



Campus de Gualtar  
4710-057 Braga



UNIVERSIDADE DO MINHO  
ESCOLA DE ENGENHARIA

Departamento de  
Informática

# Conceitos de Sistemas Informáticos

## Módulo: Arquitectura de Computadores

### *Notas de estudo*

*Alberto José Proença*

*2001/02*

### Nota introdutória

Este documento é um texto de apoio ao funcionamento do Módulo de Arquitectura de Computadores (em Conceitos de Sistemas Informáticos, 1º ano, LESI) que complementa apenas a bibliografia básica recomendada. Não pretendendo substituí-la, 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*", 5th Ed., W. Stallings, Prentice Hall, 2000 (mais informação em <http://www.williamstallings.com/COA5e.html>)
- "*Computer Systems: A Programmer's Perspective*", Randal Bryant and David O'Hallaron, Prentice Hall, 2002 (apenas em versão Beta, acessível através da página da disciplina)

## Índice

<b>1. Introdução à organização e arquitectura dum computador</b>	<b>3</b>
1.1. Estrutura interna dum computador.....	3
1.2. A origem do computador "moderno" .....	5
1.3. Funcionamento básico dum computador .....	6
1.4. A hierarquia duma arquitectura de barramentos .....	6
1.5. Os módulos de I/O .....	8
1.6. Níveis de abstracção num computador e mecanismos de conversão.....	8
<b>Anexo A: Representação de reais em vírgula flutuante.....</b>	<b>10</b>
<b>2. Introduction to Computer Systems and Machine-Level Representation of C Programs</b>	<b>13</b>
2.1. Information is Bits in Context.....	13
2.2. Programs are Translated by Other Programs into Different Forms .....	15
2.3. Processors Read and Interpret Instructions Stored in Memory.....	16
2.4. Caches Matter.....	19
2.5. Storage Devices Form a Hierarchy .....	20
2.6. Systems Communicate With Other Systems Using Networks .....	20
2.7. C versus Assembly Programming.....	21
2.8. Machine-Level Code .....	22
2.9. Summary .....	22
<b>3. Análise do funcionamento do processador</b>	<b>24</b>
3.1. Operações num processador .....	24
3.2. Localização dos operandos .....	24
3.3. Formato das instruções.....	25
3.4. Tipos de instruções presentes num processador .....	26
3.5. Registos visíveis ao programador.....	26
3.6. Modos de acesso aos operandos .....	28
3.7. Caracterização das arquitecturas RISC (resumo) .....	28
3.8. Instruções de <i>input/output</i> .....	29
3.9. Ordenação de <i>bytes</i> numa palavra.....	30

## 1. Introdução à organização e arquitectura dum computador

### 1.1. Estrutura interna dum computador

### 1.2. A origem do computador "moderno"

### 1.3. Funcionamento básico dum computador

### 1.4. A hierarquia dum arquitectura de barramentos

### 1.5. Os módulos de I/O

### 1.6. Níveis de abstracção num computador e mecanismos de conversão

#### Anexo A: Representação de reais em vírgula flutuante

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

A ilustração da **estrutura interna dum computador** em termos funcionais – a unidade central de processamento, a memória, os módulos para Entrada/Saída de informação, a interligação dos componentes - é efectuada com recurso a figuras do livro do Stallings, sendo aprofundado no próximo capítulo e complementado com bibliografia adicional ("*How Computers Work*"; a indicar no sumário).

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**.

A análise das diversas formas de representar informação dentro de um computador será efectuada ao longo das diversas aulas: desde os textos e suportes multimédia que representam a forma de os seres humanos se comunicarem entre si, às formas compactas de representação de quantidades numéricas, e aos comandos específicos para execução directa nos computadores. A representação de quantidades numéricas é aprofundada com o estudo da representação de valores do tipo *int* e do tipo *float* (designação de números reais de precisão simples na linguagem C), sendo razoavelmente coberto no Cap. 8 do Stallings; os anexos 1 e 2 do livro do Tanenbaum apresentam o mesmo assunto com maior clareza; um resumo dos aspectos mais relevantes da representação de valores em **vírgula flutuante** é incluído no Anexo A deste capítulo. A **representação de programas** é coberta com mais detalhe no capítulo seguinte.

### 1.1. Estrutura interna dum computador

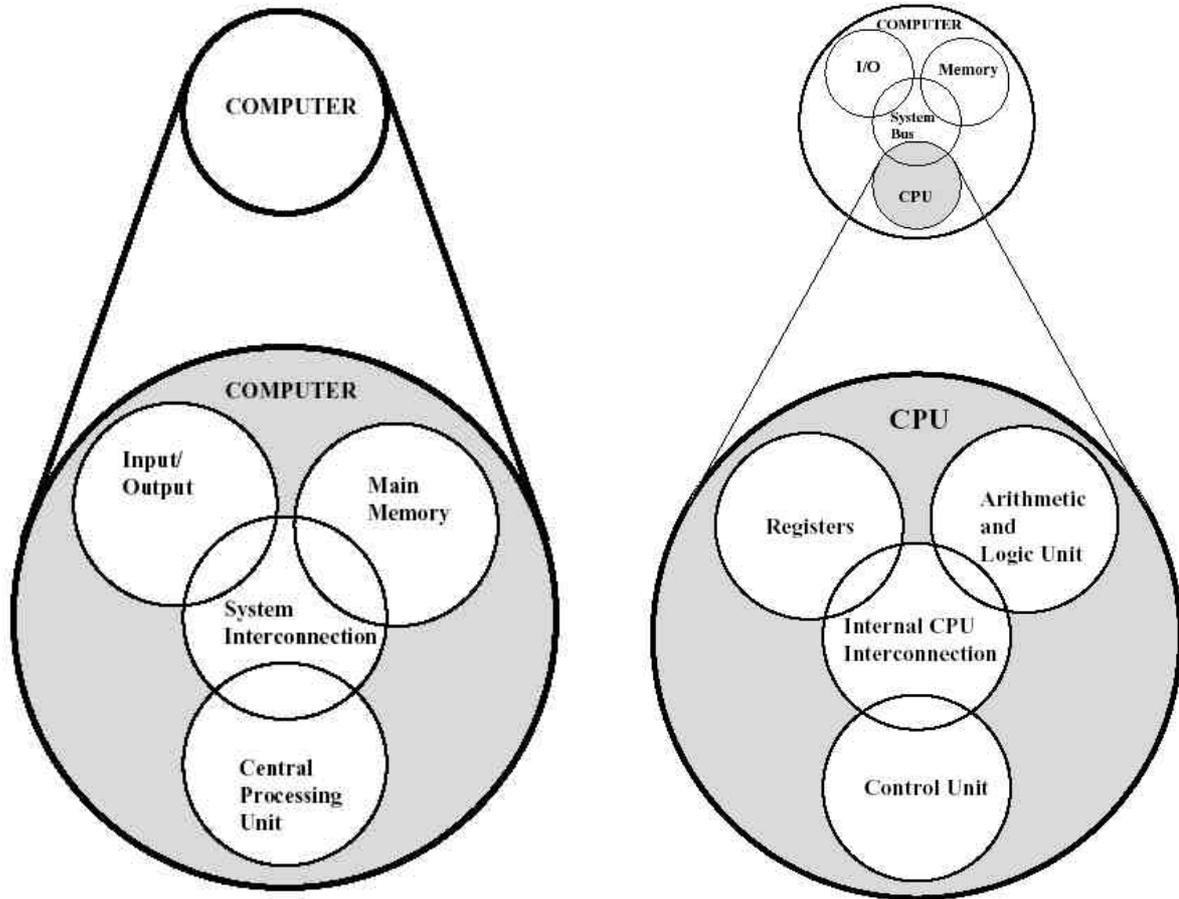
Os principais blocos funcionais que se podem encontrar num computador podem ser agrupados em apenas 3: a entidade que processa a informação, a entidade que armazena a informação que está a ser processada, e as unidades que estabelecem a ligação deste par de entidades (processador-memória) com o exterior.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

- **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, ...

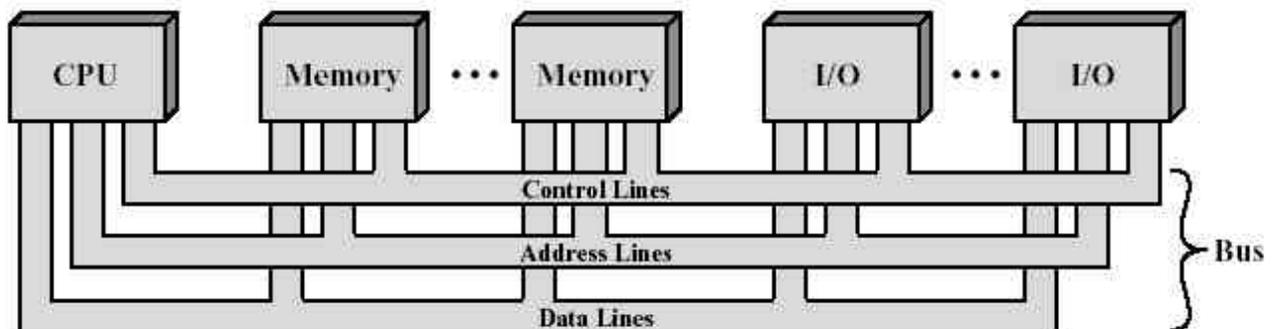
Os principais blocos que constituem um processador podem ser identificados como sendo:

- **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 ( $2^n$ )
  - **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), ...



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



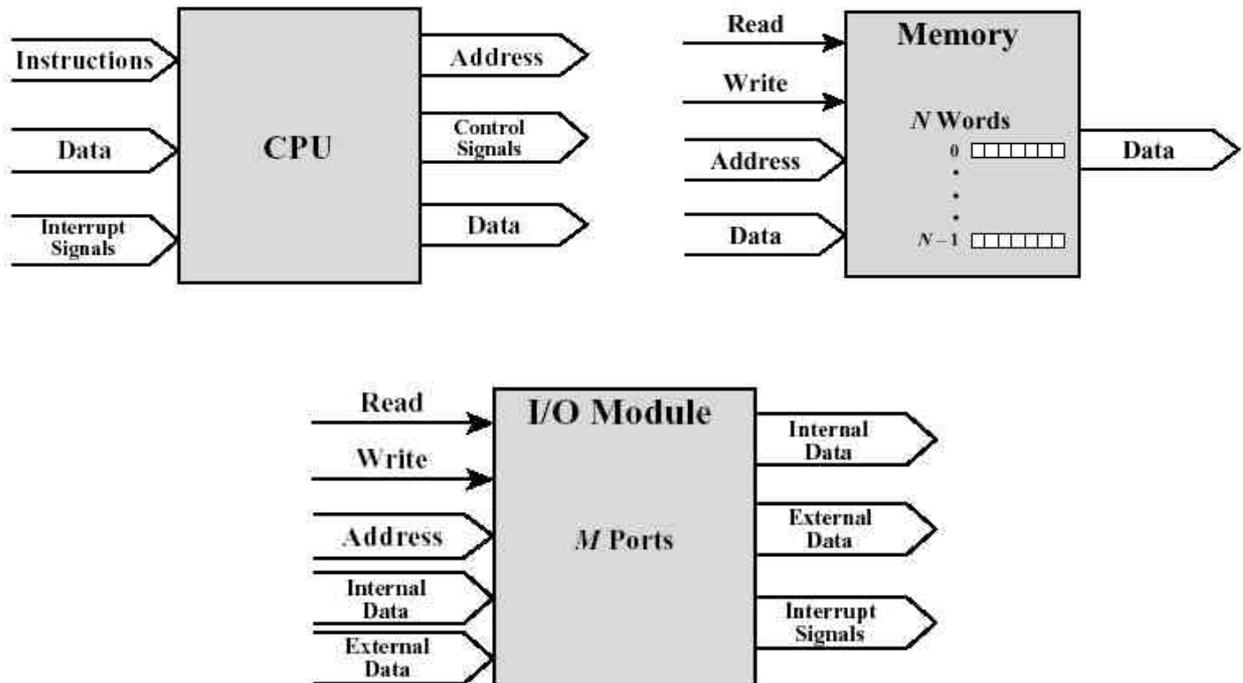
As principais categorias de barramentos são normalmente designadas por:

- **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*, ...).

Os sinais que normalmente se encontram 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:



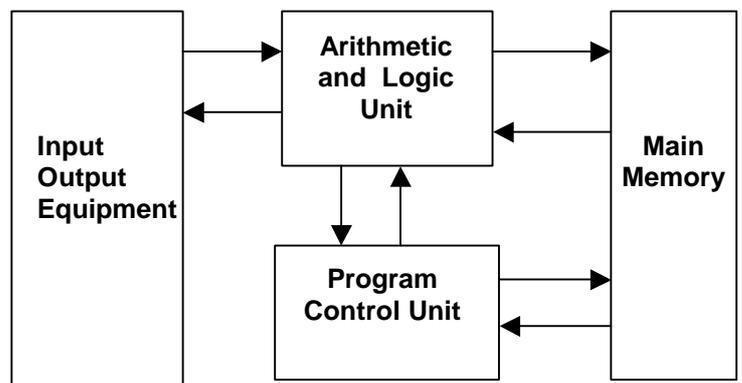
## 1.2.A origem do computador "moderno"

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

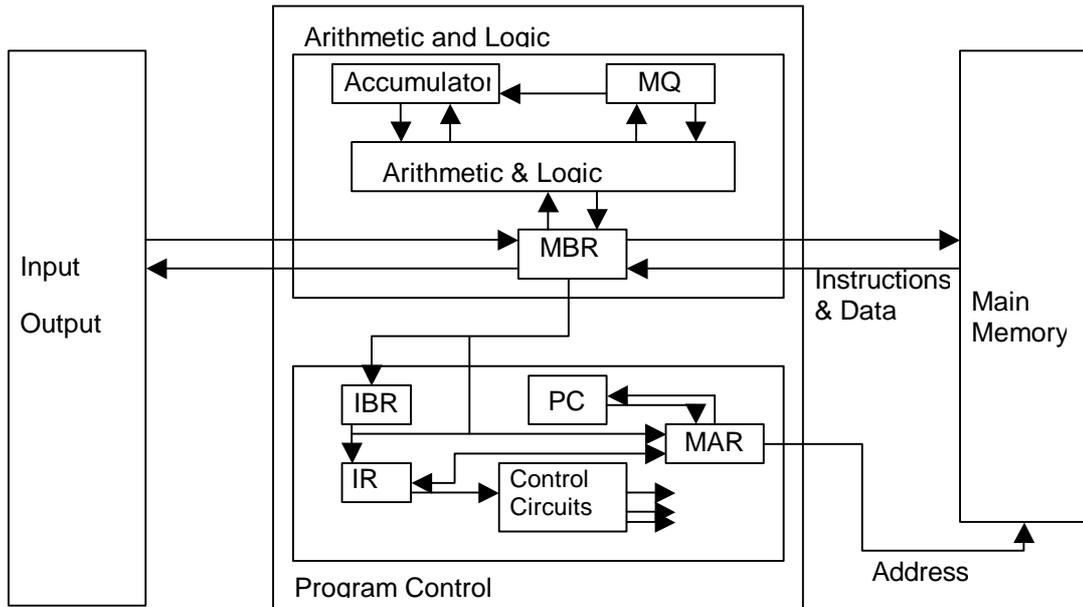
- os dados e as instruções são armazenados numa simples memória de leitura-escrita;
- os conteúdos desta memória são acedidos pelo endereço da sua localização, independentemente da informação lá contida;
- a execução ocorre de um modo sequencial (a menos que explicitamente modificado), de instrução para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



Estrutura de um computador IAS



Estrutura mais detalhada de um computador IAS

### 1.3. Funcionamento básico dum computador

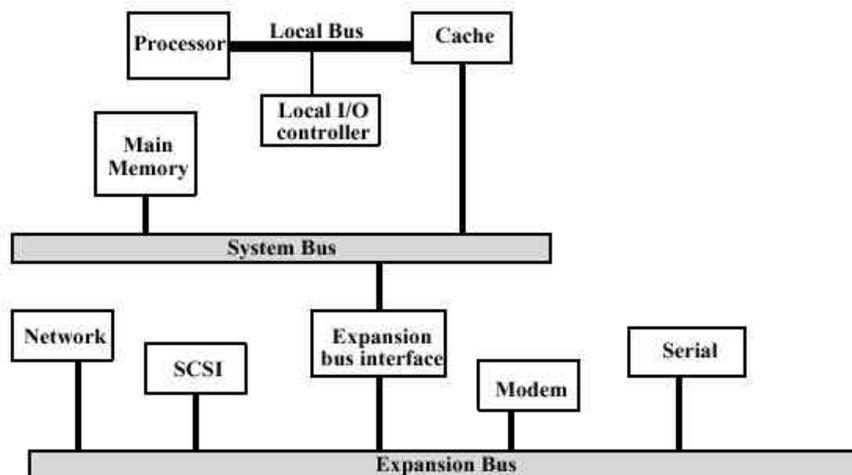
Na execução de um programa já em memória pronto a ser executado – programa em linguagem máquina - um processador executa sempre os seguintes passos:

- vai buscar uma instrução à memória e actualiza o apontador para a próxima instrução;
- decodifica a informação contida nos bits que constituem a instrução;
- executa a operação especificada, buscando os operandos quando necessário e armazenando o resultado quando solicitado; depois repete de novo o primeiro passo.

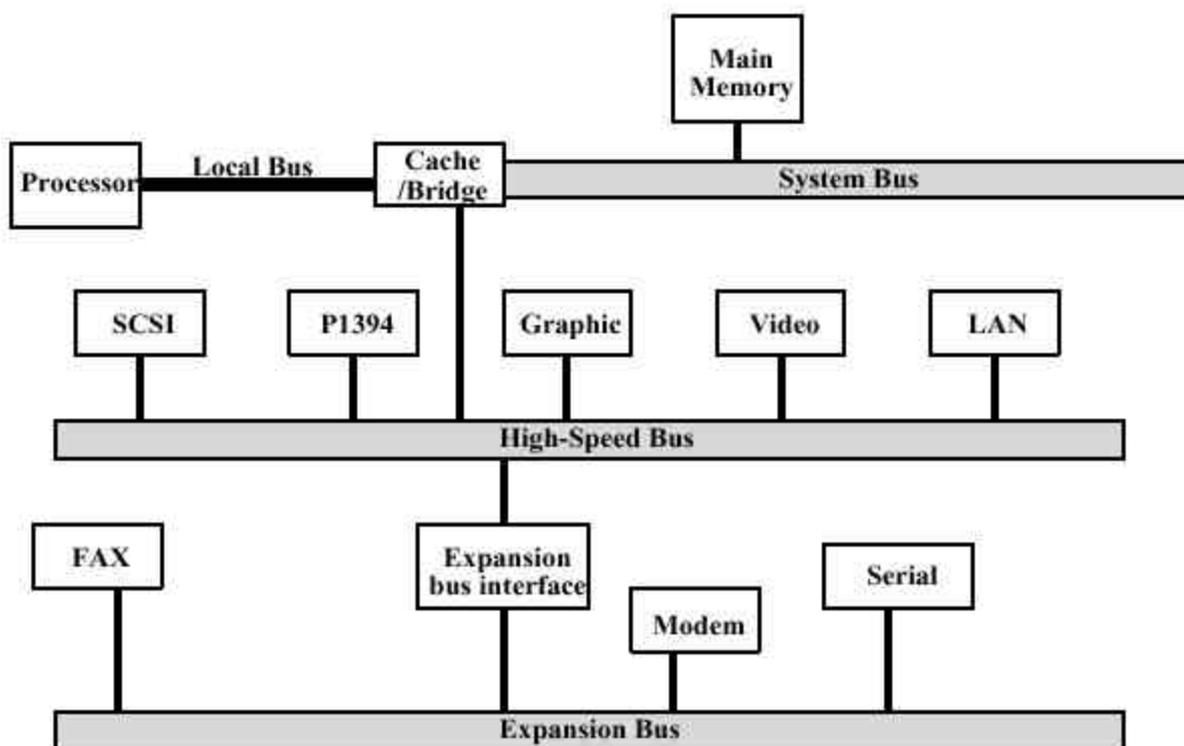
Por razões de eficiência, esses comandos deverão estar já armazenados na memória rápida do computador, codificados em formato associado ao processador que o vai executar, i.e., na sua linguagem máquina. Mais adiante se verá como se pode converter um programa escrito numa linguagem de alto nível para que ele possa ser executado pelo processador, e como será executado.

### 1.4. A hierarquia duma arquitectura de barramentos

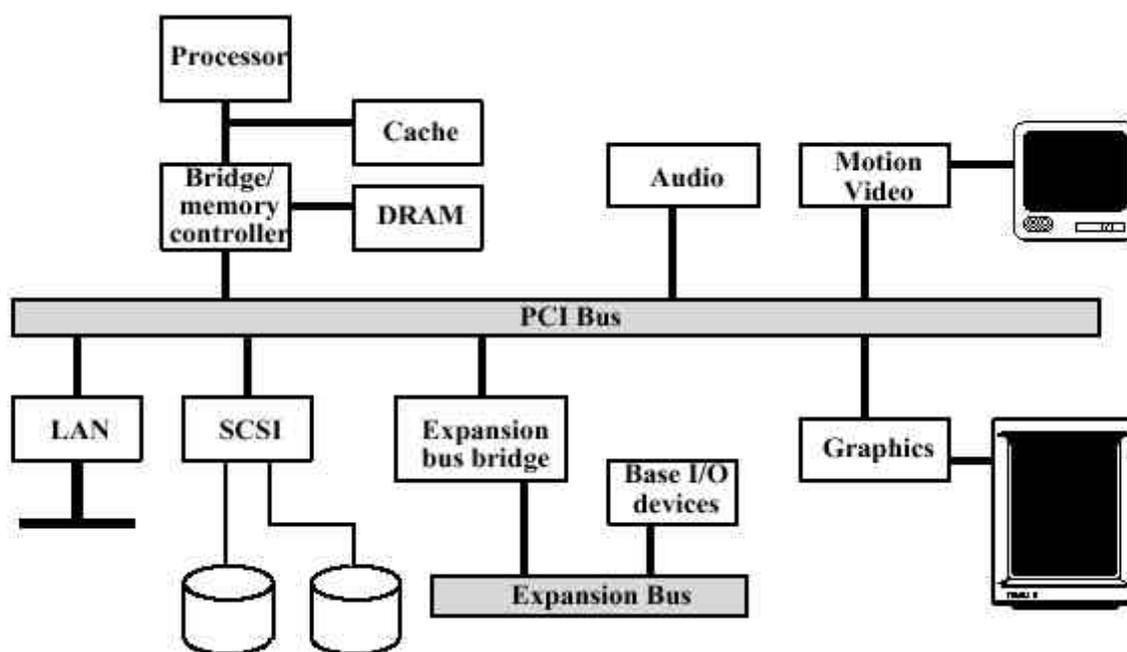
A existência de um único conjunto de barramentos para a circulação da informação entre os diversos módulos de um computador não leva em consideração que certos tipo de informação tem requisitos temporais muito mais exigentes que outros; por exemplo, a leitura de instruções da memória exige tempos de 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**:



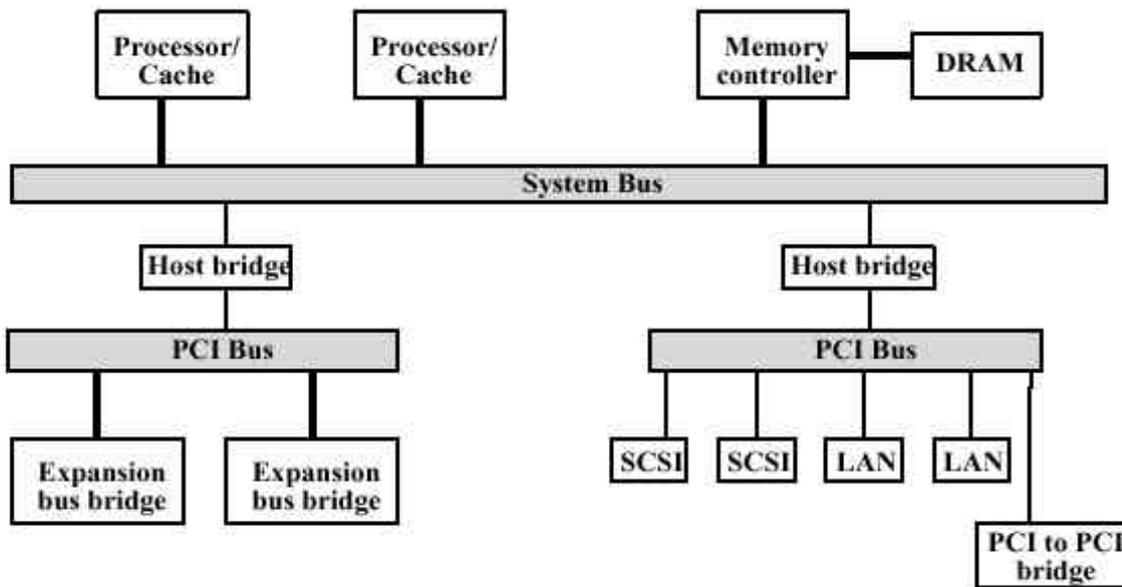
Neste tipo de organização de barramentos não se faz distinção entre periféricos com requisitos temporais exigentes de outros que não os tenham. A figura que se segue já toma em consideração essa distinção, e é uma **organização típica de uma hierarquia de barramentos de alto desempenho**:



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*

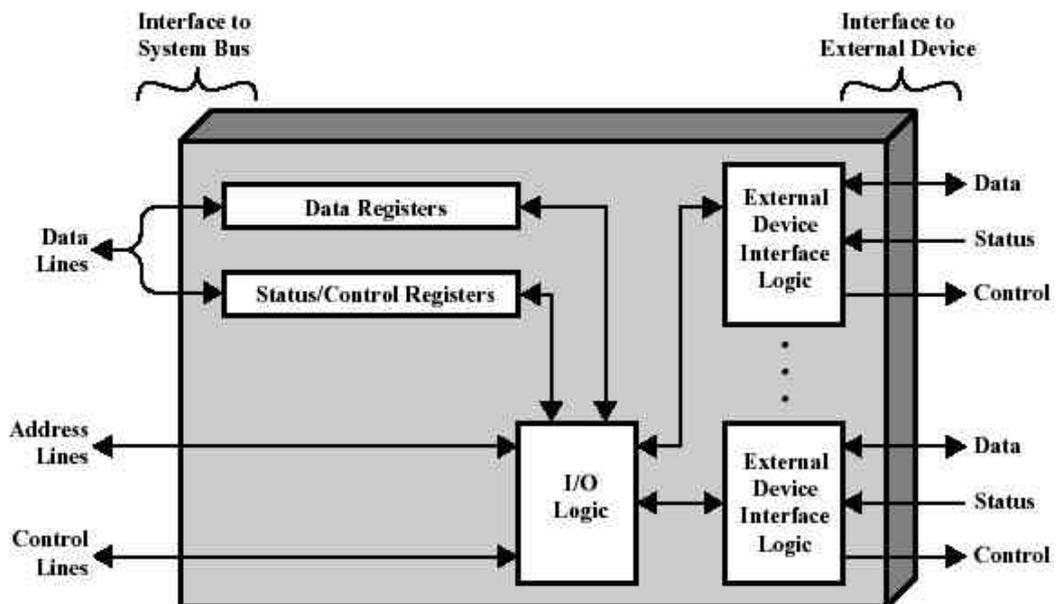


Organização de barramentos típica de um sistema servidor

### 1.5. Módulos de I/O

As principais funções de um módulo de Entrada/Saída (ou módulo I/O) são essencialmente de efectuar o interface entre o mundo digital codificado internamente de uma dada maneira (visível nas relações CPU/memória) com o(s) componentes periféricos através de ligações de dados específicas.

A estrutura básica de um módulo de I/O é a apresentada na figura:



### 1.6. 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 disciplina:

- **Nível da linguagem máquina (em binário):** instruções e variáveis totalmente codificadas em binário, sendo a codificação das instruções sempre associada a um dado processador; a sua utilização é pouco

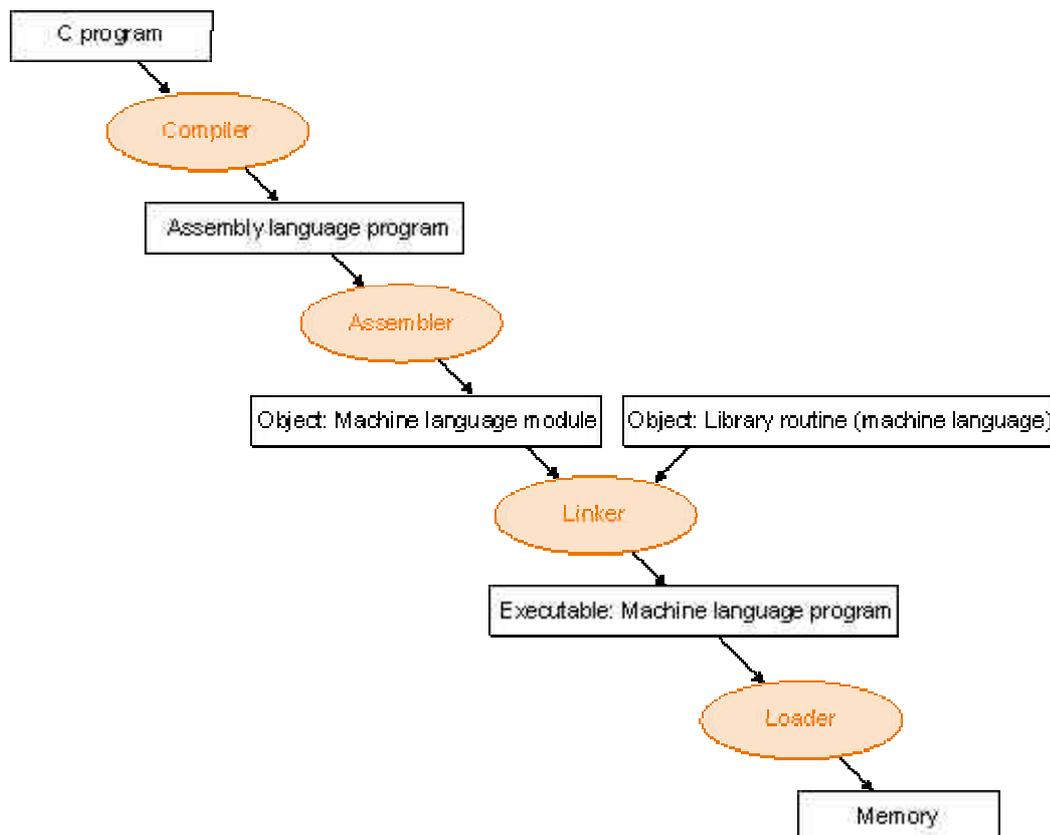
adequada para seres humanos;

- **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:

- **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)
- **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*.

O anexo B apresenta uma visão mais completa deste assunto, enquanto a figura a seguir ilustra estes níveis de abstracção e respectivos mecanismos de conversão.



Existe ainda outro mecanismo que permitem 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 noutra, mas sim a análise dum programa seguida de geração e execução do código máquina associado.

## Anexo A: Representação de reais em vírgula flutuante

- A.1. Notação científica
- A.2. Normalização na representação
- A.3. Intervalo e precisão de valores representáveis
- A.4. Formato binário dum valor em fp
- A.5. O bit escondido
- A.6. A norma IEEE 754 para valores em fp

### A.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:

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

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.

### A.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 **]. 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 otimizar 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**.

### A.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.

#### A.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.

#### A.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).

#### A.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+magnitude, 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):**

$$V = (-1)^S * (1.F) * 2^{(E-127)}$$

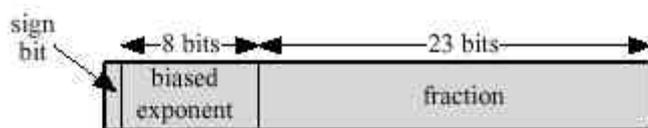
em que S, F e E representam respectivamente os valores em binário dos campos no formato em fp;

- **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 ≠ 0**; o valor decimal vem dado por

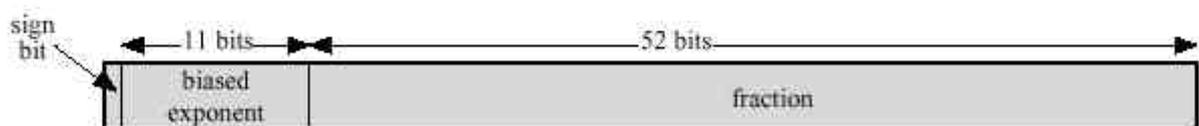
$$V = (-1)^S * (0.F) * 2^{(-126)}$$

- **representação do zero:** é o caso particular previsto em cima, onde **E = 0** e **F = 0** ;
- **representação de ±∞ :** 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 (NaN):** 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 ≠ 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) Single format



(b) Double format

## 2. Introduction to Computer Systems and Machine-Level Representation of C Programs<sup>1</sup>

### 2.1. Information is Bits in Context

### 2.2. Programs are Translated by Other Programs into Different Forms

### 2.3. Processors Read and Interpret Instructions Stored in Memory

### 2.4. Caches Matter

### 2.5. Storage Devices Form a Hierarchy

### 2.6. Systems Communicate With Other Systems Using Networks

### 2.7. C versus Assembly Programming

### 2.8. Machine-Level Code

### 2.9. Summary

### 2.1. Information is Bits in Context

A *computer system* is a collection of hardware and software components that work together to run computer programs. Specific implementations of systems change over time, but the underlying concepts do not. All systems have similar hardware and software components that perform similar functions. Randal Bryant and David O'Hallaron's book is written for programmers who want to improve at their craft by understanding how these components work and how they affect the correctness and performance of their programs.

In their classic text on the C programming language, Kernighan and Ritchie introduce readers to C using the `hello` program shown in Figure 1.

---

```

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

```

*code/intro/hello.c*

---

*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.

---

<sup>1</sup> Compiled and adapted from the Beta Draft version book of Randal E. Bryant and David R. O'Hallaron *Computer Systems: A Programmer's Perspective* (Prentice Hall, 2002, final version not yet available). These edited notes aim to introduce the students to Computer Architecture courses lectured at Dep. Informatics, University of Minho (Feb 2002).

#	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 'a' 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.

As programmers, we need to understand machine representations of numbers because they are not the same as integers and real numbers. They are finite approximations that can behave in unexpected ways. This fundamental idea is explored in detail in Chapter 2.

**Aside: The C programming language.**

C was developed from 1969 to 1977 by Dennis Ritchie of Bell Laboratories. The American National Standards Institute (ANSI) ratified the ANSI C standard in 1989. The standard defines the C language and a set of library functions known as the *C standard library*. Kernighan and Ritchie describe ANSI C in their classic book, which is known affectionately as “K&R” [41]. In Ritchie’s words [65], C is “quirky, flawed, and an enormous success.” So why the success?

- *C was closely tied with the Unix operating system.* C was developed from the beginning as the system programming language for Unix. Most of the Unix kernel, and all of its supporting tools and libraries, were written in C. As Unix became popular in universities in the late 1970s and early 1980s, many people were exposed to C and found that they liked it. Since Unix was written almost entirely in C, it could be easily ported to new machines, which created an even wider audience for both C and Unix.
- *C is a small, simple language.* The design was controlled by a single person, rather than a committee, and the result was a clean, consistent design with little baggage. The K&R book describes the complete language and standard library, with numerous examples and exercises, in only 261 pages. The simplicity of C made it relatively easy to learn and to port to different computers.
- *C was designed for a practical purpose.* C was designed to implement the Unix operating system. Later, other people found that they could write the programs they wanted, without the language getting in the way.

C is the language of choice for system-level programming, and there is a huge installed base of application level programs as well. However, it is not perfect for all programmers and all situations. C pointers are a common source of confusion and programming in C also lacks explicit support for useful abstractions such as classes, objects, and exceptions. Newer languages such as C++ and Java address these issues for application level programs.

**End Aside.**

## 2.2. Programs are Translated by Other Programs into Different Forms

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 3. The programs that perform the four phases (*preprocessor*, *compiler*, *assembler*, and *linker*) are known collectively as the *compilation system*.

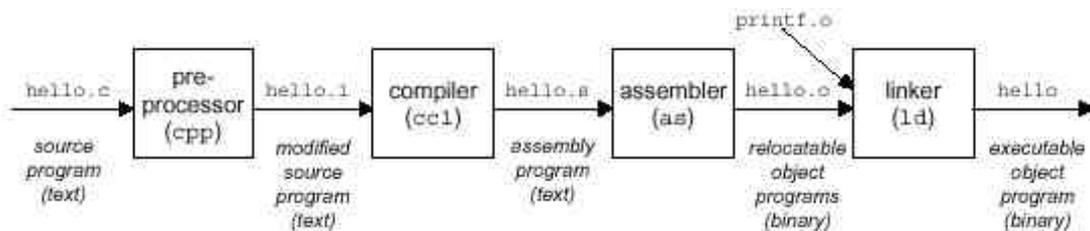


Figure 3: The compilation system.

- *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.

### Aside: The GNU project.

GNU is one of many useful tools developed by the GNU (short for GNU's Not Unix) project. The GNU project is a tax-exempt charity started by Richard Stallman in 1984, with the ambitious goal of developing a complete Unix-like system whose source code is unencumbered by restrictions on how it can be modified or distributed. As of 2002, the GNU project has developed an environment with all the major components of a Unix operating system, except for the kernel, which was developed separately by the Linux project. The GNU environment includes the EMACS editor, GCC compiler, GDB debugger, assembler, linker, utilities for manipulating binaries, and other components.

The GNU project is a remarkable achievement, and yet it is often overlooked. The modern **open source** movement (commonly associated with Linux) owes its intellectual origins to the GNU project's notion of *free software* ("free" as in "free speech" not "free beer"). Further, Linux owes much of its popularity to the GNU tools, which provide the environment for the Linux kernel.

**End Aside.**

### 2.3. 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 on a Unix system, we type its name to an application program known as *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.

#### 2.3.1. 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 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, as we will get to its various details in stages in the course of the book.

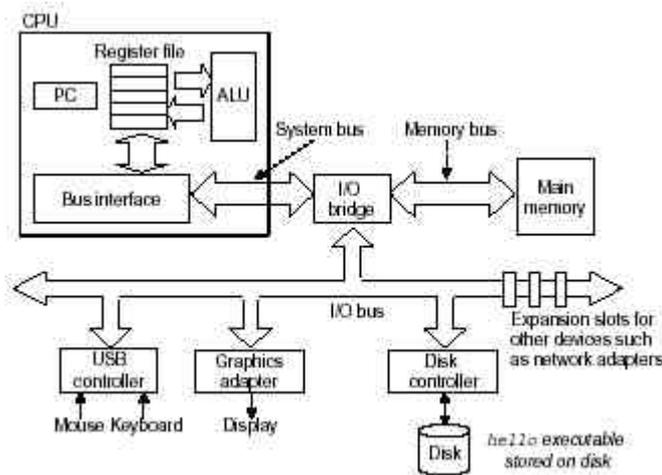


Figure 4: **Hardware organization of a typical system.** CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.

#### 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* 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 server-class systems such as Intel Itaniums and high-end Sun SPARC5 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. And 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, type `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 and over: 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.

Chapter 4 (of the book) has much more to say about how processors work.

### 2.3.2. Running the hello Program

Given this 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 5.

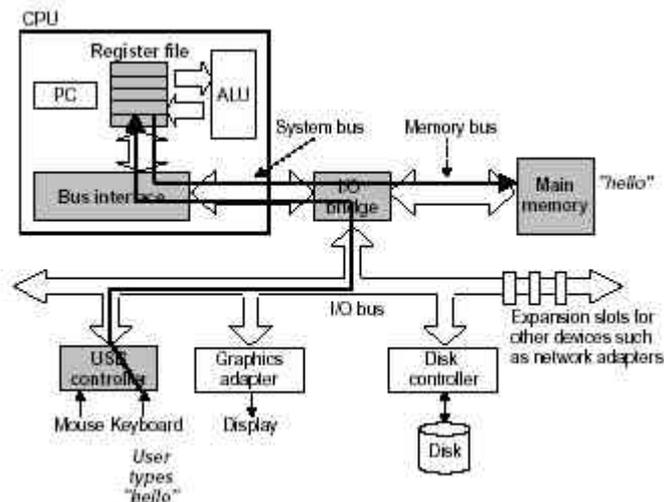


Figure 5: Reading the `hello` command from the keyboard.

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.

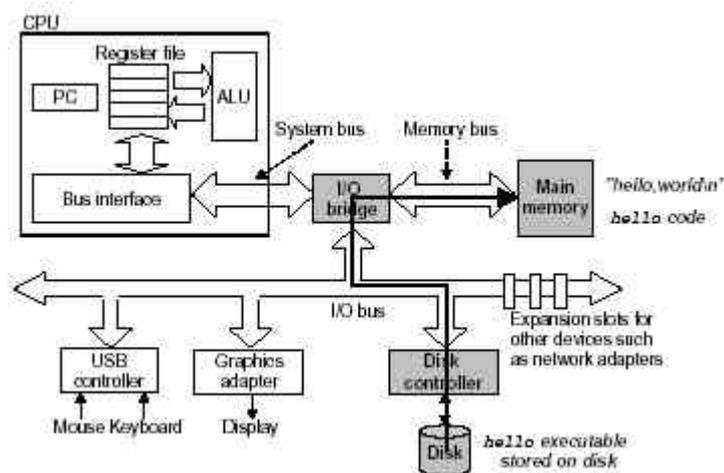


Figure 6: Loading the executable from disk into main memory.

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

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 instructions 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 7.

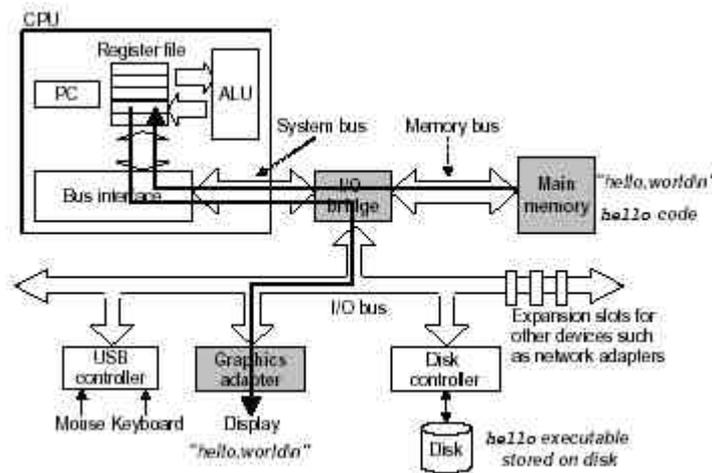


Figure 7: Writing the output string from memory to the display.

## 2.4. Caches Matter

An important lesson from this simple example is that a system spends a lot of time moving information from one place to another. 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 of 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 *the memories* (or simply caches) that serve as temporary staging areas for information that the processor is likely to need in the near future. Figure 8 shows the cache memories in a typical system. A *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)*.

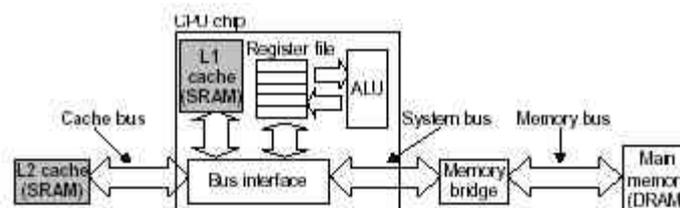


Figure 8: Cache memories.

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 (of the book).

## 2.5. Storage Devices Form a Hierarchy

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 9.

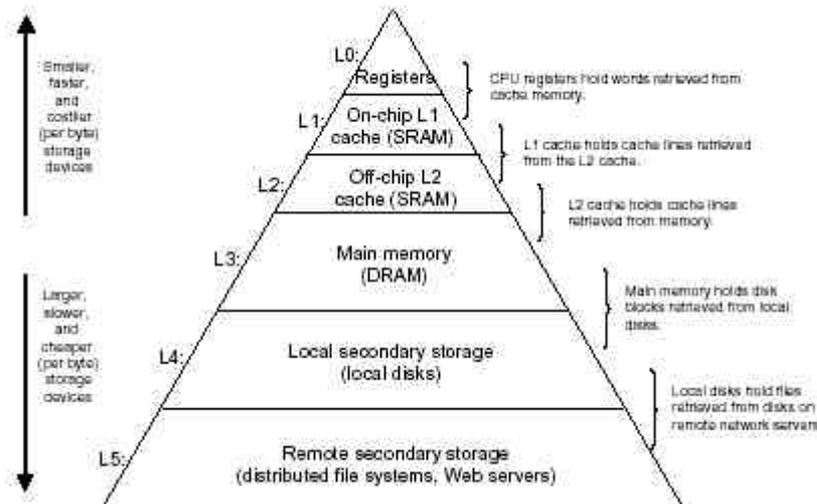


Figure 9: An example of a memory hierarchy.

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 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 system with distributed file systems, the local disk serves as a cache for data stored on the disk 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.

## 2.6. Systems Communicate With Other Systems Using Networks

Up to this point in our tour of systems, we have treated a system as an isolated collection of hardware and software. In practice, modern systems are often linked to other systems by networks. From the point of view of an individual system, the network can be viewed as just another I/O device, as shown in Figure 14. When the system copies a sequence of bytes from main memory to the network adapter, the data flows across the network to another machine, instead of say, to local disk drive. Similarly, the system can read data sent from other machines and copy this data to its main memory.

With the advent of global networks such as the Internet, copying information from one machine to another has become one of the most important uses of computer systems. For example, applications such as email, instant messaging, the World Wide Web, FTP, and telnet are all based on the ability to copy information over a network.

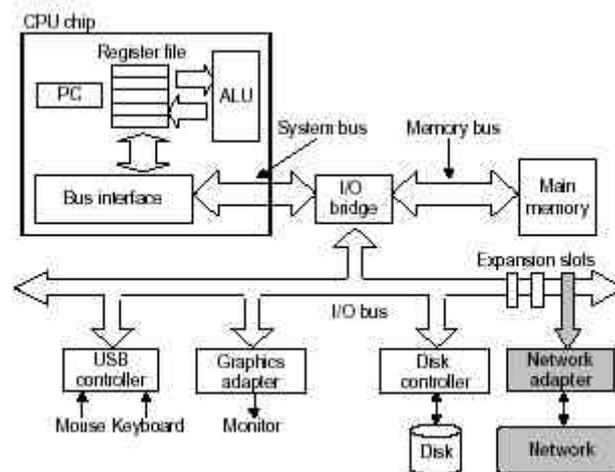


Figure 14: A network is another I/O device.

## 2.7. C versus Assembly Programming

When programming in a high-level language, such as C, we are shielded from the detailed, machine-level implementation of our program. In contrast, when writing programs in assembly code, a programmer must specify exactly how the program manages memory and the low-level instructions the program uses to carry out the computation. Most of the time, it is much more productive and reliable to work at the higher level of abstraction provided by a high-level language. The type checking provided by a compiler helps detect many program errors and makes sure we reference and manipulate data in consistent ways. With modern, optimizing compilers, the generated code is usually at least as efficient as what a skilled, assembly-language programmer would write by hand. Best of all, a program written in a high-level language can be compiled and executed on a number of different machines, whereas assembly code is highly machine specific.

Even though optimizing compilers are available, being able to read and understand assembly code is an important skill for serious programmers. By invoking the compiler with appropriate flags, the compiler will generate a file showing its output in assembly code. Assembly code is very close to the actual machine code that computers execute. Its main feature is that it is in a more readable textual format, compared to the binary format of object code. By reading this assembly code, we can understand the optimization capabilities of the compiler and analyze the underlying inefficiencies in the code. As we will experience in Chapter 5 (of the book), programmers seeking to maximize the performance of a critical section of code often try different variations of the source code, each time compiling and examining the generated assembly code to get a sense of how efficiently the program will run. Furthermore, there are times when the layer of abstraction provided by a high-level language hides information about the runtime behavior of a program that we need to understand. For example, when writing concurrent programs using a thread package (covered in Chapter 11 of the book), it is important to know what type of storage is used to hold the different program variables. This information is visible at the assembly code level. The need for programmers to learn assembly code has shifted over the years from one of being able to write programs directly in assembly to one of being able to read and understand the code generated by optimizing compilers.

A brief history of the Intel architecture is the starting point of the companion document *Machine-Level Programs on Linux/IA32*<sup>2</sup>. Intel processors have grown from rather primitive-bit processors in 1978 to the mainstream machines for today's desktop computers. The architecture has grown correspondingly with new features added and the 16-bit architecture transformed to support 32-bit data and addresses. The result is a

<sup>2</sup> To update this document, please add the following Intel family member to the existing list:

**Pentium 4:** (2001, 42 Mtransistors). Added 8-byte integer and floatingpoint formats to the vector instructions, along with 144 new instructions for these formats. Intel shifted away from Roman numerals in their numbering convention.

rather peculiar design with features that make sense only when viewed from a historical perspective. It is also laden with features providing backward compatibility that are not used by modern compilers and operating systems. We will focus on the subset of the features used by `gcc` and Linux. This allows us to avoid much of the complexity and arcane features of IA32.

## 2.8. Machine-Level Code

The compiler does most of the work in the overall compilation sequence, transforming programs expressed in the relatively abstract execution model provided by C into the very elementary instructions that the processor executes. The assembly code representation is very close to machine code. Its main feature is that it is in a more readable textual format, as compared to the binary format of object code. Being able to understand assembly code and how it relates to the original C code is a key step in understanding how computers execute programs.

The assembly programmer's view of the machine differs significantly from that of a C programmer. Parts of the processor state are visible that are normally hidden from the C programmer:

- The program counter (called `%eip`) indicates the address in memory of the next instruction to be executed.
- The integer register file contains eight named locations storing `32` values. These registers can hold addresses (corresponding to C pointers) or integer data. Some registers are used to keep track of critical parts of the program state, while others are used to hold temporary data, such as the local variables of a procedure.
- The condition code registers hold status information about the most recently executed arithmetic instruction. These are used to implement conditional changes in the control flow, such as is required to implement `if` or `while` statements.
- The floating-point register file contains eight locations for storing floating-point data.

Whereas C provides a model where objects of different data types can be declared and allocated in memory, assembly code views the memory as simply a large, byte-addressable array. Aggregate data types in C such as arrays and structures are represented in assembly code as contiguous collections of bytes. Even for scalar data types, assembly code makes no distinctions between signed or unsigned integers, between different types of pointers, or even between pointers and integers.

The program memory contains the object code for the program, some information required by the operating system, a runtime stack for managing procedure calls and returns, and blocks of memory allocated by the user, (for example, by using the `malloc` library procedure).

The program memory is addressed using *virtual* addresses. At any given time, only limited subranges of virtual addresses are considered valid. For example, although the `32` addresses of IA32 potentially span a 4-gigabyte range of address values, a typical program will only have access to a few megabytes. The operating system manages this virtual address space, translating virtual addresses into the physical addresses of values in the actual processor memory.

A single machine instruction performs only a very elementary operation. For example, it might add two numbers stored in registers, transfer data between memory and a register, or conditionally branch to a new instruction address. The compiler must generate sequences of such instructions to implement program constructs such as arithmetic expression evaluation, loops, or procedure calls and returns.

## 2.9. Summary

A computer system consists of hardware and systems software that cooperate to run application programs. Information inside the computer is represented as groups of bits that are interpreted in different ways, depending on the context. Programs are translated by other programs into different forms, beginning as ASCII text and then translated by compilers and linkers into binary executable files.

Processors read and interpret binary instructions that are ~~sed~~ in main memory. Since computers spend most of their time copying data between memory, I/O devices, and the CPU registers, the storage devices in a system are arranged in a hierarchy, with the CPU registers at the top, followed by multiple levels of hardware cache memories, DRAM main memory, and disk storage. Storage devices that are higher in the hierarchy are faster and more costly per bit than those lower in the hierarchy. Storage devices that are higher in the hierarchy serve as caches for devices that are lower in the hierarchy. Programmers can optimize the performance of their C programs by understanding and exploiting the memory hierarchy.

Finally, networks provide ways for computer systems to communicate with one another. From the viewpoint of a particular system, the network is just another I/O device.

## 3. Análise do funcionamento do processador

- 3.1. Operações num processador
- 3.2. Localização dos operandos
- 3.3. Formato das instruções
- 3.4. Tipos de instruções presentes num processador
- 3.5. Registos visíveis ao programador
- 3.6. Modos de acesso aos operandos
- 3.7. Caracterização das arquitecturas RISC (resumo)
- 3.8. Instruções de *input/output*
- 3.9. Ordenação de *bytes* numa palavra

A análise do funcionamento do processador dum computador - o principal elemento responsável por processar a informação dentro de um computador - complementa a visão genérica sobre o funcionamento dum computador na execução de programas escritos numa dada linguagem HLL, apresentada no capítulo anterior, e concentra-se **no funcionamento do processador** e sua relação com a **memória**, introduzindo-se igualmente várias questões genéricas independentes da família do processador seleccionado

### 3.1. 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 - vêm directamente na própria instrução. Assim, se a operação pretendida é "a=b+c", uma instrução em linguagem *assembly* poderá ser "add a,b,c".

A maioria das arquitecturas de processadores existentes actualmente (e normalmente designadas por arquitecturas RISC, *Reduced Instruction Set Computers*) 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 modo encadeado (*pipeline* na terminologia anglo-saxónica).

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 (a= a+b).

### 3.2. 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.

### 3.3. Formato das instruções

A codificação das instruções *assembly* para linguagem máquina deverá ser compacta para que as instruções ocupem pouca memória e para que a sua transferência para o CPU seja rápida.

- **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**.

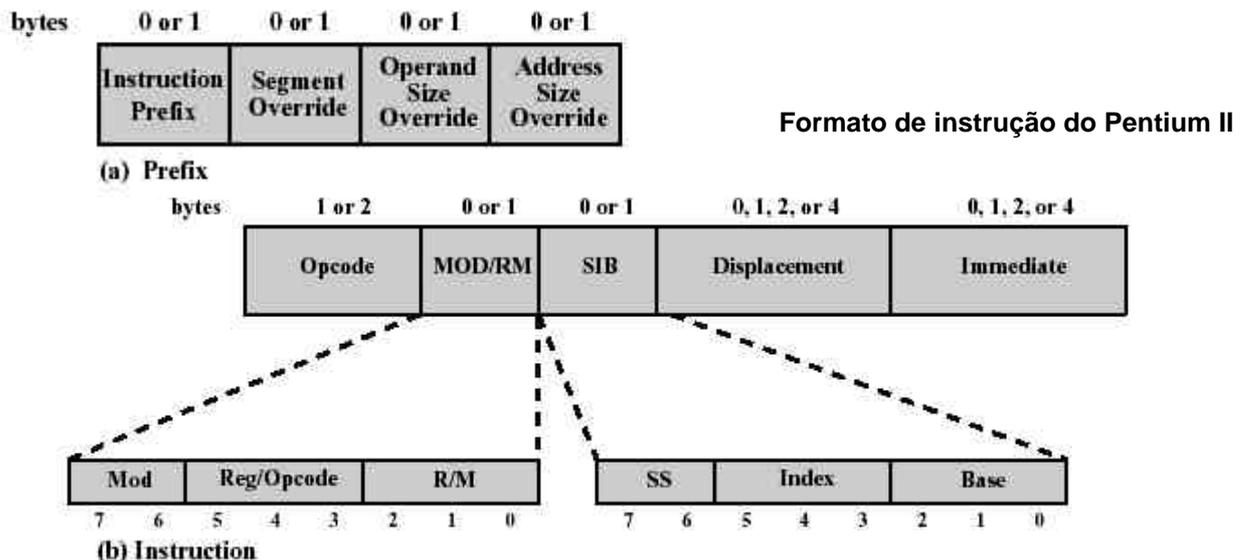
- **Campos duma instrução**

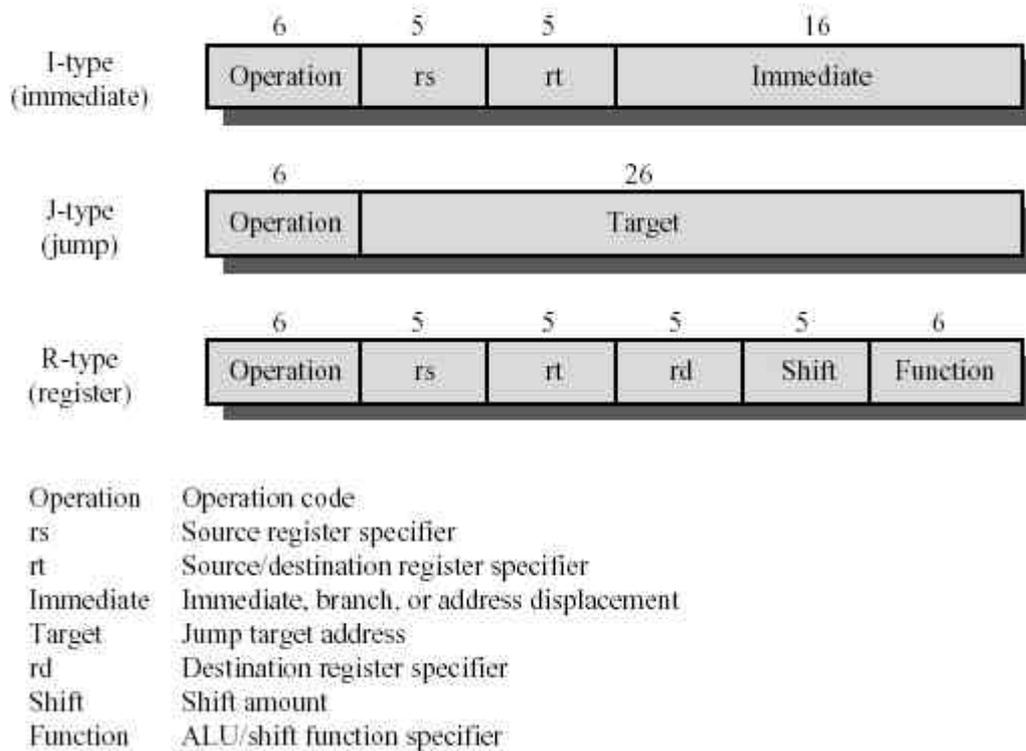
No formato de cada instrução em linguagem máquina é sempre possível identificar um conjunto de campos com informação bem definida: um campo que caracterize a operação a efectuar (normalmente designado por **opcode**) e tantos campos quantos o n.º de operandos que for possível especificar.

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 MIPS

#### 3.4. Tipos de instruções presentes num processador

O conjunto de instruções que cada processador suporta é bastante variado. Contudo é possível identificar e caracterizar grupos de instruções que se encontram presentes em qualquer arquitectura:

- **operações aritméticas, lógicas, ...:** soma, subtração e multiplicação com inteiros e fp, e operações lógicas AND, OR, NOT, ShiftLeft/Right 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.

#### 3.5. Registos visíveis ao programador

Um dos aspectos essenciais para um programador em *assembly* 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*.

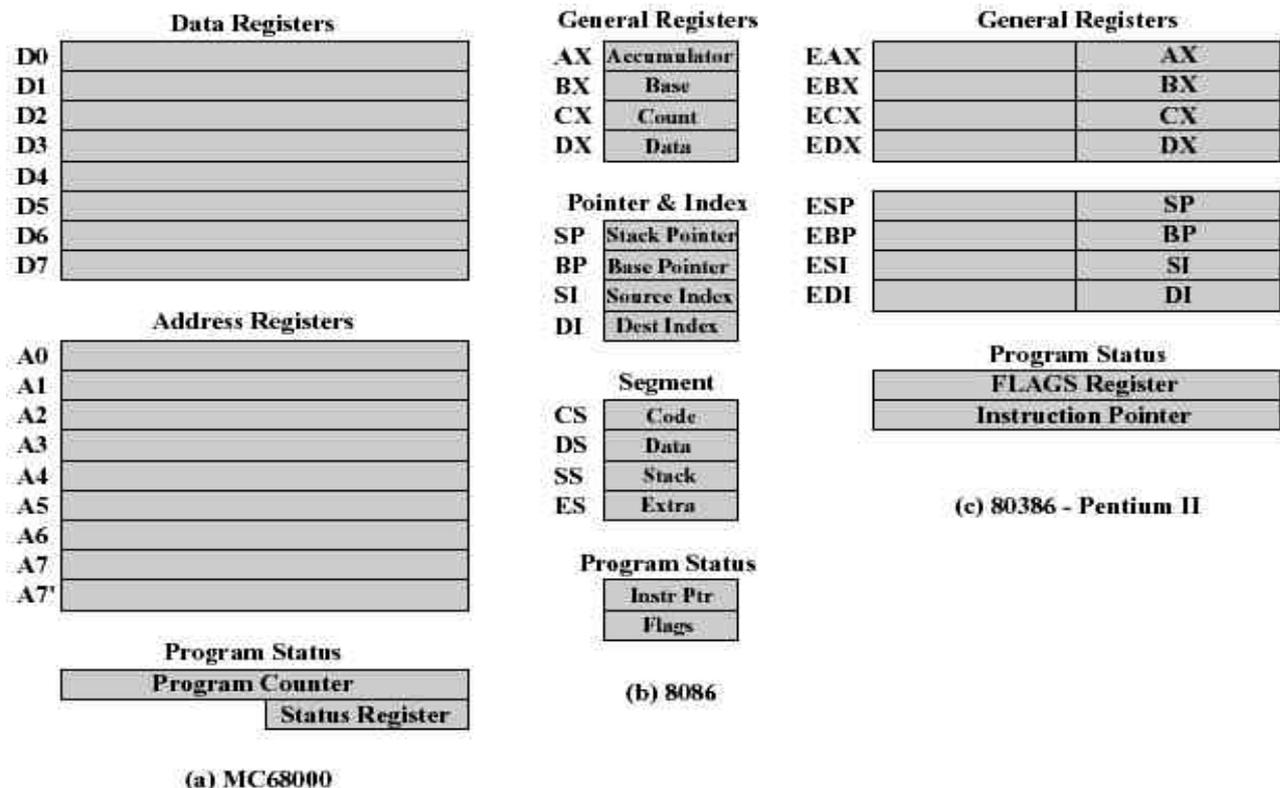
• **Em arquitecturas CISC (M680x0, Ix86/IA-32)**

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, conseqüentemente, 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 IA32), 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 ilustra a organização dos registos das arquitecturas CISC referidas:



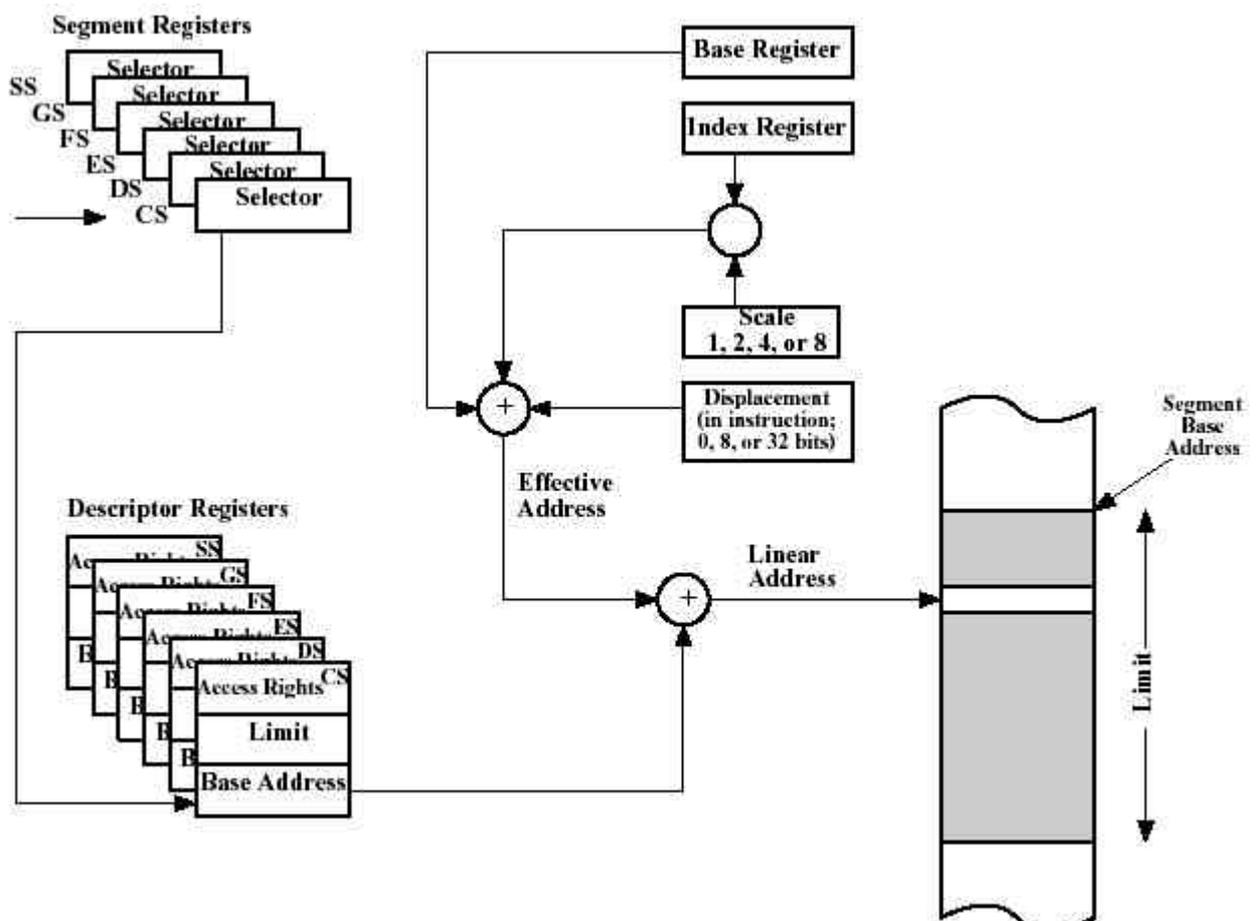
### 3.6. Modos de acesso aos operandos

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:



### 3.7. 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, 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 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 conseqüente 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 acessos 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!).

### 3.8. 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 (ver a secção sobre Módulos de I/O no capítulo 1):

- 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 numa porta - *in* - ou de escrita numa porta - *out*.

### 3.9. 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, IBM 370;
- 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.