Arquitetura de Computadores

MIEI – 2017/18
DI-FCT/UNL
Aula 9

Alguns µ-processadores Intel

- Cada nova geração aumentou a velocidade e capacidade:
  Proc. – dados/endereços
  - 8080 – 8bits/16bits
  - 8086/8088 – 16bits/20bits *(usados nos primeiros IBM/PC)*
  - 80286 – 16bits/24bits *(usado no IBM/AT)*
  - 80386 – 32bits/32bits *(usado no IBM/PS2)*
  - 80486, Pentium (P5), Pentium Pro (P6), …
- Existe compatibilidade
  - continuam a existir instruções com dados de 8bits no 80386 e seguintes

Diagrama do 8086

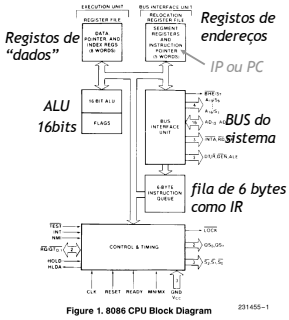
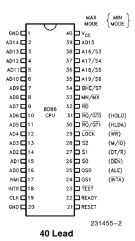


Figure 1. 8086 CPU Block Diagram



40 Lead

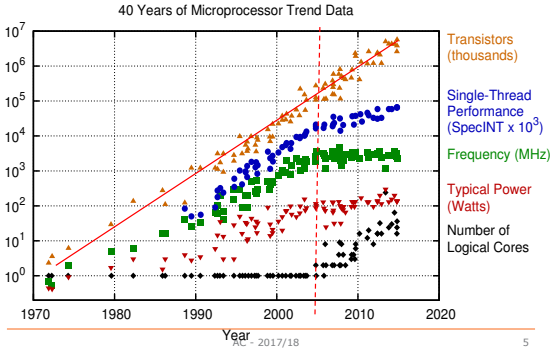
Figure 2. 8086 Pin Configuration

Table 2-1. Processor Performance Over Time and Other Key Features of the Intel Architecture

Intel Processor	Date of Product Intro-duction	Perfor-mance in MIPs[1]	Max. CPU Frequency at Intro-duction	No. of Transis-tors on the Die	Main CPU Register Size[2]	Extern. Data Bus Size[2]	Max. Extern. Addr. Space	
8086	1978	0.8	8 MHz	29 K	16	16	1 MB	(20bits)
Intel 286	1982	2.7	12.5 MHz	134 K	16	16	16 MB	(24bits)
Intel386™ DX	1985	6.0	20 MHz	275 K	32	32	4 GB	(32bits)
Intel486™ DX	1989	20	25 MHz	1.2 M	32	32	4 GB	
Pentium®	1993	100	60 MHz	3.1 M	32	64	4 GB	
Pentium Pro	1995	440	200 MHz	5.5 M	32	64	64 GB	(36bits)

Fonte: Intel architecture software developer's manual

Observação de Moore ("Lei de Moore")



Capacidades das arquitecturas

- O número de bits nos endereços (p.e. IP) determina a capacidade máxima de memória endereçável
  - Logo o maior programa, de código e dados
  - O tamanho dos apontadores em C
- O número de linhas no bus de endereços determina a capacidade máxima de memória realmente acessível
- O número de bits dos registos gerais determina o tamanho dos dados operados pelas instruções
- O número de linhas no bus de dados determina a capacidade máxima de transferência dos dados em cada ciclo de acesso à memória

Tamanho do endereço

- Palavras de memória que se pode endereçar (teoricamente):
  - 8 bits – 2^8 endereços =256 endereços
  - 16 bits – 2^{16} = 64 K
  - 32 bits – 2^{32} = 4 G
  - 64 bits – 2^{64} = 16 E
- Se memória endereçada ao byte:
  - 32 bits → 4 GBytes
- Se a memória endereçada por palavras de 2 bytes:
  - 32 bits → 8 GBytes

Múltiplos

- Nota sobre os unidades em bases 2 vs base 10 (exemplo com bytes):

Decimal			**Binary**				
Value	Metric		Value	IEC		*JEDEC*	
1000 (10^3)	kB	kilobyte	**1024 (2^{10})**	KiB	kibibyte	***KB***	***kilobyte***
1000^2	MB	megabyte	**1024^2**	MiB	mebibyte	***MB***	***megabyte***
1000^3	GB	gigabyte	**1024^3**	GiB	gibibyte	***GB***	***gigabyte***
1000^4	TB	terabyte	**1024^4**	TiB	tebibyte	***TB***	***terabyte***
1000^5	PB	petabyte	**1024^5**	PiB	pebibyte	***PB***	***petabyte***
1000^6	EB	exabyte	**1024^6**	EiB	exbibyte	***EB***	***exabyte***
1000^7	ZB	zettabyte	**1024^7**	ZiB	zebibyte	***ZB***	***zettabyte***
1000^8	YB	yottabyte	**1024^8**	YiB	yobibyte	***YB***	***yottabyte***

Conclusão

- O resultado desta evolução:
  - Arquiteturas muito complexas e difíceis de compreender e programar
  - Instruções de tamanho variável (de 1 a 18? bytes…)
  - ISA muito complexos, difíceis de interpretar pelo hardware
  - Os programadores (e os compiladores) "refugiam-se" num subconjunto do ISA …
  - Surgem arquiteturas RISC – *Reduced Instruction Set Computer*
  - Nos CISC, existe uma "tradução" para micro-código, mais simples. Este é executado pelos componentes do CPU
- O desempenho não vem só dos GHz...
  - Pipelines e Paralelismo interno
  - Paralelismo - multi-processadores

Exemplo Intel 80386 / **IA-32** (1985)

- CPU com 32 bits de dados e de endereços, com BUS de 32 bits:
  - dimensão do IP, registos de dados, número de linhas no BUS de dados e de endereços: 32
  - endereça até 4G de bytes de memória
  - transfere e opera até 4bytes de cada vez
- Base para os i486, Pentium/P5 (i586), Pentium Pro/P6 (i686)
  - 486 – FPU interna

Nomes dos registos em *Assembly*

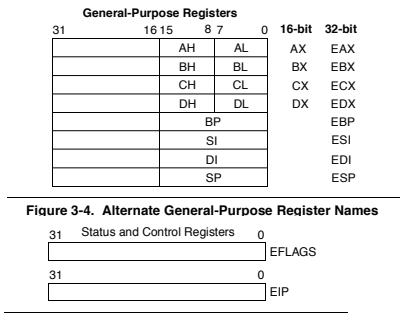


Figure 3-4. Alternate General-Purpose Register Names

Figure 3-3. Application Programming Registers

Principais tipos de dados

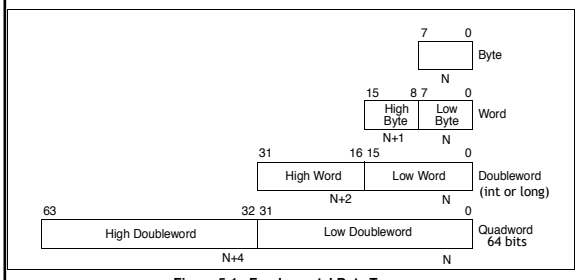
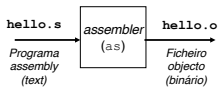


Figure 5-1. Fundamental Data Types

Programação em *Assembly*

- Ninguém programa diretamente em código máquina
- Programa-se em ***assembly***:
  - Texto, segundo regras próprias, onde se usa mnemónicas para as instruções, números em diversas bases, nomes simbólicos, etc.
- Um programa, o ***assembler***, traduz para código máquina:

```
as –o hello.o  hello.s
```



Assembly Unix para Intel

- Registos prefixados com % → **`%eax`**, **`%al`**, etc
- Valores imediatos prefixados com $ → **`$20`**, **`$0x1A`**, etc
- Instruções tipicas: ***mnemónica src, dst***
  - Dimensão dos dados da instrução na mnemónica:
  - mov**b** (8bits), mov**w** (16bits), mov**l** (32bits)
  - Pode ser subentendido pelos operandos (%eax-32bits; %al-8bits)

```
movl $20, %eax
mov $20, %eax
```


Assembly Unix para Intel

- Existem várias "instruções" que controlam o *assembler* e ajudam o programador mas que não se traduzem diretamente em instruções máquina:
  - Comentários: /* */ ou #
  - Etiquetas (labels) – para referenciar posições de memória
  - Preencher memória com valores (variáveis): .ascii, .byte, .int ...
  - Definir símbolos para valores (tipo *defines* do C)
  - etc

Reescrevendo Helloworld

```
    EXIT = 1          # usando simbolos para constantes
    WRITE = 4
    LINUX_SYSCALL = 0x80
.data           # secção de dados (variáveis)
msg:    .ascii    "Hello, world!\n"  # um vetor de caracteres
    LEN = . - msg      # LEN representa o tamanho do vetor
.text           # secção de código
.global _start   # exportar o simbolo _start (inicio do programa)
_start:  mov    $LEN,%edx
         mov    $msg,%ecx
         mov    $1,%ebx
         mov    $WRITE,%eax        # pedir write ao sistema
         int    $LINUX_SYSCALL     # chama o sistema

         mov    $0,%ebx
         mov    $EXIT,%eax          # pedir o exit ao sistema
         int    $LINUX_SYSCALL     # chama o sistema
```


Programação usando o Assembler

- O *assembler* verifica a validade da instrução *assembly*: se existe a instrução máquina tendo em conta os operandos indicados
  - Exemplo: *mov %eax, %cx* não existe!
- Resolve as etiquetas obtendo o endereço respetivo
- Também interpreta e faz as conversões de várias bases de numeração
  - Decimal, hexadecimal, binário, caracteres
  - Nomes simbólicos para constantes
- O *assembler* codifica a instrução no respectivo código máquina

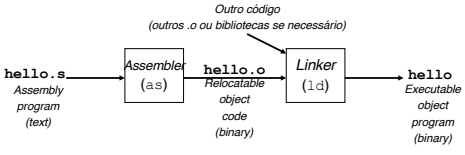
Produzindo o executável em Linux

- O *assember* (as no nosso caso) traduz para código máquina (.o):

```
as –o hello.o hello.s
```

- O *linker* (ld) produz o executável (formato ELF):

```
ld –o hello hello.o
```



Um programa em Unix/Linux

- Um programa, para ser executável num SO tem de seguir algumas regras:
  - O ficheiro contendo o executável tem um formato específico, descrevendo o programa (exemplo: ELF)
  - Apresenta código e dados em duas secções distintas e claramente identificadas
  - Indica o endereço onde começar a execução (exemplo: _start)
  - O *linker* (ld) produz este tipo de executáveis
- O código pode ter diversas origens:
  - *assembly* ou diversas linguagens de programação;
  - compiladas em momentos diferentes (ex: de bibliotecas)

Vendo o resultado do assembler

```
$ objdump  -D hello.o
hello.o:     file format elf32-i386
Disassembly of section .text:
00000000 <_start>:
0:       ba 0e 00 00 00          movl   $0xe,%edx
5:       b9 00 00 00 00          movl   $0x0,%ecx
a:       bb 01 00 00 00          movl   $0x1,%ebx
f:       b8 04 00 00 00          movl   $0x4,%eax
14:      cd 80                   int    $0x80
16:      bb 00 00 00 00          movl   $0x0,%ebx
1b:      b8 01 00 00 00          movl   $0x1,%eax
20:      cd 80                   int    $0x80
Disassembly of section .data:
00000000 <msg>:
0:       48
1:       65
2:       6c
3:       6c
4:       6f
 . . . etc
```


Codificação de cada instrução máquina

- O formato depende do tipo de instrução
  - O código da operação ou *opcode* (sempre!)
  - A indicação do tamanho dos dados
  - A especificação dos operandos: tipo e localização ou valor
- Exemplo: instrução com 3 operandos:

  op3 ← ***operação***(*op1, op2*)
  - Exige codificar em bits: a operação, os tipos de operandos (registos, endereço ou valor imediato) e os valores dos operandos (qual o registo, qual o valor ou qual o endereço de memória).
  - Possível codificação:

opc/typ	size	op1	op2	op3

Alternativas

- Instruções com outro número de operandos?

  op1 ← operação(op1, op2)

  op ← operação(op)
- Instruções sem operandos ou que assumem locais pré-definidos para operandos?
- Instruções operando sobre registos do CPU?
  - código do registo necessita de menos bits
- Usar operandos de que tamanho? (8, 16, 32 ?)
- **Nos Intel existem de todos os tipos**
  - As instruções não são todas do mesmo tamanho
  - *Fetch* e *decode* e restante CPU mais complicado
  - ler mais informação da memória e suportar vários tamanhos de dados