FAC

- Tipos de computadores
  - Celular
  - IoT
  - Notebook
  - Desktop
  - Servidores
  - Geladeiras e outros eletrodomésticos
    - Sistemas embarcados
  - Computador de bordo
- Um jogo é o que mais exige esforço computacional
- Como é formado um computador

  Softwares de aplicação → Software de Sistemas → Hardware

Software de sistemas

Software que fornece serviços normalmente úteis, incluindo sistemas operacionais, compiladores, carregadores, e montadores.

Existem muitos tipos de software de sistemas, mas dois tipos são fundamentais em todos os sistemas computacionais modernos: um sistema operacional e um compilador. Um sistema operacional fornece a interface entre o programa do usuário e o hardware e disponibiliza diversos serviços e funções de supervisão. Entre as funções mais importantes estão:
▪ Manipular as operações básicas de entrada e saída
▪ Alocar armazenamento e memória
▪ Providenciar o compartilhamento protegido do c

Software de aplicação

- Manda uma chamada ao software de sistema (sistema operacional)

    - O sistema recebe e manda o hardware fazer algo
    - Driver é um software de aplicação que oferece as chamadas básicas para o software de sistema
- Níveis de código
    1. Aplicação

       ↧ Código fonte
    2. Linguagem de alto nível (muito provavelmente)
        - Mais produtiva
        - C / Python / Java / JS
        - ↧ Compilação ⇒ Processo de tradução completo que gera um **executável**
        - ↧ OU Interpretação ⇒ Lê uma instrução, traduz e executa
    3. Linguagem de montagem (Assembly)
        - Ainda não é a linguagem que o computador entende
        - Aproxima a linguagem de alto nível a linguagem de máquina
        - Linguagem binária em que vemos quando abrimos um exe no bloco de notas
        - Linker junta as bibliotecas ao seu código e gera a linguagem de montagem
        - ↧ Montagem / Linking - Feito pelo montador (Assembler)
    4. Linguagem de máquina (binário)
    5. Lógica digital

Obs:

1. O **Assembler** (montador) faz a tradução do Assembly (linguagem de montagem) para linguagem de máquina (binário). Essa tradução é 1 para 1 (1/1) [CAI NA PROVA]
    a. A volta sempre é verdadeira
    b. É possível voltar o binário para Assembly

2. O compilador / interpretador faz a tradução da linguagem de alto nível para linguagem de montagem (Assembly). Essa tradução é 1 para n.
    a. Traduzir Assembly para alto nível pode não voltar ao código original

A Linguagem de montagem

está relacionada com uma arquitetura (x86, x64, ARM, Risc-V, Mips, entre outras) e é chamada de Instruction Set Achitecture (ISA) ou Arquitetura de Conjunto de Instruções.

Sistema operacional

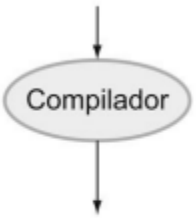
Programa de supervisão que gerencia os recursos de um computador, em favor dos programas executados nessa máquina.

Compilador

Um programa que traduz as instruções de linguagem de alto nível para instruções de linguagem assembly.

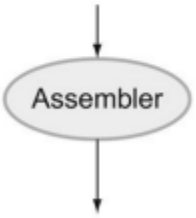
Programa em linguagem de alto nível (em C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Programa em linguagem de máquina binária (para o MIPS)

```
swap:
      multi $2, $5,4
      add   $2, $4,$2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```



Programa em linguagem de máquina binária (para o MIPS)

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

1.4. Sob as tampas

Agora que olhamos por trás do programa para descobrir como ele funciona, vamos abrir a tampa do computador para aprender sobre o hardware dentro dele. O hardware de qualquer computador realiza as mesmas funções básicas: entrada, saída, processamento e armazenamento de dados. A forma como essas funções são realizadas é o principal tema deste livro, e os capítulos subsequentes lidam com as diferentes partes destas quatro tarefas.

Quando tratamos de um aspecto importante neste livro, tão importante que esperamos que você se lembre dele para sempre, nós o enfatizamos identificando-o como um item "Colocando em perspectiva". Há aproximadamente uma dúzia desses itens no livro; o primeiro descreve os cinco componentes de um computador que realizam as tarefas de entrada, saída, processamento e armazenamento de dados.

Dois dos principais componentes dos computadores são: os **dispositivos de entrada**, como o teclado e o mouse, e os **dispositivos de saída**, como a caixa de som. Como o nome sugere, a entrada alimenta o computador, e a saída é o resultado da computação, enviado para o usuário. Alguns dispositivos, como redes sem fio, fornecem tanto entrada quanto saída para o computador.

dispositivo de entrada

Um mecanismo por meio do qual o computador é alimentado com informações, como o teclado e o mouse.

dispositivo de saída

Um mecanismo que transmite o resultado de uma computação para o usuário ou para outro computador.

Os capítulos 5 e 6 descrevem dispositivos de entrada e saída (E/S) em mais detalhes, mas vamos dar um passeio preliminar pelo hardware do computador, começando com os dispositivos de E/S externos.

Colocando em perspectiva

Os cinco componentes de um computador são: entrada, saída, memória,

---

caminho de dados e controle; os dois últimos, às vezes, são combinados e chamados de processador. A Figura 1.5 mostra a organização padrão de um

unidade central de processamento (CPU)
Também chamada de processador. A parte ativa do computador, que contém o caminho de dados, e o controle, que soma, testa números e sinaliza aos dispositivos de E/S para que sejam ativados etc.

A memória é onde os
programas são mantidos quando estão sendo executados; ela também contém os dados necessários aos programas em execução. A memória é constituída de chips DRAM. DRAM significa RAM dinâmica (Dynamic Random Access Memory). Várias DRAMs são usadas em conjunto para conter as instruções e os dados de um programa. Ao contrário das memórias de acesso sequencial, como as fitas magnéticas, a parte RAM do termo DRAM significa que os acessos à memória levam o mesmo tempo, independentemente da parte da memória lida.

Descer até as profundezas de qualquer componente de hardware revela os interiores da máquina. Dentro do processador, existe outro tipo de memória – a memória cache. A memória cache consiste em uma memória pequena e rápida que age como um buffer para a memória DRAM

memória volátil
Armazenamento, como a DRAM, que conserva os dados apenas enquanto estiver recebendo energia.

memória não volátil
Uma forma de memória que conserva os dados mesmo na ausência de energia e que é usada para armazenar programas entre execuções. Um disco de DVD é não volátil.

memória principal

Também chamada memória primária. A memória usada para armazenar os programas enquanto estão sendo executados; normalmente consiste na DRAM nos computadores atuais.

memória secundária

Memória não volátil usada para armazenar programas e dados entre execuções; normalmente consiste em memória flash nos PMDs e discos magnéticos nos servidores.

disco magnético

(também chamado de disco rígido) Uma forma de memória secundária não volátil composta por discos giratórios cobertos com um material de gravação magnético. Por serem dispositivos mecânicos rotativos, os tempos de acesso são cerca de 5 a 20 milissegundos e o custo por gigabyte em 2012 era de US$ 0,05 a US$ 0,10.

memória flash

Uma memória semicondutora não volátil. Ela é mais barata e mais lenta que a DRAM, porém mais cara por bit e mais rápida que os discos magnéticos. Os tempos de acesso são cerca de 5 a 50 microssegundos, e o custo por gigabyte em 2012 era de US$ 0,75 a US$ 1,00

PSEUDOINSTRUÇÕES PARA INSTRUÇÕES

```
move $a0, $s0
add $a0, $a0, $zero

li $v0, 9
addi $v0, $zero, 9

la $r1, nome_da_variável
lui $r1, nome_da_variável >> 16 # Carregue a parte alta do endereço da variável em $r1
addiu $r1, $r1, nome_da_variável & 0xffff # Adicione a parte baixa do endereço da variável a $r1

blt $r1, $r2, etiqueta
slt $t0, $r1, $r2 # Verifique se $r1 < $r2
```

```
bne $t0, $zero, etiqueta # Se o resultado for 1, salte para a etiqueta

bgt $r1, $r2, etiqueta
slt $t0, $r2, $r1 # Verifique se $r2 < $r1
beq $t0, $zero, etiqueta # Se o resultado for 0, salte para a etiqueta

ble $r1, $r2, etiqueta
slt $t0, $r2, $r1 # Verifique se $r2 < $r1
beq $t0, $zero, etiqueta # Se o resultado for 0, salte para a etiqueta
slt $t0, $r1, $r2 # Verifique se $r1 < $r2
beq $t0, $zero, etiqueta # Se o resultado for 0, salte para a etiqueta

bge $r1, $r2, etiqueta
slt $t0, $r2, $r1 # Verifique se $r2 < $r1
beq $t0, $zero, etiqueta # Se o resultado for 0, salte para a etiqueta
slt $t0, $r1, $r2 # Verifique se $r1 < $r2
beq $t0, $one, etiqueta # Se o resultado for 1, salte para a etiqueta
```

Os dois formatos de instrução MIPS até aqui são R e I. Os 16 primeiros bits são iguais: ambos contêm um campo op, indicando a operação básica; um campo rs, indicando um dos operandos origem; e um campo rt, que especifica o outro operando origem, exceto para load word, em que especifica o registrador destino. O formato R divide os 16 últimos bits em um campo rd, especificando o registrador destino; um campo shamt, explicado na Seção 2.6; e o campo funct, que particulariza a operação específica das instruções no formato R. O formato I mantém os 16 bits finais como um único campo de endereço

Valor ASCII	Sinal	Valor ASCII	Sinal	Valor ASCII	Sinal	Valor ASCII	Sinal	Valor ASCII	Sinal	Valor ASCII	Sinal
32	Espaç	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Escrevendo um programa em Assembly MIPS

Um programa no assembly MIPS possui a seguinte estrutura:

```
.data
  # alocação de memória
.text
main: # rótulo (onde o programa começa a ser executado)
  # código em assembly
  # encerramento do código, não necessariamente por último
```

- Memória principal (temporária): cache e ram
- Memória secundária (persistente): Disco (HD, pendrive)

- .data ⇒ Alocação na memória **principal**

Obs:

- **Memória principal** é a memória volátil onde dados ficam disponíveis para o processador em complemento aos registradores. Geralmente composta pela cache e pela RAM.

    - **Memória secundária** é a unidade persistente, onde os dados são armazenados. Quando a energia para de passar por lá, os dados somem.
- Sistema operacional é quem conhece o hardware ou generaliza a chamada para o software de sistema

Declaração de dados na seção .data

Só é possível declarar variáveis nessa seção

- O dado vai para a memória principal
    - A operação só acontece com os dados que estão no registrador.
- Segue o seguinte formato
    - Rótulo: .tipo valor1, valor2, … , valor N

Os possíveis tipos são:

- **word w1, w2, … , wn**: representam palavras de 32 bits por w1, w2, … , wn que são inteiros.
- **half h1, h2, … , hn**: representam dados de 16 bits (2 bytes).
- **byte b1, b2, … , bn**: representam dados de 8 bits (1 byte).
- **asciiz str**: representa uma cadeia de caracter dada em *str* (entre aspas). A cadeia é automaticamente terminada pelo caracter nulo.

Princípio do design 1

Simplicidade favorece a regularidade

- Implementação simples melhora o desempenho
- Outras instruções aritméticas
    - sub: subtração
    - mul: multiplicação, essa instrução aritmética precisa de cuidado com o tanto de dígitos que o “a” consegue armazenar na base binária, o limite pode ser excedido facilmente

Princípio do design 2

Menor significa mais rápido.
Uma quantidade muito grande de registradores pode aumentar o tempo do ciclo do clock simplesmente porque os sinais eletrônicos levam mais tempo quando precisam atravessar uma distância maior.

Princípio do design 3

Torne o caso comum mais rápido

- O uso de constantes pequenas é muito comum
- Essas instruções evitam uma instrução *lw* num registrador
- **Obs:** evitar usar a contante 0 em instruções imediatas

Representação em linguagem de máquina

Todas as instruções são traduzidas para binário pelo montador (*assembler*). Os códigos binários gerados são chamados de linguagem de máquina. A conversão é pautada em 3 formatos de representação: tipo **R**, **I** e **J**.

Formato tipo R

A instruções do tipo R são representadas num binário de 32 bits segregado da seguinte forma:

Onde:

- op: código de operação (opcode);
- rs: número do 1° registrador de origem;
- rt: número do 2° registrador de origem;
- rd: número do registrador de destino;
- shamt: tamanho do deslocamento;
  - “shift ammount”
- function: código da função

- Complementa o opcode

São instruções do tipo R

- Aritméticas
- Lógicas
- De deslocamento

Ex:

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

00000010001100100100000000100000

Formato tipo I

São representadas num binário de 32 bits da seguinte forma:

- op: código de operação;
- rs: operador de origem;
- rt: registrador de destino (ou origem para sw);

São instruções do tipo I

- Imediatas
- De acesso à memória

  Obs: A capacidade máxima de uma constante é -2^{15} a $2^{15}-1$

```
sw $t0, 0($s0)
;   rt, const(rs)
```

Operações lógicas

São instruções para manipulação de bits.

- shift left: sll reg1, reg2, shamt
  - shift left logical

- shift right: srl reg1, reg2, shamt
  - shift right logical

- e lógico (bit a bit): and reg1, reg2, reg3
  - reg1 = reg2 E reg3
  - andi = reg1, reg2, const

- ou lógico (bit a bit): or reg1, reg2, reg3
  - ori reg1, reg2, const
  - reg1 = reg2 ou reg3

- não (ou) lógico (bit a bit): nor reg1, reg2, reg3
  - nor $t1, $s0, $zero
  - reg1 = NÃO(reg2 ou reg3)

Instrução	Formato	op	rs	rt	rd	shamt	funct	endereço
`add`	R	0	reg	reg	reg	0	32_{dec}	n.a.
`sub` (subtract)	R	0	reg	reg	reg	0	34_{dec}	n.a.
`add immediate`	I	8_{dec}	reg	reg	n.a.	n.a.	n.a.	constante
`lw` (load word)	I	35_{dec}	reg	reg	n.a.	n.a.	n.a.	endereço
`sw` (store word)	I	43_{dec}	reg	reg	n.a.	n.a.	n.a.	endereço

FIGURA 2.5 Codificação de instruções MIPS.
Na tabela, "reg" significa um número de registrador entre 0 e 31, "endereço" significa um endereço de 16 bits, e "n.a." (não se aplica) significa que esse campo não aparece nesse formato. Observe que as instruções `add` e `sub` têm o mesmo valor no campo op; o hardware usa o campo funct para decidir sobre a variante da operação: somar (32) ou subtrair (34).

Nome	Número do registrador	Uso	Preservado na chamada?
$zero	0	O valor constante 0	n.a.
$v0–$v1	2–3	Valores para resultados e avaliação de expressões	não
$a0–$a3	4–7	Argumentos	não
$t0–$t7	8–15	Temporários	não
$s0–$s7	16–23	Valores salvos	sim
$t8–$t9	24–25	Mais temporários	não
$gp	28	Ponteiro global	sim
$sp	29	Stack pointer	sim
$fp	30	Frame pointer	sim
$ra	31	Endereço de retorno	sim

Linguagem assembly do MIPS

Categoria	Instrução	Exemplo	Significado	Comentários
Aritmética	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Três operandos; dados nos registradores
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Três operandos; dados nos registradores
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Usada para somar constantes
Transferência de dados	load word	lw $s1,20($s2)	$s1 = Memória[$s2 + 20]	Dados da memória para o registrador
	store word	sw $s1,20($s2)	Memória[$s2 + 20] = $s1	Dados do registrador para a memória
	load half	lh $s1,20($s2)	$s1 = Memória[$s2 + 20]	Halfword da memória para registrador
	load half unsigned	lhu $s1,20($s2)	$s1 = Memória[$s2 + 20]	Halfword da memória para registrador
	store half	sh $s1,20($s2)	Memória[$s2 + 20] = $s1	Halfword de um registrador para memória
	load byte	lb $s1,20($s2)	$s1 = Memória[$s2 + 20]	Byte da memória para registrador
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memória[$s2 + 20]	Byte da memória para registrador
	store byte	sb $s1,20($s2)	Memória[$s2 + 20] = $s1	Byte de um registrador para memória
	load linked word	ll $s1,20($s2)	$s1 = Memória[$s2 + 20]	Carrega word como 1ª metade do swap atômico
	store condition, word	sc $s1,20($s2)	Memória[$s2+20]=$s1;$s1=0 or 1	Armazena word como 2ª metade do swap atômico
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Carrega constante nos 16 bits mais altos
Lógica	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Três operadores em registrador; AND bit a bit
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Três operadores em registrador; OR bit a bit
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Três operadores em registrador; NOR bit a bit
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	AND bit a bit registrador com constante
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	OR bit a bit registrador com constante
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Deslocamento à esquerda por constante
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Deslocamento à direita por constante
Desvio condicional	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Testa igualdade; desvio relativo ao PC
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Testa desigualdade; relativo ao PC
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compara menor que; usado com beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compara menor que sem sinal
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compara menor que constante
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compara menor que constante sem sinal
Desvio incondicional	jump	j 2500	go to 10000	Desvia para endereço de destino
	jump register	jr $ra	go to $ra	Para switch e retorno de procedimento
	jump and link	jal 2500	$ra = PC + 4; go to 10000	Para chamada de procedimento

Deslocamento lógico

Ex:

1011 ⇒ srl 1 ⇒ 0101 1 (LSB)

Insere 0 | Joga para fora

- Não conserva o sinal, joga tudo para

Deslocamento aritmético

Ex1:

1011 ⇒ sra 1 (qtd. de casas) ⇒ 1101 1 (LSB)

(conserva sinal) Insere 1 | Joga para fora

Ex2:

0101 ⇒ sra 1 ⇒ 0010 1 (LSB)

- C

Instruções de desvio

Desvio condicional

- Desvia o fluxo se uma condição for satisfeita

```
beq rs, rt, label (branch if equal)
;se o conteúdo dos registradores for igual ele realiza a instrução,
;se não, continua o código normalmente
```

- Se rs == rt, desvia para a instrução rotulada por label

```
bne rs, rt, label (branch if not equal)
```

- Se rs ≠ rt, desvia para a instrução rotulada por label

Desvio incondicional

- j label (jump)
  - Desvia para a instrução com rótulo label
  - Instrução de desvio incondicional

Formato tipo J

op	endereço
6 bits	26 bits

1 - Exemplo

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

variável	registrador
f	$t0
g	$t1
h	$t3
i	$t4
j	$t5

1.1

```
beq $t4, $t5, soma ; se forem iguais, pule para soma
sub $t0, $t1, $t3 ; f = g - h, se forem iguais pularia essa linha
j sair ; (jump) pula para a posição do sair

soma:
  add $t0, $t1, $t3 ; f = g + h

sair:
  ...
```

1.2

```
bne $t4, $t5, sub
add $t0, $t1, $t3
j sair

sub:
  $t0, $t1, $t3

sair:
  ...
```

1.3

```
beq $t4, $t5, soma
bne $t4, $t5, subtrai

soma:
  add $t0, $t1, $t3
  j sair

subtrai:
  sub $t0, $t1, $t3

sair:
  ...
```

2 - Exemplo

```
i = 0; // move $t0, $zero
while (i != n)
{
  k = k + 1;
  i++;
}
```

variável	registrador
i	$t0
j	$t1
k	$t2
n	$s0

2.1

```
soma:
  addi $t2, $t2, 1 ; # k = k + 1
  addi $t0, $t0, 1 ; # i = i + 1
  bne $t0, $s0, soma ; # se i != n irá voltar para o rótulo

;# do { ... } while ();
```

2.2

```
laco:
  beq $t0, $s0, sair
  addi $t2, $t2, 1 ;# k = k + 1;
  addi $t0, $t0, 1 ;# i++
  j laco

sair:
  ...

;# define for e while pois verifica a condição antes de executar
```

Obs: Pseudoinstruções

bge	≥
bgt	>
ble	≤
blt	<

O hardware para executá-las seria mais lento.

- Verificar igualdade e desigualdade requer apenas uma instrução, enquanto >, ≥, <, ≤ requer mais

Instruções de comparação

slt/slti rd, rs, rt/const

- se rs < rt, rd = 1, caso contrário, rd = 0

Ex

-

(set on less (immediate))

```
if (i < j ) k++;
```

variável	registrador
i	$t0
j	$t1
k	$t2

```
slt $t9, $t0, $t1 ;# se i < j, $t9 = 1
beq $t9, $zero, sair
addi $t2, $t2, 1 ;# k++

sair:
   ...
```

Funções em Assembly e implementação de condicionais

Como implementar >, ≥, < e ≤?

Contamos com as instruções:

- slt (rs < rt?) ⇒ 1
  - “set on less than”
  - “se rs < rt retorna 1, se não retorna 0
- beq (devia se rs == rt)
  - beq rs, rt, label
- bne (devia se rs ≠ rt)

- bne rs, rt, label

Desvio se “>”

Digamos $s0 > $s1

```
slt $t0, $s1, $s0     ; $t0 = 1 se $s1 < $s0
bne $t0, $zero, maior ; $t0 = 0 $s1 >= $s0
```

Desvio se “≥”

Digamos $s0 ≥ $s1

- “Set on less than”

```
slt $t0, $s0, $s1          ; $t0 = 0 <=> $s0 </ $s1
beq $t0, $zero, maiorIgual ; $t0 = 0  => $s0 >= $s1
```

Desvio se “≤”

$s0 ≤ $s1

```
slt $t0, $s1, $s0
beq $t0, $zero, menorIgual
```