Campus de Gualtar
4710-057 Braga

UNIVERSIDADE DO MINHO
ESCOLA DE ENGENHARIA

Departamento de
Informática

Introdução aos Sistemas de Computação

Notas de estudo

Alberto José Proença

30-Mar-03

Nota introdutória

Este documento é um texto de apoio ao funcionamento de disciplinas na área da Arquitectura de Computadores, complementando apenas a bibliografia básica recomendada. Não pretendendo substitui-la, adapta e resume alguns aspectos considerados essenciais durante a leccionação da matéria; baseia-se em documentos de anos anteriores e integra e complementa excertos de documentos de livros recomendados:

- "*Computer Organization and Architecture - Designing for Performance*", 6th Ed., W. Stallings, Prentice Hall, 2002 (mais informação em http://williamstallings.com/COA6e.html); adiante referido por COA
- "*Computer Systems: A Programmer's Perspective*", Randal Bryant and David O'Hallaron, Prentice Hall, 2003 (mais informação em http://csapp.cs.cmu.edu/); adiante referido por CSAPP
- *Structured Computer Organization*, 4th Ed., Andrew S. Tanenbaum, Prentice Hall, 1999; adiante referido por SCO

Índice

O estudo da organização e arquitectura dum computador começa por uma reflexão sobre **o que é um computador**. No contexto desta disciplina iremos considerar um computador (ou **sistema de computação**) como uma sistema (máquina) que tem como finalidade **processar informação** e que suporta alterações à sua funcionalidade através da sua **programação**.

A **estrutura interna dum computador**, em termos funcionais, é constituída por pelo menos uma unidade central de processamento, uma memória, módulos para Entrada/Saída de informação, e pela interligação destes componentes. A ilustração desta estrutura será feita ao longo destas notas com recurso a figuras de livros da especialidade, sendo complementada com bibliografia adicional disponível na Web[1].

Mas a primeira questão que se coloca logo no início é a seguinte: o que é "informação" e como é que essa informação se encontra representada dentro de um computador? A **representação da informação** num computador permite uma melhor compreensão do modo de **funcionamento de um computador**.

1. Representação da informação num computador

O "tijolo" para a representação de qualquer tipo de informação num computador é um valor que pode ser numericamente representado por um dígito binário, o **bit** (do inglês ***bi**nary digi**t***).

Existem diversos tipos de informação que são normalmente representados num computador, podendo-se destacar:

- **textos**, que representam a forma básica dos seres humanos contactarem entre si e na coordenada tempo;
- **números**, que embora se possam considerar como parte dos textos, têm representação específica mais compacta, permitindo melhorar nos computadores a eficiência dos cálculos numéricos ("*computare*" em latim...);
- **conteúdos multimédia**, que representam, de forma compacta e com reduzida quebra na qualidade, imagens (incl. fotografias), audio e material audiovisual;
- **código para execução no processador**, que representa de forma compacta o conjunto de comandos que um processador deve executar para processar a informação conforme especificado num programa.

Na representação de textos, o código actualmente mais usado na representação do alfabeto latino, é o que foi proposto pelos americanos numa tentativa de normalizarem as trocas de informação entre equipamentos: o código **ASCII** (*American Standard Code for Information Interchange*). Esta codificação permite representar, com apenas 7 bits, as 26+26 letras do alfabeto, os algarismos, os sinais de pontuação e operadores aritméticos, para além de diversos sinais de controlo:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

[1] Recomenda-se a leitura da revista dedicada ao tema de referência "*How Computers Work*" da editora de *Smart Computing*, http://www.smartcomputing.com/editorial/stoc.asp?guid=a9opjpko&vol=5&iss=3&type=5

A tabela abaixo representa uma outra visão do mesmo código com 7 bits:

ASCII value	Character	Control character	ASCII value	Character	ASCII value	Character	ASCII value	Character
000	(null)	NUL	032	(space)	064	@	096	
001	☺	SOH	033	!	065	A	097	a
002	☻	STX	034	"	066	B	098	b
003	♥	ETX	035	#	067	C	099	c
004	♦	EOT	036	$	068	D	100	d
005	♣	ENQ	037	%	069	E	101	e
006	♠	ACK	038	&	070	F	102	f
007	(beep)	BEL	039	'	071	G	103	g
008	◘	BS	040	(	072	H	104	h
009	(tab)	HT	041	)	073	I	105	i
010	(line feed)	LF	042	*	074	J	106	j
011	(home)	VT	043	+	075	K	107	k
012	(form feed)	FF	044	,	076	L	108	l
013	(carriage return)	CR	045	-	077	M	109	m
014	♫	SO	046	.	078	N	110	n
015	☼	SI	047	/	079	O	111	o
016	►	DLE	048	0	080	P	112	p
017	◄	DC1	049	1	081	Q	113	q
018	↕	DC2	050	2	082	R	114	r
019	!!	DC3	051	3	083	S	115	s
020	π	DC4	052	4	084	T	116	t
021	§	NAK	053	5	085	U	117	u
022	▬	SYN	054	6	086	V	118	v
023	↨	ETB	055	7	087	W	119	w
024	↑	CAN	056	8	088	X	120	x
025	↓	EM	057	9	089	Y	121	y
026	→	SUB	058	:	090	Z	122	z
027	←	ESC	059	;	091	[	123	{
028	(cursor right)	FS	060	<	092	\	124	|
029	(cursor left)	GS	061	=	093	]	125	}
030	(cursor up)	RS	062	>	094	^	126	~
031	(cursor down)	US	063	?	095	_	127	⌂

Para mais informação sobre este tipo de codificação e respectivas variantes que contemplam os caracteres especiais de outras línguas (acentos, por ex.), existem extensões do código ASCII para 8-bits[2].

Embora este tipo de codificação com apenas 8 bits seja suficiente para a representação destes caracteres e dos caracteres gregos e cirílicos, ele é manifestamente insuficiente para a maioria das línguas do extremo oriente. Após várias propostas de normalização baseadas em representações de 2 bytes fossem sendo propostas ao longo dos tempos, a norma **Unicode** parece ser a que está actualmente a ter maior aceitação[3].

Na representação de **valores numéricos** para efeitos de cálculo, existem formas mais eficientes de representar valores da classe dos **inteiros** que utilizando o formato de codificação de texto. Estas formas tomam em consideração a representação numérica dos sistemas de numeração de bases diferentes, e na representação de valores com sinal existem codificações que não só são eficientes na ocupação de espaço, como também permitem implementações eficientes e compactas das operações, ao nível do *hardware*. O Anexo A apresenta com mais detalhe a representação de inteiros.

No caso dos valores **reais**, a representação normalmente utilizada é baseada na notação em vírgula flutuante, seguindo a norma **IEEE 754**[4]. O Anexo B está dedicado a uma apresentação mais detalhada deste assunto.

A representação eficiente de **conteúdos multimédia** apenas teve um impacto significativo com a ampla divulgação da Internet. A necessidade de se compactar os ficheiros de imagem para uma dimensão aceitável para uma rápida comunicação numa Internet de baixa velocidade, levou ao desenvolvimento de

[2] Recomenda-se uma visita ao site http://www.jimprice.com/jim-asc.htm
[3] Para uma breve análise histórica dos diversos tipos de codificação de caracteres, recomenda-se também uma visita ao *site* http://tronweb.super-nova.co.jp/characcodehist.html
[4] Para mais informação sugere-se uma visita a http://grouper.ieee.org/groups/754/

algoritmos de compactação de informação altamente eficientes.

Se se considerar que a **imagem** captada por uma máquina fotográfica digital de média resolução (1600x1200 pontos ou pixels, de "*picture elements*"), em que cada pixel requer 8x3 (R+G+B) mais 8 (controlo) bits para se poder representar uma imagem com qualidade, temos uma só imagem a precisar de 8 MB para ser representada! Se se não aplicar algoritmos eficientes de compactação, esta forma "bruta" de representação é pouco eficiente. Os algoritmos que actualmente são mais utilizados são os desenvolvidos pelo "*Joint Photographic Experts Group*", mais conhecidos por **JPEG**[5]. A imagem referida captada pela máquina fotográfica acima referida, já é normalmente armazenada na própria máquina usando um de 2 algoritmos de compactação disponibilizados pelo fabricante: alta compressão em detrimento da qualidade (ocupando cerca de 200 KB), ou menor compressão e maior qualidade (cerca de 1 MB).

Considerando idênticos poderão ser feitos para o caso de **filmes**, onde a necessidade de se representar 25 imagens por cada segundo de visualização, impõe valores extremamente elevados de representação da informação em bits, se não houver algoritmos muito eficientes de compactação. Neste caso os algoritmos com maior divulgação e aceitação a nível mundial são os desenvolvidos pelo "*Moving Picture Experts Group*", ou **MPEG**[6].

A norma MPEG contempla não apenas a imagem, mas também a componente de **audio**, i.e., está orientada à representação de material audio-visual. Como tal incorpora nos seus algoritmos uma camada (a 3) dedicada ao som, e como tal mais conhecida como **MP3**[7].

Embora os programas de computador sejam escritos sob a forma de textos e usando linguagens de programação, esta forma textual de representação dos comandos de um processador não é eficiente. A **representação de programas** codificados de acordo com a especificação do fabricante da arquitectura de um dado processador (ou família de processadores), segue normalmente um formato específico e relativo ao conjunto de instruções (*instruction set*) suportado por esse processador (ou família). Este assunto será abordado mais adiante.

1.1 Information is Bits in Context *(retirado de CSAPP)*

Modern computers store and process information represented as two-valued signals. These lowly binary digits, or bits, form the basis of the digital revolution. The familiar decimal, or base-10, representation has been in use for over 1000 years, having been developed in India, improved by Arab mathematicians in the 12th century, and brought to the West in the 13th century by the Italian mathematician Leonardo Pisano, better known as Fibonacci. Using decimal notation is natural for ten-fingered humans, but binary values work better when building machines that store and process information. Two-valued signals can readily be represented, stored, and transmitted, for example, as the presence or absence of a hole in a punched card, as a high or low voltage on a wire, or as a magnetic domain oriented clockwise or counterclockwise. The electronic circuitry for storing and performing computations on two-valued signals is very simple and reliable, enabling manufacturers to integrate millions of such circuits on a single silicon chip.

In isolation, a single bit is not very useful. When we group bits together and apply some *interpretation* that gives meaning to the different possible bit patterns, however, we can represent the elements of any finite set. For example, using a binary number system, we can use groups of bits to encode nonnegative numbers. By using a standard character code, we can encode the letters and symbols in a document.

Representing letters and symbols

In their classic text on the C programming language, Kernighan and Ritchie introduce readers to C using the `hello` program shown below.

[5] Recomenda-se uma visita ao *site* http://www.jpeg.org/public/jpeghomepage.htm
[6] Recomenda-se uma visita ao *site* http://mpeg.telecomitalialab.com/
[7] Recomenda-se uma visita ao *site* http://www.mpeg.org/MPEG/mp3.html

code/intro/hello.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5 printf("hello, world\n");
6 }
```

code/intro/hello.c

Figure 1: The hello program.

Although `hello` is a very simple program, every major part of the system must work in concert in order for it to run to completion. In a sense, the goal of this book is to help you understand what happens and why, when you run `hello` on your system. We will begin our study of systems by tracing the lifetime of the `hello` program, from the time a programmer creates it, until it runs on a system, prints its simple message, and terminates. As we follow the lifetime of the program, we will briefly introduce the key concepts, terminology, and components that come into play. Later chapters will expand on these ideas.

Our `hello` program begins life as a *source program* (or *source file*) that the programmer creates with an editor and saves in a text file called `hello.c`. The source program is a sequence of bits, each with a value of 0 or 1, organized in 8-bit chunks called *bytes*. Each byte represents some text character in the program. Most modern systems represent text characters using the ASCII standard that represents each character with a unique byte-sized integer value. For example, Figure 2 shows the ASCII representation of the `hello.c` program.

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	<sp>	w	o	r	l	d	\	n	"	)	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

Figure 2: The ASCII text representation of `hello.c`.

The `hello.c` program is stored in a file as a sequence of bytes. Each byte has an integer value that corresponds to some character. For example, the first byte has the integer value 35, which corresponds to the character '#'. The second byte has the integer value 105, which corresponds to the character '`i`', and so on. Notice that the invisible newline character '\n', which is represented by the integer value 10, terminates each text line. Files such as `hello.c` that consist exclusively of ASCII characters are known as *text files*. All other files are known as *binary files*.

The representation of `hello.c` illustrates a fundamental idea: All information in a system — including disk files, programs stored in memory, user data stored in memory, and data transferred across a network — is represented as a bunch of bits. The only thing that distinguishes different data objects is the context in which we view them. For example, in different contexts, the same sequence of bytes might represent an integer, floating-point number, character string, or machine instruction.

Representing numbers

We consider the three most important encodings of numbers. *Unsigned* encodings are based on traditional binary notation, representing numbers greater than or equal to 0. *Two's-complement* encodings are the most common way to represent signed integers, that is, numbers that may be either

positive or negative. *Floating point* encodings are a base-two version of scientific notation for representing real numbers. Computers implement arithmetic operations, such as addition and multiplication, with these different representations, similar to the corresponding operations on integers and real numbers.

Computer representations use a limited number of bits to encode a number, and hence some operations can *overflow* when the results are too large to be represented. This can lead to some surprising results. For example, on most of today's computers, computing the expression

```
200 * 300 * 400 * 500
```

yields -884,901,888. This runs counter to the properties of integer arithmetic—computing the product of a set of positive numbers has yielded a negative result.

On the other hand, integer computer arithmetic satisfies many of the familiar properties of true integer arithmetic. For example, multiplication is associative and commutative, so that computing any of the following C expressions yields -884,901,888:

```
(500 * 400) * (300 * 200)
((500 * 400) * 300) * 200
((200 * 500) * 300) * 400
400 * (200 * (300 * 500))
```

The computer might not generate the expected result, but at least it is consistent!

Floating-point arithmetic has altogether different mathematical properties. The product of a set of positive numbers will always be positive, although overflow will yield the special value . On the other hand, floating-point arithmetic is not associative due to the finite precision of the representation. For example, the C expression `(3.14+1e20)-1e20` will evaluate to `0.0` on most machines, while `3.14+(1e20- 1e20)` will evaluate to `3.14`.

Our treatment of this material is very mathematical. We start with the basic definitions of the encodings and then derive such properties as the range of representable numbers, their bit-level representations, and the properties of the arithmetic operations. We believe it is important for you to examine this material from such an abstract viewpoint, because programmers need to have a solid understanding of how computer arithmetic relates to the more familiar integer and real arithmetic. Although it may appear intimidating, the mathematical treatment requires just an understanding of basic algebra. We recommend you work the practice problems as a way to solidify the connection between the formal treatment and some real-life examples.

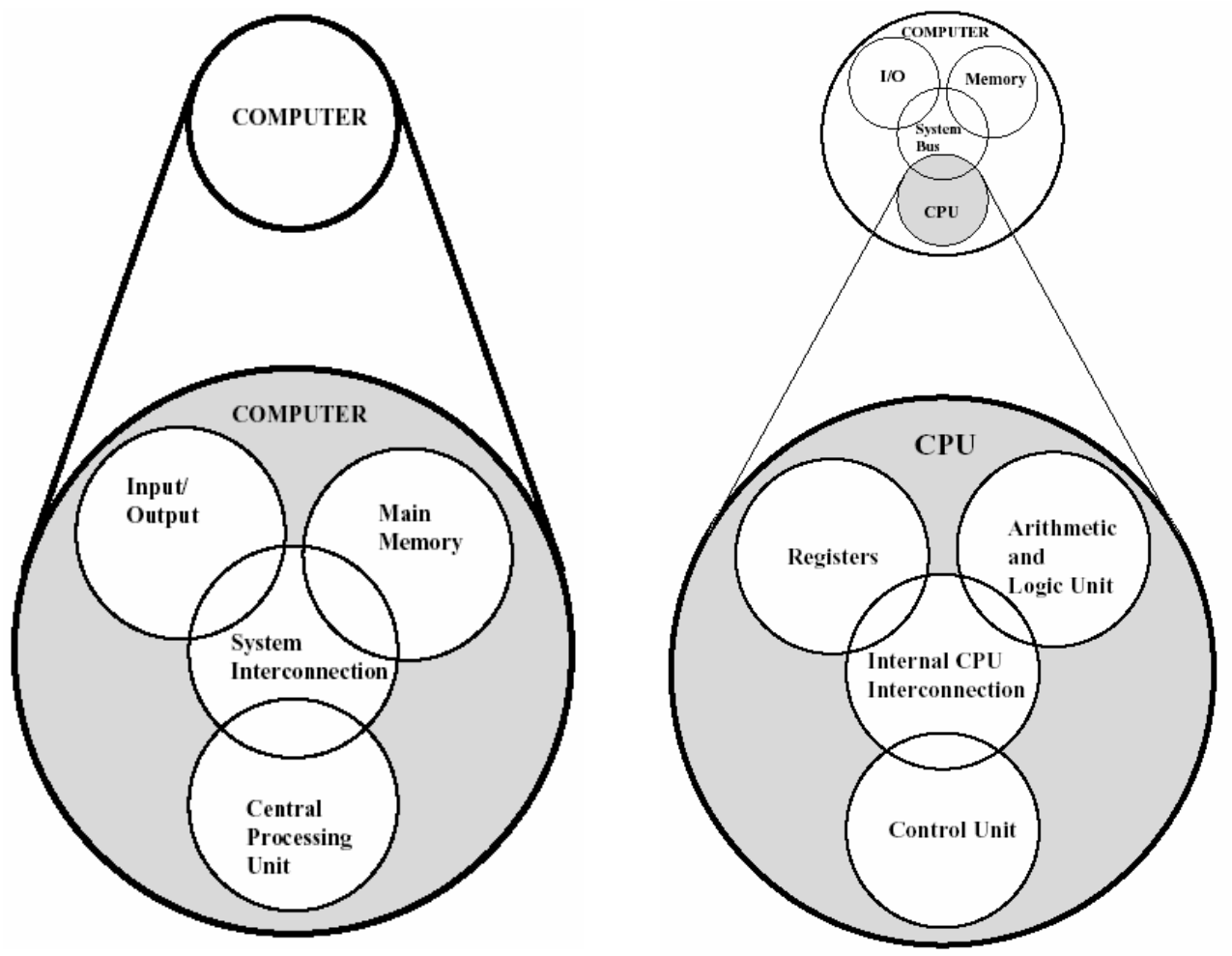
The C++ programming language is built upon C, using the exact same numeric representations and operations. Everything said in this chapter about C also holds for C++. The Java language definition, on the other hand, created a new set of standards for numeric representations and operations. Whereas the C standard is designed to allow a wide range of implementations, the Java standard is quite specific on the formats and encodings of data. We highlight the representations and operations supported by Java at several places in chapter 2.

2. Estrutura interna dum computador

Os principais blocos funcionais que se podem encontrar num computador podem ser agrupados em apenas 3: a entidade que processa a informação, a entidade que armazena a informação que está a ser processada, e as unidades que estabelecem a ligação deste par de entidades (processador-memória) com o exterior. Mais concretamente, os blocos são (as figuras são de COA):

- **Processador(es),** incluindo uma ou mais Unidades Centrais de Processamento **CPU**, e eventualmente processadores auxiliares ou **coprocessadores** para execução de funções matemáticas, gráficas, de comunicações, ...
  Os principais blocos que constituem um processador podem ser identificados como sendo:
  - **conjunto de registos** para armazenar temporariamente a informação que vem da memória ou os

valores de variáveis (da aplicação ou de gestão do sistema);
  - **unidades funcionais** (aritméticas, lógicas, de vírgula flutuante,...) para operar sobre as variáveis;
  - **unidade de controlo**, que emite a sequência de sinais adequados ao funcionamento do processador e para actuação noutros componentes do computador.
- **Memória principal**, onde é armazenada toda a informação que o CPU vai necessitar de manusear; encontra-se organizada em **células** que podem ser directa e individualmente **endereçadas** pelo CPU (ou por outro componente que também possa aceder directamente à memória); cada célula tem normalmente **8 bits** de dimensão (todos os processadores disponíveis comercialmente lidam com esta dimensão de célula); a dimensão máxima de memória física que um computador pode ter está normalmente associada à largura n do barramento de endereços ($\mathbf{2^n}$)
- **Dispositivos de Entrada/Saída (I/O) e respectivos controladores**, incluindo:
  - dispositivos que fazem interface com o ser humano: monitor, teclado, rato, impressora, colunas de som, ...
  - dispositivos que armazenam grandes quantidades de informação, também designados por memória secundária: disco, banda magnética, CD-ROM, ...
  - dispositivos de interface para comunicação com outros equipamentos: interfaces vídeo, placas de rede local, modems, interface RDIS, ...
  - dispositivos internos auxiliares, como um temporizador, um controlador de interrupções, um controlador de acessos directos à memória (DMA), ...



Para melhor se compreender o sistema de interligação entre os diversos componentes num computador, vamos analisá-los cada individualmente:

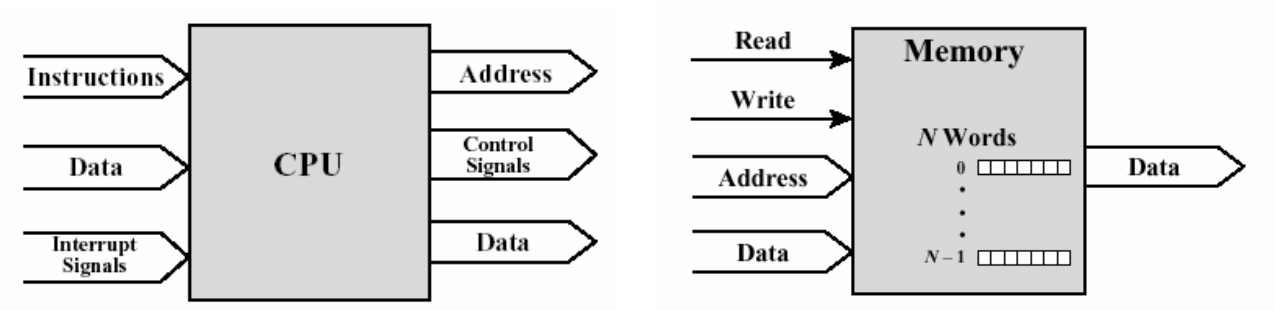
- **ligações no CPU**: para leitura das instruções (em memória) e de dados (em memória ou nos dispositivos de I/O); para escrita de dados resultantes do processamento efectuado (na memória ou em I/O), para enviar sinais de controlo para outras unidades (de leitura, de escrita, de

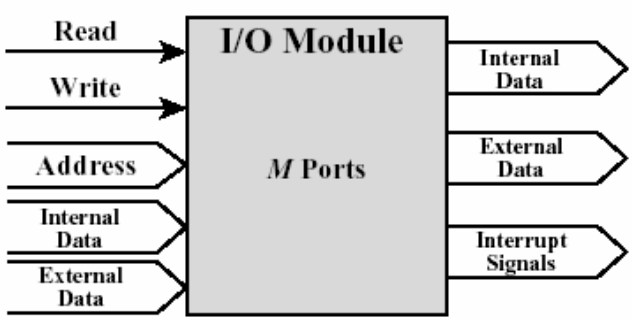
reconhecimento de pedidos feitos ao CPU, ...) e para receber e intervir quando lhe são solicitadas interrupções ao normal funcionamento da aplicação em execução (i.e., àinterrupções ao processo em execução);

- **ligações na memória**: para receber e enviar dados (sob o ponto de vista da memória, instruções são "dados"), para receber os endereços que especificam a localização dos dados a ler/escrever, e para receber sinais de controlo (leitura, escrita, *timing*, ...) ou enviar (*timing*, ...);

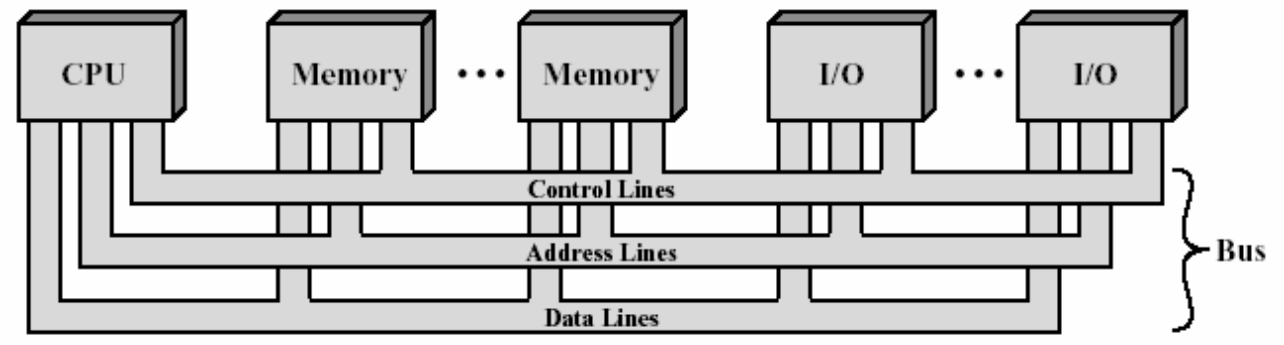
- **ligações nos dispositivos de I/O**: sob o ponto de vista do CPU, não existe distinção relativamente aos acessos à memória; i.e., recebem e enviam dados a pedido do CPU (ou doutro circuito apropriado), recebem endereços para especificar os portos (*ports*) de I/O (incluem módulos e os registos internos associados a cada periférico a controlar - registos de controlo de estado e de dados), recebem sinais de controlo idênticos aos da memória e enviam sinais de controlo associados ao processo de interrupção; adicionalmente há que lembrar que estes dispositivos recebem/enviam dados para os periféricos que lhe estão ligados.

Os sinais presentes na interligação de cada um desses módulos dum computador - CPU, memória, I/O - aos diversos barramentos para suportar a transferência de informação entre as diversas partes, estão representados na figura seguinte (de COA):





O sistema de interligação dos diversos componentes presentes num computador designa-se genericamente por barramentos (*bus*); estes barramentos são constituídos por um elevado número de ligações físicas, podendo estar agrupados de forma hierárquica.



As principais categorias de barramentos são normalmente designadas por:

- **Barramento de dados,** que têm por função transportar a informação (códigos dos programas e dados) entre os blocos funcionais dum computador; quanto maior a sua "largura", maior o número de bits que é possível transportar em simultâneo;
- **Barramento de endereços**, que têm por função transportar a identificação/localização ("endereço") dos sítios onde se pretende ler ou escrever dados (por ex., o endereço de uma célula de memória ou de um registo de estado de um controlador);
- **Barramento de controlo**, que agrupa todo o conjunto de sinais eléctricos de controlo do sistema, necessários ao bom funcionamento do computador como um todo (por ex., sinais para indicar que a informação que circula no barramento de dados é para ser escrita e não lida da célula de memória cuja localização segue no barramento de endereços; sinais para pedir o *bus*; sinal de *reset*; ...).

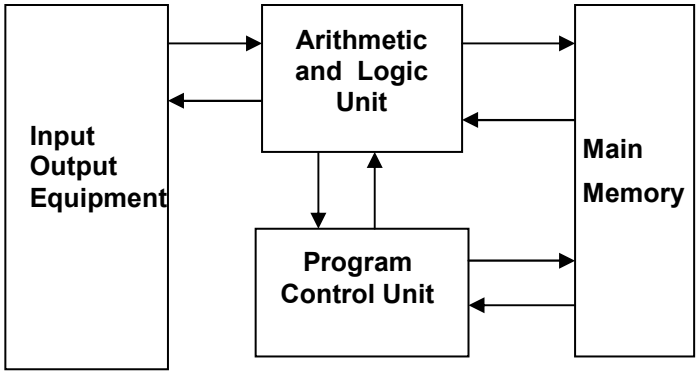
2.1 A origem do computador "moderno"

O modelo de computador proposto por von Neumann em 1945 tinha as seguintes características:

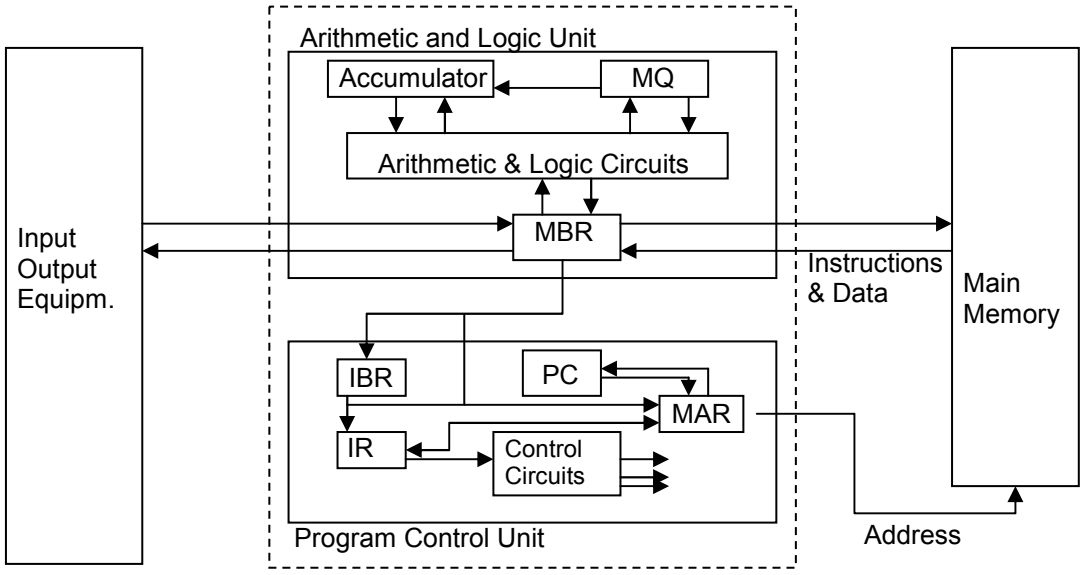
- os dados e as instruções são armazenados numa simples memória de leitura-escrita;
- os conteúdos desta memória são acedidos pelo endereço da sua localização, independentemente da informação lá contida;
- a execução ocorre de um modo sequencial (a menos que explicitamente modificado), de instrução para instrução.

Em 1946 von Neumann e os seus colegas projectaram um novo computador no Princeton Institute for Advanced Studies (IAS), o qual ficou conhecido como o computador IAS, e que é considerado por vários autores como o percursor dos computadores existentes hoje em dia, também conhecidos como máquinas von Neumann.

A sua estrutura é apresentada na figura ao lado, sendo mais detalhada na figura em baixo.



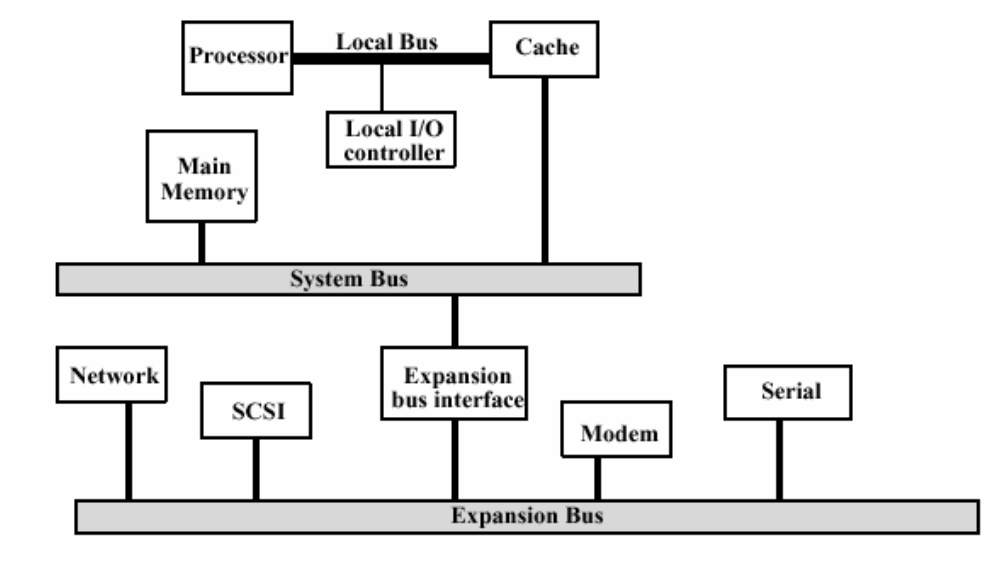
Estrutura de um computador IAS



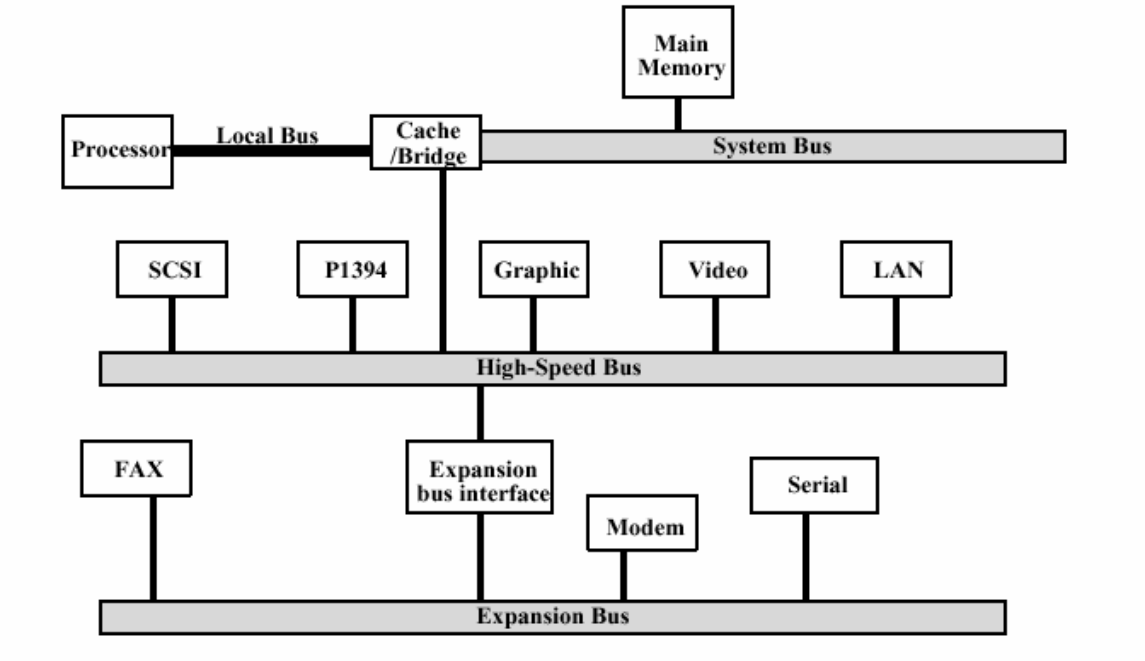
Estrutura mais detalhada de um computador IAS

2.2 A hierarquia duma arquitectura de barramentos

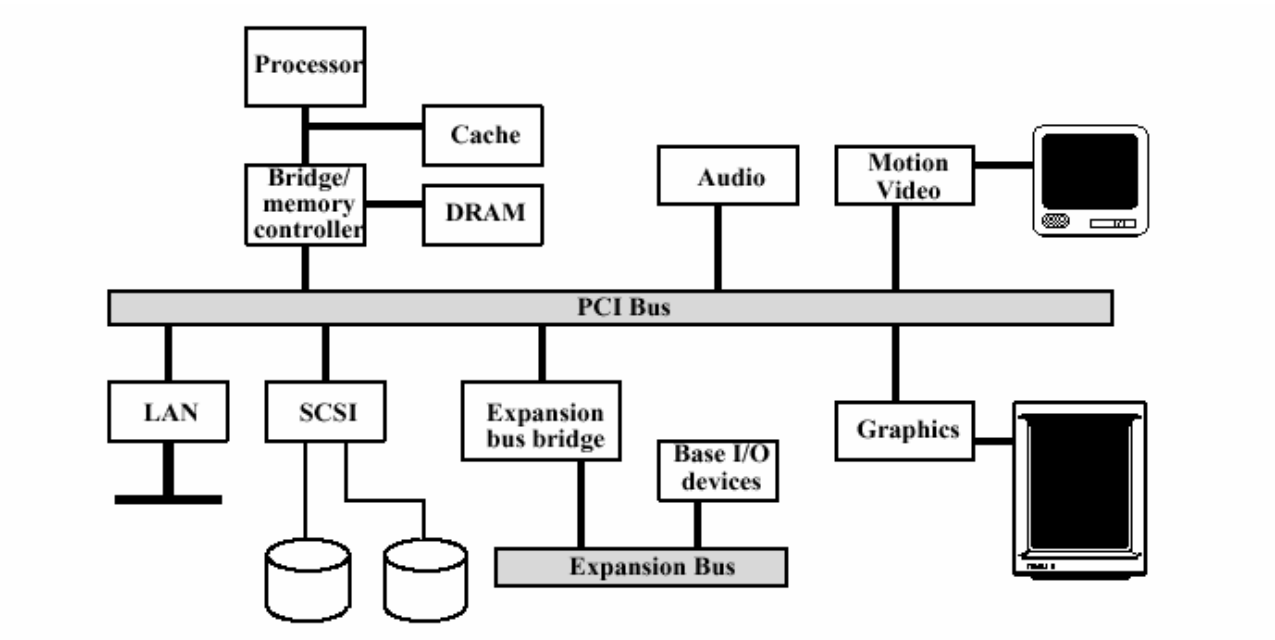
A existência de um único conjunto de barramentos para a circulação da informação entre os diversos módulos de um computador não leva em consideração que certos tipo de informação tem requisitos temporais muito mais exigentes que outros; por exemplo, a leitura de instruções da memória exige tempos de resposta muito mais curtos que as comunicações em rede com outros computadores, ou até a leitura de caracteres de um teclado. Daqui surgiu a necessidade de se organizar hierarquicamente a organização dos barramentos. Eis a **organização típica duma arquitectura hierárquica de barramentos**:



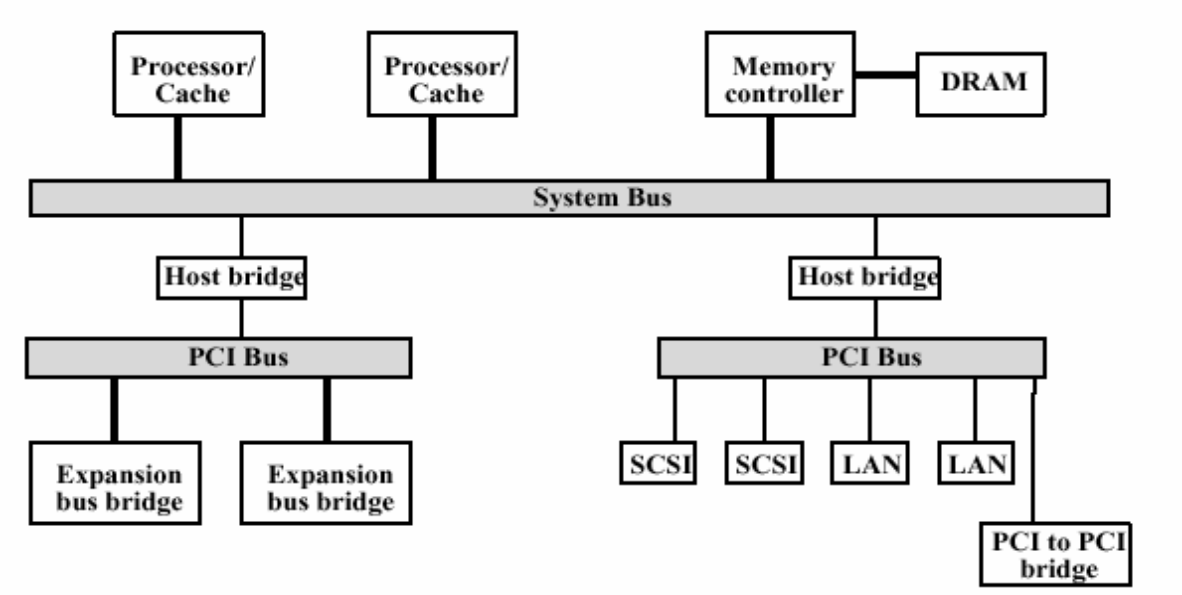
Neste tipo de organização de barramentos não se faz distinção entre periféricos com requisitos temporais exigentes de outros que não os tenham. A figura que se segue já toma em consideração essa distinção, e é uma **organização típica de uma hierarquia de barramentos de alto desempenho**:



Merece especial destaque uma referência à organização interna de computadores que utilizam o barramento conhecido como *Peripheral Component Interface*, também mais conhecido por PCI. Aqui podemos encontrar 2 tipos de organização, consoante o fim a que se destinam: para utilização individual (***desktop system***) ou para funcionar como servidor (***server system***). Eis uma ilustração de cada um deles:



Organização de barramentos típica de um sistema *desktop*



Organização de barramentos típica de um sistema servidor

2.3 Hardware Organization of a System

To understand what happens to our `hello` program when we run it, we need to understand the hardware organization of a typical system, which is shown in Figure 1.4. This particular picture is modeled after the family of Intel Pentium systems, but all systems have a similar look and feel. Don't worry about the complexity of this figure just now. We will get to its various details in stages throughout the course of the book.

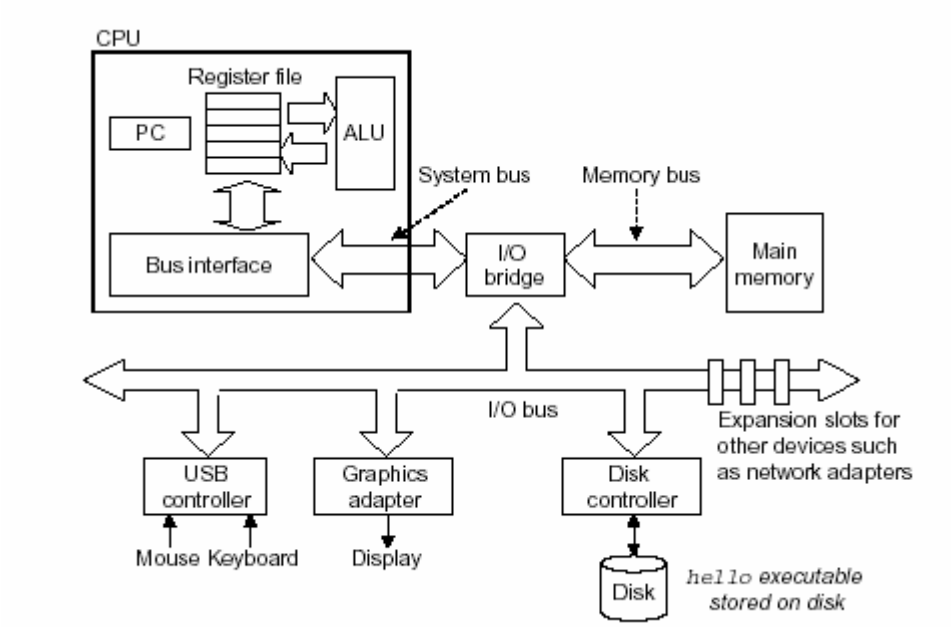


Figure 1.4: Hardware organization of a typical system.

Buses

Running throughout the system is a collection of electrical conduits called *buses* that carry bytes of information back and forth between the components. Buses are typically designed to transfer fixed-sized chunks of bytes known as *words*. The number of bytes in a word (the *word size*) is a fundamental system parameter that varies across systems. For example, Intel Pentium systems have a word size of 4 bytes, while serverclass systems such as Intel Itaniums and high-end Sun SPARCS have word sizes of 8 bytes. Smaller systems that are used as embedded controllers in automobiles and factories can have word sizes of 1 or 2 bytes. For simplicity, we will assume a word size of 4 bytes, and we will assume that buses transfer only one word at a time.

I/O Devices

Input/output (I/O) devices are the system's connection to the external world. Our example system has four I/O devices: a keyboard and mouse for user input, a display for user output, and a disk drive (or simply disk) for long-term storage of data and programs. Initially, the executable `hello` program resides on the disk. Each I/O device is connected to the I/O bus by either a *controller* or an *adapter*. The distinction between the two is mainly one of packaging. Controllers are chip sets in the device itself or on the system's main printed circuit board (often called the *motherboard*). An adapter is a card that plugs into a slot on the motherboard. Regardless, the purpose of each is to transfer information back and forth between the I/O bus and an I/O device.

Chapter 6 has more to say about how I/O devices such as disks work. In Chapter 11, you will learn how to use the Unix I/O interface to access devices from your application programs. We focus on the especially interesting class of devices known as networks, but the techniques generalize to other kinds of devices as well.

Main Memory

The *main memory* is a temporary storage device that holds both a program and the data it manipulates while the processor is executing the program. Physically, main memory consists of a collection of *Dynamic Random Access Memory (DRAM)* chips. Logically, memory is organized as a linear array of bytes, each with its own unique address (array index) starting at zero. In general, each of the machine

instructions that constitute a program can consist of a variable number of bytes. The sizes of data items that correspond to C program variables vary according to type. For example, on an Intel machine running Linux, data of type `short` requires two bytes, types `int`, `float`, and `long` four bytes, and type `double` eight bytes. Chapter 6 has more to say about how memory technologies such as DRAM chips work, and how they are combined to form main memory.

Processor

The *central processing unit* (CPU), or simply *processor*, is the engine that interprets (or *executes*) instructions stored in main memory. At its core is a word-sized storage device (or *register*) called the *program counter* (PC). At any point in time, the PC points at (contains the address of) some machine-language instruction in main memory.

From the time that power is applied to the system, until the time that the power is shut off, the processor blindly and repeatedly performs the same basic task, over and over again: It reads the instruction from memory pointed at by the program counter (PC), interprets the bits in the instruction, performs some simple *operation* dictated by the instruction, and then updates the PC to point to the *next* instruction, which may or may not be contiguous in memory to the instruction that was just executed.

There are only a few of these simple operations, and they revolve around main memory, the *register file*, and the *arithmetic/logic unit* (ALU). The register file is a small storage device that consists of a collection of word-sized registers, each with its own unique name. The ALU computes new data and address values. Here are some examples of the simple operations that the CPU might carry out at the request of an instruction:

- *Load:* Copy a byte or a word from main memory into a register, overwriting the previous contents of the register.
- *Store:* Copy a byte or a word from a register to a location in main memory, overwriting the previous contents of that location.
- *Update:* Copy the contents of two registers to the ALU, which adds the two words together and stores the result in a register, overwriting the previous contents of that register.
- *I/O Read:* Copy a byte or a word from an I/O device into a register.
- *I/O Write:* Copy a byte or a word from a register to an I/O device.
- *Jump:* Extract a word from the instruction itself and copy that word into the program counter (PC), overwriting the previous value of the PC.

3. Níveis de abstracção num computador e mecanismos de conversão

Na utilização de um computador é possível identificar vários níveis de abstracção, sendo os mais relevantes, no âmbito desta temática:

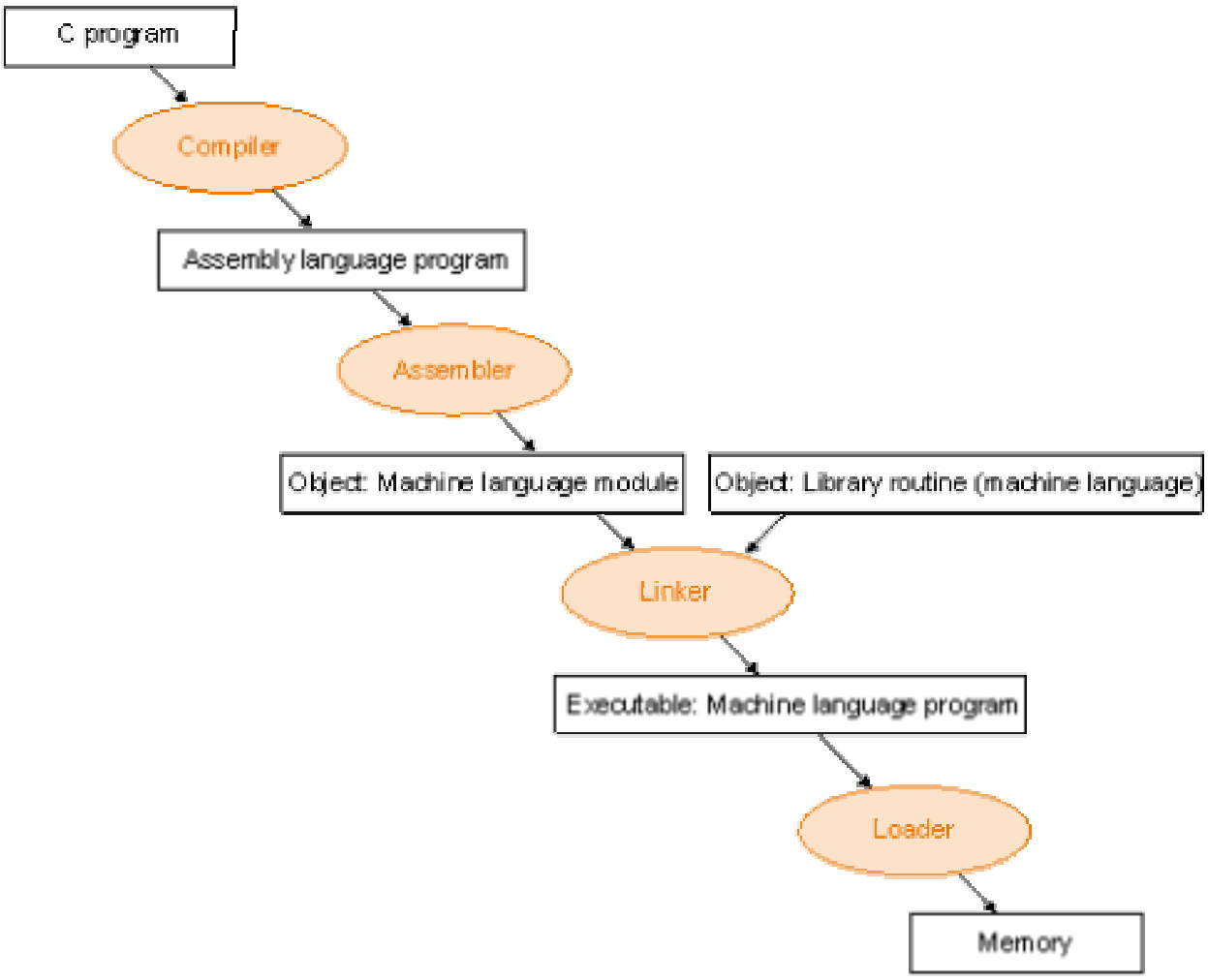
- **Nível da linguagem máquina (em binário)**: instruções e variáveis totalmente codificadas em binário, sendo a codificação das instruções sempre associada a um dado processador e tendo como objectivo a execução eficiente e rápida dos comandos; a sua utilização é pouco adequada para seres humanos;
- **Nível da linguagem *assembly*** (tradução literal do inglês: "de montagem"): equivalente ao nível anterior, mas em vez da notação puramente binária, a linguagem usa mnemónicas para especificar as operações pretendidas, bem como os valores ou localizações dos operandos; embora este nível seja melhor manuseado por seres humanos, ele ainda é inteiramente dependente do conjunto de instruções dum dado processador, isto é, não é portátil entre processadores de famílias diferentes, e as estruturas que manipula, quer de controlo, quer de dados, são de muito baixo nível;
- **Nível das linguagens HLL** (*High Level Languages*, como o C, Pascal, FORTRAN, ...): linguagens mais poderosas e mais próximas dos seres humanos, que permitem a construção de programas para execução eficiente em qualquer processador.

Dado que o processador apenas "entende" os comandos em linguagem máquina, é necessário converter os programas escritos em linguagens dos níveis de abstracção superiores para níveis mais baixos, até eventualmente se chegar à linguagem máquina. Estes tradutores ou conversores de níveis são

normalmente designados por:

- **Compiladores**: programas que traduzem os programas escritos em HLL para o nível de abstracção inferior, i.e., para *assembly*; a maioria dos compiladores existentes incluem já os dois passos da tradução para linguagem máquina, isto é, traduzem de HLL directamente para linguagem máquina binária, sem necessitarem de um *assembler*;
- ***Assemblers*** : programas que lêem os ficheiros com os programas escritos em linguagem de montagem - *assembly language* – e os convertem para linguagem máquina em binário, i.e., "montam" as instruções em formato adequado ao processador (também designados por "montadores" na literatura brasileira).

A figura a seguir ilustra estes níveis de abstracção e respectivos mecanismos de conversão.



Existe ainda outro mecanismo que permite executar programas escritos em HLL sem usar a compilação: a interpretação. Com um interpretador, as instruções de HLL são analisadas uma a uma, e o interpretador gera código em linguagem máquina e executa de imediato esse código, sem o guardar. Não há propriamente uma tradução de um programa noutro, mas sim a análise dum programa seguida de geração e execução do código máquina associado.

3.1 Programs are Translated by Other Programs into Different Forms *(retirado de CSAPP)*

The `hello` program begins life as a high-level C program because it can be read and understood by human beings in that form. However, in order to run `hello.c` on the system, the individual C statements must be translated by other programs into a sequence of low-level *machine-language* instructions. These instructions are then packaged in a form called an *executable object program* and stored as a binary disk file. Object programs are also referred to as *executable object files*.

On a Unix system, the translation from source file to object file is performed by a *compiler driver*:

```
unix> gcc -o hello hello.c
```

Here, the GCC compiler driver reads the source file `hello.c` and translates it into an executable object file `hello`. The translation is performed in the sequence of four phases shown in Figure 1.3. The programs that perform the four phases (*preprocessor*, *compiler*, *assembler*, and *linker*) are known collectively as the *compilation system*.

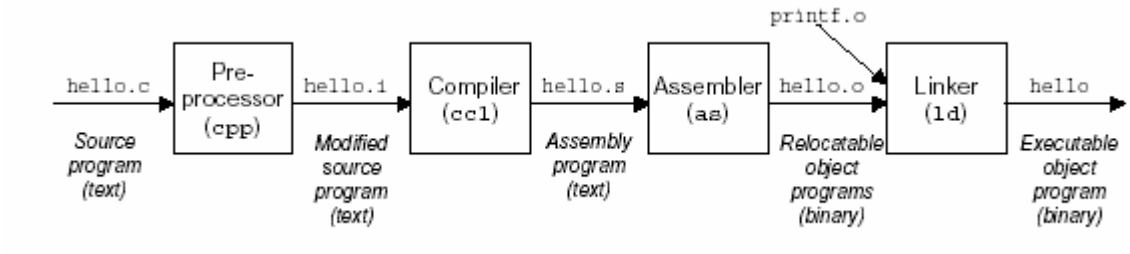


Figure 1.3: The compilation system.

- *Preprocessing phase.* The preprocessor (`cpp`) modifies the original C program according to directives that begin with the `#` character. For example, the `#include <stdio.h>` command in line 1 of `hello.c` tells the preprocessor to read the contents of the system header file `stdio.h` and insert it directly into the program text. The result is another C program, typically with the `.i` suffix.
- *Compilation phase.* The compiler (`cc1`) translates the text file `hello.i` into the text file `hello.s`, which contains an *assembly-language program*. Each statement in an assembly-language program exactly describes one low-level machine-language instruction in a standard text form. Assembly language is useful because it provides a common output language for different compilers for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language.
- *Assembly phase.* Next, the assembler (`as`) translates `hello.s` into machine-language instructions, packages them in a form known as a *relocatable object program*, and stores the result in the object file `hello.o`. The `hello.o` file is a binary file whose bytes encode machine language instructions rather than characters. If we were to view `hello.o` with a text editor, it would appear to be gibberish.
- *Linking phase.* Notice that our `hello` program calls the `printf` function, which is part of the *standard C library* provided by every C compiler. The `printf` function resides in a separate precompiled object file called `printf.o`, which must somehow be merged with our `hello.o` program. The linker (`ld`) handles this merging. The result is the `hello` file, which is an *executable object file* (or simply *executable*) that is ready to be loaded into memory and executed by the system.

It Pays to Understand How Compilation Systems Work

For simple programs such as `hello.c`, we can rely on the compilation system to produce correct and efficient machine code. However, there are some important reasons why programmers need to understand how compilation systems work:

- *Optimizing program performance.* Modern compilers are sophisticated tools that usually produce good code. As programmers, we do not need to know the inner workings of the compiler in order to write efficient code. However, in order to make good coding decisions in our C programs, we do need a basic understanding of assembly language and how the compiler translates different C statements into assembly language. For example, is a `switch` statement always more efficient than a sequence of `if-then-else` statements? Just how expensive is a function call? Is a `while` loop more efficient than a `do` loop? Are pointer references more

efficient than array indexes? Why does our loop run so much faster if we sum into a local variable instead of an argument that is passed by reference? Why do two functionally equivalent loops have such different running times? In Chapter 3, we will introduce the Intel IA32 machine language and describe how compilers translate different C constructs into that language. In Chapter 5 you will learn how to tune the performance of your C programs by making simple transformations to the C code that help the compiler do its job. And in Chapter 6 you will learn about the hierarchical nature of the memory system, how C compilers store data arrays in memory, and how your C programs can exploit this knowledge to run more efficiently.

- *Understanding link-time errors.* In our experience, some of the most perplexing programming errors are related to the operation of the linker, especially when you are trying to build large software systems. For example, what does it mean when the linker reports that it cannot resolve a reference? What is the difference between a static variable and a global variable? What happens if you define two global variables in different C files with the same name? What is the difference between a static library and a dynamic library? Why does it matter what order we list libraries on the command line? And scariest of all, why do some linker-related errors not appear until run time? You will learn the answers to these kinds of questions in Chapter 7
- *Avoiding security holes.* For many years now, *buffer overflow bugs* have accounted for the majority of security holes in network and Internet servers. These bugs exist because too many programmers are ignorant of the stack discipline that compilers use to generate code for functions. We will describe the stack discipline and buffer overflow bugs in Chapter 3 as part of our study of assembly language.

3.2 Processors Read and Interpret Instructions Stored in Memory

At this point, our `hello.c` source program has been translated by the compilation system into an executable object file called `hello` that is stored on disk. To run the executable file on a Unix system, we type its name to an application program known as a *shell*:

```
unix> ./hello
hello, world
unix>
```

The shell is a command-line interpreter that prints a prompt, waits for you to type a command line, and then performs the command. If the first word of the command line does not correspond to a built-in shell command, then the shell assumes that it is the name of an executable file that it should load and run. So in this case, the shell loads and runs the `hello` program and then waits for it to terminate. The `hello` program prints its message to the screen and then terminates. The shell then prints a prompt and waits for the next input command line.

The Operating System Manages the Hardware

When the shell loaded and ran the `hello` program, and when the `hello` program printed its message, neither program accessed the keyboard, display, disk, or main memory directly. Rather, they relied on the services provided by the *operating system*. We can think of the operating system as a layer of software interposed between the application program and the hardware, as shown in Figure 1.10. All attempts by an application program to manipulate the hardware must go through the operating system.

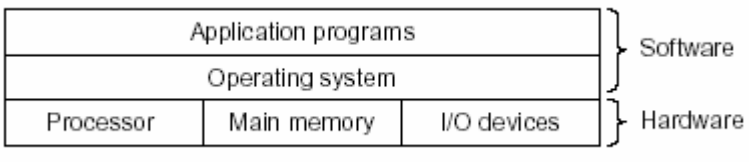


Figure 1.10: Layered view of a computer system.

The operating system has two primary purposes: (1) to protect the hardware from misuse by runaway applications, and (2) to provide applications with simple and uniform mechanisms for manipulating complicated and often wildly different low-level hardware devices. The operating system achieves both goals via the fundamental abstractions shown in Figure 1.11: *processes*, *virtual memory*, and *files*. As this figure suggests, files are abstractions for I/O devices, virtual memory is an abstraction for both the main memory and disk I/O devices, and processes are abstractions for the processor, main memory, and I/O devices.

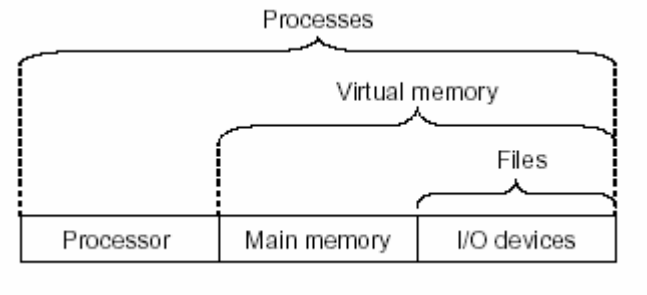


Figure 1.11: Abstractions provided by an operating system.

3.3 Running the `hello` Program

Given a simple view of a system's hardware organization and operation, we can begin to understand what happens when we run our example program. We must omit a lot of details here that will be filled in later, but for now we will be content with the big picture.

Initially, the shell program is executing its instructions, waiting for us to type a command. As we type the characters "`./hello`" at the keyboard, the shell program reads each one into a register, and then stores it in memory, as shown in Figure 1.5.

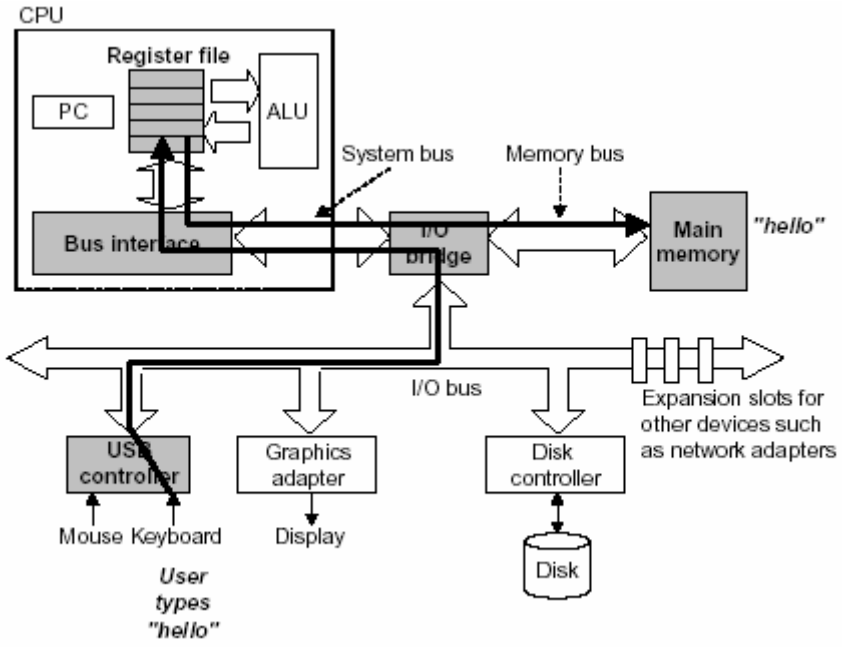


Figure 1.5: Reading the `hello` command from the keyboard.

When we hit the `enter` key on the keyboard, the shell knows that we have finished typing the command. The shell then loads the executable `hello` file by executing a sequence of instructions that

copies the code and data in the `hello` object file from disk to main memory. The data include the string of characters "`hello, world\n`" that will eventually be printed out.

Using a technique known as *direct memory access* (DMA, discussed in Chapter 6), the data travels directly from disk to main memory, without passing through the processor. This step is shown in Figure 1.6.

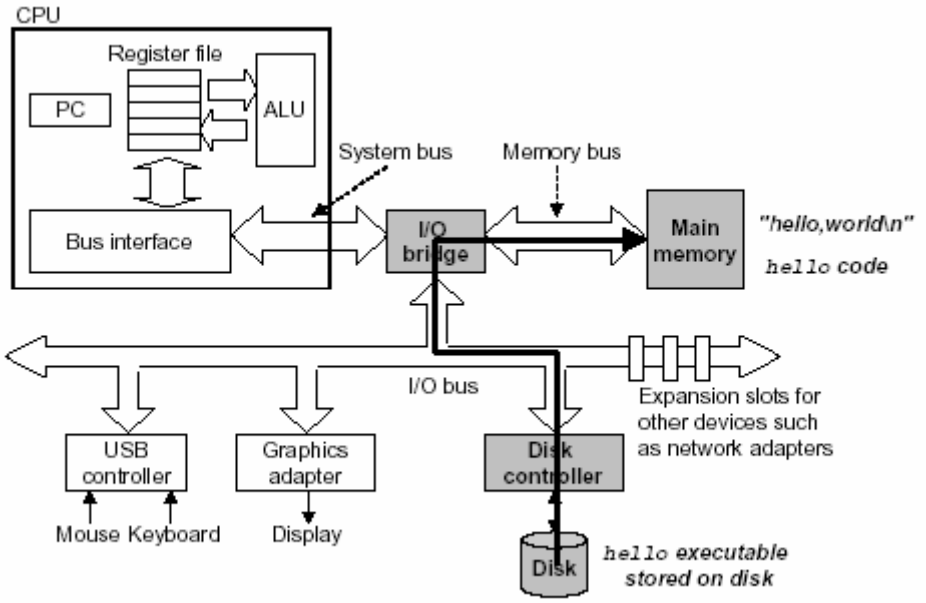


Figure 1.6: Loading the executable from disk into main memory.

Once the code and data in the `hello` object file are loaded into memory, the processor begins executing the machine-language instructions in the `hello` program's `main` routine. These instruction copy the bytes in the "`hello, world\n`" string from memory to the register file, and from there to the display device, where they are displayed on the screen. This step is shown in Figure 1.7.

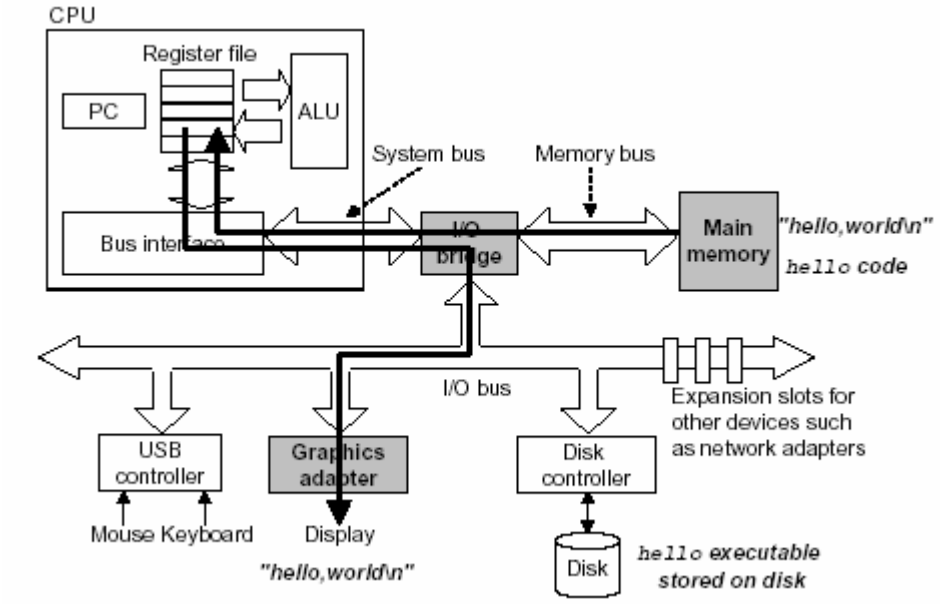


Figure 1.7: Writing the output string from memory to the display.

4. Execução de instruções num computador

Na execução de um programa já em memória pronto a ser executado – programa em linguagem máquina - o CPU executa cada instrução numa sequência de passos elementares, assim agrupados:

- **Fetch**:
  - lê uma instrução da localização em memória especificada pelo registo IP (*Instruction Pointer*)
  - incrementa o IP de modo a ficar a apontar para a próxima instrução
  - carrega a instrução que vem da memória no seu IR (*Instruction Register*)
- **Execute**:
  - analisa a instrução para determinar o tipo de operação e operandos
  - se a instrução necessita de operandos, calcula a sua localização
  - se necessário, vai buscar o(s) operandos(s)
  - executa a operação especificada na instrução
  - guarda o resultado da operação efectuada
  - volta ao passo inicial para ir buscar nova instrução

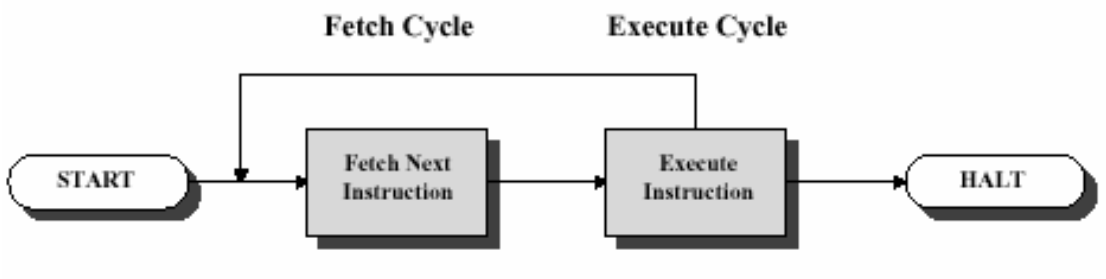


Figure 3.3: Basic Instruction Cycle (*do livro COA*).

```
public class Interp {
    static int PC;                          // program counter holds address of next instr
    static int AC;                          // the accumulator, a register for doing arithmetic
    static int instr;                       // a holding register for the current instruction
    static int instr_type;                  // the instruction type (opcode)
    static int data_loc;                    // the address of the data, or -1 if none
    static int data;                        // holds the current operand
    static boolean run_bit = true;          // a bit that can be turned off to halt the machine

    public static void interpret(int memory[ ], int starting_address) {
        // This procedure interprets programs for a simple machine with instructions having
        // one memory operand. The machine has a register AC (accumulator), used for
        // arithmetic. The ADD instruction adds an integer in memory to the AC, for example
        // The interpreter keeps running until the run bit is turned off by the HALT instruction
        // The state of a process running on this machine consists of the memory, the
        // program counter, the run bit, and the AC. The input parameters consist of
        // of the memory image and the starting address.

        PC = starting_address;
        while (run_bit) {
```

```
        instr = memory[PC];                    // fetch next instruction into instr
        PC = PC + 1;                           // increment program counter
        instr_type = get_instr_type(instr);    // determine instruction type
        data_loc = find_data(instr, instr_type); // locate data (-1 if none)
        if (data_loc >= 0)                     // if data_loc is -1, there is no operand
        data = memory[data_loc];               // fetch the data
        execute(instr_type, data);             // execute instruction
        }
    }
    private static int get_instr_type(int addr) { ... }
    private static int find_data(int instr, int type) { ... }
    private static void execute(int type, int data){ ... }
}
```

Fig. 2-3. An interpreter for a simple computer (written in Java) (*do livro SCO*).

Um CPU suporta ainda um mecanismo que permite a **interrupção** deste ciclo de passos encadeados, que lhe permita atender a uma solicitação externa (dum periférico, dum temporizador, por anomali do hardware) ou interna (como resultado da execução do programa, como por exemplo, uma divisão por 0). Quando um sinal de interrupção é activado, o CPU suspende a execução do programa, salvaguardando o seu contexto (como o IP e outros registos).

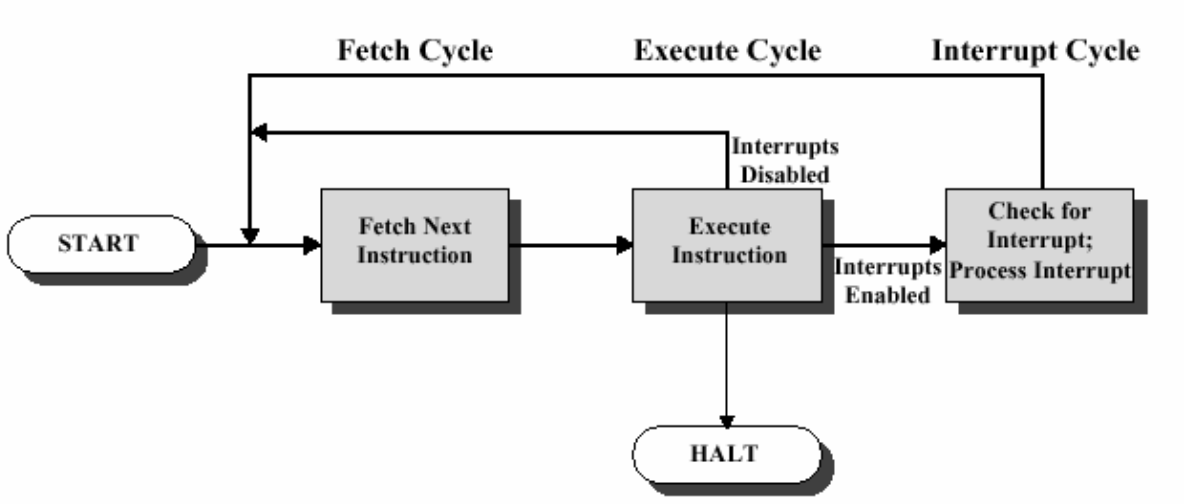


Figure 3.9: Instruction Cycle with Interrupts (*do livro SCO*).

4.1 Acessos à memória na execução de instruções

Durante a execução de cada instrução, o CPU necessita de aceder à memória para ir buscar a instrução (*Fetch*), e o(s) operando(s) sempre que estes estejam em memória. O processo de aceder à memória para ir buscar uma instrução é idêntico ao de buscar um operando para efectuar uma operação (apresentado com detalhe na próxima sub-secção); i.e., o CPU coloca o conteúdo do IP/PC no barramento de endereços, activa o sinal de leitura à memória no barramento de controlo, e o conteúdo da(s) célula(s) de memória indicada(s) no barramento de endereços é colocado no barramento de dados, de modo que o CPU o possa ler e colocar no registo de instrução (IR).

A sub-secção seguinte apresenta com mais detalhe os acessos à memória nas operações de leitura e escrita de dados na memória.

4.2 Accessing Main Memory (*retirado de CSAPP*)

Data flows back and forth between the processor and the DRAM main memory over shared electrical conduits called *buses*. Each transfer of data between the CPU and memory is accomplished with a series of steps called a *bus transaction*. A *read transaction* transfers data from the main memory to the CPU. A *write transaction* transfers data from the CPU to the main memory.

A *bus* is a collection of parallel wires that carry address, data, and control signals. Depending on the particular bus design, data and address signals can share the same set of wires, or they can use different sets. Also, more than two devices can share the same bus. The control wires carry signals that synchronize the transaction and identify what kind of transaction is currently being performed. For example, is this transaction of interest to the main memory, or to some other I/O device such as a disk controller? Is the transaction a read or a write? Is the information on the bus an address or a data item?

Figure 6.6 shows the configuration of a typical desktop system. The main components are the CPU chip, a chipset that we will call an *I/O bridge* (which includes the memory controller), and the DRAM memory modules that comprise main memory. These components are connected by a pair of buses: a *system bus* that connects the CPU to the I/O bridge, and a *memory bus* that connects the I/O bridge to the main memory.

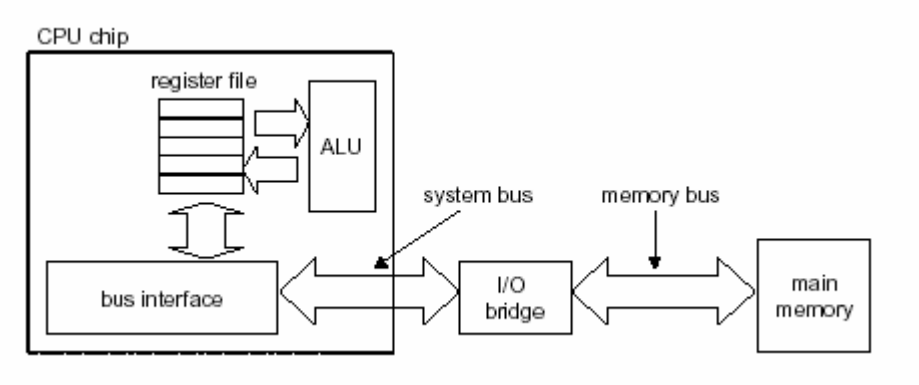


Figure 6.6: Typical bus structure that connects the CPU and main memory.

The I/O bridge translates the electrical signals of the system bus into the electrical signals of the memory bus. As we will see, the I/O bridge also connects the system bus and memory bus to an I/O bus that is shared by I/O devices such as disks and graphics cards. For now, though, we will focus on the memory bus.

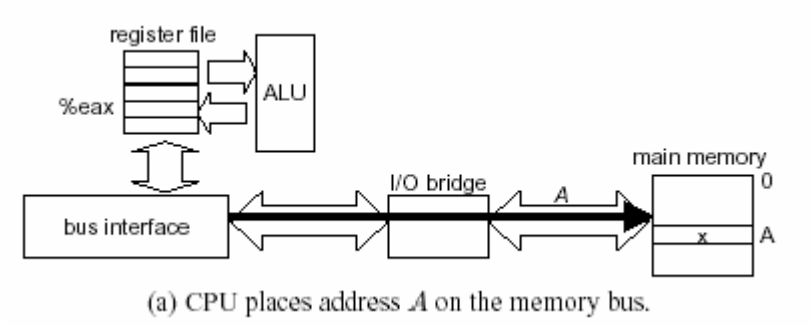
Consider what happens when the CPU performs a load operation such as

```
movl A,%eax
```

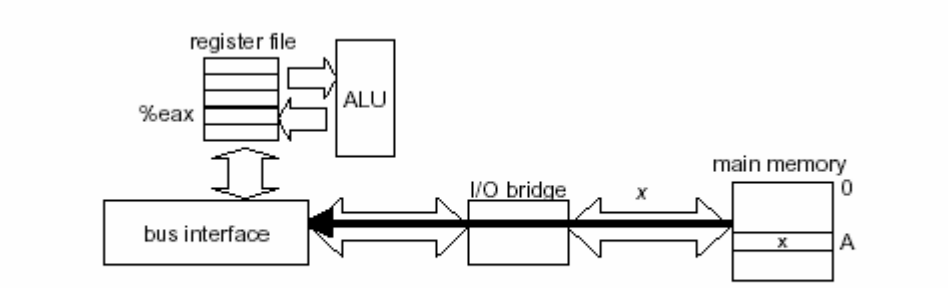
where the contents of address A are loaded into register `%eax`. Circuitry on the CPU chip called the *bus interface* initiates a read transaction on the bus.

The read transaction consists of three steps. First, the CPU places the address A on the system bus[8]. The I/O bridge passes the signal along to the memory bus[9] (Figure 6.7(a)). Next, the main memory senses the address signal on the memory bus[10], reads the address from the memory bus, fetches the data word from the DRAM, and writes the data to the memory bus[11]. The I/O bridge translates the memory bus signal into a system bus signal, and passes it along to the system bus (Figure 6.7(b)). Finally, the CPU senses the data on the system bus, reads it from the bus, and copies it to register `%eax` (Figure 6.7(c)).

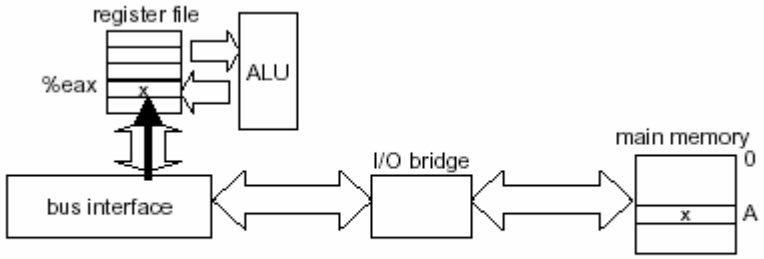
[8] Mais concretamente, no barramento de endereços, *Address Bus*
[9] Idem
[10] Em conjunto com o sinal de controlo, indicando que a operação é de leitura da memória
[11] Neste caso, no barramento de dados, *Data Bus*



(a) CPU places address A on the memory bus.



(b) Main memory reads A from the bus, retrieves word x, and places it on the bus.



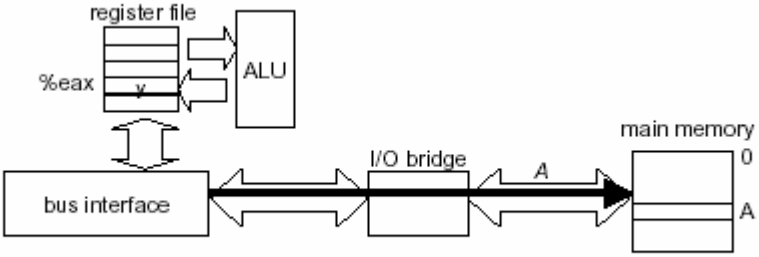
(c) CPU reads word x from the bus, and copies it into register `%eax`.

Figure 6.7: Memory read transaction for a load operation: `movl A,%eax`.

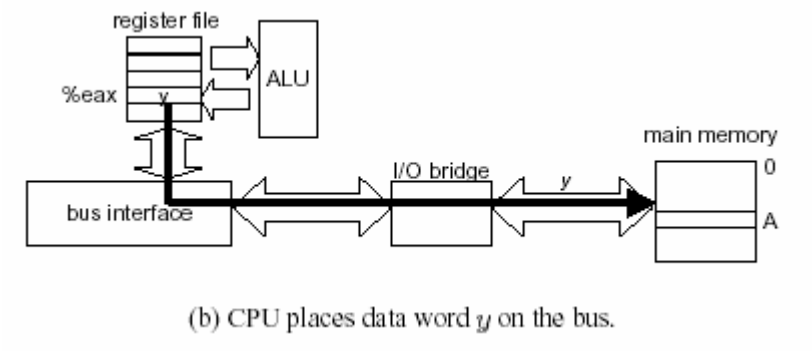
Conversely, when the CPU performs a store instruction such as

```
movl %eax,A
```

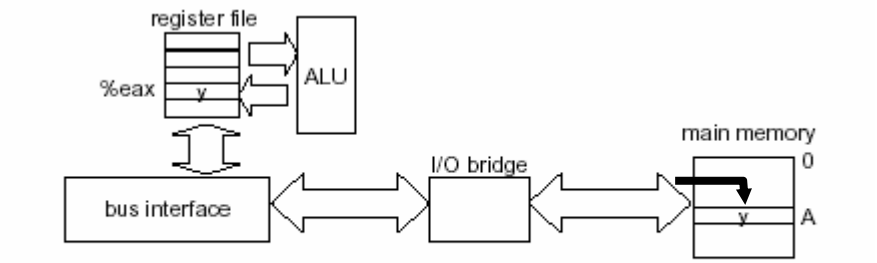
where the contents of register `%eax` are written to address `A`, the CPU initiates a write transaction. Again, there are three basic steps. First, the CPU places the address on the system bus. The memory reads the address from the memory bus and waits for the data to arrive (Figure 6.8(a)). Next, the CPU copies the data word in `%eax` to the system bus (Figure 6.8(b)). Finally, the main memory reads the data word from the memory bus and stores the bits in the DRAM (Figure 6.8(c)).



(a) CPU places address A on the memory bus. Main memory reads it and waits for the data word.



(b) CPU places data word y on the bus.



(c) Main memory reads data word y from the bus and stores it at address A.

Figure 6.8: Memory write transaction for a store operation: `movl %eax,A`.

4.3 Instruction-Level Parallelism (*retirado de SCO*)

Computer architects are constantly striving to improve performance of the machines they design. Making the chips run faster by increasing their clock speed is one way, but for every new design, there is a limit to what is possible by brute force at that moment in history. Consequently, most computer architects look to parallelism (doing two or more things at once) a way to get even more performance for a given clock speed.

Parallelism comes in two general forms: instruction-level parallelism and processor level parallelism. In the former, parallelism is exploited within individual instructions to get more instructions/sec out of the machine. In the latter, multiple CPUs work together on the same problem. Each approach has its own merits. In this section we will look at instruction-level parallelism.

Pipelining

It has been known for years that the actual fetching of instructions from memory is a major bottleneck in instruction execution speed. To alleviate this problem, computers going back at least as far as the IBM Stretch (1959) have had the ability to fetch instructions from memory in advance, so they would be there when they were needed. These instructions were stored in a set of registers called the **prefetch buffer**[12]. This way, when an instruction was needed, it could usually be taken from the prefetch buffer rather than waiting for a memory read to complete.

In effect, prefetching divides instruction execution into two parts: fetching and actual execution. The concept of a **pipeline** carries this strategy much further. Instead of dividing instruction execution into only two parts, it is often divided into many parts, each one handled by a dedicated piece of hardware, all of which can run in parallel.

[12] O CPU dos primeiros PC's, o Intel 8088, também tinha um *prefetch buffer* com 6 bytes.

Fig. 2-4(a) illustrates a pipeline with five units, also called stages. Stage 1 fetches the instruction from memory and places it in a buffer until it is needed. Stage 2 decodes the instruction, determining its type and what operands it needs. Stage 3 locates and fetches the operands, either from registers or from memory. Stage 4 actually does the work of carrying out the instruction, typically by running the operands through the data path of the CPU. Finally, stage 5 writes the result back to the proper register.

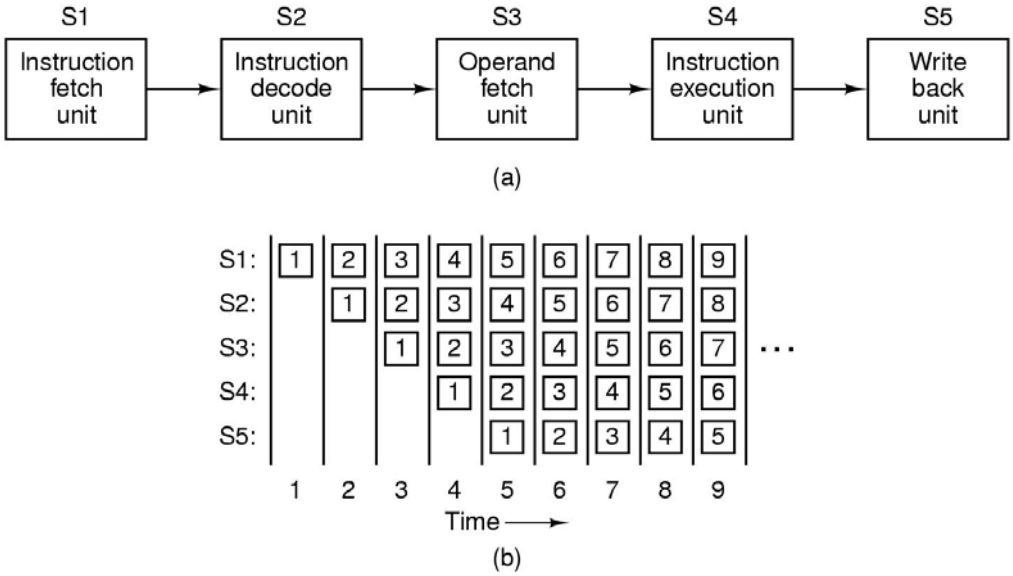


Figure 2-4. (a) A five stage pipeline. (b) The state of each stage as a function of time. Nine clock cycles are illustrated.

In Fig. 2.4(b) we show how the pipeline operates as a function of time. During clock cycle 1, stage S1 is working on instruction 1, fetching it from memory. During cycle 2, stage S2 decodes instruction 1, while stage S1 fetches instruction 2. During cycle 3, stage S3 fetches the operands for instruction 1, stage S2 decodes instruction, and stage S3 fetches the third instruction. (…) Finally, during cycle 5, S5 writes the result of instruction 1 back, while the other stages work on the following instructions.

Suppose that the cycle time of this machine is 2 nsec[13]. Then it takes 10 nsec for an instruction to progress all the way through the five-stage pipeline. At first glance, with an instruction taking 10 nsec, it might appear that the machine can run at 100 MIPS[14], but in fact it does much better than this. At every clock cycle (2 nsec), one new instruction is completed, so the actual rate of processing is 500 MIPS, not 100 MIPS[15].

Superscalar Architectures

If one pipeline is good, then surely two pipelines are better. One possible design for a dual pipeline CPU, based on Figure 2-4, is shown in Fig. 2-5. Here a single instruction fetch unit fetches pairs of instructions together and puts each one into its own pipeline, complete with its own ALU for parallel operation. To be able to run in parallel, the two instructions must not conflict over resource usage (e.g., registers), and neither must depend on the result of the other. As with a single pipeline, either the compiler must guarantee this situation to hold (i.e., the hardware does not check ands gives incorrect results if the instructions are not compatible), or conflicts are detected and eliminates during execution using extra hardware.

[13] O que corresponde à utilização de um *clock* com uma frequência de 500MHz
[14] ***M**illions of **I**nstructions **P**er **S**econd*
[15] Nota: o aumento do desempenho muito raramente é proporcional ao nº de níveis de *pipeline*, pois nem sempre se consegue manter o *pipeline* cheio; por ex., quando uma instrução precisa do resultado da anterior, ou sempre que há instruções de salto, o *pipeline* poderá ter de ser empatado.

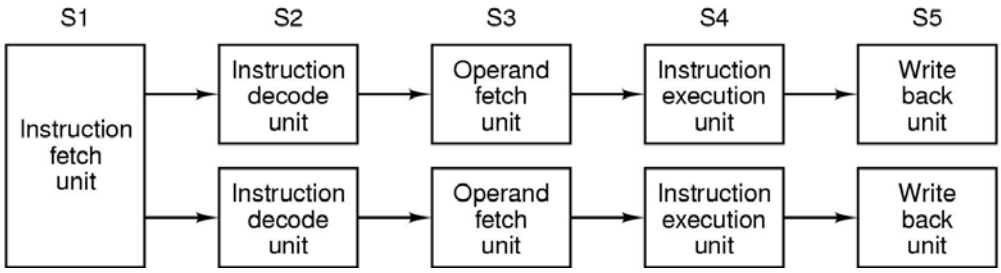


Figure 2-5. Dual five-stage pipelines with a common instruction fetch unit.

Although pipelines, single or double, are mostly used on RISC[16] machines (the 386 and its predecessors did not have any), starting with 486 Intel began introducing pipelines into its CPUs. The 486 had one pipeline and the Pentium had two five-stages pipelines roughly as Fig. 2-5, although the exact division of work between stages 2 and 3 (called decode-1 and decode-2) was slightly different than in our example. The main pipeline, called the **u pipeline**, could execute a arbitrary Pentium instruction. The second pipeline, called the **v pipeline**, could execute only simple integer instructions.

Complex rules determined whether a pair of instructions were compatible so they could be executed in parallel. If the instructions in a pair were not simple enough or incompatible, only the first one was executed (in the u pipeline). The second one was then held and paired with the instruction following it. Instructions were always executed in order.

4.4 Caches Matter *(retirado de CSAPP)*

An important lesson from this simple example is that a system spends a lot of time moving information from one place to another[17]. The machine instructions in the `hello` program are originally stored on disk. When the program is loaded, they are copied to main memory. As the processor runs the program, instructions are copied from main memory into the processor. Similarly, the data string "`hello,world\n`", originally on disk, is copied to main memory, and then copied from main memory to the display device. From a programmer's perspective, much of this copying is overhead that slows down the "real work" of the program.

Thus, a major goal for system designers is make these copy operations run as fast as possible. Because of physical laws, larger storage devices are slower than smaller storage devices. And faster devices are more expensive to build than their slower counterparts. For example, the disk drive on a typical system might be 100 times larger than the main memory, but it might take the processor 10,000,000 times longer to read a word from disk than from memory.

Similarly, a typical register file stores only a few hundred bytes of information, as opposed to millions of bytes in the main memory. However, the processor can read data from the register file almost 100 times faster than from memory. Even more troublesome, as semiconductor technology progresses over the years, this *processor-memory gap* continues to increase. It is easier and cheaper to make processors run faster than it is to make main memory run faster.

To deal with the processor-memory gap, system designers include smaller faster storage devices called *cache memories* (or simply caches) that serve as temporary staging areas for information that the processor is likely to need in the near future. Figure 1.8 shows the cache memories in a typical system. An *L1 cache* on the processor chip holds tens of thousands of bytes and can be accessed nearly as fast as the register file. A larger *L2 cache* with hundreds of thousands to millions of bytes is connected to the processor by a special bus. It might take 5 times longer for the process to access the L2 cache than the L1 cache, but this is still 5 to 10 times faster than accessing the main memory. The L1 and L2 caches are implemented with a hardware technology known as *Static Random Access Memory* (SRAM).

[16] *Reduced Instruction Set Computer*, conceito a ser detalhado adiante
[17] Esta análise, embora não considere o impacto do *pipeline*, mostra já a relevância duma hierarquia de memória. A introdução do *pipeline* vem reforçar ainda mais essa necessidade.

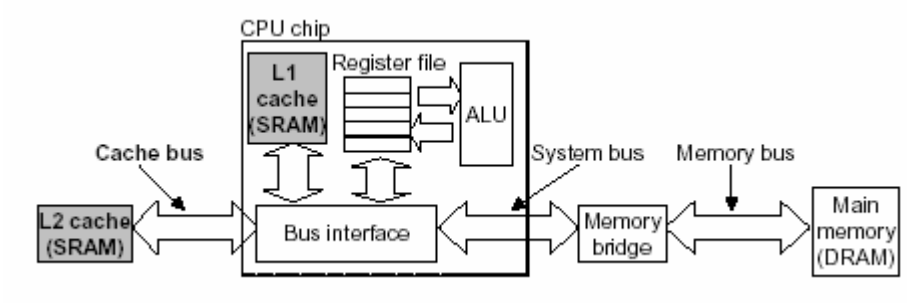


Figure 1.8: Cache memories.

One of the most important lessons in this book is that application programmers who are aware of cache memories can exploit them to improve the performance of their programs by an order of magnitude. You will learn more about these important devices and how to exploit them in Chapter 6.

4.5 Storage Devices Form a Hierarchy *(retirado de CSAPP)*

This notion of inserting a smaller, faster storage device (e.g., cache memory) between the processor and a larger slower device (e.g., main memory) turns out to be a general idea. In fact, the storage devices in every computer system are organized as a *memory hierarchy* similar to Figure 1.9. As we move from the top of the hierarchy to the bottom, the devices become slower, larger, and less costly per byte. The register file occupies the top level in the hierarchy, which is known as level 0 or L0. The L1 cache occupies level 1 (hence the term L1). The L2 cache occupies level 2. Main memory occupies level 3, and so on. The main idea of a memory hierarchy is that storage at one level serves as a cache for storage at the next lower level. Thus, the register file is a cache for the L1 cache, which is a cache for the L2 cache, which is a cache for the main memory, which is a cache for the disk. On some networked systems with distributed file systems, the local disk serves as a cache for data stored on the disks of other systems.

Just as programmers can exploit knowledge of the L1 and L2 caches to improve performance, programmers can exploit their understanding of the entire memory hierarchy. Chapter 6 will have much more to say about this.

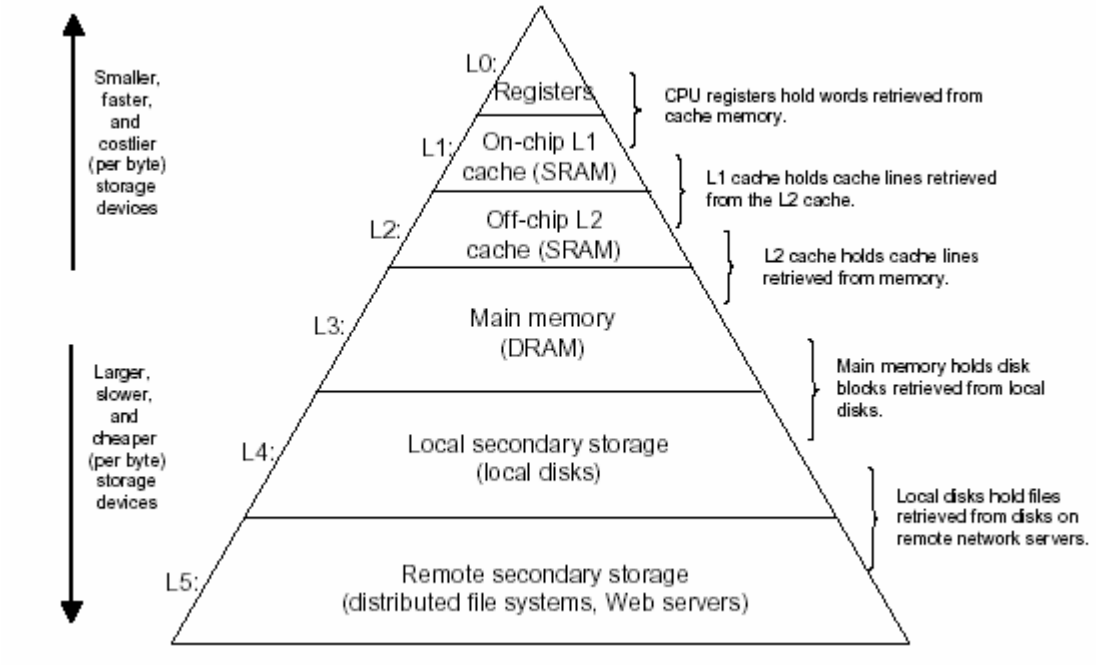


Figure 1.9: An example of a memory hierarchy.

5. Análise do nível ISA (*Instruction Set Architecture*)

O nível ISA é responsável pelo interface entre o *software* num computador – código associado a um dado programa – e o seu *hardware* – conjunto de circuitos que leva a bom termo o programa em execução. Há quem o defina como o aspecto que a máquina se apresenta a um programador em linguagem máquina. Como já não existem estes "personagens", vamos então considerar que o nível ISA aceita o código gerado por um compilador (no sentido lato, i.e., que também inclua um *assembler*). Para que um compilador produza o código adequado a cada processador, é necessário saber, entre vários outros aspectos, que tipos de dados e instruções estão disponíveis, como os operandos são manipulados, quais os registos que podem ser usados, como a memória é acedida e estruturada. È o conjunto destes aspectos que definem o nível ISA, e não o modo como os circuitos físicos são construídos. Vamos analisar alguns deles.

5.1 Operações num processador

Qualquer processador tem capacidade para efectuar **operações aritméticas e lógicas** com valores do tipo inteiro. Na especificação de uma dessas operações, a realização física dos processadores (i.e., os circuitos digitais) impõem normalmente algumas limitações: os processadores apenas efectuam operações a partir de, no máximo, 2 operandos fonte. Por outras palavras, as instruções que são utilizadas para a realização de operações no processador precisam de especificar normalmente **3 operandos**: 2 para fonte e um para destino.

- **N.º de operandos em cada instrução**

Quando o formato de instrução o permite, a especificação dos **3 operandos** para a realização de operações no processador – quer a sua localização, quer o seu valor - vem directamente na própria instrução. Assim, se a operação pretendida é "`a= b+c`", uma instrução em linguagem máquina poderá ser especificada por "`add a,b,c`", especificando `b` ou `c` uma variável ou uma constante.

A maioria das arquitecturas de processadores existentes actualmente (e normalmente designadas por arquitecturas RISC) suportam este tipo de operação com especificação explícita de 3 operandos, mas com uma restrição: todos esses operandos deverão estar em registos. O motivo para esta limitação está relacionado com a própria **filosofia RISC**: as instruções deverão ser simples e rápidas. Operandos em memória introduzem acções extras no processador que podem conduzir a atrasos significativos na execução da instrução e comprometem o seu funcionamento interno em *pipeline*.

As arquitecturas processadores de baixo custo de gerações anteriores tinham limitações para representar os 3 operandos na instrução. As mais antigas - os microprocessadores dos anos 70 - apenas especificavam **1 operando**, sendo o 2º operando fonte e o destino alocado implicitamente a um mesmo registo, designado por acumulador; a operação efectuada pelo CPU é do tipo "`Acc= Acc <op> x`". Outras, mais recentes (família Intel x86 e a sucessora Intel IA-32, e a família Motorola 68k, no início dos anos 80), especificam normalmente **2 operandos**: um deles é simultaneamente fonte e destino (i.e, `a= a+b`).

É ainda possível existirem arquitecturas que representem explicitamente **0 (zero) operandos** em operações aritméticas e lógicas. Estas arquitecturas – de que não existe actualmente nenhum exemplo comercial – consideram que os operandos, quer fonte, quer destino, se encontram sempre no topo da *stack*. Assim, uma operação nesta arquitectura não necessita de especificar nenhum operando, uma vez que o CPU já sabe implicitamente onde os ir buscar – retira-os do topo da pilha – e onde armazenar o resultado da operação – coloca-o no topo da pilha.

Localização dos operandos

- **Variáveis escalares**

Para que as instruções sejam executadas com rapidez e eficiência, os operandos deverão ser acedidos à velocidade de funcionamento do processador. Idealmente, todos os operandos - que incluem as variáveis dos programas desenvolvidos pelo utilizador - deveriam assim estar disponíveis em **registos**. Para que tal aconteça é necessário um n.º bastante elevado de registos. Contudo, com a evolução da tecnologia dos compiladores, tornou-se possível representar a maioria das variáveis escalares de qualquer programa usando apenas os **32 registos genéricos** que as arquitecturas contemporâneas baseadas em processadores RISC disponibilizam; não é este o caso da família IA-32, com um reduzido n.º de registos, e que obriga o recurso à memória para representar as variáveis escalares.

- **Variáveis estruturadas**

Na representação de variáveis estruturadas o uso de registos já se torna mais impraticável. As variáveis estruturadas são mantidas na **memória**. Dado que cada célula de memória é de 8 bits, convém não esquecer que quase todas as variáveis numéricas ocupam mais que uma célula: os inteiros nas arquitecturas de 32 bits ocupam 32 bits, enquanto os reais ocupam 32 ou 64 bits, consoante são de precisão simples ou dupla.

5.2 Formato de instruções em linguagem máquina

A codificação em linguagem máquina deverá ser compacta para que ocupem pouca memória, para que a sua transferência para o CPU seja rápida e para que a sua descodificação se efectue também com rapidez.

- **Comprimento das instruções**

Quando o custo das memórias e dos processadores era considerável, os *instruction sets* dos processadores eram compactados para pequenas dimensões (**8 bits**), à custa de compromissos com o n.º de operandos a incluir e com a utilização de instruções de **comprimento variável** (com consequências nefastas para um funcionamento em *pipeline*). Processadores projectados nos anos 70 e inícios de 80, considerados de processadores de 16-bits, centraram neste valor (**16 bits**) a dimensão mínima e/ou básica para o formato de instrução, mas incluindo sempre a possibilidade de conterem extensões de várias palavras extra.

As arquitecturas RISC tiveram como um dos seus objectivos a definição de formatos de instrução de **comprimento fixo**, e de dimensão tal que permitisse especificar os 3 operandos: **32 bits**.

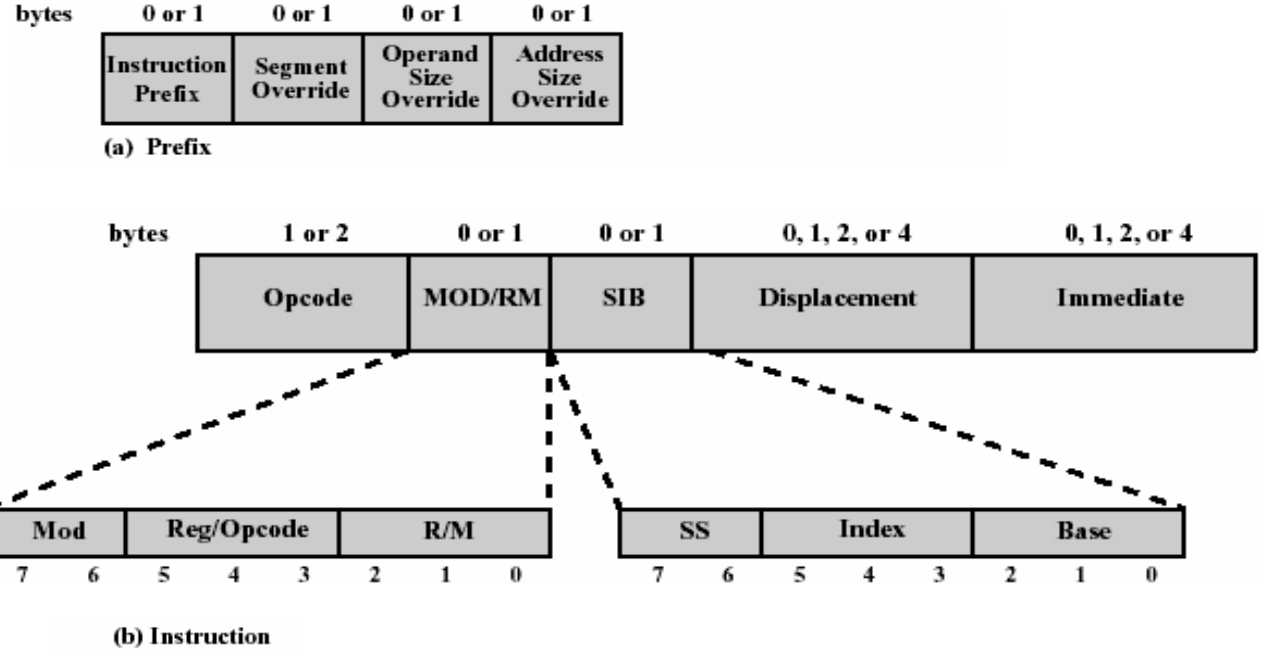
- **Campos duma instrução**

No formato de cada instrução em linguagem máquina é sempre possível identificar um conjunto de campos com informação bem definida: um campo que caracterize a operação a efectuar (normalmente designado por ***opcode***) e tantos campos quantos o n.º de operandos que for possível especificar.

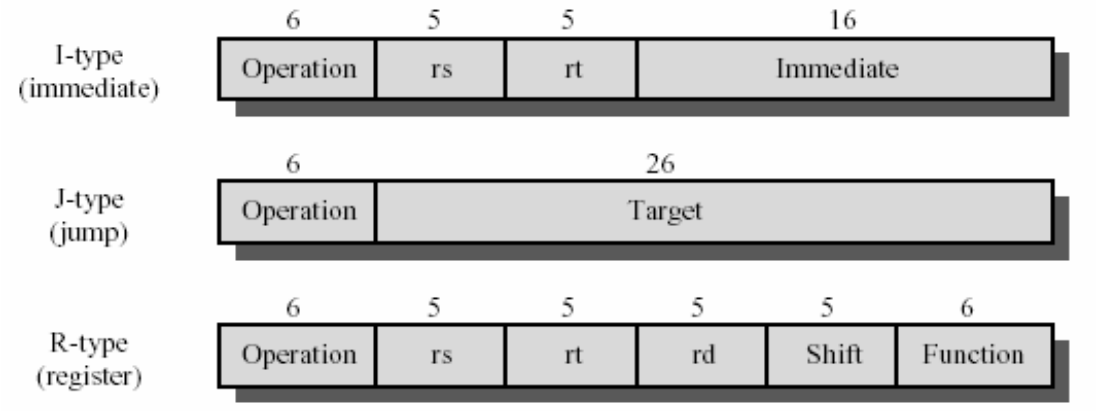
Uma análise mais detalhada dum processador RISC permite verificar como esta estrutura é homogeneamente seguida por esses fabricantes. Contudo, a organização das instruções na família IA-32 é bem mais complexa.

- **Exemplos de formatos de instrução**

As figuras que a seguir se apresentam ilustram os 2 tipos de instruções acima referidos: um derivado duma arquitectura com origem nos anos 70 (Pentium II, baseado no 8086/8), e duma arquitectura RISC (MIPS).



Formatos de instruções do Pentium



Operation	Operation code
rs	Source register specifier
rt	Source/destination register specifier
Immediate	Immediate, branch, or address displacement
Target	Jump target address
rd	Destination register specifier
Shift	Shift amount
Function	ALU/shift function specifier

Formatos de instruções do MIPS

5.3 Tipos de instruções presentes num processador

O conjunto de instruções presentes no *instruction set* de cada processador (ou família de processadores) é bastante variado. Contudo é possível identificar e caracterizar grupos de instruções que se encontram presentes em qualquer arquitectura:

- **operações aritméticas, lógicas, ...**: soma, subtracção e multiplicação com inteiros e fp, operações lógicas AND, OR, NOT, e operações de deslocamento de bits para a esquerda/direita, são as mais comuns; alguns processadores suportam ainda a divisão, quer directamente por *hardware*, quer por microprogramação;
- **para transferência de informação**: integram este grupo as instruções que transferem informação entre registos, entre registos e a memória (normalmente designadas por instruções de *load/store*), directamente entre posições de memória (suportado por ex. no M680x0, mas não disponível no IA-32 nem em qualquer arquitectura RISC), ou entre registos e a *stack*, com incremento/decremento automático do *sp* (disponível em qualquer arquitectura CISC, mas incomum em arquitecturas RISC);
- **para controlo de fluxo**: a execução de qualquer programa pressupõe normalmente a tomada de decisões quanto à próxima instrução a executar, e nas linguagens HLL existem para tal as estruturas de controlo, as quais incluem essencialmente o "`if...then...else`", os ciclos e as invocações de funções e procedimentos; as instruções em linguagem máquina que suportam estas estruturas executam testes que poderão ser seguidos de saltos condicionais ou incondicionais para uma determinada localização de memória, onde se encontre o bloco de instruções para executar; no caso das invocações, os processadores suportam ainda a salvaguarda automática do endereço de retorno.

5.4 Registos visíveis ao programador

Um dos aspectos essenciais para se programar em linguagem máquina (ou para se ler e compreender um programa escrito nesse nível) é saber de quantos registos dispõe, qual a sua dimensão, e em que circunstâncias esses registos podem ser usados.

- **Em arquitecturas RISC**

No caso das arquitecturas RISC, a grande maioria dos processadores possui 32 registos genéricos de 32 bits - para representar valores inteiros e/ou endereços de memória - para além de 32 registos de 32 bits para representação de valores em vírgula flutuante. Estes registos são considerados de **uso genérico** e podem ser usados explicitamente por qualquer instrução que aceda a operandos em registos, com excepção de um registo que contém o valor zero e que não pode ser alterado (só de leitura). Ainda visível ao programador está sempre também o apontador para a próxima instrução, o *instruction pointer* ou *program counter*.

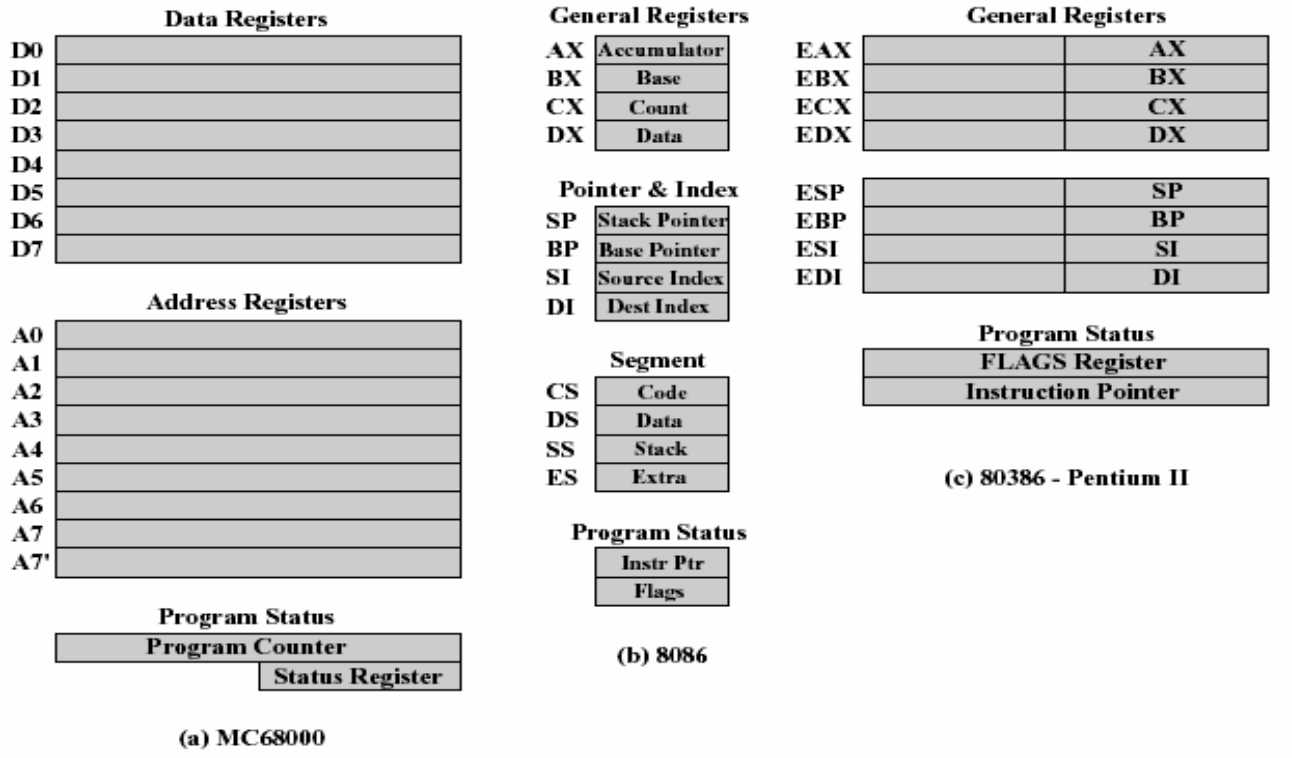
- **Em arquitecturas CISC (M680x0, Ix86/IA-32)**

A existência de um grande n.º de registos nas arquitecturas RISC, aliado à evolução da tecnologia dos compiladores dos últimos anos (em especial na geração de código), vem permitindo representar a maioria das variáveis escalares directamente em registo, não havendo necessidade de recorrer com tanta frequência à memória. Esta organização não foi contudo economicamente viável nas gerações anteriores de microprocessadores, com destaque para a família da Motorola (M680x0) e, ainda mais antiga, a família da Intel (ix86). Estes processadores dispunham de um menor n.º de registos e, consequentemente, uma diferente organização que suportasse eficientemente diversos mecanismos de acesso à memória.

No caso da família M680x0, o programador tinha disponível dois bancos de 8 registos genéricos de 32 bits: um para dados (D) e outro para apontadores para a memória (A), suportando este último banco um variado leque de modos de endereçamento à memória. Apenas um dos registos (A7) é usado implicitamente em certas operações de manuseamento da *stack*..

A família da Intel é mais complicada, por não ter verdadeiramente registos de uso genérico. A arquitectura de base dispõe efectivamente de 4 registos para conter operandos aritméticos (A, B, C e D), mais 4 para trabalhar com apontadores para a memória (BP, SP, DI e SI) e outros 4 para lidar com uma memória segmentada (CS, DS, SS e ES; a única maneira de uma arquitectura de 16 bits poder aceder a mais de 64k células de memória). Cada um destes registos não pode ser considerado de uso genérico, pois quase todos eles são usados implicitamente (isto é, sem o programador explicitar o seu uso) em várias instruções (por ex., os registos A e D funcionam de acumuladores em operações de multiplicação e divisão, enquanto o registo C é usado implicitamente como variável de contagem em instruções de controlo de ciclos). A situação complica-se ainda mais com a variação da dimensão dos registos na mesma família (de registos de 16 bits no i286 para registos de 32 bits no i386 e sucessores, normalmente designados por IA-32), pois o formato de instrução foi concebido para distinguir apenas operandos de 8 e de 16 bits, e um bit bastava; para garantir compatibilidade ao longo de toda a arquitectura, os novos processadores têm de distinguir operandos de 8, 16 e 32 bits, usando o mesmo formato de instrução!

A figura seguinte (de COA) ilustra a organização dos registos das arquitecturas CISC referidas:



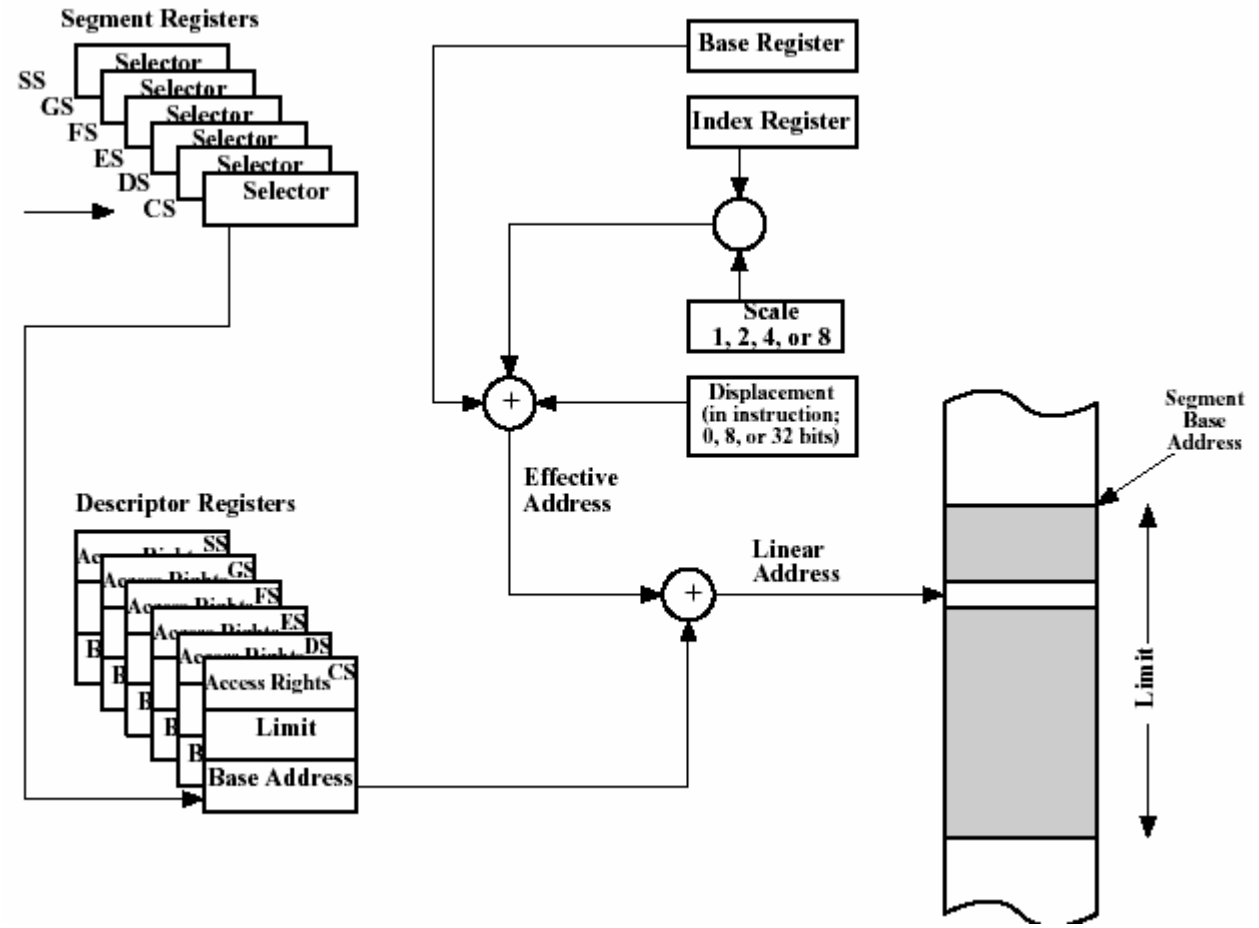
5.5 Modos de acesso aos operandos

Os operandos em *assembly* estão armazenados algures dentro do computador: ou em registos, ou na memória. A escolha de um destes modos de acesso vem indicada explicitamente no formato de instrução da operação que se pretende que o CPU efectue.

Nas arquitecturas RISC as operações aritméticas/lógicas impõem que os operandos estejam em registos no CPU: ou em **registos explicitados pelo programador** (um dos 32 registos genéricos, ou ainda de fp), ou no **registo de instrução** (fazendo parte do formato de instrução, também designado por modo de acesso imediato).

O acesso à memória nas arquitecturas RISC faz-se normalmente através de operações de *load* ou de *store*. Nestas operações, as arquitecturas RISC normalmente especificam um endereço de memória (para uma instrução de *load* ou *store*) duma única maneira: **um valor numérico de 16 bits e um registo**; a posição de memória que se acede vem dada pela soma desse valor com o conteúdo do registo especificado. Algumas arquitecturas RISC suportam ainda, alternativamente, o cálculo do endereço através da soma do conteúdo de 2 registos.

As arquitecturas CISC, como dispõem de um leque de registos mais reduzidos, necessitam de aceder mais frequentemente à memória para ir buscar/armazenar os conteúdos das variáveis. Assim, estas arquitecturas oferecem um leque mais variado de cálculo do endereço da posição de memória que se pretende aceder. A figura seguinte mostra, a título de exemplo, as opções disponibilizadas pelo Pentium II para esse cálculo:



Ordenação de *bytes* numa palavra

A dimensão de cada célula de memória (8 bits) não é suficientemente grande para armazenar integralmente o conteúdo dum registo (32 bits). Quando se pretende armazenar o conteúdo dum registo na memória, a partir de um dado endereço, duas alternativas básicas se colocam em relação à ordem em que os blocos de

8 bits da palavra são colocados na memória:

- coloca-se no 1º endereço da memória a extremidade da palavra com os 8 bits mais significativos (do lado esquerdo), seguidos dos restantes *bytes*; a esta alternativa de se privilegiar a extremidade mais significativa, designa-se por ***big-endian*** ; exemplos de arquitecturas que seguem esta alternativa: M680x0, Sun SPARC, e a maioria das arquitecturas RISC;
- coloca-se no 1º endereço da memória a extremidade da palavra com os 8 bits menos significativos (do lado direito), seguidos dos restantes *bytes*; a esta alternativa de se privilegiar a extremidade menos significativa, designa-se por ***little-endian***; exemplos de arquitecturas que seguem esta alternativa: Ix86, DEC VAX.

Algumas arquitecturas RISC suportam ambas, mas apenas uma de cada vez que é feito o *reset* do CPU.

5.6 Instruções de *input/output*

O acesso aos periféricos dum computador é feito normalmente pelos seus controladores, ligados aos barramentos do computador. Estes controladores desempenham quase todas as tarefas indispensáveis ao bom funcionamento dos periféricos; apenas necessitam de ser devidamente configurados inicialmente, posteriormente activados com comandos específicos, e estabelecer uma comunicação com a informação a transferir de/para o computador. Assim, o CPU apenas precisa de aceder a esses controladores para 3 tipos de acções, como referido atrás:

- escrita de comandos de configuração e de activação de tarefas específicas; em registo(s) de controlo;
- leitura do estado do controlador após execução das tarefas solicitadas; em registo(s) de estado;
- escrita/leitura de informação para ser comunicada com o exterior; em registo(s) de dados.

Para o desempenho destas actividades, a maioria dos processadores não dispõe de instruções específicas de *input/output*; basta apenas usar as instruções normais de acesso à memória para efectuar a leitura ou escrita dos registos dos controladores. O descodificador de endereços do computador se encarregará de gerar os sinais apropriados de *chip select* para seleccionar o controlador necessário, de acordo com o mapa de alocação da memória que o projectista do computador definiu. A este modelo de mapeamento do espaço endereçável de *input/output* no espaço endereçável de memória é designado por ***memory mapped I/O***. Todas as arquitecturas RISC seguem este modelo.

Outros processadores, como os da família Ix86, dispõem adicionalmente de instruções específicas de I/O, que implementam essencialmente uma de 2 operações de acesso a um registo dum controlador (também chamado de uma porta): operação de leitura duma porta - *in* - ou de escrita numa porta - *out*..

5.7 Caracterização das arquitecturas RISC (resumo)

Ao longo destes capítulos fizeram-se várias referências ao modelo de arquitectura RISC em oposição ao modelo que existia na década de 70 (designado por CISC[18], em oposição ao RISC) e que teve continuidade na linha dos processadores x86 da Intel, por questões de compatibilidade com arquitecturas anteriores.

As principais características que identificam uma arquitectura RISC podem ser resumidas a:

- **conjunto reduzido e simples de instruções**: a tarefa do compilador na geração de código é tanto mais simplificada quanto menor for o número de opções que tiver de escolher; assim um *instruction set* menor não só facilita a tarefa do compilador, como ainda permite a sua execução mais rápida no *hardware*, e instruções simples permitem maior rapidez de execução; foi provado que o desempenho de um processador pode também ser melhorado se se escolherem bem as instruções mais utilizadas e as optimizarem, em vez de se tentar que todas as instruções em HLL's estejam presentes em linguagem máquina; foram estas as principais motivações no desenvolvimento deste modelo alternativo;
- **uma operação por ciclo máquina**: considera-se um ciclo máquina como sendo o tempo necessário a aceder a um par de operandos em registos, efectuar uma operação na ALU e armazenar o resultado

[18] ***C**omplex **I**nstruction **S**et **C**omputer*

num registo; dentro deste espírito, uma operação aritmética/lógica com os operandos/resultado nessa condições cumpre esta especificação, assim como uma operação de comparação de operandos em registos e consequente acção de alteração do *instruction pointer* (operação de salto), ou operação de leitura ou escrita de/em memória, com uma especificação do endereço de memória que não vá para além de uma operação aritmética com operandos em registos (uma instrução de *push* ou *pop* não satisfaz esta condição, pois inclui a operação adicional de soma ou subtracção do conteúdo de um outro registo, o *stack pointer*); principal objectivo desta característica: permitir uma execução encadeada de instruções eficiente (*pipeline*);

- **operandos sempre em registos**: para satisfazer a característica enunciada antes, pois se houver operandos ou resultado em memória, a instrução será um conjunto de mais que uma operação elementar; assim, acessos à memória são apenas efectuados com instruções explícitas de *load* e *store*;
- **modos simples de endereçamento à memória**: por idêntico motivo; certas arquitecturas CISC têm modos de endereçamento tão complexos, que é necessário efectuar vários acesos à memória e somas/subtracções de endereços, que tornam qualquer mecanismo de encadeamento de instruções altamente ineficiente;
- **formatos simples de instruções**: as arquitecturas RISC têm normalmente um único tamanho de instrução (comprimento fixo), o que facilita o encadeamento de instruções pois sabe-se *a priori* a sua dimensão; nas arquitecturas CISC só se sabe o comprimento efectivo da instrução depois de se descodificar parcialmente a instrução (e por vezes em mais que um passo!).

Anexo A: Sistemas de numeração e representação de inteiros

A.1 Sistemas de numeração

Os números podem ser representados em qualquer sistema de numeração. Os seres humanos usam normalmente um sistema de numeração baseado na base 10 (com 10 dígitos diferentes). Os computadores, pelo facto de só representarem dois valores (0, 1), os dígitos binários - também conhecidos por **bits**, da contracção do inglês *binary digit* - são máquinas binárias, e por isso trabalham em base 2.

Para compreender o que significa a base em que os números são representados num dado sistema de numeração, é necessário relembrar o significado da ordem dos dígitos.
A **ordem** de um dígito dentro de um número é dada pela posição que esse dígito ocupa no número: 0 é a ordem do dígito imediatamente à esquerda da ponto (vírgula) decimal, crescendo no sentido da esquerda, e decrescendo no sentido da direita.

Exemplo

1532.64_{10}

Dígito 4 - ordem -2
Dígito 6 - ordem -1
Dígito 2 - **ordem 0**
Dígito 3 - ordem +1
Dígito 5 - ordem +2
Dígito 1 - ordem +3

A **base** utilizada determina o número de dígitos que podem ser utilizados; por exemplo, base 10 utiliza 10 dígitos (0 a 9), base 2 utiliza 2 dígitos (0 e 1), base 5 utiliza 5 dígitos (0 a 4), base 16 utiliza 16 dígitos (0 a 9, e, A a F).

A.2 Conversão entre bases

A **conversão** de um número escrito na **base *b* para a base decimal** obtém-se multiplicando cada dígito pela base *b* elevada à ordem do dígito, e somando todos estes valores.

Exemplos

1532_6 (base 6)
$1*6^3 + 5*6^2 + 3*6^1 + 2*6^0 = 416_{10}$

1532.64_{10} (base 10)
$1*10^3 + 5*10^2 + 3*10^1 + 2*10^0 + 6*10^{-1} + 4*10^{-2} = 1532.64_{10}$

1532_{13} (base 13)
$1*13^3 + 5*13^2 + 3*13^1 + 2*13^0 = 3083_{10}$

110110.011_2 (base 2)
$1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2} + 1*2^{-3} = 54.375_{10}$

Na **conversão** de um número na **base decimal para uma base *b***, o processo mais directo é composto por 2 partes:

- **divisão sucessiva da parte inteira** desse número pela respectiva base, sendo os restos obtidos com cada uma dessas divisões, os dígitos da base *b* (a começar com o menos significativo, i.e., mais junto ao ponto decimal) e os quocientes a usar na sucessão de divisões;
- **multiplicação sucessiva da parte fraccionária** desse número pela respectiva base, sendo a parte inteira de cada um dos produtos obtidos, os dígitos da base *b* (a começar com o mais significativo, i.e., mais junto ao ponto decimal), e a parte decimal a usar na sucessão de multiplicações.

Exemplo

235.375_{10}

235/2 = 117	Resto = 1	/* *bit* menos significativo int*/
117/2 = 58	Resto = 1	
58/2 = 29	Resto = 0	
29/2 = 14	Resto = 1	
14/2 = 7	Resto = 0	
7/2 = 3	Resto = 1	
3/2 = 1	Resto = 1	/* *bit* mais significativo int*/
0.375*2 = 0.750	P. int. = 0	/* *bit* mais significativo frac*/
0.75*2 = 1.5	P. int. = 1	
0.5*2 = 1.0	P. int. = 1	/* *bit* menos significativo frac*/

$235.625_{10} = 11101011.011_2$

Outro processo de converter de uma base decimal para outra base utiliza **subtracções sucessivas**, mas apenas é utilizado na **conversão para a base binária**, e mesmo nesta para valores que não ultrapassam a ordem de grandeza dos milhares e normalmente apenas para inteiros.
A grande vantagem deste método é a sua rapidez de cálculo mental, sem ajuda de qualquer máquina de calcular, desde que se saiba de cor a **"tabuada" das potências de 2**:

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9
1	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**

Ajuda também saber como se comportam as potências de 2 para expoentes com mais que um dígito. Sabendo que 2^{10} = 1024 (= **1K**, ~= 10^3), e que $2^{1x} = 2^x * 2^{10} = 2^x * 1K$, é possível a partir daqui extrapolar não apenas todos os restantes valores entre 2^{10} e 2^{19}, como ainda ter uma noção da **ordem de grandeza de um valor binário com qualquer número de dígitos** (bits):

Exemplos

Tabela de potências de 2^{10} a 2^{19}

2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}
1K	**2K**	**4K**	**8K**	**16K**	**32K**	**64K**	**128K**	**256K**	**512K**

Tabela de potências de 2 com expoentes variando de 10 em 10

2^{10}	2^{20}	2^{30}	2^{40}	2^{50}	2^{60}	2^{70}	2^{80}	2^{90}	2^{x0}
1K	**1M**	**1G**	**1T**	**1P**	**1?**				
~10^3	**~10^6**	**~10^9**	**~10^{12}**	**~10^{15}**	**~10^{18}**	**~10^{21}**	**~10^{24}**	**~10^{27}**	**~10^{3*x}**

Com base nesta informação, é agora possível pôr em prática o método das subtracções sucessivas para converter um nº decimal num binário: procura-se a maior potência de 2 imediatamente inferior ao valor do nº decimal, e subtrai-se essa potência do nº decimal; o expoente da potência indica que o nº binário terá um bit 1 nessa ordem; com o resultado da subtracção repete-se o processo até chegar ao resultado 0.

Exemplo

1081.625_{10}

$1081.625 - 2^{10} = 57.625$
$57.625 - 2^{5} = 24.625$
$25.625 - 2^{4} = 9.625$
$9.625 - 2^{3} = 1.625$
$1.625 - 2^{0} = 0.625$
$0.625 - 2^{-1} = 0.125$
$0.125 - 2^{-3} = 0$

$1\,0\,0\,0\,0\,1\,1\,1\,0\,0\,1.\,1\,0\,1_2$
10 9 8 7 6 5 4 3 2 1 0 -1 -2 –3 **< ordem**

Os processadores utilizam um determinado número de bits para representar um número. A quantidade de bits utilizados determina a gama de valores representáveis. Tal como qualquer outro sistema de numeração - onde a gama de valores representáveis com ***n*** dígitos é b^n - a mesma lógica aplica-se à representação de valores binários.
Sendo ***n*** o número de bits utilizados, a **gama de valores representáveis em binário, usando *n* bits é 2^n**.

A.3 Base hexadecimal

O sistema de numeração de **base hexadecimal** (16) é frequentemente utilizada como forma alternativa de representação de valores binários, não apenas pela facilidade de conversão entre estas 2 bases, como ainda pela menor probabilidade de erro humano na leitura/escrita de números.

Tal como referido anteriormente, são utilizados 16 dígitos: 0, 1, ..., 9, A, B, C, D, E, F.

Exemplos

431210 em hexadecimal

4312 / 16 = 269	Resto = 8
269 / 16 = 16	Resto = 13 (dígito D)
16 / 16 = 1	Resto = 0
1 / 16 = 0	Resto = 1

Logo, 431210 = 10D816

2AF316 em decimal

2 * 163 + 10 * 162 + 15 * 161 + 3 * 160 = 1099510

A motivação para usar hexadecimal é a facilidade com que se converte entre esta base e binário. Cada dígito hexadecimal representa um valor entre 0 e 15; cada conjunto de 4 bits representa também um valor no mesmo intervalo. Pode-se então aproveitar esta característica nos 2 tipos de conversão: de binário, agrupando os bits de 4 em 4 a partir do ponto decimal, e convertendo-os; para binário, convertendo cada dígito hexadecimal em 4 bits.

Exemplos

```
      2    A    F  (hexadecimal)
    0010 1010 1111 (binário)
```

2AF$_{16}$ = 001010101111$_2$

```
    1101 0101 1011 (binário)
      D    5    B  (hexadecimal)
```

110101011011$_2$ = D5B$_{16}$ (também comum representar como 0xD5B, ou ainda 0xd5b)

A.4 Números negativos

Os computadores lidam com números positivos e números negativos, sendo necessário encontrar uma representação para números com sinal negativo. Existe uma grande variedade de opções, das quais apenas se destacam 4, sendo apenas 3 as actualmente usadas para representar valores negativos:

- **sinal e amplitude/magnitude** (S+M)
- **complemento para 1**
- **complemento para 2**
- **notação em excesso** (ou *biased*)

Como o próprio nome indica, a representação **sinal e amplitude** utiliza um bit para representar o sinal, o bit mais à esquerda: **0** para indicar um valor positivo, **1** para indicar um valor negativo.

Na representação em **complemento para 1** invertem-se todos os bits de um número para representar o seu complementar: assim se converte um valor positivo para um negativo, e vice-versa. Quando o bit mais à esquerda é **0**, esse valor é positivo; se for **1**, então é negativo.

Exemplo

100$_{10}$ = 01100100$_2$ (com 8 bits)

Invertendo todos os bits:
10011011$_2$ = **–100$_{10}$**

O problema desta representação é que existem 2 padrões de bits para o **0**. Nomeadamente 0_{10} = 00000000_2 = 11111111_2. A solução encontrada consiste em representar os números em **complemento para 2** . Para determinar o negativo de um número negam-se todos os seus bits e soma-se uma unidade.

Exemplo

100$_{10}$ = 01100100$_2$ (com 8 bits)

Invertendo todos os bits:
10011011$_2$

Somando uma unidade :
10011011$_2$ + 1 = 10011100$_2$ = **–100$_{10}$**

A representação em complemento para 2 tem as seguintes características:

- o bit da esquerda indica o sinal;
- o processo indicado no parágrafo anterior serve para converter um número de positivo para negativo e de negativo para positivo;
- o 0 tem uma representação única: todos os bits a 0;
- a gama de valores que é possível representar com *n* bits é -2^{n-1} ... $2^{n-1}-1$.

Exemplo

Qual o número representado por $\mathbf{11100100_2}$ (com 8 bits)?

Como o bit da esquerda é 1 este número é negativo.
Invertendo todos os bits:
00011011_2
Somando uma unidade :
$00011011_2 + 1 = 00011100_2 = 28_{10}$
Logo:
$11100100_2 = \mathbf{-28_{10}}$

Como é que se converte um número representado em complemento para 2 com *n* bits, para um número representado com mais bits?
Resposta: basta fazer a extensão do sinal! Se o número é positivo acrescenta-se 0's à esquerda, se o número é negativo acrescenta-se 1's à esquerda.

Exemplo

Representar os seguintes números (de 8 bits) com 16 bits:

$\mathbf{01101010_2}$

Positivo, logo:
$00000000\ 01101010_2$

$\mathbf{11011110_2}$

Negativo, logo:
$11111111\ 11011110_2$

A multiplicação e a divisão têm algoritmos algo complexos que ultrapassam o âmbito destas notas de estudo. No entanto a **multiplicação e a divisão por potências de 2** realizam-se efectuando deslocamentos de bits à direita ou à esquerda, respectivamente.

Fazer o deslocamento à esquerda uma vez - num número binário - corresponde a multiplicar por 2, duas vezes corresponde a multiplicar por 4 (= 2^2), 3 vezes corresponde a multiplicar por 8 (= 2^3), e assim sucessivamente. O mesmo se aplica à divisão com o deslocamento à direita.

Exemplo

Dividir 11001010_2 (= 202_{10}) por 4:

Deslocar à direita 2 vezes:
$000110010.1_2 = 50.5_{10}$

Multiplicar 000001010_2 (= 10_{10}) por 8.

Deslocar à esquerda 3 vezes:
$001010000_2 = 80_{10}$

Esta regra também se aplica aos números em complemento para 2 desde que se mantenha o sinal.

A última notação referida no início - **notação em excesso** – tem uma vantagem sobre qualquer outra referida anteriormente: a representação numérica dos valores em binário, quer digam respeito a valores com ou sem sinal, tem o mesmo comportamento na relação entre eles. Por outras palavras, o valor em binário com todos os bits a 0 representa o menor valor inteiro, quer este tenha sinal ou não, e o mesmo se aplica ao maior valor em binário, i.e., com todos os bits a 1: representa o maior inteiro, com ou sem sinal.

Exemplo

Binário (8 bits)	**Sinal + Ampl**	**Compl p/ 1**	**Compl p/ 2**	**Excesso** (128)
$0000\ 00001_2$	+1	+1	+1	-127
...				
$1000\ 00001_2$	-1	-126	-127	+1
...				
$1111\ 11110_2$	-126	-1	-2	+126

Como o próprio nome sugere, esta codificação de um inteiro (negativo ou positivo) em binário com *n* bits é feita sempre em excesso (de 2^{n-1} ou 2^{n-1}-1). Neste exemplo com 8 bits, o valor $+1_{10}$ é representado em binário, em notação por excesso de 2^{n-1}, pelo valor $(+1_{10}$ + excesso) = $(+1_{10} + 128_{10})$ = $(0000\ 00001_2 + 1000\ 00000_2) = 1000\ 00001_2$.

A tabela que a seguir se apresenta, representando todas as combinações possíveis com 4 bits, ilustra de modo mais completo as diferenças entre estes 4 modos (+1 variante) de representar inteiros com sinal.

Binário (4 bits)	**Sinal + Ampl**	**Compl p/ 1**	**Compl p/ 2**	**Excesso** (7)	**Excesso** (8)
0000	0	0	0	-7	-8
0001	1	1	1	-6	-7
0010	2	2	2	-5	-6
0011	3	3	3	-4	-5
0100	4	4	4	-3	-4
0101	5	5	5	-2	-3
0110	6	6	6	-1	-2
0111	7	7	7	0	-1
1000	-0	-7	-8	1	0
1001	-1	-6	-7	2	1
1010	-2	-5	-6	3	2
1011	-3	-4	-5	4	3
1100	-4	-3	-4	5	4
1101	-5	-2	-3	6	5
1110	-6	-1	-2	7	6
1111	-7	-0	-1	8	7

Anexo B: Representação de reais em vírgula flutuante

B.1 Notação científica

A representação de um valor infinito de valores usando uma máquina finita vai obrigar a assumir um conjunto de compromissos, os quais, no caso dos reais, irão afectar não só a gama de valores representáveis, como ainda a sua precisão. A utilização da notação científica, do tipo:

$$\textbf{Valor} = (-1)^{S} * \textbf{Mantissa} * \textbf{Radix}^{Exp}$$

é ainda aquela que permite obter a melhor representação de um n.º real em vírgula flutuante (ou *fp* na terminologia inglesa) com um n.º limitado de dígitos. O valor do radix é de 10 na representação decimal, e pode ser 2 ou uma potência de 2 na representação interna num computador. A IBM usava nos seus *mainframes* um radix de 16, pois permitia-lhe aumentar o intervalo de representação de valores; contudo os problemas que tiveram com esta representação deram mais força à utilização do valor 2 como radix.

B.2 Normalização na representação

A notação científica permite que um mesmo n.º possa ser representado de várias maneiras com os mesmos dígitos (por ex., 43.789E+12 , .43789E14, 43789E+09). Para facilitar a sua representação - omitindo a necessidade de representar o ponto/vírgula decimal - impõe-se a adopção de uma norma de representação, e diz-se que um dado n.º fp está normalizado quando cumpre essa norma. Alguns autores consideram que um n.º está **normalizado** quando a mantissa (ou parte fraccionária, **F**) se encontra no intervalo **]Radix , 1]**. Por outras palavras, existe sempre um dígito diferente de 0 à esquerda do ponto decimal.

Num exemplo em decimal com 7 algarismos na representação de fp (5 para a mantissa e 2 para o expoente), o intervalo de representação dum fp normalizado, seria em valor absoluto [1.0000E-99, 9.9999E+99] . Existe aqui um certo desperdício na representação de fp usando 7 algarismos, pois fica excluído todo o intervalo [0.0001E-99, 1.0000E-99[. Para se poder optimizar a utilização dos dígitos na representação de fp, aceitando a representação de valores menores que o menor valor normalizado, mas com o menor valor possível do expoente, se designa esta representação de **desnormalizada**.

Todas as restantes representações designam-se por **não normalizadas**.

B.3 Intervalo e precisão de valores representáveis

Pretende-se sempre com qualquer codificação obter o maior intervalo de representação possível e simultaneamente a melhor precisão (relacionada com a distância entre 2 valores consecutivos). Existindo um n.º limitado de dígitos para a representação de ambos os valores - **F** e **Exp** - há que ter consciência das consequências de se aumentarem ou diminuírem cada um deles.

O intervalo de valores representáveis depende essencialmente do Exp, enquanto a precisão vai depender do número de dígitos que for alocado para a parte fraccionária. Numa representação em binário, a dimensão mínima a usar para fp (que será sempre um múltiplo da dimensão da célula) deverá ser pelo

menos 32. Se fosse 16, 1 bit seria para o sinal, e os restantes 15 seriam insuficientes mesmo para representar apenas a parte fraccionária (daria uma precisão de 1 em cerca de 32 000...).

Usando 32 bits para representação mínima de fp, torna-se necessário encontrar um valor equilibrado para a parte fraccionária e para o expoente. Esse valor é 8 para o expoente - permite representar uma gama da ordem de grandeza dos 1040 - e pelo menos 23 para a parte fraccionária - permite uma precisão equivalente a 7 algarismos decimais.

B.4 Formato binário dum valor em fp

Existem 3 campos a representar nos 32 bits dum valor em fp: o sinal (1 bit), a parte fraccionária (23 bits) e o expoente (8 bits). Para se efectuar qualquer operação aritmética estes 3 campos terão de ser identificados e separados para terem um tratamento distinto na unidade que processa os valores em fp. A ordem da sua representação (da esquerda para a direita) segue uma lógica:

- **sinal, S**: ficando mais à esquerda, permite usar o mesmo *hardware* (que trabalha com valores inteiros) para testar o sinal de um valor em fp;
- **expoente, E**: ficando logo a seguir vai permitir fazer comparações quanto à grandeza relativa entre valores absolutos em fp, sem necessidade de separar os 3 campos: basta comparar os valores como se de valores meramente binários se tratassem;
- **parte fraccionária, F**: é o campo mais à direita.

B.5 O bit escondido

Um valor normalizado tem sempre um dígito diferente de zero à esquerda do ponto decimal. Se o sistema de numeração é decimal, esse dígito pode ser um de entre 9 possíveis; se o sistema de numeração é binário, esse dígito só pode ser um. Assim, **e apenas na representação binária**, esse dígito à esquerda do ponto decimal toma sempre o mesmo valor, e é um desperdício do espaço de memória estar a representá-lo fisicamente. Ele apenas se torna necessário para efectuar as operações, permanecendo **escondido** durante a sua representação. Ganha-se um bit para melhorar a precisão, permitindo passar para 24 o n.º de bits da parte fraccionária (numa representação com 32 bits).

B.6 A norma IEEE 754 para valores em fp

A representação de valores em fp usando 32 bits e com o formato definido anteriormente permite ainda várias combinações para representar o mesmo valor. Por outro lado, não ficou ainda definido como representar os valores desnormalizados, bem como a representação de valores externos ao intervalo permitido com a notação normalizada.

A norma IEEE 754 define com clareza estas imprecisões, permitindo uma maior compatibilidade ao nível dos dados no porte de aplicações entre sistemas que adoptem a mesma norma. De momento todos os microprocessadores disponíveis comercialmente com unidades de fp suportam a norma IEEE 754 no que diz respeito aos valores de 32 bits. Aspectos relevantes na norma IEEE 754:

- **representação do sinal e parte fraccionária**: segue o formato definido anteriormente, sendo a parte fraccionária representada sempre em valor absoluto, e considerando o bit escondido na representação normalizada;
- **representação do expoente**: para permitir a comparação de valores em fp sem separação dos campos, a codificação do expoente deveria ser tal que os valores menores de expoente (os negativos) tivessem uma representação binária menor que os valores positivos (e maiores); as codificações usando complemento para 1 ou 2, ou ainda a representação usando sinal+amplitude, não possuem este comportamento, i.e., os valores negativos têm o bit mais significativo (à esquerda) igual a 1, o que os torna, como números binários, maiores que os números positivos; a notação que satisfaz este requisito é uma notação por excesso, na qual se faz um deslocamento na gama de valores decimais correspondentes ao intervalo de representação de n bits, de 0 a 2^(n-1),

de modo a que o 0 decimal passe a ser representado não por uma representação binária com tudo a zero, mas por um valor no meio da tabela; usando 8 bits por exemplo, esta notação permitiria representar o 0 pelo valor 127 ou 128; a norma IEEE adoptou o primeiro destes 2 valores, pelo que a representação do expoente se faz por notação **por excesso 127**; o expoente varia assim entre -127 e +128;

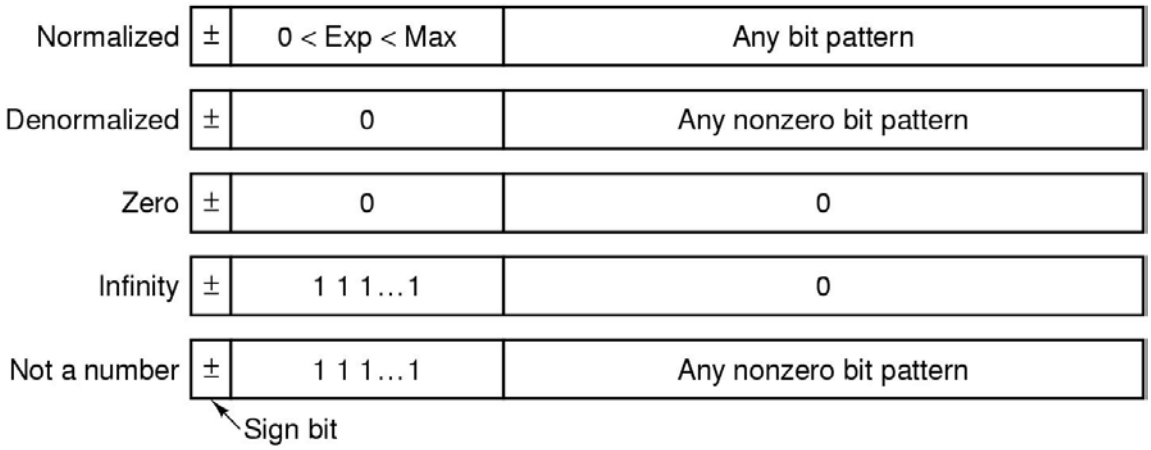
- **valor decimal de um fp em binário** (normalizado)**: V = (-1)^S * (1.F) * 2^E-127** , em que S, F e E representam respectivamente os valores em binário dos campos no formato em fp;

- **representação de valores desnormalizados**: para poder contemplar este tipo de situação a norma IEEE reserva o valor de **E = 0000 0000b** para representar valores desnormalizados, desde que se verifique também que **F diferente de 0**; o valor decimal vem dado por **V = (-1)^S * (0.F) * 2^(-126)**

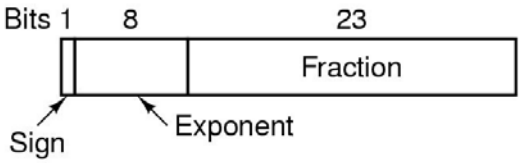
- **representação do zero**: é o caso particular previsto em cima, onde **E = 0** e **F = 0** ;

- **representação de (± infinito)**: a norma IEEE reserva a outra extremidade de representação do expoente; quando **E = 1111 1111b** e **F = 0** , são esses os "valores" que se pretendem representar;

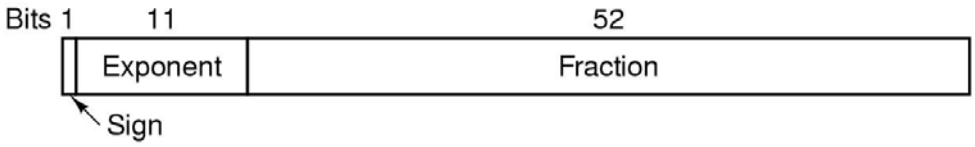
- **representação de n.º não real**: quando o valor que se pretende representar não é um n.º real (imaginário por exemplo), a norma prevê uma forma de o indicar para posterior tratamento por rotinas de excepção; neste caso **E = 1111 1111b** e **F diferente de 0** .



A norma IEEE 754 contempla ainda a representação de valores em fp que necessitem de maior intervalo de representação e/ou melhor precisão, por várias maneiras. A mais adoptada pelos fabricantes utiliza o dobro do n.º de bits, 64, pelo que é também conhecida pela representação em **precisão dupla**, enquanto a representação por 32 bits se designa por precisão simples. Para precisão dupla, a norma especifica, entre outros aspectos, que o expoente será representado por 11 bits e a parte fraccionária por 52 bits.



(a)



(b)

Representação de reais com precisão simples (a) e dupla (b)

Anexo C: Arquitectura e conjunto de instruções do IA32

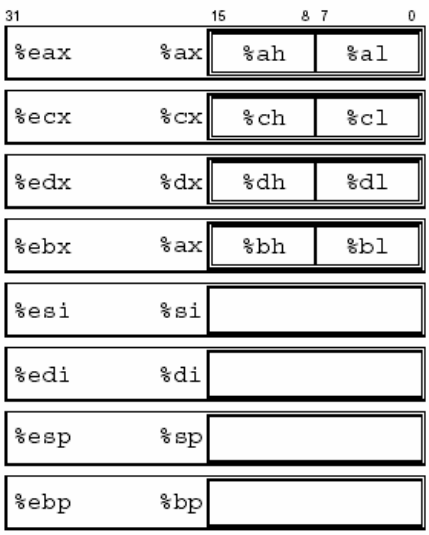


Ilustração 1- Conjunto de registos inteiros do IA32 (notação Linux)

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	E_a	$R[E_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(E_a)	$M[R[E_a]]$	Indirect
Memory	$Imm\ (E_b)$	$M[Imm + R[E_b]]$	Base + displacement
Memory	(E_b, E_i)	$M[R[E_b] + R[E_i]]$	Indexed
Memory	$Imm\ (E_b, E_i)$	$M[Imm + R[E_b] + R[E_i]]$	Indexed
Memory	$(, E_i, s)$	$M[R[E_i] \cdot s]$	Scaled indexed
Memory	$Imm\ (, E_i, s)$	$M[Imm + R[E_i] \cdot s]$	Scaled Indexed
Memory	(E_b, E_i, s)	$M[R[E_b] + R[E_i] \cdot s]$	Scaled indexed
Memory	$Imm\ (E_b, E_i, s)$	$M[Imm + R[E_b] + R[E_i] \cdot s]$	Scaled indexed

Ilustração 2 - Modos de endereçamento IA32: imediato, registo e valores em memória. O factor de escala *s* pode tomar o valor 1, 2, 4 ou 8; *Imm* pode ser uma constante de 0, 8, 16 ou 32 *bits*. O modo de endereçamento em memória reduz-se à forma *Imm*(Eb, Ei, s), em que alguns dos campos podem não estar presentes.

Tabela 1- Códigos de condições (flags) . Descrevem atributos da última operação lógica ou aritmética realizada. Usadas para realizar saltos condicionais.

Símbolo	Nome	Descrição
CF	Carry Flag	A última operação gerou transporte.
ZF	Zero Flag	A última operação teve resultado zero
SF	Sign Flag	A última operação teve resultado negativo
OF	Overflow Flag	A última operação causou overflow em complemento para dois.

Tipo	Instrução	Efeito	Descrição
Transferência de Informação	mov? S, D	D←S	Move (? = b,w,l)
	movsbl S, D	D←SignExtend(S)	Move Sign-Extended Byte
	movzbl S, D	D←ZeroExtend(S)	Move Zero-Extended Byte
	pushl S	%esp ← %esp - 4; Mem[%esp] ← S	Push
	popl D	D←Mem[%esp]; %esp ←%esp+ 4	Pop
	leal S, D	D← &S	Load Effective Address
Operações Aritméticas e Lógicas	incl D	D← D +1	Increment
	decl D	D← D –1	Decrement
	negl D	D← -D	Negate
	notl D	D← ˜D	Complement
	addl S, D	D← D + S	Add
	subl S, D	D← D – S	Subtract
	imull S, D	D← D * S	32 bit Multiply
	xorl S, D	D← D ^ S	Exclusive-Or
	orl S, D	D← D | S	Or
	andl S, D	D← D & S	And
	sall k, D	D← D << k	Left Shift
	shll k, D	D← D << k	Left Shift
	sarl k, D	D← D >> k	Arithmetic Right Shift
	shrl k, D	D← D >> k	Logical Right Shift
	imull S	%edx : %eax ← S × %eax	Signed 64 bit Multiply
	mull S	%edx : %eax ← S × %eax	Unsigned 64 bit Multiply
	cltd	%edx : %eax ← SignExtend(%eax)	Convert to Quad Word
	idivl S	%edx ← %edx : %eax mod S; %eax ← %edx:%eax ÷ S	Signed Divide
	divl S	%edx ← %edx : %eax mod S; %eax ← %edx : %eax ÷ S	Unsigned Divide
Teste	cmp? S2, S1	(CF, ZF, SF, OF) ← S1 – S2	Compare (? = b,w,l)
	test? S2, S1	(CF, ZF, SF, OF) ← S1 & S2	Test (? = b,w,l)
Instruções de set	sete R8	R8 ← ZF (Sinónimo: setz R8)	Equal/Zero
	setne R8	R8 ← ~ZF (Sinónimo: setnz R8)	Not Equal/Not Zero
	sets R8	R8 ← SF	Negative
	setns R8	R8 ← ~SF	Non Negative
	setg R8	R8 ← ~(SF^OF) & ~ZF (Sinónimo: setnle R8)	Greater (signed >)
	setge R8	R8 ← ~(SF^OF) (Sinónimo: setnl R8)	Greater or equal (signed >=)
	setl R8	R8 ← SF^OF (Sinónimo: setnge R8)	Less (signed <)
	setle R8	R8 ← (SF^OF) | ZF (Sinónimo: setng R8)	Less or equal (signed <=)
	seta R8	R8 ← ~CF & ~ZF (Sinónimo: setnbe R8)	Above (unsigned >)
	setae R8	R8 ← ~CF (Sinónimo: setnb R8)	Above or equal (unsigned >=)
	setb R8	R8 ← CF (Sinónimo: setnae R8)	Below (unsigned <)
	setbe R8	R8 ← CF & ~ZF (Sinónimo: setna R8)	Below or equal (unsigned <=)
Instruções de salto	jmp Label	%eip ← Label	Unconditional jump
	jmp *D	%eip ← *D	Indirect unconditional jump
	je Label	Jump if ZF (Sinónimo: jz)	Zero/Equal
	jne Label	Jump if ~ZF (Sinónimo: jnz)	Not Zero/Not Equal
	js Label	Jump if SF	Negative
	jns Label	Jump if ~SF	Not Negative
	jg Label	Jump if ~(SF ^OF) & ~ZF (Sinónimo: jnle)	Greater (signed >)
	jge Label	Jump if ~(SF^OF) (Sinónimo: jnl)	Greater or equal (signed >=)
	jl Label	Jump if SF^OF (Sinónimo: jnge)	Less (signed <)
	jle Label	Jump if (SF^OF) | ZF (Sinónimo: jng)	Less or equal (signed <=)
	ja Label	Jump if ~CF & ~ZF (Sinónimo: jnbe)	Above (unsigned >)
	jae Label	Jump if ~CF (Sinónimo: jnb)	Above or equal (unsigned >=)
	jb Label	Jump if CF (Sinónimo: jnae)	Below (unsigned <)
	jbe Label	Jump if CF & ~ZF (Sinónimo: jna)	Below or equal (unsigned <=)
Invocação de Procedimentos	call Label	pushl %eip; %eip= Label	Procedure call
	call *Op	pushl %eip; %eip= *Op	Procedure call
	ret	popl %eip	Procedure return
	leave	movl %ebp, %esp; pop %ebp	Prepare stack for return

D – destino [Reg | Mem] **S** – fonte [Imm | Reg | Mem] **R_8** – destino Reg 8 *bits*

D e **S** não podem ser ambos operandos em memória