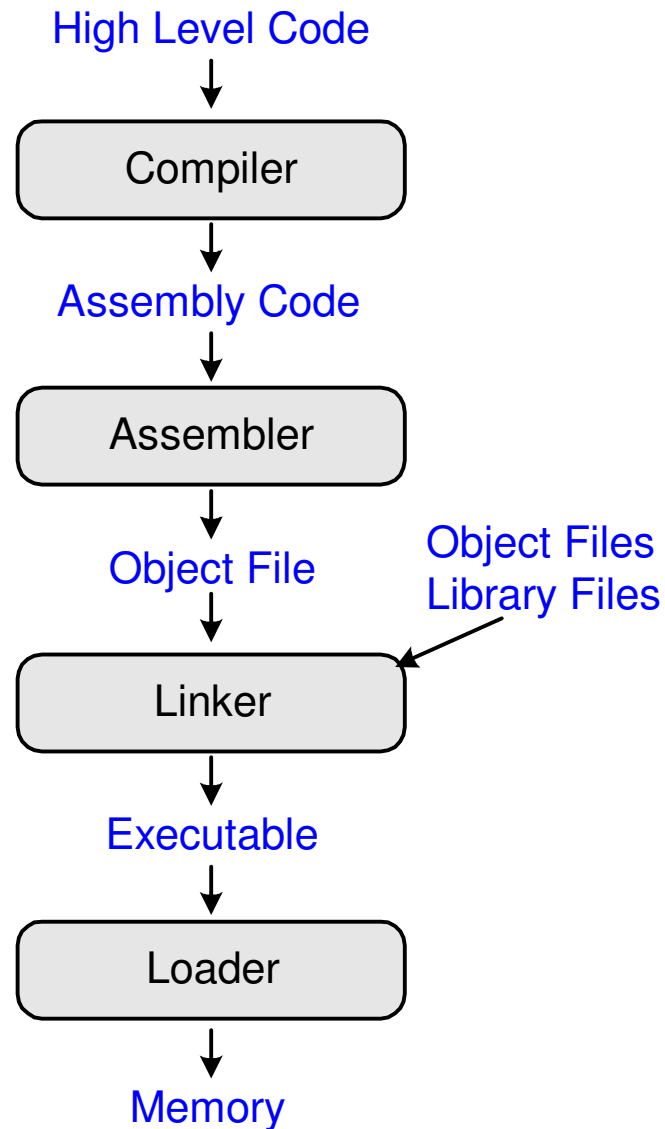
```
0x0040005C      jal    sum
...
0x004000A0  sum:  add    $v0, $a0, $a1
```

```
JTA 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)
26-bit addr 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)
                0  1  0  0  0  0  2  8
```





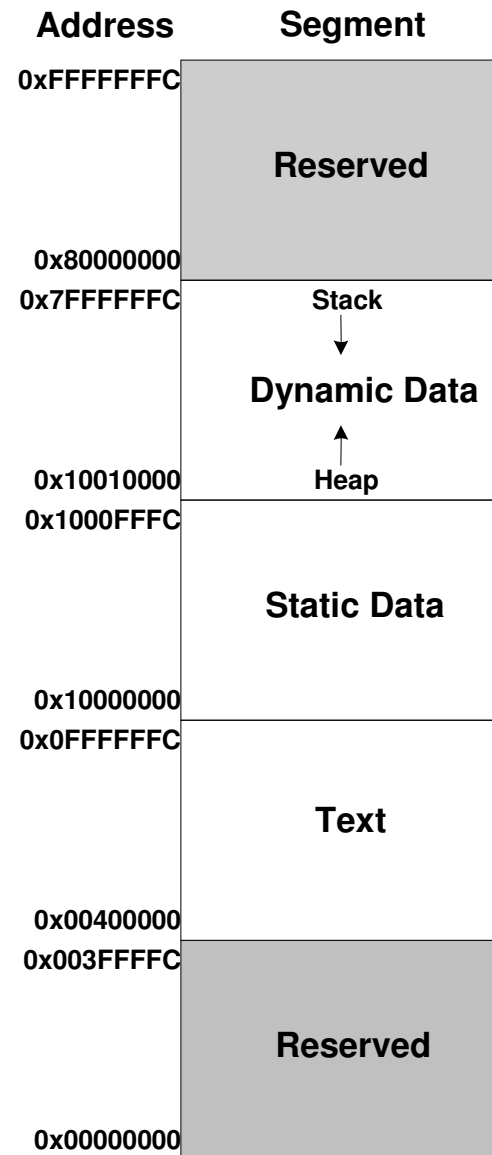
# Como Executar uma Aplicação



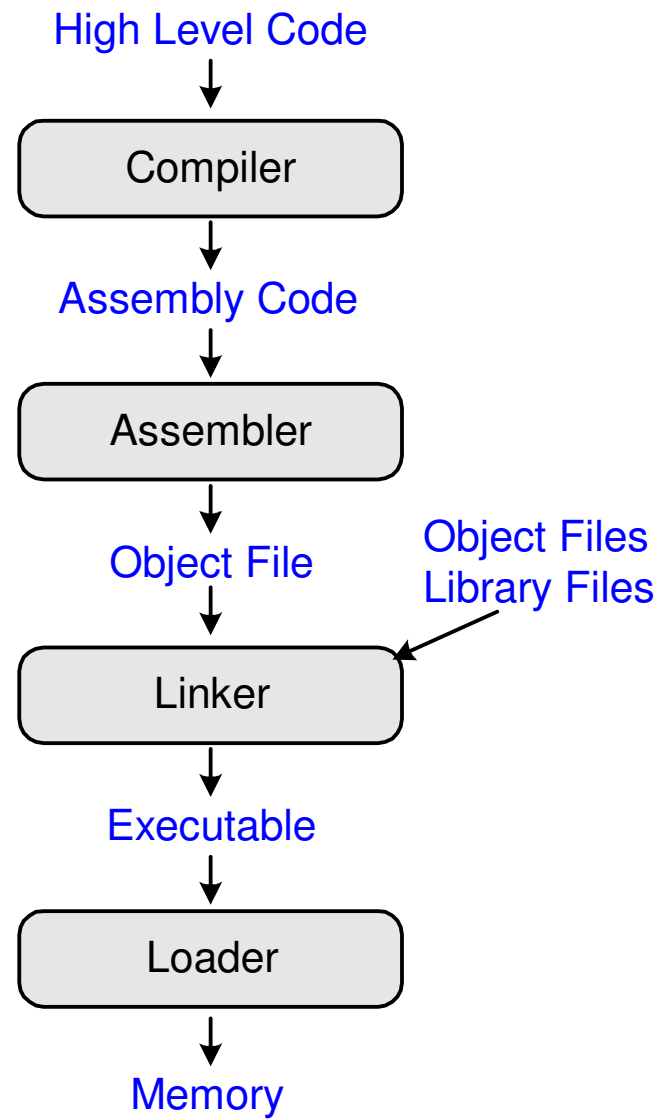
# O que Deve ser Armazenado na Memória

- Instruções (também chamado: *text*)
- Dado
  - Global/stático: alocado antes de começar a execução
  - Dinâmico: alocado pelo programa em execução
- Qual o tamanho da memória?
  - No máximo  $2^{32} = 4$  gigabytes (4 GB)
  - A partir do endereço 0x00000000 ao 0xFFFFFFFF

# Mapa de Memória MIPS



# Executando um Programa



# Exemplo: Programa em C

```
int f, g, y; // global variables
```

```
int main(void)
```

```
{
```

```
    f = 2;
```

```
    g = 3;
```

```
    y = sum(f, g);
```

```
    return y;
```

```
}
```

```
int sum(int a, int b) {
```

```
    return (a + b);
```

```
}
```

# Exemplo: Programa em Assembly

```
int f, g, y; // global

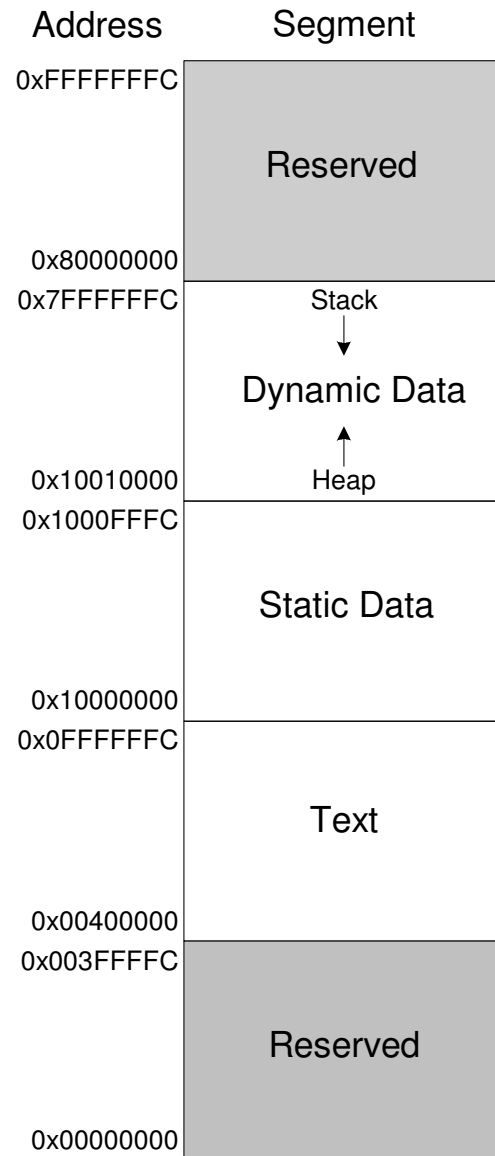
int main(void)
{
    f = 2;
    g = 3;

    y = sum(f, g);
    return y;
}

int sum(int a, int b) {
    return (a + b);
}

.data
f:
g:
y:
.text
main:
    addi $sp, $sp, -4    # stack frame
    sw   $ra, 0($sp)    # store $ra
    addi $a0, $0, 2     # $a0 = 2
    sw   $a0, f         # f = 2
    addi $a1, $0, 3     # $a1 = 3
    sw   $a1, g         # g = 3
    jal  sum            # call sum
    sw   $v0, y         # y = sum()
    lw   $ra, 0($sp)    # restore $ra
    addi $sp, $sp, 4    # restore $sp
    jr   $ra            # return to OS
sum:
    add  $v0, $a0, $a1  # $v0 = a + b
    jr  $ra             # return
```

# Mapa de Memória MIPS



# Exemplo: Tabela de Símbolos

Symbol	Address
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C



# Exemplo: Programa Executável

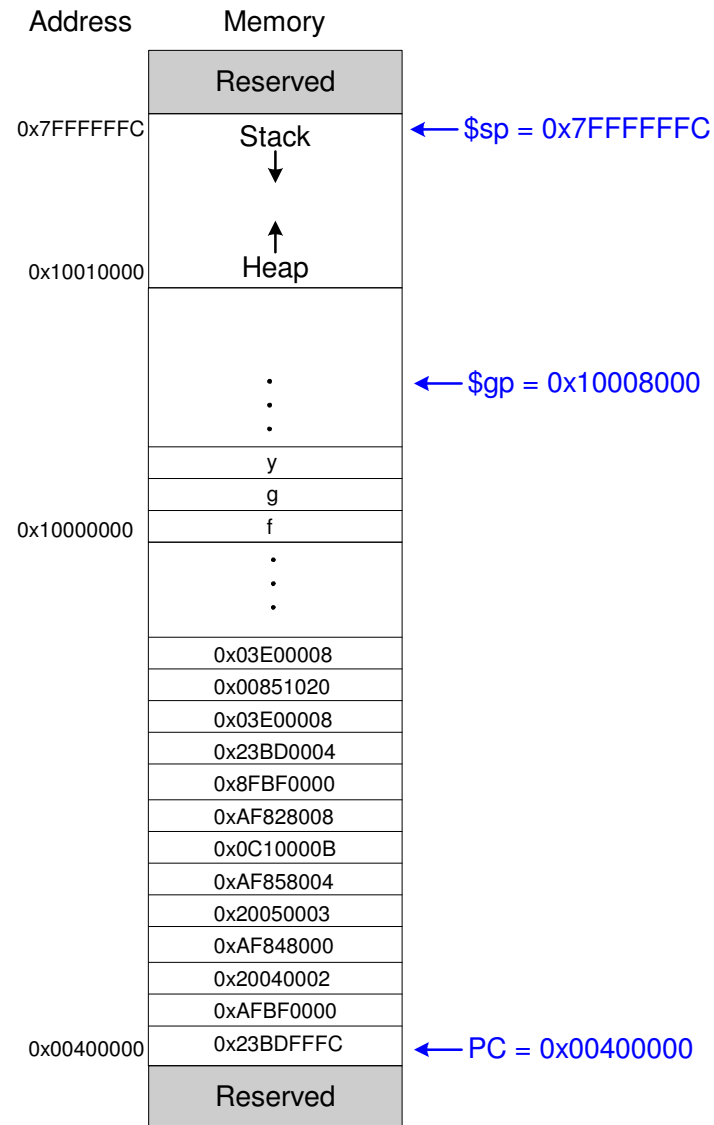
Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)
Text segment	Address	Instruction
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E00008
Data segment	Address	Data
	0x10000000	f
	0x10000004	g
	0x10000008	y

```

addi $sp, $sp, -4
sw  $ra, 0 ($sp)
addi $a0, $0, 2
sw  $a0, 0x8000 ($gp)
addi $a1, $0, 3
sw  $a1, 0x8004 ($gp)
jal  0x0040002C
sw  $v0, 0x8008 ($gp)
lw  $ra, 0 ($sp)
addi $sp, $sp, -4
jr  $ra
add $v0, $a0, $a1
jr  $ra

```

# Exemplo: Programa na Memória



# Pseudo Instruções

Pseudoinstruction	MIPS Instructions
<code>li \$s0, 0x1234AA77</code>	<code>lui \$s0, 0x1234</code> <code>ori \$s0, 0xAA77</code>
<code>mul \$s0, \$s1, \$s2</code>	<code>mult \$s1, \$s2</code> <code>mflo \$s0</code>
<code>clear \$t0</code>	<code>add \$t0, \$0, \$0</code>
<code>move \$s1, \$s2</code>	<code>add \$s2, \$s1, \$0</code>
<code>nop</code>	<code>sll \$0, \$0, 0</code>

# Exceções (Interrupções)

- Chamada de procedimento, não prevista no código, para um *exception handler*
- Causado por:
  - Hardware, também chamado *interrupção*, exemp: keyboard
  - Software, também chamado de *traps*, exemp.: instrução indefinida
- Quando uma exceção ocorre, o processador:
  - Registra a causa da exceção
  - Desvia a execução para *exception handler* no endereço de instrução 0x80000180
  - Retorna ao programa

# Registradores de Exceção

- Não faz parte do register file.
  - Cause
    - » Registra a causa da exceção
  - EPC (Exception PC)
    - » Registra o PC onde ocorreu a exceção
- **EPC e Cause: parte do Coprocessador 0**
- Move from Coprocessor 0
  - `mfc0 $t0, EPC`
  - Move o conteúdo de EPC para \$t0

# Exceções

Exception	Cause
Hardware Interrupt	0x00000000
System Call	0x00000020
Breakpoint / Divide by 0	0x00000024
Undefined Instruction	0x00000028
Arithmetic Overflow	0x00000030

# Exceções

- O Processador salva a causa e o PC em Cause e EPC
- Processador desvia para o **exception handler** (0x80000180)
- Exception handler:
  - Salva os registradores na pilha
  - Lê o registrador Cause

```
mfc0 Cause, $t0
```
  - Trata a exceção
  - Restaura os registradores
  - Retorna ao programa

```
mfc0 EPC, $k0  
jr $k0
```

# Instruções signed e Unsigned

- Soma e Subtração
- Multiplicação e Divisão
- Set less than



# Instruções

- **Soma e subtração**
  - **Signed:** add, addi, sub
    - » Executa a mesma operação que a versão unsigned
    - » Porém o processador gera exceção se overflow
  - **Unsigned:** addu, addiu, subu
    - » O processador não gera exceção se overflow
    - » **Nota:** addiu sign-extends o imediato
- **Multiplicação e Divisão**
  - **Signed:** mult, div
  - **Unsigned:** multu, divu
- **Set Less Than**
  - **Signed:** slt, slti
  - **Unsigned:** sltu, sltiu
  - **Nota:** sltiu sign-extends o imediato antes da comparação

# Instruções

- Loads

- Signed:

- » Sign-extends para criar o valor de 32-bit
    - » Load halfword: lh
    - » Load byte: lb

- Unsigned: addu, addiu, subu

- » Zero-extends para criar o valor de 32-bit
    - » Load halfword unsigned: lhu
    - » Load byte: lbu

# Ponto-Flutuante

- Floating-point coprocessor (Coprocessor 1)
- 32 registradores de 32-bit (\$f0 - \$f31)
- Valores Double-precision são mantidos em dois floating point registers
  - e.g., \$f0 e \$f1, \$f2 e \$f3, etc.
  - Assim, os registradores double-precision floating point são: \$f0, \$f2, \$f4, etc.

# Ponto-Flutuante

Name	Register Number	Usage
\$fv0 - \$fv1	0, 2	return values
\$ft0 - \$ft3	4, 6, 8, 10	temporary variables
\$fa0 - \$fa1	12, 14	procedure arguments
\$ft4 - \$ft8	16, 18	temporary variables
\$fs0 - \$fs5	20, 22, 24, 26, 28, 30	saved variables