Arquitetura de Computadores

MiEI – 2016/17

DI-FCT/UNL

Aula 7

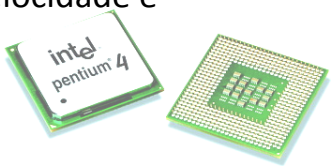
Intel e os circuitos integrados (CI)

- Intel- (Integrated Electronics Corporation)
  - Fundada em 1968 por funcionários vindos da Fairchild Semiconductor
    - Na origem (com a Texas Instruments) dos circuitos integrados (*microchip*)
  - Micro-processador
    - um único circuito integrado contendo todas as funcionalidades da unidade central de processamento (CPU) do computador
      - Intel 4004 – µ-proc. de 4bits para calculadoras

Nota: Já existiam processadores e computadores!

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

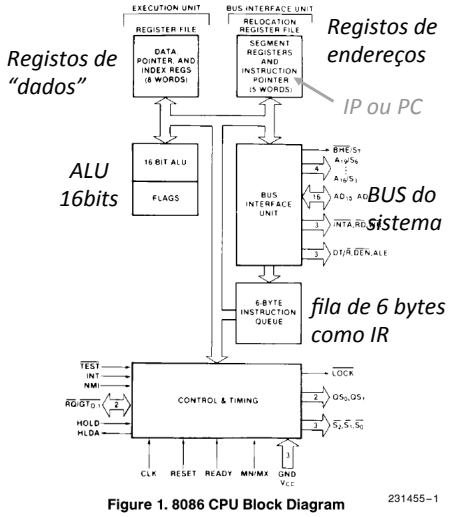
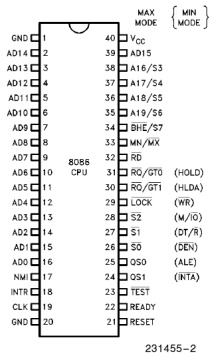


Figure 1. 8086 CPU Block Diagram

231455-1



231455-2

40 Lead

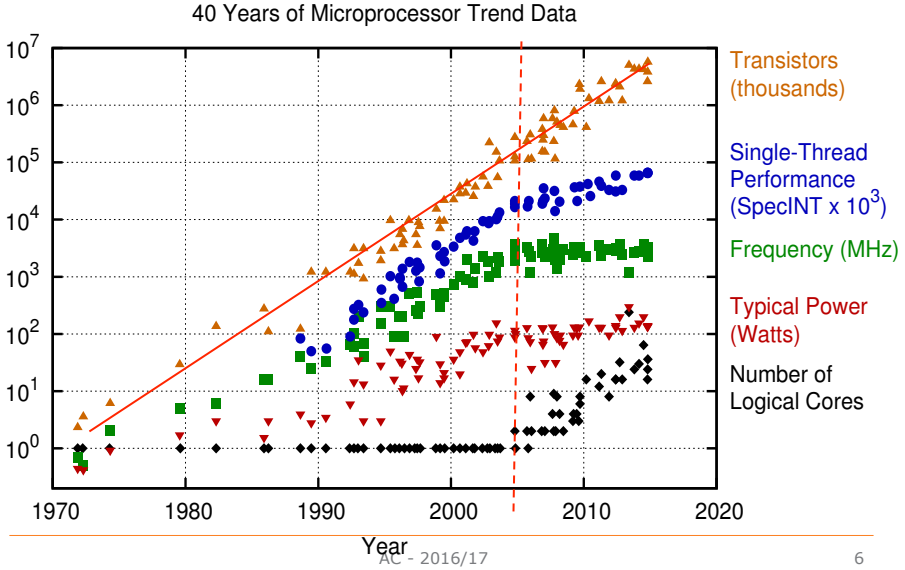
Figure 2. 8086 Pin Configuration

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
Intel 286	1982	2.7	12.5 MHz	134 K	16	16	16 MB
Intel386™ DX	1985	6.0	20 MHz	275 K	32	32	4 GB
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

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***

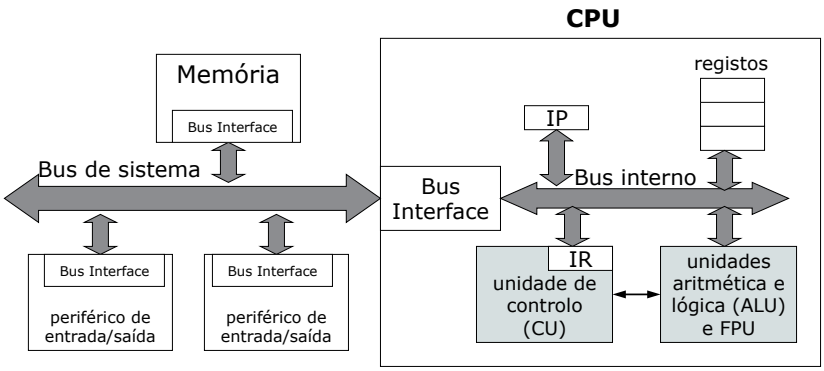
Conclusão

- 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 …
  - Internamente, existe uma "tradução" para micro-código, mais simples. Este é realmente executado pelo CPU
- O desempenho não vem só dos GHz...
  - Pipelines e Paralelismo interno
  - Paralelismo - multi-processadores
- Extensão do IA-32 para 64 bits:
  - AMD com AMD64; Intel com EM64T (x86-64)

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

Arquitectura de Von Neumann



Registos e nomes em *Assembly*

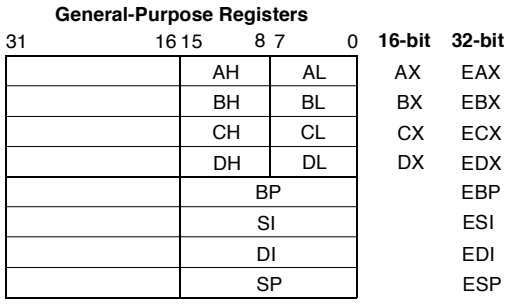


Figure 3-4. Alternate General-Purpose Register Names

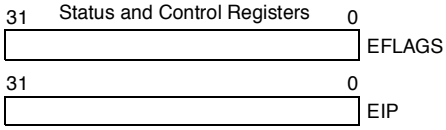


Figure 3-3. Application Programming Registers

13

Principais tipos de dados

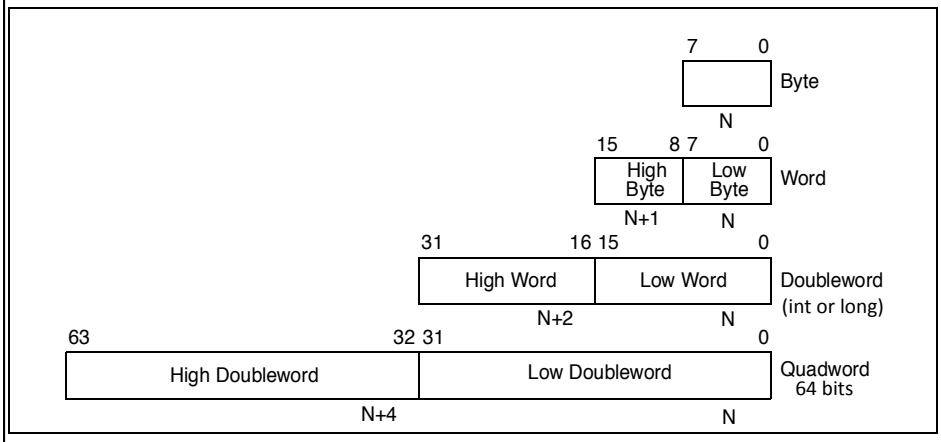


Figure 5-1. Fundamental Data Types

Helloworld em assembly Linux / Intel 32bits

```
.data           # secção de dados (variáveis)
msg:    .ascii    "Hello, world!\n"  # um vetor de caracteres
    len = 14    # len representa o tamanho do texto em msg

.text            # secção de código
.global _start    # exportar o simbolo _start (inicio do programa)
_start:  movl    $len,%edx
         movl    $msg,%ecx
         movl    $1,%ebx
         movl    $4,%eax       # pedir write ao sistema
         int     $0x80         # chama o sistema

         movl    $0,%ebx
         movl    $1,%eax       # pedir o exit ao sistema
         int     $0x80         # chama o sistema
```


Assembly Unix para Intel

- Sintaxe base: ***mnemónica src, dst***
- Dimensão dos dados dada na mnemónica:
  - mov**b** (8bits), mov**w** (16bits), mov**l** (32bits)
  - Pode ser subentendido pelos operandos (%eax-32bits; %al-8bits)
- Existem várias directivas 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
```

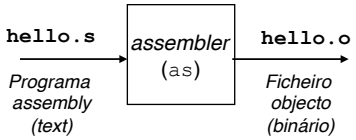

Programação usando o Assembler

- 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

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
```



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
```

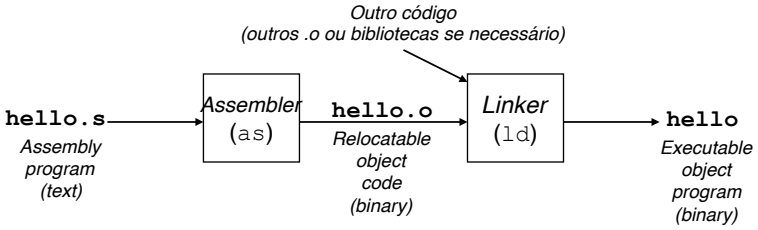
Produzindo o executável em Linux

- 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)

Codificação de cada instrução máquina

- O formato depende do tipo de instrução. Pode incluir:
  - 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 e os tipos de operandos (registos, endereço ou valor imediato) e o valor dos operandos: qual o registo, qual o valor ou qual o endereço de memória.
  - Possível codificação:

opcode	size	op1	op2	op3

Outras alternativas

- Instruções com que 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
- Usa 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

Codificação dos operandos nas instruções

- O código da operação indica a dimensão dos dados, o tipo de operandos e seus endereços
- Alguns exemplos de variantes de MOV no Intel

Mov register/32	reg1	reg2	**`mov %eax, %edx`** *reg1 → reg2*
Mov immed/32	reg1	valor imediato/32	**`mov $4, %eax`** *valor → reg1*
Mov immed/16	reg1	valor imed/16	**`add $4, %ax`** *valor → reg1* *(2 bytes)*
Mov memory/16	reg1	endereço	**`add (100), %ax`** *Mem[endereço] → reg1* *(2 bytes)*

Alternativas para os operandos

- Como ter instruções de menor tamanho:
  - Instruções com menos operandos
  - Instruções com operandos em registos do CPU
    - usa menos bits para endereçar os registos
  - Instruções com menos endereços ou valores imediatos nos operandos
    - ou na instrução indica-se um registo onde está o endereço do operando
  - Instruções com operandos implícitos
    - existem registos/endereços pré-definidos para a operação

Critérios na escolha dos operandos (e formato das instruções)

- Facilidade de programação - formas variadas de referir os operandos facilitam a vida ao programador
- Menor número de instruções - instruções com maior expressividade fazem com que os programas tenham menos instruções (embora mais longas e CPU mais complexo)
- Instruções mais curtas - limitando o número de operandos (ou obrigando a que sejam registos); mais rápido, mas mais instruções no programa
- Mais bits para constantes - k bits permitem representar 2k valores; logo mais bits é melhor, mas alonga as instruções
- Fases de "fetch" e descodificação mais rápidas - quanto menos operandos mais rápida e simples é a fase de fetch e obtenção dos operandos; instruções simples simplificam a unidade de controlo e permite mais optimizações

Operandos imediatos

- O operando faz parte da própria instrução
  - trazido para o CPU (para o *Instruction Register*), quando este faz *fetch* da instrução

Exemplo: mov **$4**, %eax EAX ← 4

  - Serve para especificar constantes na instrução
  - Pode reduzir o tamanho do programa e aumentar a velocidade da sua execução
  - Indicado no assembly por **$**

Operandos imediatos (2)

- Exemplos usando etiquetas ou nomes simbólicos para os valores:

```
Var:   .int 3  # define uma doubleword em memória com o valor 3
CONST = -5

mov   $Var, %eax          EAX ← Var
```

o endereço da variável será colocado no registo EAX

```
mov   $CONST, %eax              EAX ← -5
```

a representação de -5 será copiada para EAX

Operandos em registos

- O operando está contido num dos registos do CPU
  - Pressupõe que uma instrução anterior deixou o registador com o valor desejado

Exemplo: mov **%ax**, **%dx** DX ← AX

  - Acesso muito rápido ao operando
  - Servem como variáveis temporárias (ou de cópias das variáveis em memória)
  - Indicado no assembly por **%**
  - **Limitado pelo número de registos do CPU**

Endereçamento direto da memória

- O endereço está contido na instrução
  - O endereço na instrução indica a posição de memória onde está o operando

  Exemplo: mov **(100)**, %eax EAX ← Memória[100]
- A instrução contém o endereço, com o número de bits suficiente para alcançar toda a memória
- Indicado no assembly por **( )**
- É obrigatório conhecer essa posição quando se escreve o programa:
  - Pode ser difícil saber e pode obrigar a que o programa seja carregado numa zona específica da memória
  - **O *assembler* ajuda com as etiquetas (labels)**
    - **Funcionam como o nome das variáveis**

Endereçamento direto da memória (2)

- Exemplo de endereçamento **direto** da memória usando etiqueta em vez do valor do endereço:

```
Var:  .int 3  # define uma doubleword em memória com o valor 3

…
mov   Var, %eax        EAX ← memoria[Var]
```

Equivale a:

```
mov   (Var), %eax          »
```

Endereçamento indirecto por registo

- Um registo contém o endereço de memória onde está o operando
  - Pressupõe que uma instrução anterior deixou o registo com o endereço desejado

Exemplo: mov **(%ebx)**, %ax AX ← Memoria[EBX]

  - O registo referencia a memória, como um *pointer*
    - Facilita a programação (p.e. percorrer vetores)
  - Indicado no assembly por um registo entre **( )**

Transferências de dados (MOV)

Table 6-1. Move Instruction Operations

Type of Data Movement	Source → Destination
From memory to a register *(load)*	Memory location → General-purpose register Memory location → Segment register
From a register to memory *(store)*	General-purpose register → Memory location Segment register → Memory location
Between registers	General-purpose register → General-purpose register General-purpose register → Segment register Segment register → General-purpose register General-purpose register → Control register Control register → General-purpose register General-purpose register → Debug register Debug register → General-purpose register
Immediate data to a register	Immediate → General-purpose register
Immediate data to memory	Immediate → Memory location

Não transfere de memória para memória

Resumo - Combinações de operandos

	Origem	Destino	*Exemplo*	*Correspondência C (aprox)*
mov **movb** **movw** **movl**	*Imm*	*Reg*	`mov $0x4,%eax`	`temp = 0x4;`
		MemDir	`movl $-147,VAR`	`VAR = -147;`
		MemInd	`movl $45, (%edx)`	`*p = 45;`
	Reg	*Reg*	`mov %eax,%edx`	`temp2 = temp1;`
		MemDir	`mov %eax,VAR`	`VAR = temp;`
		MemInd	`mov %eax,(%edx)`	`*p = temp;`
	MemDir	*Reg*	`mov VAR,%edx`	`temp = VAR;`
	MemInd	*Reg*	`mov (%eax),%edx`	`temp = *p;`

De forma semelhante para as operações aritméticas e lógicas

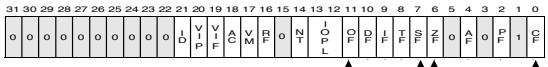
Operações aritmeticas e lógicas (exemplos)

- Exemplos em *assembly*:
  - Aritméticas:
    - add $5, %eax
    - add %bx, %cx
    - inc %al
  - Lógicas:
    - and %ecx, %eax
    - or $0x2F, %ah
    - not %ax
  - Estas correspondem a instruções de tamanhos diferentes, com operadores de tipos diferentes (8, 16 e 32 bits)
  - Nestes exemplos a dimensão dos dados pode ser inferida pelo *assember* a partir dos registos usados
- Estas operações alteram as *flags!* (*Overflow*, *Carry*, *etc…*)

Operações aritméticas e lógicas

- Operações:
  - aritméticas: add, sub, inc, dec, cmp, ...
  - lógicas: and, or, xor, not, ...
- Realizadas pela ALU
- Cada operação altera o estado de algumas *flags*

EFLAGS:



Exemplos:

- OF – *Overflow*
- SF – *Sign*
- ZF – *Zero*
- CF – *Carry*

ADD – Adição de números

add op1, op2

op2←op2+op1

- soma os bits de ambos os operandos
- o resultado é guardado na segunda referência (que não pode ser imediata)
- altera as *flags* de acordo com o resultado
- exemplos:

```
add $5, %eax        -- end. Imediato/reg e 32bits
add %ax, %bx        -- end. registo e 16bits
add (100), %al      -- end. direto e 8bits
```

Exemplos ADD e flags

```
    1110   14   -2
+   1010   10   -6
--------  ---- ----
  1 1000    8?  -8
CF=1  ZF=0  OF=0  SF=1
```

Interpretação sem e com sinal

```
  0110    6   +6
+ 1101   13   -3
------- ---- ----
1 0011    3?   3
CF=1  ZF=0  OF=0  SF=0
```

```
    1001   9   -7
+   0100   4    4
--------  --- ----
    1101  13   -3
CF=0  ZF=0  OF=0  SF=1
```

- Adição de 2 números com sinais diferentes nunca produz *overflow*
- Adição de 2 números com sinais iguais só dá *overflow* se o sinal do resultado for diferente

Exemplos ADD (2)

- Exemplos em que o resultado é inválido se em complemento para 2.
  - A OF e CF indicam que o resultado está errado, dependendo da representação ser em complemento para 2 ou não.

```
   1101   13   -3
+  1010   10   -6
------- ----- ----
 1 0111    7?  +7?
CF=1 ZF=0 OF=1 SF=0
```

```
   1000    8  -8
+  1000    8  -8
------------------
 1 0000    0?  0?
CF=1 ZF=1 OF=1 SF=0
```

```
    0101   5    +5
+   0110   6    +6
--------  ---- -----
    1011  11    -5?
CF=0 ZF=0 OF=1 SF=1
```

```
    0111    7    +7
+   0111    7    +7
--------  ----- -----
    1110   14    -2?
CF=0 ZF=0 OF=1 SF=1
```

Carry Flag -- CF

- CF a 1: se há transporte do bit mais significativo
- Interpretação a fazer no programa:
  - uma op. aritmética sobre números sem sinal deu um resultado fora do domínio de valores válidos

```
mov $0xFF, %al      1111 1111      FF  (255)
add $4, %al         0000 0100      04     ( 4)
                    ---------------    -----
                  1 0000 0011    1 03  (259)
```

CF = 1

AL = 0000 0011 (em 8bits, limite até 255)

Overflow Flag -- OF

- OF a 1: se uma op. aritmética sobre números com sinal, dá resultado fora do domínio de valores válidos

```
mov $4, %al         0000 0100      (4)
add 0x7F, %al       0111 1111    (127)
                    ---------------
                    1000  0011   (-125?  sem sinal é 131)
```

→ a soma de dois números positivos não pode dar negativo!

OF = 1 (4+127 > limite de 127)

Nota:

CF = 0 → *em números sem sinal: 4+127=131 < 255*

Sign Flag -- SF

- SF = ao valor do bit mais significativo do resultado, que é o bit de sinal na representação em complemento a 2

SF = 0 se positivo
SF = 1 se negativo

Zero Flag -- ZF

- ZF = 1 se resultado da última operação efectuada pela ALU for zero
- ZF = 0 se o resultado for diferente de zero

```
mov $2, %ax
sub $2, %ax     → ZF = 1

mov $3, %ax
sub  $2, %ax    → ZF = 0
```