

# Computadores e Programação

## Visão Geral dos Sistemas (Aula 2 e 3)

### Sistema de Computador

Consistem de hardware e sistemas de software que funcionam juntos para executar aplicações do usuário. As implementações específicas podem variar com o tempo, mas os conceitos fundamentais não mudam.

### Bits + contexto

Um programa começa com um programa fonte escrito por um programador, em uma linguagem de alto nível e armazenado em um arquivo (*hello.c*)

O programa fonte é uma **sequência de bits** (0s e 1s), organizados em grupos de 8 bits chamados **bytes**.

Cada byte representa um **caractere** alfa numérico (de texto) A maioria dos sistemas modernos representa caracteres usando o padrão ASCII.

Arquivos como *hello.c*, constituídos exclusivamente de caracteres ASCII (i.e., sem formatação), são conhecidos como **arquivos texto**.

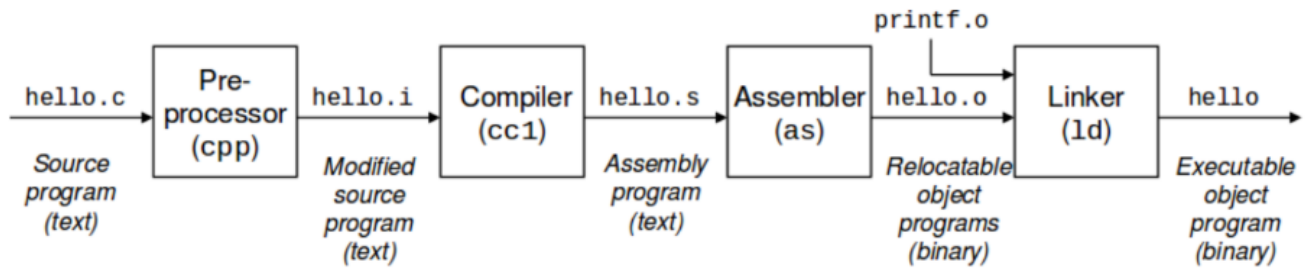
Todos os outros arquivos são conhecidos como **arquivos binários**

- Toda informação em um sistema (incluindo arquivos no disco, programas ou dados do usuário na memória, dados transferidos na rede) é representada como conjunto de bits.
- A única coisa que diferencia é o contexto no qual a informação (conjunto de bits) é vista (ex., a mesma sequência de bits pode ser vista como um número inteiro, uma string, uma instrução de máquina, etc.)

### Tradução de programas

- Para executar o programa *hello* no sistema, as sentenças C devem ser traduzidas por outros programas em uma sequência de instruções de linguagem de máquina
- Essas instruções são então empacotadas no formato chamado programa (ou arquivo) objeto executável e armazenado como arquivo binário no disco
- No UNIX, a tradução de arquivo fonte em arquivo executável pode ser feita pelo programa gcc:

# Sistema de Compilação



A tradução de programas é feita em quatro fases:

1. Pré-processamento (para gerar arquivo fonte com as inclusões especificadas)
2. Compilação (para gerar código de montagem)
3. Montagem (para gerar código binário relocável)
4. Ligação (para gerar código binário executável)

Os programas que executam as quatro fases de tradução são chamados coletivamente de sistema de compilação

## Pré-Processamento

O pré-processador (`gcc -E`) modifica o programa original C tratando as diretivas que começam com o caracter `#`

Ex: a diretiva `# include < stdio.h >` diz para o preprocessor ler o conteúdo do arquivo `stdio.h` e inserir o conteúdo no programa fonte, gerando o arquivo `hello.i`

## Compilador

O compilador (`gcc -S`) traduz o arquivo texto `hello.i` no arquivo texto `hello.s`, o qual contém o programa em linguagem de montagem

Cada sentença no programa em linguagem de montagem descreve exatamente uma instrução de máquina em um formato de texto padrão

A linguagem de montagem provê uma linguagem de saída comum para diferentes compiladores e linguagens de alto nível

## Montagem

O montador (`gcc -c`) traduz o arquivo `hello.s` em instruções de linguagem de máquina, empacotadas no formato conhecido como programa objeto relocável e armazena em um arquivo `hello.o`

Trata-se de um arquivo binário, cujos bytes codificam instruções em linguagem de máquina (ao invés de caracteres)

## Ligação

O programa hello chama a função printf que não é implementada em hello. c (faz parte da biblioteca C padrão)

A função printf reside em um arquivo objeto precompilado chamado printf.o, o qual deve ser incorporado ao programa hello. o (ligação)

O ligador (gcc) trata essa incorporação

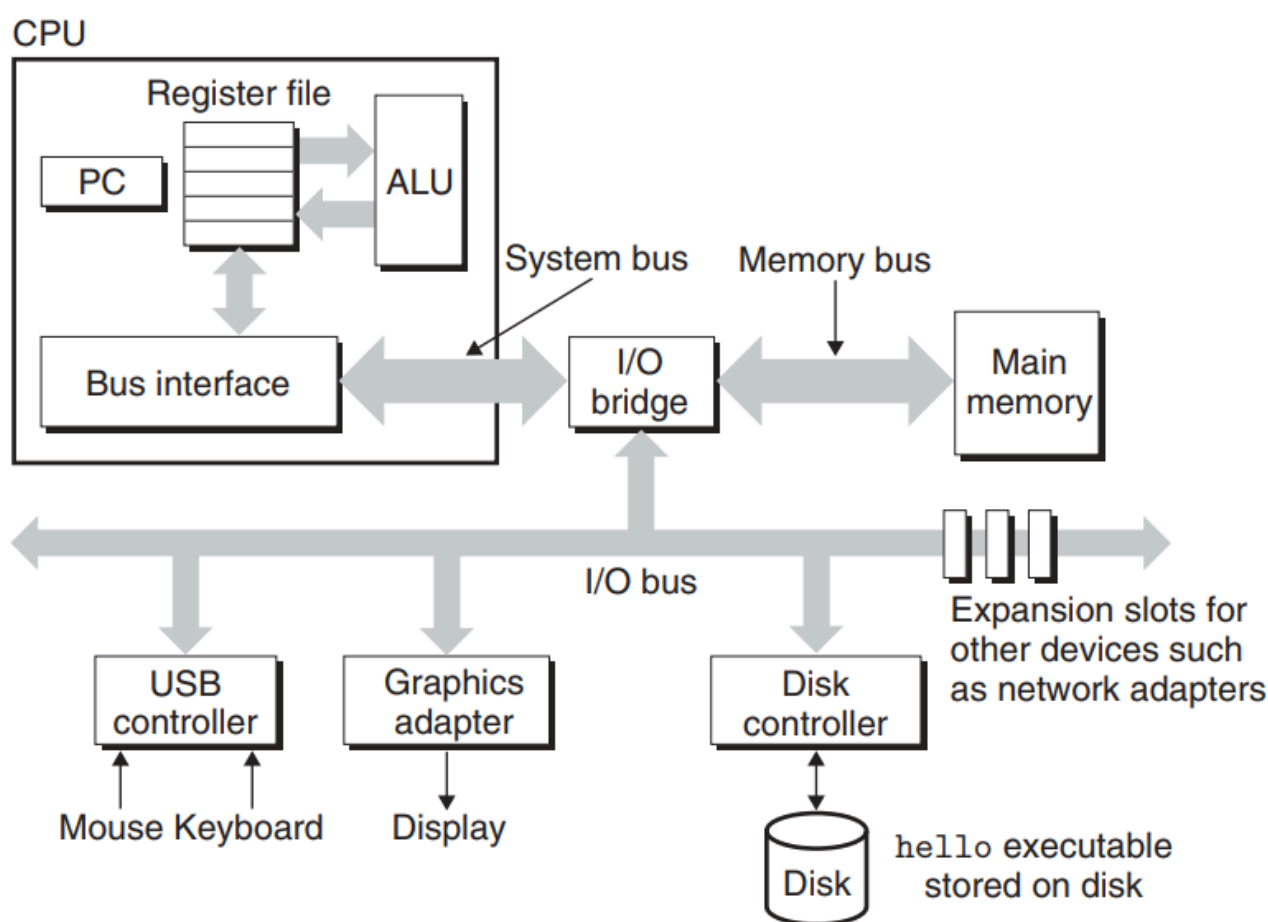
O resultado é o arquivo hello, um arquivo objeto executável que está pronto para ser carregado na memória e executado pelo sistema

## Leitura e interpretação de instruções

Nesse ponto, nosso programa fonte hello. c pode ser executado no shell Unix

O shell é um interpretador de linha de comando que exibe um prompt e espera pela entrada de comandos

## Organização do hardware de um sistema



### Barramentos:

- Condutores elétricos que carregam bytes de informação entre os demais componentes
- São projetados para transferir bytes agrupados em palavras (ex., 4bytes, 8bytes)

### Dispositivos de E/S

- Conexão do sistema com o mundo externo

## **Memória Principal**

- Dispositivo de armazenamento temporário que armazena programas e os dados usados por eles enquanto o processador está executando o programa
- Logicamente a memória é organizada como um vetor de bytes, cada byte com seu próprio endereço Espaço máximo de endereçamento:
  - 4G (232) bytes, sem extensão, arquitetura de 32 bits
  - 16EXA (264) bytes, teórico na arquitetura de 64 bits, embora 256 TERA (248) bytes usado nos processadores x86-64 atuais

## **Processador**

A CPU (Unidade Central de Processamento) é a máquina que interpreta e executa as instruções armazenadas na memória principal

O PC (Ponteiro ou Contador de Programa) contém o endereço da próxima instrução a ser executada em linguagem de máquina

Operações básicas executadas pela CPU:

- carga/armazenamento de dados da memória/ registrador
- operações lógicas/ aritméticas nos registradores
- desvio para outros pontos do programa

## **Uso de Cache**

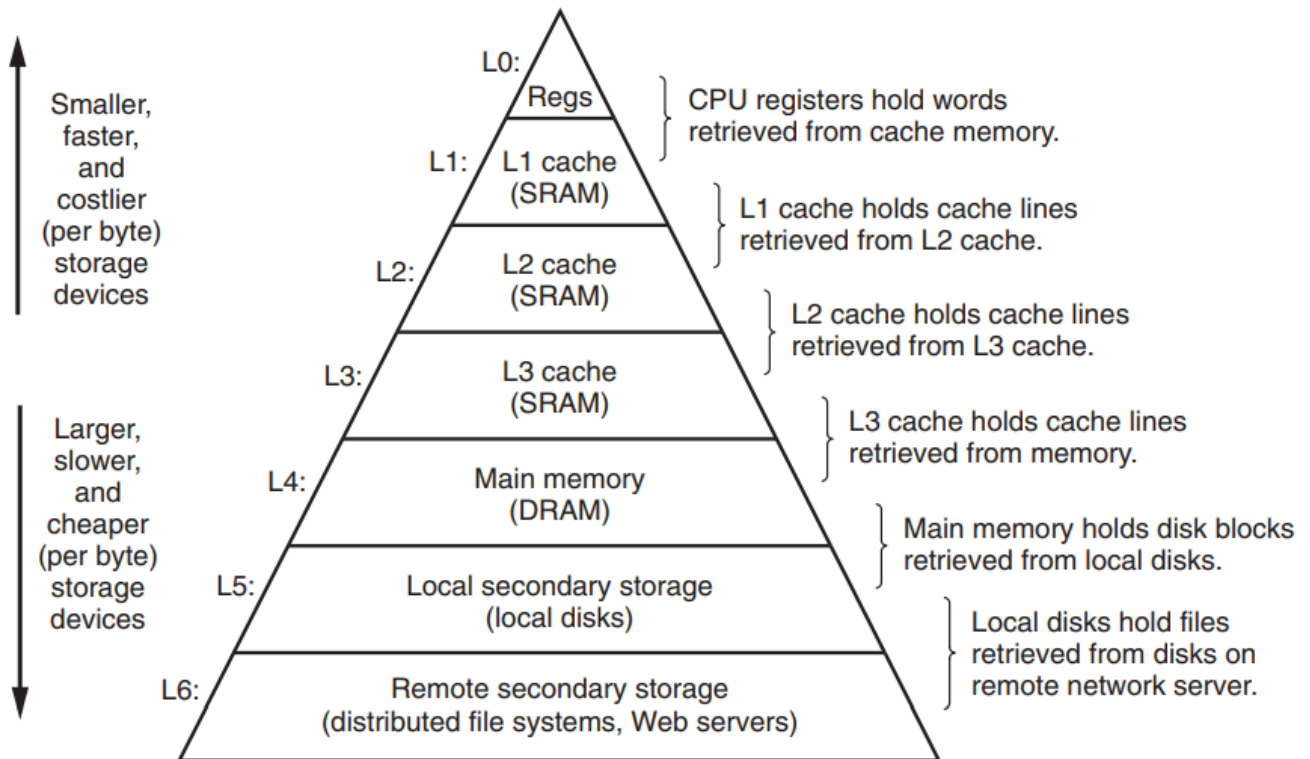
Cache é uma memória bem mas rápida (e cara) que a memória principal e de tamanho bem menor

Idéia básica: duplicar em cache uma área da memória para permitir o acesso mais rápido à informação (que teria que ser buscada na memória mais lenta se a cache não existisse)

Cache só é eficiente se houver localidade (tendência dos programas de acessar instruções e dados em regiões localizadas proximamente)

Acesso e atualizações da cache são feitos pelo hardware de forma

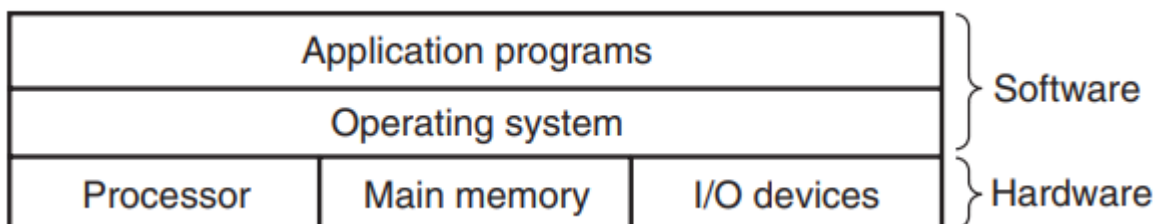
transparente



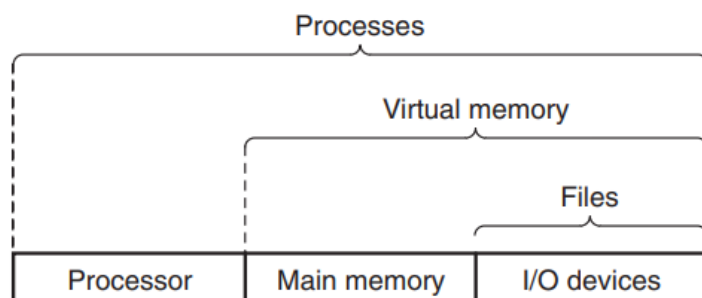
## Funcionalidades de Sistema Operacional

O Sistema Operacional (SO) tem duas funções básicas:

1. Proteger o hardware do uso inapropriado pelas aplicações
2. Prover mecanismos elementares para interação das aplicações com os dispositivos de hardware, simplificando a tarefa dos desenvolvedores de aplicações



**Figure 1.11**  
Abstractions provided by an operating system.



## Processos

Caracteriza um **programa em execução**

Vários processos podem executar **concorrentemente**, i.e., a execução das instruções de um processo pode alternar com a execução das instruções de outro processo num mesmo computador

O SO usa o mecanismo de **troca de contexto** para alternar entre a execução de um processo e outro

Para isso, **informações de contexto** são armazenadas para cada processo (estado das variáveis, ponteiro de programa, pilha de execução, conteúdo da memória, etc.)

## Threads

Threads é o nome dado às diferentes **unidades de execução** de um processo  
As threads compartilham o código e variáveis globais do processo

**Multithreading** é um modelo de programação com importância crescente, devido à demanda por aplicações com melhor desempenho e uso apropriado das arquiteturas multicore

## Memória Virtual

Abstração que dá a cada processo a ilusão de que ele possui uso exclusivo da memória principal

Todo processo tem a mesma visão da memória principal, chamada espaço de endereçamento virtual

O espaço de endereçamento virtual é dividido em áreas de propósito específico:

1. Código e dados: carregados diretamente a partir do conteúdo do arquivo objeto executável
2. Heap: espaço de dados criado dinamicamente (ex., malloc free)
3. Bibliotecas compartilhadas: código e dados para bibliotecas compartilhadas
4. Pilha: uso especial para implementar chamadas de funções (cresce e diminui dinamicamente)
5. Memória virtual do kernel: o kernel é a parte do SO que fica sempre residente na memória em área que as aplicações não podem ler escrever executar diretamente

## Funcionamento

Para que a abstração de memória virtual funcione, é preciso um esquema sofisticado de interação entre o hardware e o SO

Páginas da memória virtual são mapeadas em páginas da memória física  
Requer tradução pelo hardware de todo endereço de memória gerado pelo processador

A idéia básica consiste em armazenar o conteúdo da memória virtual no disco e usar a memória principal como cache para o disco

## Resumo de **Resumo ArqCompS0**

### Abstrações de S0

#### Arquivo

Um arquivo é uma sequência de bytes com um determinado tamanho (em bytes)

Todo dispositivo de E/S (disco, teclado, monitor, rede) é modelado como um arquivo

Toda tarefa de E/S é executada lendo e escrevendo em arquivos através de chamadas de sistema do S0

**Vantagem:** Usa-se primitivas de nível mais alto e não é preciso conhecer a tecnologia de cada dispositivo

#### Redes

Os sistemas de computador modernos estão normalmente ligados entre si através de infraestruturas de redes

Do ponto de vista de um sistema isolado, a rede pode ser vista como mais um dispositivo de E/S

O processador pode copiar dados entre a memória principal e o adaptador de rede (em geral via DMA), permitindo a comunicação entre diferentes máquinas

### Concorrência e Paralelismo

**Concorrência** é o termo usado para referenciar a noção geral de um sistema com atividades simultâneas

**Paralelismo** é o termo usado para referenciar o **uso da concorrência para fazer um sistema executar mais rápido**

O paralelismo pode ser explorado em diferentes níveis de abstração:

- Threads

Vários fluxos de controle dentro do mesmo processo

Os sistemas de computador podem ser: uniprocessador (um único processador sob o controle do S0) ou multiprocessador (vários processadores sob o controle do mesmo S0)

Os sistemas multiprocessador têm se tornado mais comuns com o advento das tecnologias multicore e hyperthreading

Hyperthreading (ou multithreading simultâneo)

Técnica que permite uma única CPU executar vários fluxos de controle de

um mesmo processo

Requer várias cópias de um mesmo hardware (como PC e registradores), enquanto outras partes são compartilhadas (ex., unidade de aritmética de ponto flutuante)

Requer bem menos ciclos de relógio (clock) para chavear entre threads

- Instrução

No passado, vários ciclos de relógio por instrução

Hoje, em processadores superescalares, várias instruções por ciclo de clock, com uso de pipelining

Pipelining

As ações requeridas para executar uma instrução são divididas em diferentes passos e o hardware do processador é organizado como uma série de estágios, cada um executando um desses passos

Hardware pode executar mais de uma instrução simultaneamente, mas com comportamento equivalente ao sequencial

- SIMD (Single-Instruction, Multiple-Data)

No nível mais baixo, os processadores modernos possuem hardware especial que permite uma única instrução disparar várias operações para serem executadas em paralelo (ex., instruções Intel e AMD que permitem somar mais de um par de números fracionários em paralelo)

Maior eficiência para aplicações envolvendo processamento vetorial e matrizes

## Representação e Manipulação da Informação

Os computadores armazenam e processam informações representadas como valores binários

Os circuitos eletrônicos para armazenar, processar e transmitir sinais binários são muito mais simples de construir e manter

Consideraremos as três principais formas de representação de números:

- **Sem sinal:** notação binária tradicional, representando números maiores ou iguais a zero
- **Complemento a 2:** forma mais comum para representar números negativos  
Exemplo:

$$0101 = -0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = -0 + 5 = 5$$

$$1011 = -1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -8 + 3 = -5$$

- **Ponto-flutuante:** versão na base binária da notação científica para representar números reais (veremos detalhes mais a frente)

A representação de números no computador usa uma quantidade limitada de bits

Em razão disso, algumas operações podem causar **estouro** (overflow)



quando os resultados obtidos são maiores que o que é possível representar

Acontece também o arredondamento de valores (aproximações, a serem estudadas com detalhes em cálculo numérico)

## Armazenamento de informações no Computador (Aula 4)

### Endereços de Memória

- Cada byte da memória é identificado por um número único, chamado endereço
- O conjunto de todos os possíveis endereços é chamado espaço de endereçamento virtual (visto pelos programas como um vetor de bytes monolítico)

Por ex., o valor de um ponteiro em C é o endereço virtual do primeiro byte de um bloco contínuo de armazenamento

### Sistema de Numeração

Permite quantificar coisas e é formado por:

- Conjunto de algarismos
- Conjunto de regras que estabelecem como representar quantidades numéricas com os algarismos
- Conjunto de operações que podem ser efetuadas com as quantidades numéricas

### Bases de Potenciação

Os números podem ser representados em qualquer base de potenciação (ou simplesmente base)

Uma base K requer K símbolos diferentes para representar os dígitos de 0 a (K-1)

- O sistema binário possui 2 símbolos: **0 1**
- O sistema octal possui 8 símbolos: **0 1 2 3 4 5 6 7**
- O sistema decimal possui 10 símbolos: **0 1 2 3 4 5 6 7 8 9**
- O sistema hexadecimal possui 16 símbolos: **0 1 2 3 4 5 6 7 8 9 A B C D E F**

### Notação Posicional

A forma geral de representação e interpretação de quantidades numéricas usa a idéia de **valor posicional**

A posição dos algarismos determina o seu valor

O valor do  $i$ -ésimo dígito  $D$  é  $D * Base^i$

$i$  é a posição do algarismo no número, sendo  $i = 0$  a posição menos significativa, à direita

O valor total é a soma dos valores relativos

$$1010_2 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 10_{10}$$

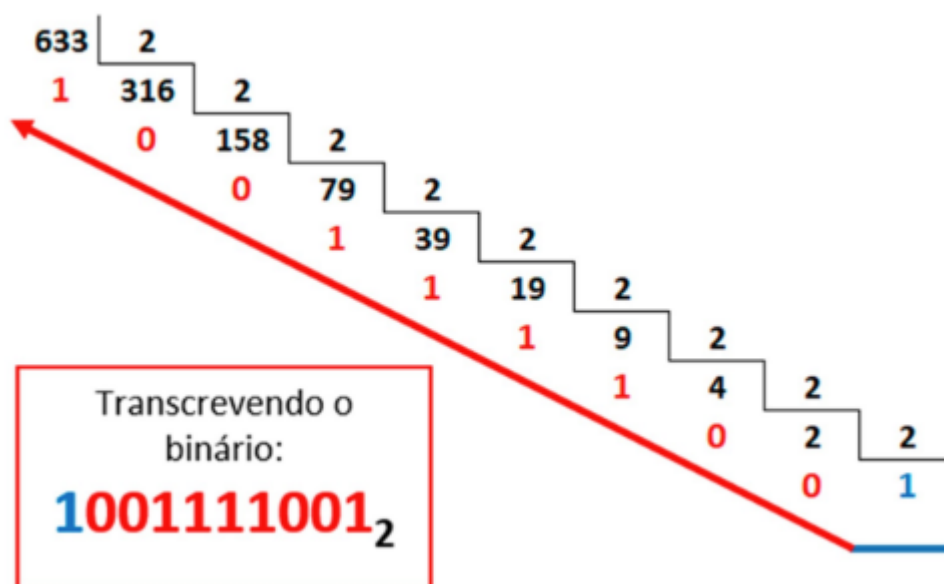
$$1010_8 = 1 * 8^3 + 0 * 8^2 + 1 * 8^1 + 0 * 8^0 = 520_{10}$$

$$1010_{10} = 1 * 10^3 + 0 * 10^2 + 1 * 10^1 + 0 * 10^0 = 1010_{10}$$

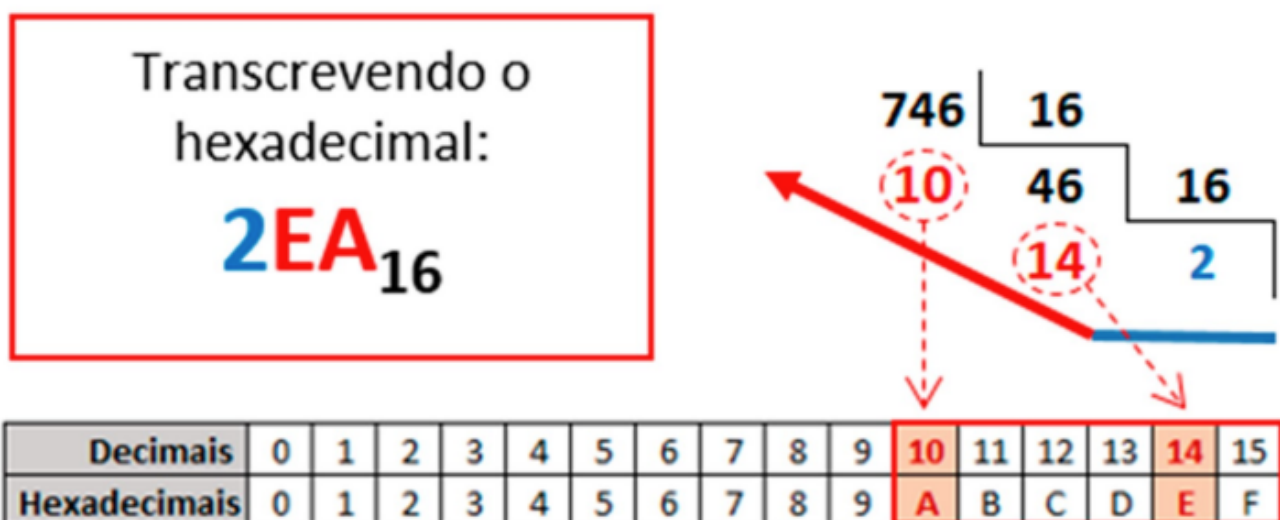
$$1010_{16} = 1 * 16^3 + 0 * 16^2 + 1 * 16^1 + 0 * 16^0 = 4112_{10}$$

## Conversão entre bases

**Decimal** → **Binário**



**Decimal** → **Hexadecimal**



## Binário → Decimal

7	6	5	4	3	2	1	0	Posição
1	1	0	1	1	1	1	0	Algarismo
2	2	2	2	2	2	2	2	Base

$$\begin{array}{cccccccc}
 1 \times 2^7 & + & 1 \times 2^6 & + & 0 \times 2^5 & + & 1 \times 2^4 & + & 1 \times 2^3 & + & 1 \times 2^2 & + & 1 \times 2^1 & + & 0 \times 2^0 \\
 \hline
 128 & + & 64 & + & 0 & + & 16 & + & 8 & + & 4 & + & 2 & + & 0 \\
 \hline
 & & & & & & & & & & & & & & = 222_{10}
 \end{array}$$

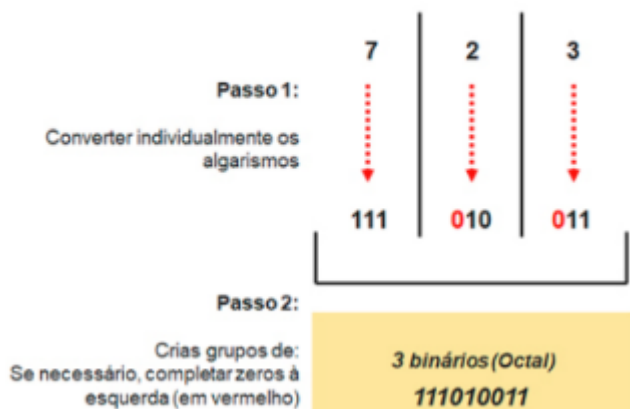
## Hexadecimal → Decimal

3	2	1	0	Posição
1	A	C	2	Algarismo
16	16	16	16	Base

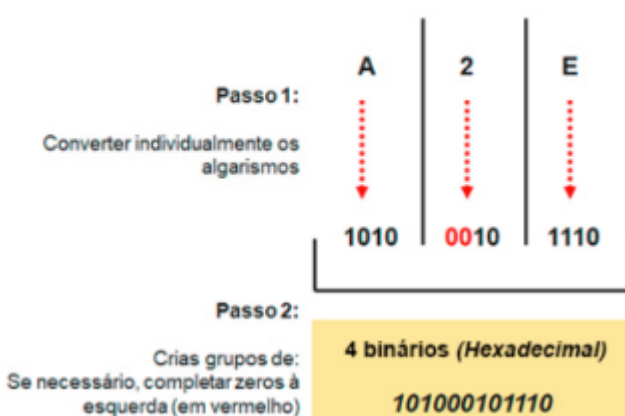
$$\begin{array}{cccc}
 1 \times 16^3 & + & A \times 16^2 & + & C \times 16^1 & + & 2 \times 16^0 \\
 \hline
 1 \times 16^3 & + & 10 \times 16^2 & + & 12 \times 16^1 & + & 2 \times 16^0 \\
 \hline
 & & & & & & 4096 + 2560 + 192 + 2 \\
 \hline
 & & & & & & = 6850_{10}
 \end{array}$$

## Binário, Hexadecimal → Binário

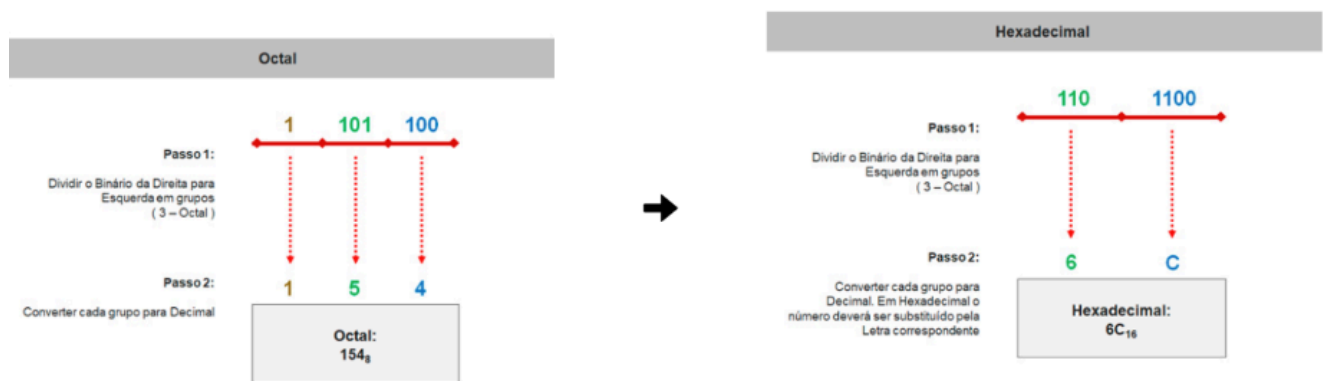
723<sub>8</sub> para a base 2



A2E<sub>16</sub> para a base 2



## Binário → Octal, Hexadecimal



## Tamanho de Palavra e formatos de dados

O tamanho da palavra em um computador indica o tamanho nominal de um número inteiro ou de um ponteiro de dado

Como um endereço virtual é codificado em uma palavra, então o tamanho máximo do espaço de endereçamento virtual para uma máquina com palavra de  $w$  bits é  $2^w$  (endereços indo de 0 a  $(2^w - 1)$ ) Ex: palavra = 32 bits, 232 bytes endereçáveis (4GBytes)

Os computadores e compiladores suportam vários formatos de dados, usando diferentes formas de codificação (ex., inteiros, ponto-flutuante) e diferentes tamanhos (ex., 2, 4, 8 bytes)

## Endereçamento de um objeto

Na quase totalidade das máquinas, um objeto com vários bytes é armazenado em uma sequência contígua de bytes

Dado pelo menor dos endereços dos bytes do objeto

Seja  $x$  uma variável inteira, se  $\&x = 0x100$  (endereço de  $x$ ), os quatro bytes de  $x$  são armazenados nos bytes: 0x100, 0x101, 0x102, 0x103

## Ordenação dos Bytes na memória

## MSB e LSB

Seja um inteiro com representação  $[x_{w-1}, x_{w-2}, x_{w-3}, \dots, x_1, x_0]$

$x_{w-1}$  é o bit mais significativo

(MSB ou MOST SIGNIFICANT BIT)

$x_0$  é o **bit menos significativo**

(LSB ou LEAST SIGNIFICANT BIT)

## Para ordenar os bytes na memória há duas convenções

- 1 **little endian:** os bytes menos significativos primeiro (LSB no endereço mais baixo) (ex., máquinas Intel)
- 2 **big endian:** os bytes mais significativos primeiro (MSB no endereço mais baixo) (ex., máquinas IBM e Sun)

Valor em hexa: 0x01234567

	0x100	0x101	0x102	0x103	
...	01	23	45	67	...

## Big Endian

	0x100	0x101	0x102	0x103	
...	67	45	23	01	...

## Little Endian

## # Representação e armazenamento no computador (Aula 5)

## Operações com bits

A linguagem C oferece operações booleanas que podem ser executadas no nível de bits:

- (or), & (and),  $\sim$  (not),  $\wedge$  (ou-exclusivo)

- ex.,  $\sim 0x41 = \sim 01000001 = 10111110 = 0xBE$
- $0x69 \& 0x55 = 01101001 \& 01010101 = 01000001 = 0x41$
- $0x69 \wedge 0x69 = 0x00$

Um uso comum das operações com bits é na implementação de **máscaras**

Uma **máscara** é um **padrão de bits** que indica um conjunto selecionado de bits dentro de uma palavra

- AND ( `&` ): "E"  $0 \& 1 = 0$ ,  $1 \& 1 = 1$
- OR ( `|` ): "OU"  $0 | 1 = 1$ ,  $1 | 1 = 1$
- NOT ( `~` ) "NEGAÇÃO"  $\sim 0 = 1$ ,  $\sim 1 = 0$
- XOR ( `^` ) "COMPARAÇÃO"  $0 \wedge 1 = 1$ ,  $1 \wedge 1 = 0$

## Operações Lógicas

- `||` → OR lógico
  - `&&` → AND lógico
  - `!` → Complemento Lógico
- As operações lógicas tratam qualquer argumento diferente de ZERO como TRUE e o argumento ZERO como FALSE
- Elas retornam `0x00` ou `0x01` apenas, indicando resultado FALSE ou TRUE, respectivamente

## Operações de Deslocamento de bits em C

### Deslocamento à direita

A expressão  $x \gg K$  desloca K bits a direita

Em geral, as máquinas suportam duas formas de deslocamento à direita:

#### **lógico e aritmético**

No deslocamento à direita **lógico**, zeros são inseridos ao se deslocar

ex.,  $10101111 \gg 4 = 00001010$

No deslocamento à direita **aritmético**, o MSB (bit  $X_n - 1$ ) é copiado a cada deslocamento (**equivale a dividir por  $2^K$ , preservando o sinal do número**)

ex.,  $10101111 \gg 4 = 11111010$

$(-81/16 = -5 - 1/16 = -6)$  (arredondando)

## Representação de Inteiros em C

### Inteiros Positivos

A linguagem C define vários tipos de números inteiros com tamanhos variados: **char, short int, int, long int, long long int**

Além do tamanho, é possível indicar se o número é sempre positivo (unsigned) ou positivo/negativo (o default)

### Inteiros Negativos

A representação mais comum para números negativos é complemento a 2 (C2)

O bit mais significativo da palavra tem peso negativo

ex:  $0001 = -0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 1$

$0110 = -0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 = 6$

$$1010 = -1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = -6$$

$$1111 = -1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = -1$$

Observe que para qualquer valor inteiro  $x$  em  $C2$

$\sim x$  (complementação bit a bit) é equivalente a  $-x - 1$

## Ponto Flutuante

Representa números reais racionais na forma  $V = x.2^Y$

Útil para representar números muito grandes ( $|V| \gg 0$ ), usando números normalizados, ou muito próximos de zero ( $|V| \ll 1$ ), usando números não normalizados

Funciona como uma aproximação para a aritmética real

## Número Binários Fracionários

A notação  $b_m b_{m-1} \dots b_1 b_0 . b_{-1} b_{-2} \dots b_{-n}$  representa o número  $b = \sum_{i=-n}^m 2^i b_i$

### Exemplos

- $101.11_2 = 1.2^2 + 0.2^1 + 1.2^0 + 1.2^{-1} + 1.2^{-2} = 5\frac{3}{4} = \frac{23}{4}$
- Deslocando o ponto duas casas à esquerda equivale a dividir por  $2^2$  levando a  $(101.11_2).2^{-2} = 1.2^0 + 0.2^{-1} + 1.2^{-2} + 1.2^{-3} + 1.2^{-4} = 1.0111_2 = 1\frac{7}{16} = \frac{23}{16}$
- Deslocando o ponto duas casas à direita equivale a multiplicar por  $2^2$  levando a  $(101.11_2).2^2 = 1.2^4 + 0.2^3 + 1.2^2 + 1.2^1 + 1.2^0 = 10111.2 = 23$

$$\begin{array}{ll} .1 & \frac{1}{2} = 1 - \frac{1}{2} = 1 - 2^{-1} \\ .11 & \frac{3}{4} = 1 - \frac{1}{4} = 1 - 2^{-2} \\ .111 & \frac{7}{8} = 1 - \frac{1}{8} = 1 - 2^{-3} \\ \vdots & \end{array}$$

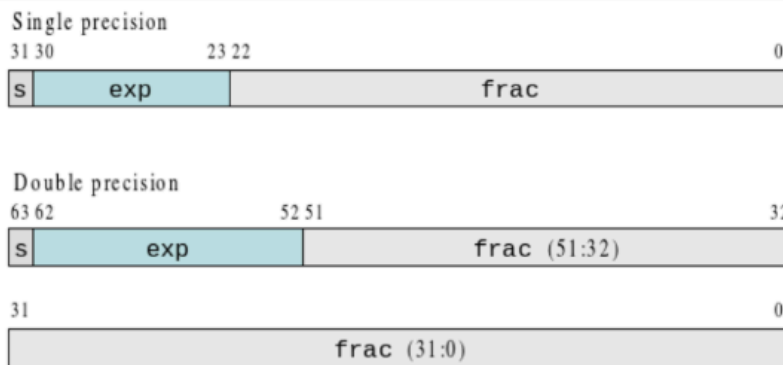
$$\begin{array}{ll} .111 \dots 1(n \text{ dígitos}) & \frac{2^n - 1}{2^n} = 1 - 2^{-n} = 1 - \epsilon \\ 1.111 \dots 1(n \text{ dígitos}) & \frac{2^{n+1} - 1}{2^n} = 2 - 2^{-n} = 2 - \epsilon \end{array}$$

## Ponto Flutuando do IEEE

$$V = (-1)^s \cdot M \cdot 2^E$$

- M = mantissa (entre 1 e  $2 - \epsilon$  ou entre 0 e  $1 - \epsilon$ )
- E = valor da potência de 2

Formatos: precisão simples (*float*) e precisão dupla (*double*)



## Precisão Simples IEEE

### Normalizado

1. Normalized



2. Denormalized



3a. Infinity



3b. NaN



### Não normalizado, Infinito e Não é um Número (NaN)



Não Normalizado:  $\text{exp} = 0$ , significando  $E = -126$

- $M = f$  ( e não há o 1 implícito)
- -126 para transição suave entre normalizados e não normalizados

Infinito:  $\text{exp} = 255$  e  $f = 0$

- $+\infty$  ou  $-\infty$  dependendo do bit de sinal

NaN:  $\text{exp} = 255$  e  $f \neq 0$

Resultado para operação numérica inválida como  $\sqrt{-1}$ ,  $\infty - \infty$

## Precisão Dupla IEEE

Precisão Dupla IEEE: Não Normalizado, Infinito e NaN

Não Normalizado:  $\text{exp} = 0$ , significando  $E = -1022$

- $M = f$  ( e não há o 1 implícito)
- -1022 para transição suave entre normalizados e não normalizados

Infinito:  $\text{exp} = 0x7FF$  e  $f = 0$

- $+\infty$  ou  $-\infty$  dependendo do bit de sinal

NaN:  $\text{exp} = 0x7FF$  e  $f \neq 0$

Resultado para operação numérica inválida como  $\sqrt{-1}$ ,  $\infty - \infty$

## Aula 6 - Representação de programas em linguagem de montagem

### Programas em linguagem de montagem

Ler código em linguagem de montagem gerado por um compilador envolve um **conjunto de habilidades distintas daquelas necessárias para escrever código** diretamente:

é preciso compreender as transformações usadas pelos compiladores para converter construções da linguagem de alto nível em código de máquina

Técnicas de **otimização** usadas pelos compiladores podem rearranjar a ordem de execução do programa:

eliminando computações desnecessárias, substituindo operações lentas, ou mesmo convertendo computações recursivas por sequências iterativas

## Perspectiva histórica das arquiteturas Intel

### Codificação de programas

Dados dois programas, `pl.c` e `p2.c`, eles podem ser compilados fazendo:

```
gcc -m32 -O1 -o p p1. C p2. C
```

A opção `-O1` diz ao compilador para usar o **nível 1** de otimização

O sistema de compilação transforma programas expressos no modelo de execução da linguagem de alto nível em instruções elementares que o processador executa

A habilidade de entender o código de montagem e sua relação com o código fonte é um passo essencial para compreender como os computadores executam os programas

- As instruções IA32 podem ter de 1 a 15 bytes de tamanho
- As instruções são codificadas de tal forma que as operações mais comuns requerem um número menor de bytes comparado às demais
- O formato das instruções é projetado de tal forma que a partir de uma posição inicial há uma única codificação dos bytes em instrução de máquina

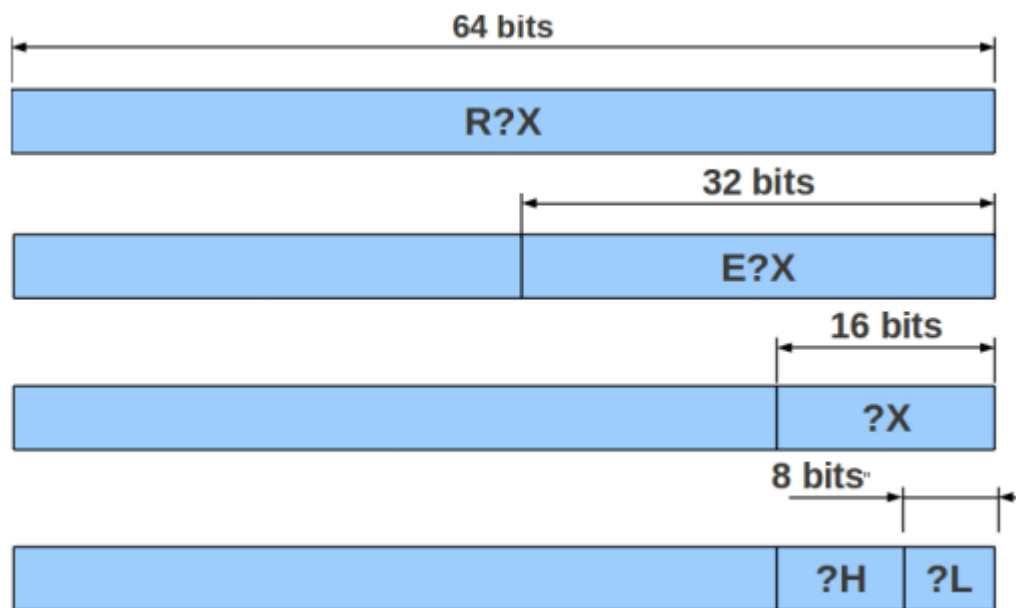
O disassembler determina o código em linguagem de montagem com base apenas na sequência de bytes do código de máquina e na arquitetura para a qual ele foi gerado (não depende da linguagem fonte do programa original)

### ATT versus Intel

- O código de montagem IA32 pode ser mostrado em diferentes formatos
- O formato ATT é o formato padrão das ferramentas gcc e objdump (e será usado ao longo do curso)
- O formato Intel é usado por outras ferramentas (ex., Microsoft), incluindo a própria documentação Intel
- Há várias diferenças entre os dois formatos
- Fazendo `gcc -O1 -S -masm=intel ex11.c` forçamos o uso do formato Intel

## Características da arquitetura x86

## Nomeclatura



## Sufixos nas Instruções de Montagem Formato ATT

C declaration	Intel data type	Assembly code suffix	Size (bytes)
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
long int	Double word	l	4
long long int	—	—	4
char *	Double word	l	4
float	Single precision	s	4
double	Double precision	l	8
long double	Extended precision	t	10/12

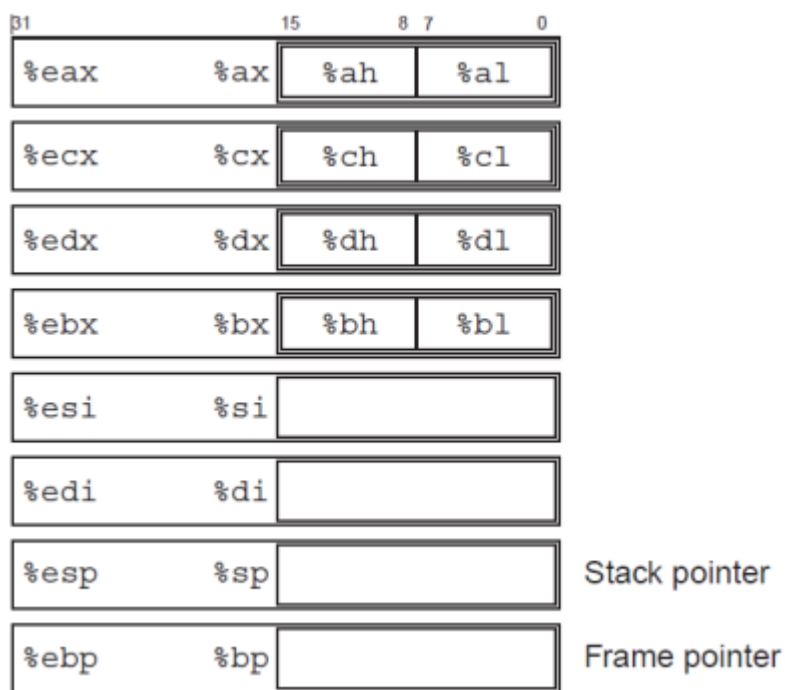
Tem que casar com o tamanho do operando destino

- **b** (byte): `movb %bl, %al`
- **w** (word, 2 bytes): `movw %bx, %ax` (razões históricas para “word”)
- **l** (long, 4 bytes): `movl %ebx, %eax`
- **q** (quad, 8 bytes): `movl %rbx, %rax`

Sufixos para instruções de ponto flutuante

- **s** (single, 32 bits, mantissa com 23 bits)
- **l** (long, 64 bits, mantissa com 52 bits)
- **t** (ten bytes, 80 bits, mantissa com 64 bits)

## Registradores de propósito geral do 80386



Tinham finalidade específica nas arquiteturas anteriores de 16 bits (razão dos nomes), mas, com o endereçamento linear (flat addressing) de 4GB, o uso específico é enormemente reduzido e os **seis primeiros** podem ser considerados de uso geral, ainda que algumas instruções usem registradores fixos como fonte e destino

- **EAX** - Acumulador, usado em operações aritméticas.
- **ECX** - Contador, usado em loops.
- **EDX** - Registrador de dados, usado em operações de entrada/saída e em multiplicações e divisões. É também uma extensão do Acumulador.
- **EBX** - Base, usado para apontar para dados no segmento DS (8086).
- **ESI** - Índice da fonte de dados a copiar (Source Index). Aponta para dados a serem copiados para DS:EDI (segmento DS, posição EDI).
- **EDI** - Índice do destino de dados a copiar (Destination Index). Aponta para o destino dos dados a serem copiados de DS: ESI.
- **ESP** - Apontador da Pilha (Stack Pointer). Aponta para o topo da pilha (endereço mais baixo dos elementos da pilha).
- **EBP** - Apontador da base do frame (registro de ativação). Acesso a argumentos de procedimentos passados pela pilha.

## Registrador de Flags

Armazena códigos de condições setadas por operações lógicas e aritméticas, que podem ser testados por instruções específicas  
x86 não suporta acesso direto ao registrador das flags  
Para modificar ou ler o eflags

Necessário utilizar as instruções privilegiadas pushf (16 bits) ou pushaf (32 bits)

## **Registadores de Segmentos**

- CS - Segmento do Código
- DS - Segmento de Dados
- ES - Segmento com dados extra
- FS - Segmento com mais dados
- GS - Segmento com ainda mais dados
- SS - Segmento da Pilha (Stack)

## **Registadores para Operações de Ponto Flutuante**

Para operações de ponto flutuante existe uma pilha especial em hardware (FPU stack) com 8 registradores, referenciados de ST(0) a ST(7), sendo ST(0) o topo da pilha

Os registradores possuem 80 bits, suportando operações de

- precisão simples (32 bits, sufixo s)
- precisão dupla (64 bits, sufixo l)
- precisão dupla estendida (80 bits, sufixo t)

As operações sobre a pilha FPU podem ser

- unárias, sobre ST (0)
- binárias, com ST (0) e ST0; ou ST(0) e operando em memória

## **Aula 7 - Formatos de operandos e instruções de movimentação de dados IA32**

### **Operandos das Instruções IA32**

#### **Valores de Entrada**

- constantes,
- conteúdo de um registrador, ou o conteúdo armazenado na memória

#### **Localizações de destino**

- registradores ou
- na memória

### **Formatos de operandos**

### 1. Imediato

Constantes numéricas no formato ATT são escritas com \$ seguido de um valor numérico (decimal ou hexadecimal)

- `movl $-20, %eax` :armazena -20 no registrador %eax
- `movl $0x2F, %ebx` :armazena 0x2F no registrador %ebx

Qualquer valor que possa ser representado em 32 bits pode ser usado como constante numérica

### 2. Registrador

Registrador Denota o conteúdo de um dos registradores, ex.:

- `movb $10, %ah`
- `movw $10, %ax`
- `movl $10, %eax`

Veja que o sufixo das instruções e o tamanho do destino (1 byte, 2 bytes ou 4 bytes) têm que ser compatíveis

### 3. Memória

Formato mais geral de referenciar memória:  $I(E_b, E_i, s)$

Endereço de memória =  $I + R[E_b] + R[E_i] * s$

-  $I$  = deslocamento do tipo imediato (mas sem o ) –  $\$R[E_b]$  representa o valor armazenado no registrador base  $E_b$

-  $R[E_i]$  representa o valor armazenado no registrador de índice  $E_i$

-  $s$  é o fator de escala (1, 2, 4 ou 8), relacionado ao tamanho do tipo dos objetos da estrutura

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

Todos os outros formatos são simplificações desse formato geral (com ausência de um ou mais elementos)

O formato completo é usual para referenciar elementos de vetores ou estruturas de dados

Exemplo:

$$260(\%ecx, \%edx) = 0x104 + 0x1 + 0x3 = 0x108 = 0x13$$

$$260_{10} = 104_{16}$$

## Modos de endereçamento

- Absoluto: `movl 17, %eax`
  - $R[\%eax] = \text{Mem}[17]$  (endereço absoluto 17 referenciado)
- Imediato: `movl $17, %eax`
  - $R\%eax = 17$  (valor inteiro constante)
- Indireto: `movl (%ecx), %eax`
  - $R[\%eax] = \text{Mem}[R[\%ecx]]$  (conteúdo de %ecx é o endereço de memória onde está o valor a ser armazenado em %eax)
- Deslocamento: `movl 8(%ebp), %edx`
  - Operando =  $\text{Mem}[R[E_s] + I] = \text{Mem}[R[\%ebp] + 8]$
- Indexado: `movl 16(%ecx, %eax, 4), %edx`
  - Operando =  $\text{Mem}[R[E_b] + s * R[E_i] + I] = \text{Mem}[R[\%ecx] + R[\%eax] * 4 + 16]$

## Instruções de Movimentação de Dados

Instruction		Effect	Description
<b>MOV</b>	<i>S, D</i>	$D \leftarrow S$	Move
<b>movb</b>		Move byte	
<b>movw</b>		Move word	
<b>movl</b>		Move double word	
<b>MOVS</b>	<i>S, D</i>	$D \leftarrow \text{SignExtend}(S)$	Move with sign extension
<b>movsbw</b>		Move sign-extended byte to word	
<b>movsbl</b>		Move sign-extended byte to double word	
<b>movswl</b>		Move sign-extended word to double word	
<b>MOVZ</b>	<i>S, D</i>	$D \leftarrow \text{ZeroExtend}(S)$	Move with zero extension
<b>movzbw</b>		Move zero-extended byte to word	
<b>movzbl</b>		Move zero-extended byte to double word	
<b>movzwl</b>		Move zero-extended word to double word	
<b>pushl</b>	<i>S</i>	$R[\%esp] \leftarrow R[\%esp] - 4;$ $M[R[\%esp]] \leftarrow S$	Push double word
<b>popl</b>	<i>D</i>	$D \leftarrow M[R[\%esp]];$ $R[\%esp] \leftarrow R[\%esp] + 4$	Pop double word

Instruções que copiam dados de uma localização para outra (ex., de um registrador para uma posição de memória)

A generalidade dos formatos de operandos permite que uma mesma instrução básica execute diferentes tipos de movimentações de dados

S = source (fonte)      D = destination (destino)

Copiar um valor de um lugar para outro requer duas instruções, a primeira para carregar o valor na fonte no registrador, e a segunda para gravar esse valor do registrador no destino de mesmo tipo

## Movimentação básica

- **movb S,D**: S→D (move um byte)
  - Qualquer registrador de bit único ( `%ah - %bh` , `%al - %bl` )
- **movw S,D**: S→D (move uma palavra de 16 bits → 2 Bytes)
  - Qualquer registrador de 16bits ( `%ax - %bp` )
- **movl S,D**: S→D (move uma palavra dupla de 32 bits → 5 Bytes)
  - Qualquer registrador de 32bits ( `%eax - %ebp` )
- Destino D tem que ser compatível com o sufixo da instrução!

## Movimentação com extensão do sinal

- **movsbw S,D**: sinalEstendido(S)→D (de byte para palavra)
- **movsbl S,D**: sinalEstendido(S)→D (de byte para palavra dupla)
- **movswl S,D**: sinalEstendido(S) →D (de palavra para palavra dupla)  
Bit de sinal repetido nos bits à esquerda, fazendo com que o valor da representação em C2 se mantenha

## Movimentação com extensão de 0s

- **movzbw S,D**: zeroEstendido(S)D (de byte para palavra)
- **movzbl S,D**: zeroEstendido(S)→D (de byte para palavra dupla)
- **movzwl S,D**: zeroEstendido(S)→D (de palavra para palavra dupla)  
Completa os bits à esquerda com 0

## Operando das instruções

O operando fonte (S) designa um valor imediato (antecedido de \$), ou armazenado em um registrador ou em memória

O operando destino (D) designa uma localização que é um registrador ou uma posição de memória

**Restrição com mov**: Uma posição de memória não pode ser copiada diretamente para outra posição de memória

Precisa-se copiar da memória para um registrador e depois do registrador para a memória

## Pilha

### Operações de 32bit



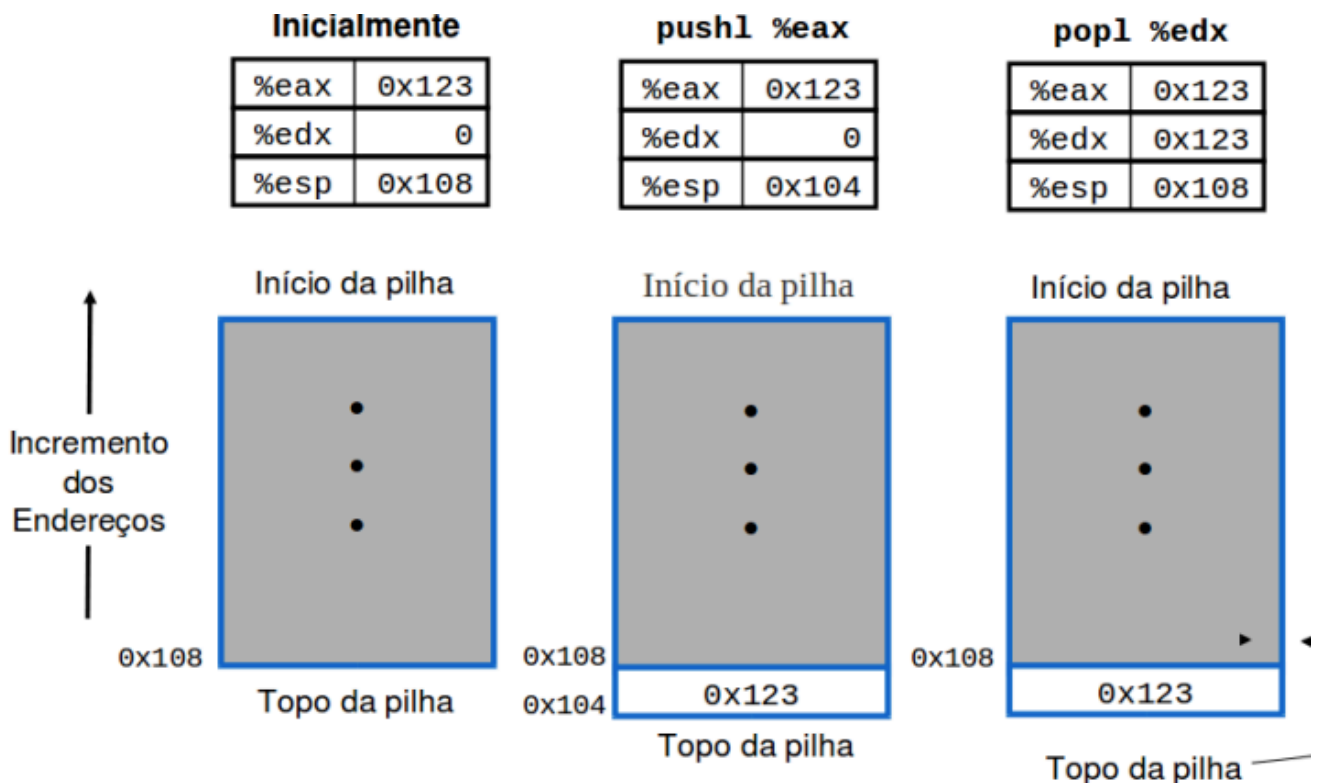
- **pushl S**: abre 4 posições na pilha, fazendo  $\%esp \leftarrow \%esp - 4$ , e atualiza o novo topo com o valor S (4 bytes) ( $M[\%esp] \leftarrow S$ )
- **popl D**: copia 4 bytes do topo para D ( $D \leftarrow M[\%esp]$ ) e decrementa topo, liberando 4 bytes ( $\%esp \leftarrow \%esp + 4$ )

## Equivalências

$$\text{pushl } \%ebp \equiv \begin{cases} \text{subl } \$4, \%esp \\ \text{movl } \%ebp, (\%esp) \end{cases}$$

$$\text{popl } \%ebp \equiv \begin{cases} \text{movl } (\%esp), \%ebp \\ \text{addl } \$4, \%esp \end{cases}$$

Segue regra LIFO (Last-in, first-out) ou "último a entrar, primeiro a sair"



## Variáveis e Ponteiros

Ponteiros em C são endereços no código de montagem.

Desreferenciar (dereferencing) um ponteiro (pegar o valor apontado por ele envolve carregá-lo num registrador e usar esse registrador para referenciar um endereço de memória.

Variável local, como **x**, pode ser mantida em registrador (ao invés da memória), para acesso mais rápido.

## Aula 8 - Operações lógicas e aritméticas IA32

### Operações aritméticas com inteiros

Instruction		Effect	Description
leal	$S, D$	$D \leftarrow \&S$	Load effective address
INC	$D$	$D \leftarrow D + 1$	Increment
DEC	$D$	$D \leftarrow D - 1$	Decrement
NEG	$D$	$D \leftarrow -D$	Negate
NOT	$D$	$D \leftarrow \sim D$	Complement
ADD	$S, D$	$D \leftarrow D + S$	Add
SUB	$S, D$	$D \leftarrow D - S$	Subtract
IMUL	$S, D$	$D \leftarrow D * S$	Multiply
XOR	$S, D$	$D \leftarrow D \wedge S$	Exclusive-or
OR	$S, D$	$D \leftarrow D \mid S$	Or
AND	$S, D$	$D \leftarrow D \& S$	And
SAL	$k, D$	$D \leftarrow D \ll k$	Left shift
SHL	$k, D$	$D \leftarrow D \ll k$	Left shift (same as SAL)
SAR	$k, D$	$D \leftarrow D \gg_A k$	Arithmetic right shift
SHR	$k, D$	$D \leftarrow D \gg_L k$	Logical right shift

As operações lógicas e aritméticas são divididas em quatro grupos:

- Operações unárias: um único operando é fonte e destino da operação  
O operador D pode ser memória ou registrador:
  - `inc D`:  $D+1 \rightarrow D$  (incremento de 1)
  - `dec D`:  $D-1 \rightarrow D$  (decremento de 1)
  - `neg D`:  $-D \rightarrow D$  (negativo do número)
  - `not D`:  $\sim D \rightarrow D$  (complemento do número bit a bit)

Exemplo: `incl (%esp)` → incrementa o inteiro no topo da pilha

- Operações binárias: o segundo operando é usado como fonte e destino da operação

O operando S pode ser: imediato, registrador ou memória.

O operando D pode ser: registrador ou memória

- `add S, D`:  $D + S \rightarrow D$  (adição)
- `sub S, D`:  $D - S \rightarrow D$  (subtração)
- `imul S, D`:  $D * S \rightarrow D$  (multiplicação, resultado em 32 bits)
- `xor S, D`:  $D \wedge S \rightarrow D$  ("ou-exclusivo" lógico bit a bit)
- `or S, D`:  $D \mid S \rightarrow D$  ("ou" lógico bit a bit)
- `and S, D`:  $D \& S \rightarrow D$  ("e" lógico bit a bit)

Exemplo: `subl %eax, %edx` [ $(\%edx - \%eax) \rightarrow \%edx$ ]

# Exemplos

Endereço	Valor
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Registrador	Valor
%eax	0x100
%ecx	0x1
%edx	0x3

## Efeitos das instruções:

- `addl %ecx, (%eax)` ( $\text{Mem}[0x100] = 0xFF + 0x1 = 0x100$ )
- `subl %edx, 4(%eax)` ( $\text{Mem}[0x104] = 0xAB - 0x3 = 0xA8$ )
- `imull $16, (%eax, %edx, 4)` ( $\text{Mem}[0x10C] = 16 \times 0x11 = 272$  (ou  $0x110$ ))

Não pode somar/subtrair memória com memória: `addl (%eax), (%esp)` - não é válido!

- Operações de deslocamento: à esquerda ou à direita, lógico ou aritmético  
Nas operações de deslocamento de bits, a quantidade de bits deslocados é passada no primeiro argumento
  - `sal k, D`:  $D < k \rightarrow D$  (deslocamento aritmético à esquerda)
  - `shi k, D`:  $D < k \rightarrow D$  (deslocamento lógico à esquerda = sal)
  - `sar k, D`:  $D > k \rightarrow D$  (deslocamento aritmético à direita)
  - `shr k, D`:  $D > k \rightarrow D$  (deslocamento lógico à direita)
- Operação de endereço efetivo de carga: copia um endereço de memória para um registrador

- A instrução **leal** (*load effective address*) é uma variante de **movl**
- **leal** tem a forma de uma instrução de movimentação de memória, mas ela não referencia a memória de fato
- O primeiro argumento da instrução referencia uma posição de memória, mas, ao invés de ler o conteúdo dessa posição, a **instrução copia o endereço efetivo de memória para o registrador destino**
- **leal** pode ser usada para gerar **ponteiros de memória**

### Definição e exemplo

```
leal S, D [&S→D];
leal (%edx), %eax [ %edx→%eax]
```

- A instrução **leal** pode ser usada também para descrever operações aritméticas de forma compacta: ex., `leal 7(%edx, %edx, 4), %eax` (assumindo `%edx = x`, temos `%eax = 5x + 7`)

## Aula 9 - Controle do fluxo de execução e instruções condicionais

### Operações Aritméticas Especiais

Instruction		Effect	Description
<code>imull S</code>	<i>S</i>	$R[\%edx]:R[\%eax] \leftarrow S \times R[\%eax]$	Signed full multiply
<code>mull S</code>	<i>S</i>	$R[\%edx]:R[\%eax] \leftarrow S \times R[\%eax]$	Unsigned full multiply
<code>cltd</code>		$R[\%edx]:R[\%eax] \leftarrow \text{SignExtend}(R[\%eax])$	Convert to quad word
<code>idivl S</code>	<i>S</i>	$R[\%edx] \leftarrow R[\%edx]:R[\%eax] \bmod S;$ $R[\%eax] \leftarrow R[\%edx]:R[\%eax] \div S$	Signed divide
<code>divl S</code>	<i>S</i>	$R[\%edx] \leftarrow R[\%edx]:R[\%eax] \bmod S;$ $R[\%eax] \leftarrow R[\%edx]:R[\%eax] \div S$	Unsigned divide

Instrução	Efeito	Descrição
<code>imull S</code>	<code>%edx:%eax ← S x %eax</code>	Mult. completa (64 bits) com sinal
<code>mull S</code>	<code>%edx:%eax ← S x %eax</code>	Mult. completa (64 bits) sem sinal

Instrução	Efeito	Descrição
<code>cld</code>	$\text{\%edx:\%eax} \leftarrow \text{estende sinal}(\text{\%eax})$	Estende 32 bits para 64 bits
<code>idivl S</code>	$\text{\%edx} \leftarrow \text{\%edx:\%eax} \bmod S$ (resto)	Divisão com sinal
	$\text{\%eax} \leftarrow \text{\%edx:\%eax} \div S$ (quociente)	
<code>divl S</code>	$\text{\%edx} \leftarrow \text{\%edx:\%eax} \bmod S$ (resto)	Divisão sem sinal
	$\text{\%eax} \leftarrow \text{\%edx:\%eax} \div S$ (quociente)	

## Registradores de códigos de condição

A CPU mantém registradores de código de condição, setados bit-a-bit, que armazenam atributos das últimas operações lógicas e aritméticas executadas. Códigos de condição são setados como **efeito colateral de uma instrução**.

## Construções Condicionais

Construções condicionais (ex., while, for, if-else) requerem o teste de alguma condição relativa ao valor de variáveis do programa e podem causar um desvio na sequência de instruções.

- **Desvios incondicionais**  
São programados com a instrução `jmp`, desviando a execução para outra parte do programa.
- **Desvios condicionais**  
São baseados em códigos de condição (ou flags) setados nos registradores de códigos de condição.

## Códigos de condição

Os códigos de condição mais comuns são:

1. **CF** (Carry Flag): a última operação gerou um bit extra, usado para detectar overflow (estouro) em operações sem sinal.
2. **ZF** (Zero Flag): o resultado da última operação foi zero.
3. **SF** (Sign Flag): o resultado da última operação foi negativo.
4. **OF** (Overflow Flag): a última operação causou um overflow de operação com sinal.

Obs:

A instrução `leal` não altera nenhum dos códigos de condição (já que seu

uso é para cálculo de endereços de memória)

Nas operações de deslocamento, CF é setado com o último bit deslocado para fora e OF é setado em zero

Nas operações lógicas, CF e OF são setados em zero

**Atenção:** instruções add e dec setam OF e ZF, mas deixam CF inalterado.

## Overflow

Nunca pode acontecer overflow quando somamos dois números de sinais opostos

Quando a soma de dois números do mesmo sinal dá um resultado de sinal oposto ocorre overflow

Overflow também ocorre quando trocamos de sinal o menor negativo inteiro, pois a magnitude resultante não tem representação inteira positiva em complemento a dois

## Classe de instruções TEST e CMP

Instruction		Based on	Description
CMP	$S_2, S_1$	$S_1 - S_2$	Compare
cmpb		Compare byte	
cmpw		Compare word	
cmpq		Compare double word	
TEST	$S_2, S_1$	$S_1 \& S_2$	Test
testb		Test byte	
testw		Test word	
testq		Test double word	

Setam os códigos de condição sem alterar qq outro registrador

- test

Similar à instrução and, mas sem alterar operando destino

**test S1, S2      (testa S2 & S1)**

Uso típico: testl %eax,%eax - para checar se %eax >, < ou = 0  
ou usa um dos operandos como máscara, escolhendo quais bits testar

- cmp

Similar à instrução sub, mas sem alterar operando destino

**cmp S1, S2      (testa S2 - S1)**

## Instruções SET

Instruction	Synonym	Effect	Set condition
<code>sete D</code>	<code>setz</code>	$D \leftarrow ZF$	Equal / zero
<code>setne D</code>	<code>setnz</code>	$D \leftarrow \sim ZF$	Not equal / not zero
<code>sets D</code>		$D \leftarrow SF$	Negative
<code>setns D</code>		$D \leftarrow \sim SF$	Nonnegative
<code>setg D</code>	<code>setnle</code>	$D \leftarrow \sim(SF \wedge OF) \ \& \ \sim ZF$	Greater (signed >)
<code>setge D</code>	<code>setnl</code>	$D \leftarrow \sim(SF \wedge OF)$	Greater or equal (signed >=)
<code>setl D</code>	<code>setnge</code>	$D \leftarrow SF \wedge OF$	Less (signed <)
<code>setle D</code>	<code>setng</code>	$D \leftarrow (SF \wedge OF) \mid ZF$	Less or equal (signed <=)
<code>seta D</code>	<code>setnbe</code>	$D \leftarrow \sim CF \ \& \ \sim ZF$	Above (unsigned >)
<code>setae D</code>	<code>setnb</code>	$D \leftarrow \sim CF$	Above or equal (unsigned >=)
<code>setb D</code>	<code>setnae</code>	$D \leftarrow CF$	Below (unsigned <)
<code>setbe D</code>	<code>setna</code>	$D \leftarrow CF \mid ZF$	Below or equal (unsigned <=)

Exemplo:

```
cmpl b,a
```

```
setl D
```

(*set less*, i.e.,  $D = 1$  se  $a < b$ )

Seja `%edx=a` e `%eax=b`, checar se  $a < b$

```
cmpl %eax, %edx
```

```
setl %al
```

```
movzbl %al, %eax
```

Para obter resultado com 32 bits, a instrução `movzbl %al, %eax` copia `%al` para `%eax` e zera os 24 bits superiores

## Instruções de Desvio (JUMP)

Uma instrução de desvio pode fazer a execução desviar para uma nova posição do programa (rompendo a ordem sequencial de instruções listadas). O endereço da instrução de destino (próxima instrução a ser executada) é normalmente indicado por um label.

Ao gerar o código objeto, o montador determina os endereços dos labels e



decodifica os endereços alvos das instruções de desvio

Variáveis (int) a, b, c, d  
nos registradores %eax, %ebx, %ecx, %edx

if (a==b) c=d; d=a+c;

```
        cmpl %eax, %ebx
        jne depois_if
        movl %edx, %ecx
depois_if: movl %eax, %edx
        addl %ecx, %edx
```

## Instrução jmp

Instruction	Synonym	Jump condition	Description
jmp <i>Label</i>		1	Direct jump
jmp <i>*Operand</i>		1	Indirect jump
j <sub>e</sub> <i>Label</i>	j <sub>z</sub>	ZF	Equal / zero
j <sub>ne</sub> <i>Label</i>	j <sub>nz</sub>	~ZF	Not equal / not zero
j <sub>s</sub> <i>Label</i>		SF	Negative
j <sub>ns</sub> <i>Label</i>		~SF	Nonnegative
j <sub>g</sub> <i>Label</i>	j <sub>nle</sub>	~(SF ^ OF) & ~ZF	Greater (signed >)
j <sub>ge</sub> <i>Label</i>	j <sub>nl</sub>	~(SF ^ OF)	Greater or equal (signed >=)
j <sub>l</sub> <i>Label</i>	j <sub>nge</sub>	SF ^ OF	Less (signed <)
j <sub>le</sub> <i>Label</i>	j <sub>ng</sub>	(SF ^ OF)   ZF	Less or equal (signed <=)
j <sub>a</sub> <i>Label</i>	j <sub>nbe</sub>	~CF & ~ZF	Above (unsigned >)
j <sub>ae</sub> <i>Label</i>	j <sub>nb</sub>	~CF	Above or equal (unsigned >=)
j <sub>b</sub> <i>Label</i>	j <sub>nae</sub>	CF	Below (unsigned <)
j <sub>be</sub> <i>Label</i>	j <sub>na</sub>	CF   ZF	Below or equal (unsigned <=)

A instrução jmp desvia incondicionalmente

Tipo de desvio

- Direto (via label), codificado como parte da instrução (ex., jmp FIM)
- Indireto (via registrador ou posição de memória), codificado com auxílio do operador "\*"

- `jmp * %eax` - usa o conteúdo do registrador como endereço de desvio
- `jmp * (%eax)` - usa o conteúdo da memória endereçada pelo registrador como endereço de desvio

As demais instruções de desvio são condicionais e podem usar apenas destino direto (via label)

- `je` (ou `jz`): ZF (igual/zero)
  - `jne` (ou `jnz`):  $\sim$ ZF (diferente/não-zero)
  - `js`: SF (negativo)
  - `jns`:  $\sim$ SF (não negativo)
  - `jg` (ou `jnle`):  $\sim$ (SF $\wedge$ OF) &  $\sim$ ZF (maior com sinal)<sup>a</sup>
  - `jge` (ou `jnl`):  $\sim$ (SF $\wedge$ OF) (maior ou igual com sinal)
  - `jl` (ou `jnge`): SF $\wedge$ OF (menor com sinal)
  - `jle` (ou `jng`): (SF $\wedge$ OF) | ZF (menor ou igual com sinal)
- <sup>a</sup> ^ é a operação ou-exclusivo (XOR)
- `ja` (ou `jnbe`):  $\sim$ CF &  $\sim$ ZF (acima sem sinal )
  - `jae` (ou `jnb`):  $\sim$ CF (acima ou igual sem sinal)
  - `jb` (ou `jnae`): CF (abaixo sem sinal)
  - `jbe` (ou `jna`): CF | ZF (abaixo ou igual sem sinal )

Estas instruções operam sobre tipos UNSIGNED

## Codificação de Labels

**Relativo ao PC:** codifica a diferença entre o endereço da instrução alvo e o endereço da instrução imediatamente após o JUMP (requer 1, 2 ou 4 bytes)

**Absoluto:** codifica o endereço alvo diretamente (requer 4 bytes)

```
int acumulador = 0;
```

```
int sub(int x, int y) {  
    int t;  
    if (x > y)  
        t = 2*x - y;  
    else  
        t = y - x + 7 ;  
    acumulador += t;  
    return t;  
}
```

```
sub: pushl   %ebp  
      movl   %esp, %ebp  
      movl   8(%ebp), %edx  
      movl   12(%ebp), %ecx  
      cmpl   %ecx, %edx  
      jle    .L2  
      leal   (%edx,%edx), %eax  
      subl   %ecx, %eax  
      jmp    .L3  
.L2: subl   %edx, %ecx  
      leal   7(%ecx), %eax  
.L3: addl   %eax, acumulador  
      popl   %ebp  
      ret
```

## Aula 10 - Tradução de expressões condicionais e repetições para linguagem de montagem

### IF-Else

```

if (test-expr)
    then-statement
else
    else-statement

```

```

t = test-expr
if (!t)
    goto false
then-statement
goto done
false:
    else-statement
done:

```

(a) Original C code

```

1  int absdiff(int x, int y) {
2      if (x < y)
3          return y - x;
4      else
5          return x - y;
6  }

```

(b) Equivalent goto version

```

1  int gotodiff(int x, int y) {
2      int result;
3      if (x >= y)
4          goto x_ge_y;
5      result = y - x;
6      goto done;
7  x_ge_y:
8      result = x - y;
9  done:
10     return result;
11 }

```

(c) Generated assembly code

```

x at %ebp+8, y at %ebp+12
1  movl    8(%ebp), %edx    Get x
2  movl    12(%ebp), %eax   Get y
3  cmpl    %eax, %edx       Compare x:y
4  jge     .L2              if >= goto x_ge_y
5  subl    %edx, %eax        Compute result = y-x
6  jmp     .L3              Goto done
7  .L2:                      x_ge_y:
8  subl    %eax, %edx        Compute result = x-y
9  movl    %edx, %eax        Set result as return value
10 .L3:                      done: Begin completion code

```

## Do-While

```
do
    body-statement
while (test-expr)
```

```
loop:
    body-statement
    t = test-expr
    if (t)
        goto loop
```

(a) C code

```
1  int fact_do(int n)
2  {
3      int result = 1;
4      do {
5          result *= n;
6          n = n-1;
7      } while (n > 1);
8      return result;
9  }
```

(b) Register usage

Register	Variable	Initially
%eax	result	1
%edx	n	<i>n</i>

(c) Corresponding assembly-language code

```
Argument: n at %ebp+8
Registers: n in %edx, result in %eax
1  movl    8(%ebp), %edx    Get n
2  movl    $1, %eax        Set result = 1
3  .L2:                    loop:
4  imull    %edx, %eax      Compute result *= n
5  subl    $1, %edx        Decrement n
6  cmpl    $1, %edx        Compare n:1
7  jg      .L2             If >, goto loop
Return result
```

## While

```
while (test-expr)
    body-statement
```

```

    t = test-expr
    if (!t)
        goto done
loop:
    body-statement
    t = test-expr
    if (t)
        goto loop
done:
```

(a) C code

```

1  int fact_while(int n)
2  {
3      int result = 1;
4      while (n > 1) {
5          result *= n;
6          n = n-1;
7      }
8      return result;
9  }
```

(b) Equivalent goto version

```

1  int fact_while_goto(int n)
2  {
3      int result = 1;
4      if (n <= 1)
5          goto done;
6      loop:
7          result *= n;
8          n = n-1;
9          if (n > 1)
10             goto loop;
11     done:
12         return result;
13 }
```

(c) Corresponding assembly-language code

```

Argument: n at %ebp+8
Registers: n in %edx, result in %eax
1  movl    8(%ebp), %edx    Get n
2  movl    $1, %eax        Set result = 1
3  cmpl    $1, %edx        Compare n:1
4  jle     .L7             If <=, goto done
5  .L10:                  loop:
6  imull    %edx, %eax      Compute result *= n
7  subl    $1, %edx        Decrement n
8  cmpl    $1, %edx        Compare n:1
9  jg      .L10            If >, goto loop
10 .L7:                  done:
    Return result
```

## For

```

int fat_for (int n) {
    int i;
    int ret = 1;
    for (i=2; i<=n; i++) {
        ret *= i;
    }
    return ret;
}

```

```

fat_for:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %ecx
    movl    $1, %eax
    movl    $2, %edx
    cmpl    $1, %ecx
    jle     .L3
.L6: imull  %edx, %eax
    addl    $1, %edx
    cmpl    %edx, %ecx
    jge     .L6
.L3: popl   %ebp
    ret

```

```

for (init-expr; test-expr;
      update-expr)
    body-statement

```

```

    init-expr
    t = test-expr
    if (!t)
        goto done
loop:
    body-statement
    update-expr
    t = test-expr
    if (t)
        goto loop
done:

```

## CMOV

Copia um valor para um registrador, dependendo de valores dos códigos de condição

Instruction	Synonym	Move condition	Description
<code>cmove S, R</code>	<code>cmovz</code>	ZF	Equal / zero
<code>cmovne S, R</code>	<code>cmovnz</code>	$\sim$ ZF	Not equal / not zero
<code>cmovs S, R</code>		SF	Negative
<code>cmovns S, R</code>		$\sim$ SF	Nonnegative
<code>cmovg S, R</code>	<code>cmovnle</code>	$\sim$ (SF ^ OF) & $\sim$ ZF	Greater (signed >)
<code>cmovge S, R</code>	<code>cmovnl</code>	$\sim$ (SF ^ OF)	Greater or equal (signed >=)
<code>cmovl S, R</code>	<code>cmovnge</code>	SF ^ OF	Less (signed <)
<code>cmovle S, R</code>	<code>cmovng</code>	(SF ^ OF)   ZF	Less or equal (signed <=)
<code>cmova S, R</code>	<code>cmovnbe</code>	$\sim$ CF & $\sim$ ZF	Above (unsigned >)
<code>cmovae S, R</code>	<code>cmovnb</code>	$\sim$ CF	Above or equal (Unsigned >=)
<code>cmovb S, R</code>	<code>cmovnae</code>	CF	Below (unsigned <)
<code>cmovbe S, R</code>	<code>cmovna</code>	CF   ZF	below or equal (unsigned <=)

## Switch Case

### Switch em C

```
int switch_eg(int x, int n) {
    int result = x;
    switch (n){
        case 100:
            result *= 13;
            break
        case 102:
            result += 11;
            break;
        case 104:
        case 106:
            result *= result;
            break;
        default:
            result = 0;
    }
    return result;
}
```

### Switch em assembly



```

    x at %ebp+8, n at %ebp+12
1    movl    8(%ebp), %edx          Get x
2    movl    12(%ebp), %eax        Get n
    Set up jump table access
3    subl    $100, %eax            Compute index = n-100
4    cmpl    $6, %eax              Compare index:6
5    ja      .L2                    If >, goto loc_def
6    jmp     *.L7(, %eax, 4)        Goto *jt[index]
    Default case
7    .L2:                          loc_def:
8    movl    $0, %eax              result = 0;
9    jmp     .L8                    Goto done
    Case 103
10   .L5:                          loc_C:
11   movl    %edx, %eax             result = x;
12   jmp     .L9                    Goto rest
    Case 100
13   .L3:                          loc_A:
14   leal    (%edx,%edx,2), %eax     result = x*3;
15   leal    (%edx,%eax,4), %eax     result = x+4*result
16   jmp     .L8                    Goto done
    Case 102
17   .L4:                          loc_B:
18   leal    10(%edx), %eax          result = x+10
    Fall through
19   .L9:                          rest:
20   addl    $11, %eax              result += 11;
21   jmp     .L8                    Goto done
    Cases 104, 106
22   .L6:                          loc_D
23   movl    %edx, %eax              result = x
24   imull   %edx, %eax              result *= x
    Fall through
25   .L8:                          done:
    Return result

```