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.)

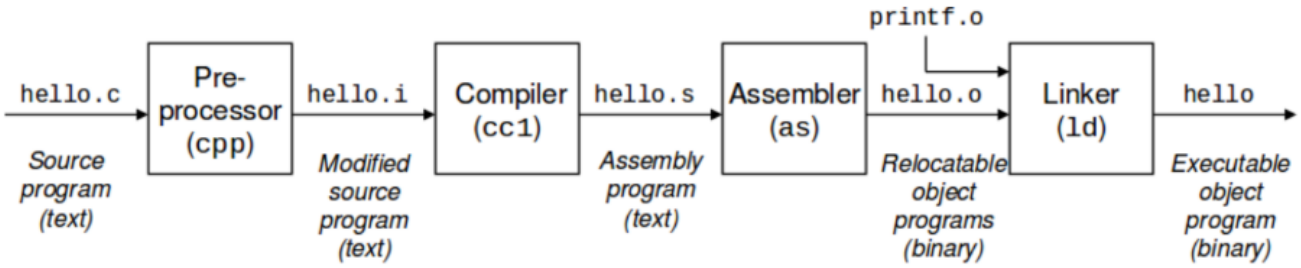
Tradução de programas

- 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 preprocessador 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)

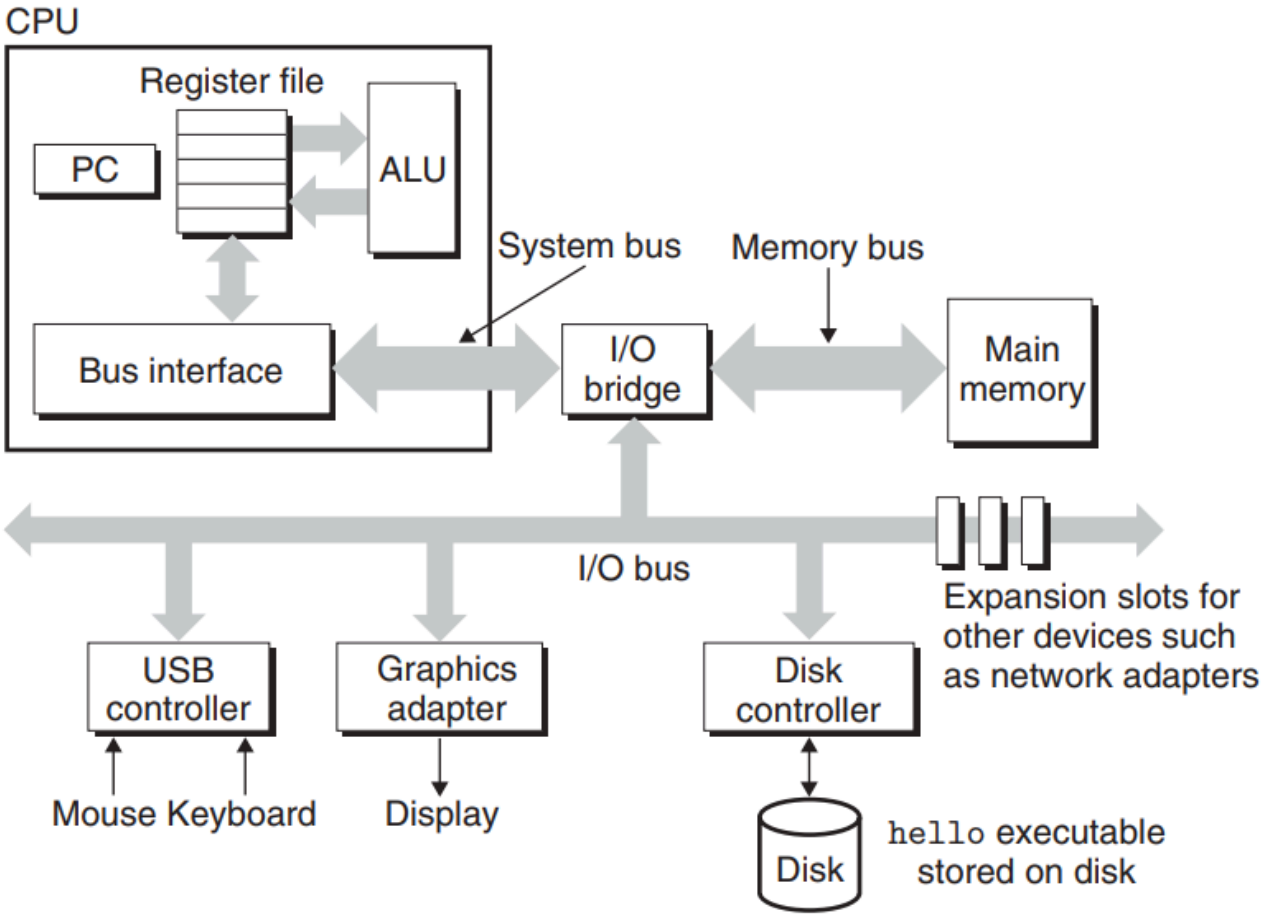
Ligação

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

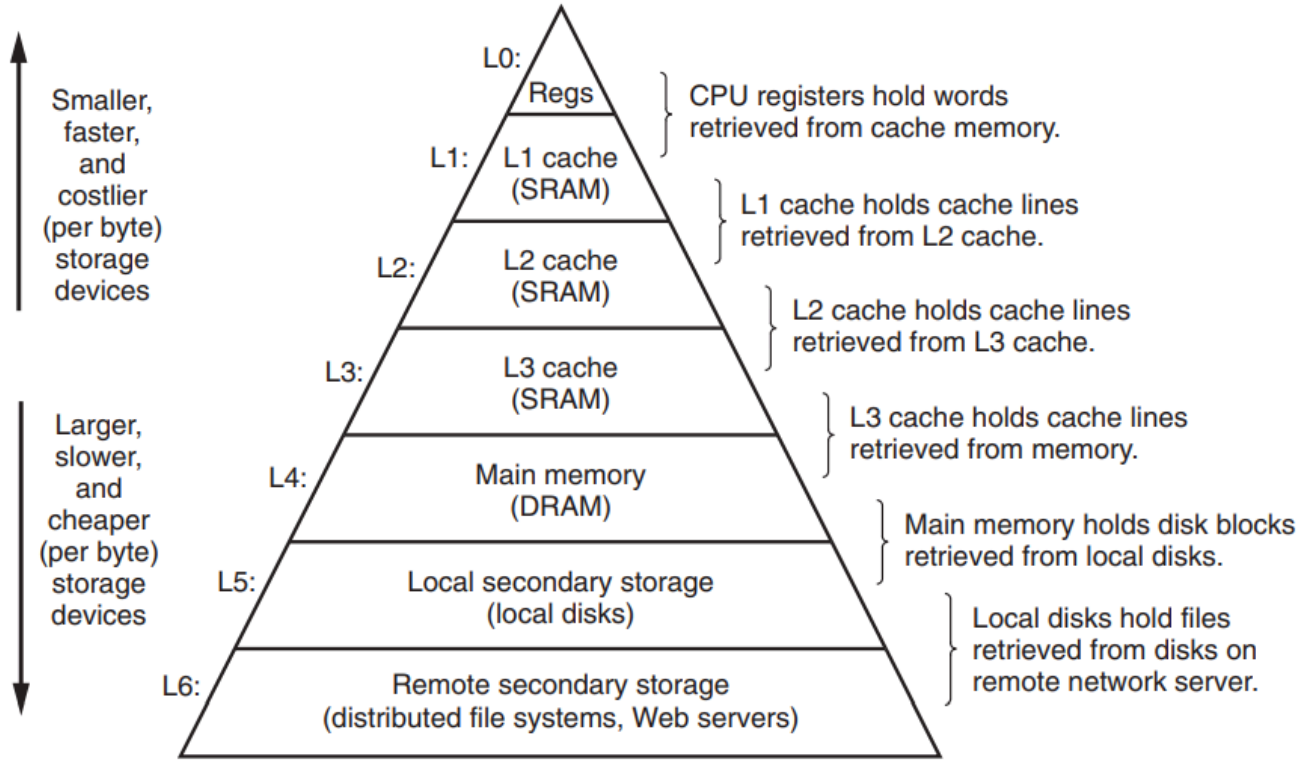
Processador

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. O 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

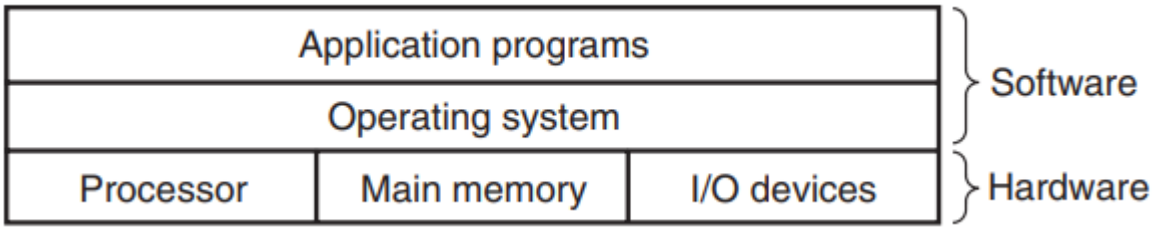
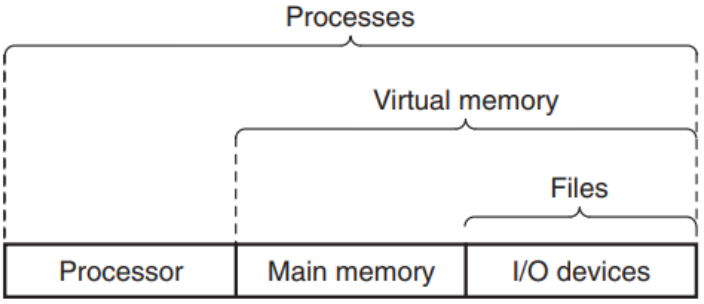


Figure 1.11
Abstractions provided by an operating system.



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 ArqCompSO

Abstrações de SO

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 SO
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 abstracão:

- Threads
  Vários fluxos de controle dentro do mesmo processo
  Os sistemas de computador podem ser: uniprocessador (um único processador sob o controle do SO) ou multiprocessador (vários processadores sob o controle do mesmo SO)
  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
- Instrucão
  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 numeros:

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

$$0101 = -0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = -0 + 5 = 5$$
$$1011 = -1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^1 = -8 + 3 = -5$$

- **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 pontenciação (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

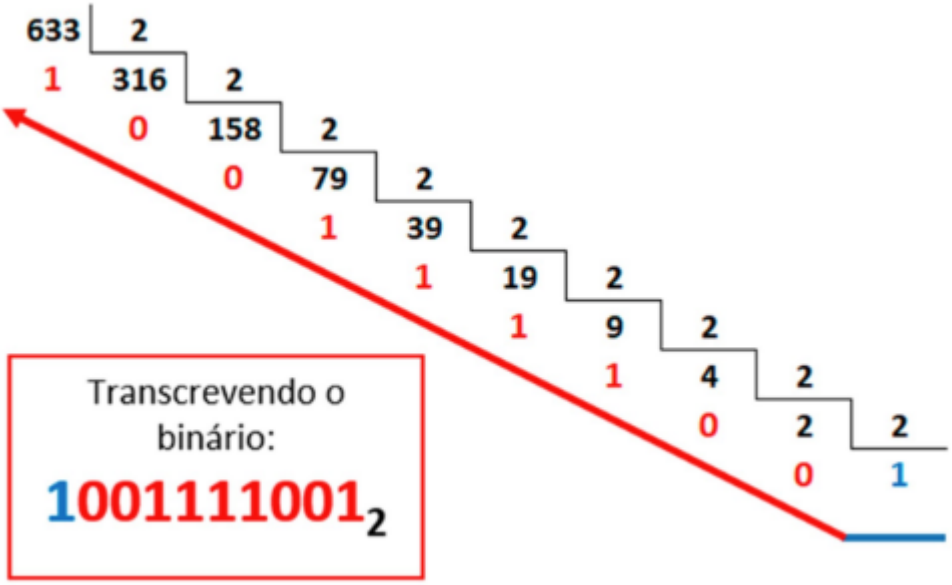
A forma geral de representação e interpretação de quantidades numéricas usa a idéia de **valor posicional**
A posição dos algarismos determina o seu valor
O valor do i-ésimo dígito D é $D * Base^i$
i é a posição do algarismo no número, sendo i = 0 a posição menos significativa, à direita

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

Decimais	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimais	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

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

$$1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

$$128 + 64 + 0 + 16 + 8 + 4 + 2 + 0$$

$$= 222_{10}$$

Hexadecimal → Decimal

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

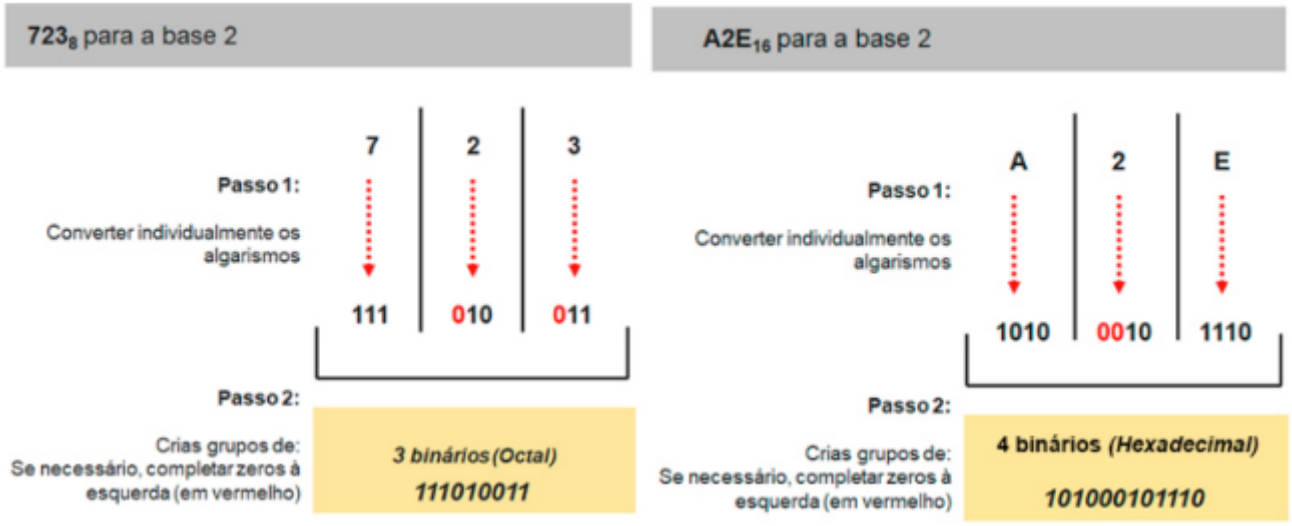
$$1 \times 16^3 + A \times 16^2 + C \times 16^1 + 2 \times 16^0$$

$$1 \times 16^3 + 10 \times 16^2 + 12 \times 16^1 + 2 \times 16^0$$

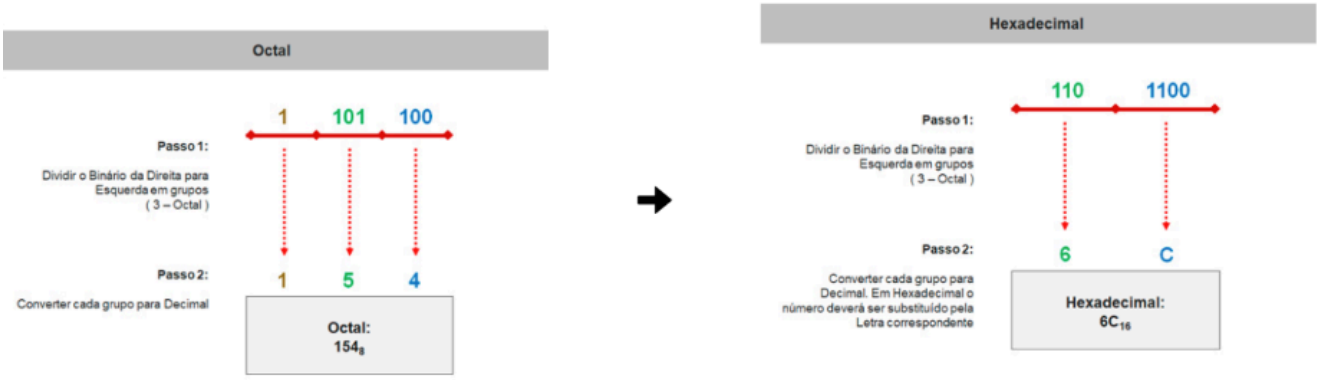
$$4096 + 2560 + 192 + 2$$

$$= 6850_{10}$$

Binário, Hexadecimal → Binário



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

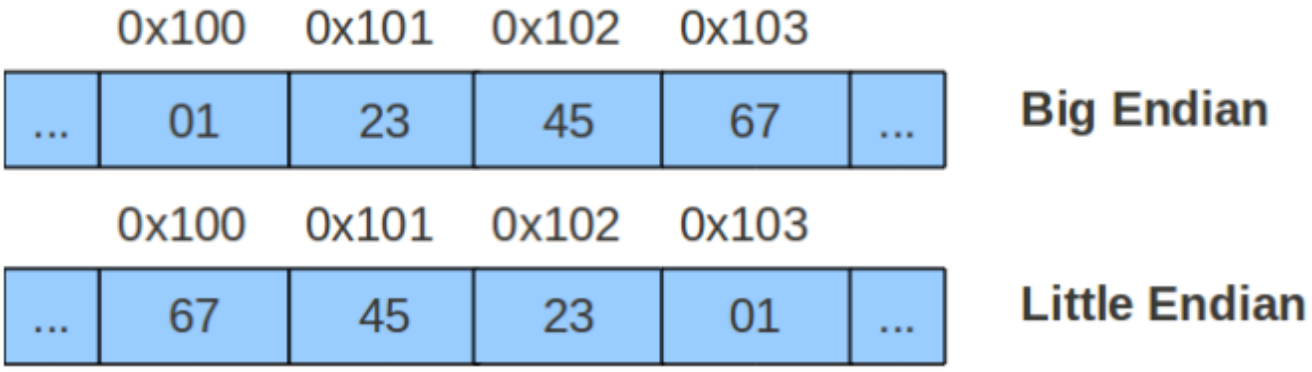
MSB e LSB

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



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), ~ (not), ∧ (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 (`~`) "NEGAÇÂO" $\sim 0 = 1$, $\sim 1 = 0$
- XOR (`^`) "COMPARAÇÂO" $0 \wedge 1 = 1$, $1 \wedge 1 = 0$

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 $b_m b_{m-1} \cdots b_1 b_0 . b_{-1} b_{-2} \cdots b_{-n}$ 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 $(101.11_2).2^2 = 1.2^4. + 0.2^3 + 1.2^2 + 1.2^1 + 1.2^0 = 10111._2 = 23$

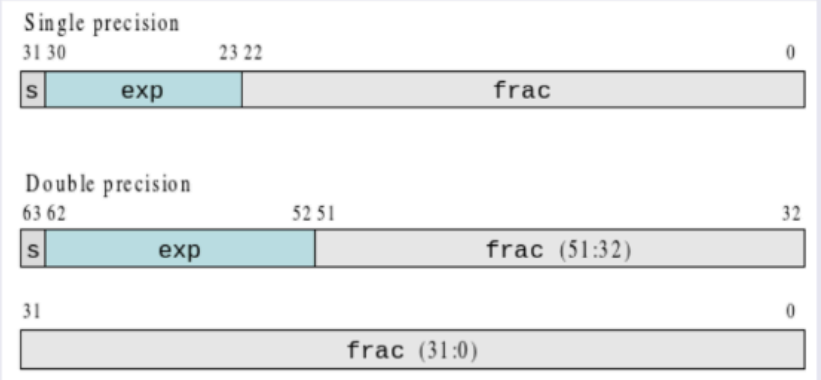
$$
\begin{aligned}
.1 \quad & \frac{1}{2} = 1 - \frac{1}{2} = 1 - 2^{-1} \\
.11 \quad & \frac{3}{4} = 1 - \frac{1}{4} = 1 - 2^{-2} \\
.111 \quad & \frac{7}{8} = 1 - \frac{1}{8} = 1 - 2^{-3} \\
& \vdots \\
.111\cdots1(\text{n dígitos}) \quad & \frac{2^n-1}{2^n} = 1 - 2^{-n} = 1 - \epsilon \\
1.111\cdots1(\text{n dígitos}) \quad & \frac{2^{n+1}-1}{2^n} = 2 - 2^{-n} = 2 - \epsilon
\end{aligned}
$$

Ponto Flutuando do IEEE

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

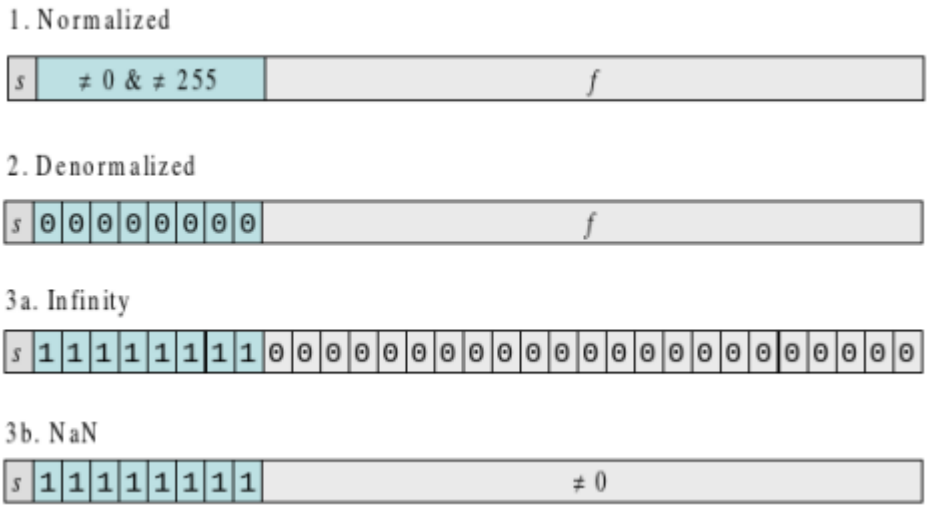
- M = mantissa (entre 1 e $2 - \epsilon$ ou entre 0 e $1 - \epsilon$)
- E = valor da potência de 2

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



Precisão Simples IEEE

Normalizado



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

Não Normalizado: exp = 0, significando E = -126

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

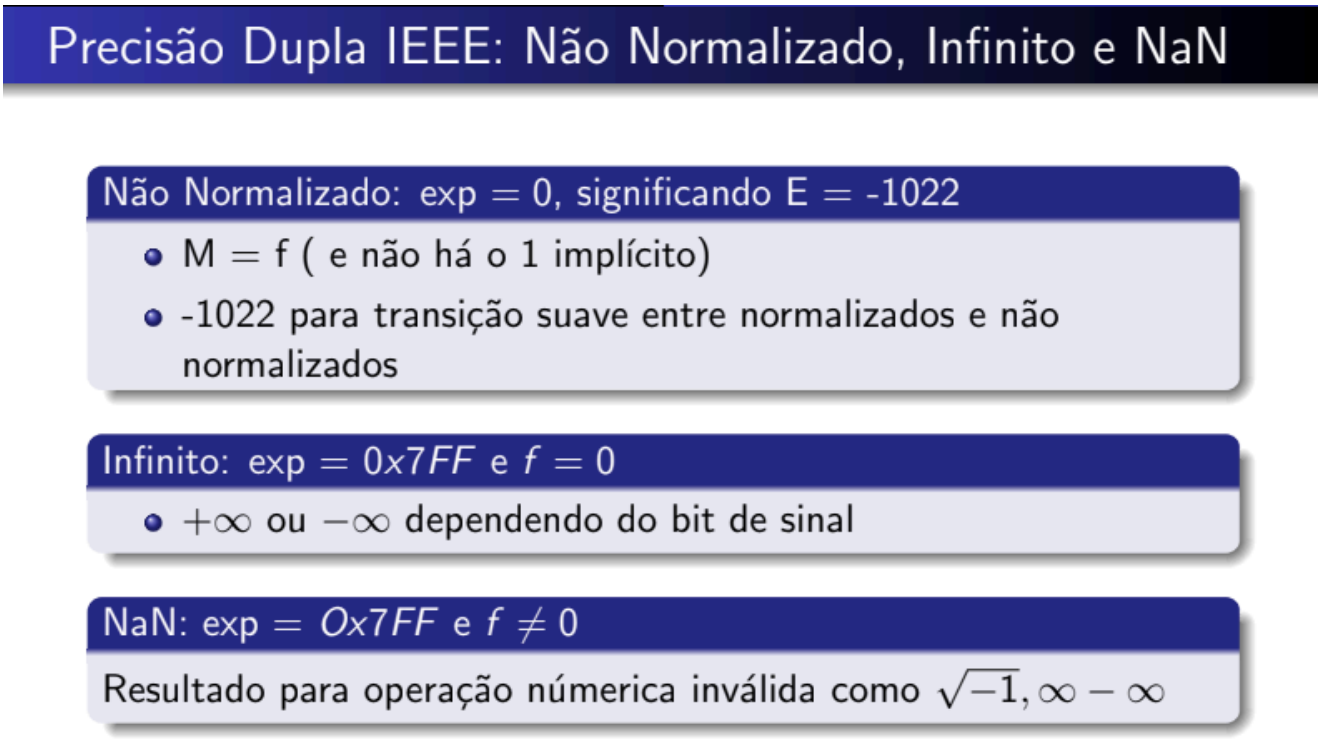
Infinito: exp = 255 e $f = 0$

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

NaN: exp = 255 e $f \neq 0$

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

Precisão Dupla IEEE



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 -01 -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

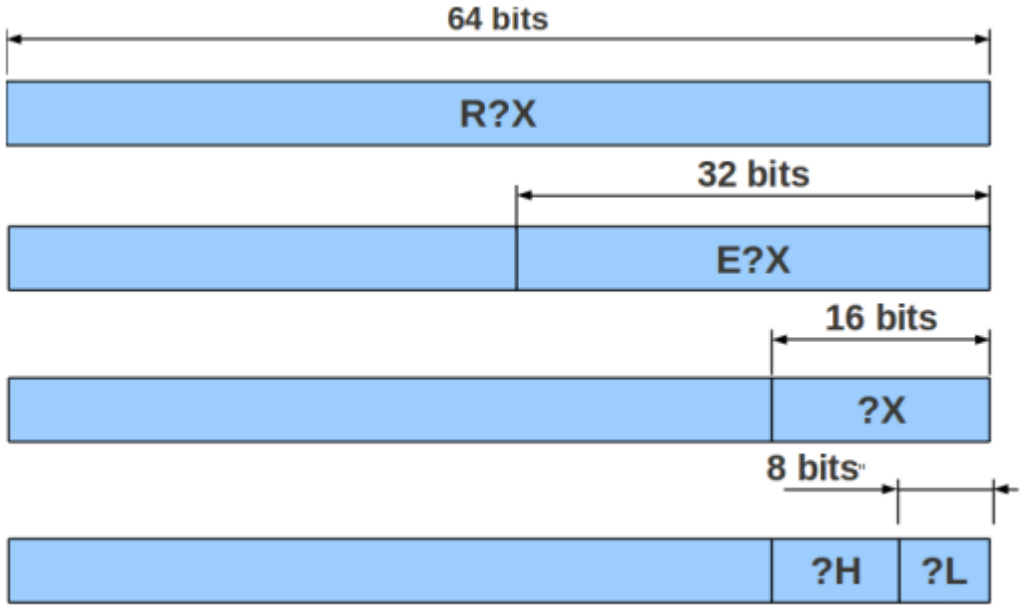
O disassembler determina o código em linguagem de montagem com base apenas na sequência de bytes do código de máquina e na arquitetura para a qual ele foi gerado (não depende da linguagem fonte do programa original)

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

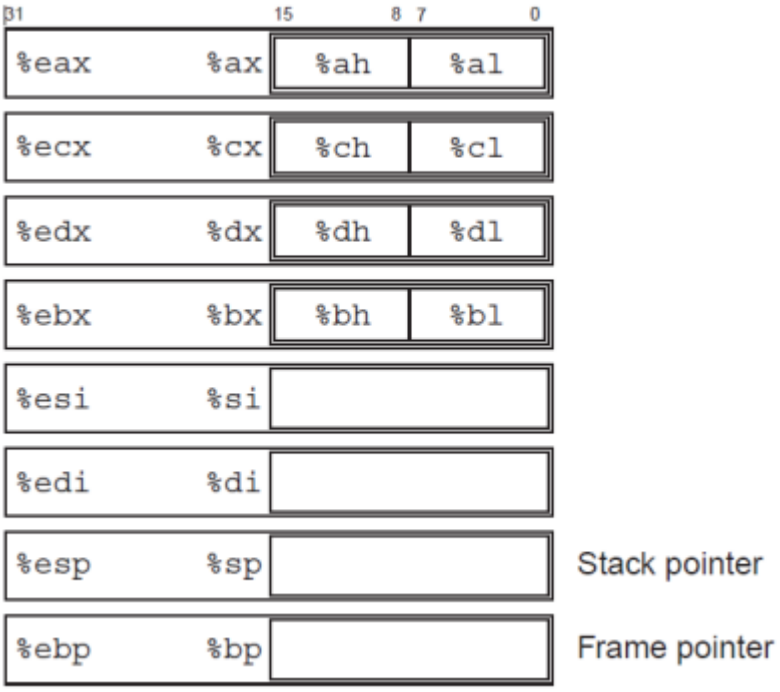
Tem que casar com o tamanho do operando destino

- **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** - Indice 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 operacões 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)

Registradores 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)

Registradores 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 registador %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)
0 formato completo é usual para referenciar elementos de vetores ou estruturas de dados

Exemplo:

$$260(\%\text{ecx}, \%\text{edx}) = \quad 0\text{x}104 + 0\text{x}1 + 0\text{x}3 = 0\text{x}108 = 0\text{x}13$$
$$260_{10} = 104_{16}$$

Modos de endereçamento

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

Instruções de Movimentação de Dados

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

$$\text{S} = \text{source (fonte)} \qquad \text{D} = \text{destination (destino)}$$

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

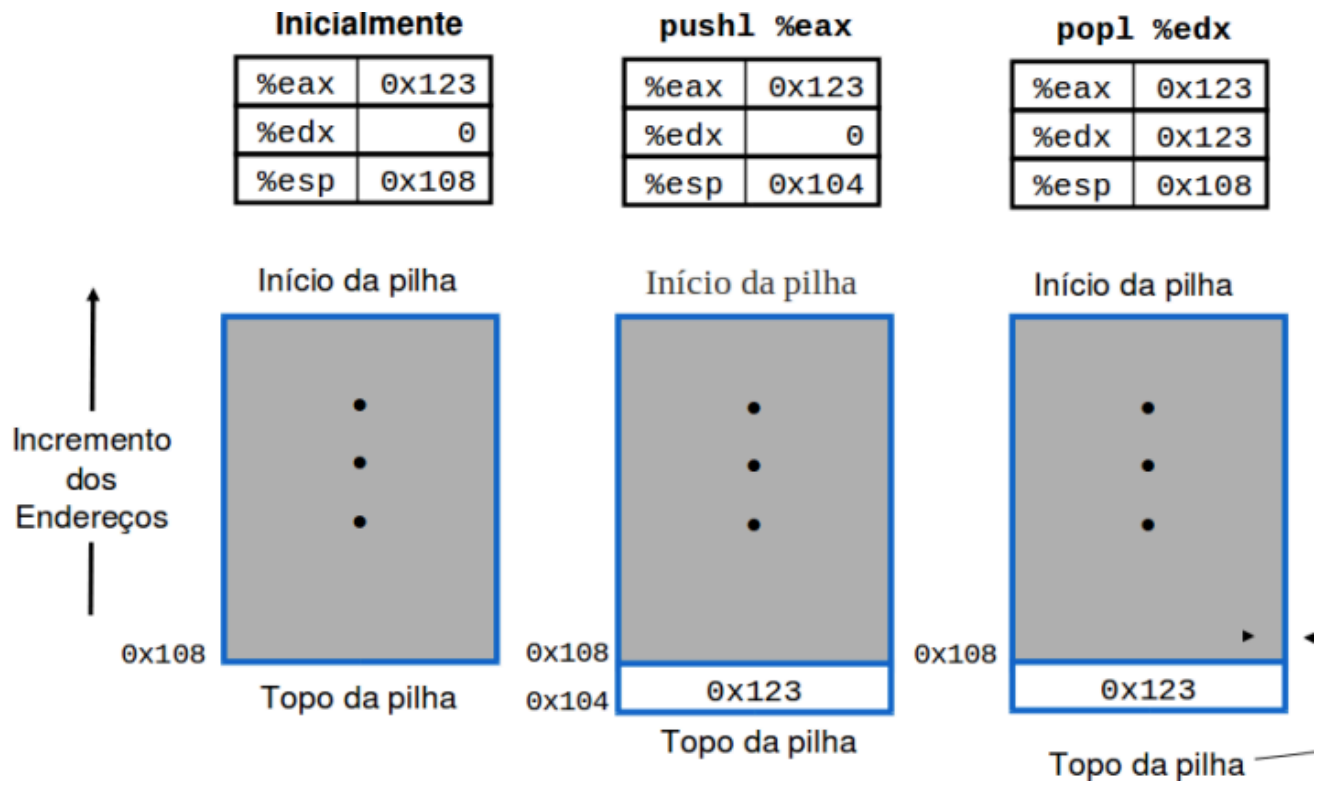
- **pushl S**: abre 4 posições na pilha, fazendo %esp←%esp-4, e atualiza o novo topo com o valor S (4 bytes) (M[%esp] ←S)
- **popl D**: copia 4 bytes do topo para D (D←M[%esp]) e decrementa topo, liberando 4 bytes (%esp←%esp+4)

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	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \& S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	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→D (incremento de 1)
  - `dec D`: D-1→D (decremento de 1)
  - `neg D`: -D→D (negativo do número)
  - `not D`: ~D→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→D (adição)
  - `sub S, D`: D - S→D (subtração)
  - `imul S, D`: D * S→D (multiplicação, resultado em 32 bits)
  - `xor S, D`: D^S→D ("ou-exclusivo" lógico bit a bit)
  - `or S, D`: D | S→D ("ou" lógico bit a bit)
  - `and S, D`: D & S→D ("e" lógico bit a bit)

Exemplo: subl %eax, %edx [(%edx - %eax) → %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) (Mem[0x100] = 0xFF + 0x1 = 0x100)
- subl %edx,4(%eax) (Mem[0x104] = 0xAB - 0x3 = 0xA8)
- imull $16,(%eax, %edx, 4) (Mem[0x10C] = 16 x 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→D (deslocamento aritmético à esquerda)
  - `shi k, D`: D < k→D (deslocamento lógico à esquerda = sal)
  - `sar k, D`: D > k→D (deslocamento aritmético à direita)
  - `shr k, D`: D > k→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 memoria, 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
imull	S	R[%edx]:R[%eax] ← S × R[%eax]	Signed full multiply
mull	S	R[%edx]:R[%eax] ← S × R[%eax]	Unsigned full multiply
cltd		R[%edx]:R[%eax] ← SignExtend(R[%eax])	Convert to quad word
idivl	S	R[%edx] ← R[%edx]:R[%eax] mod S; R[%eax] ← R[%edx]:R[%eax] ÷ S	Signed divide
divl	S	R[%edx] ← R[%edx]:R[%eax] mod S; R[%eax] ← R[%edx]:R[%eax] ÷ S	Unsigned divide

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

Instrução	Efeito	Descrição
`cltd`	%edx:%eax ← estende sinal(%eax)	Estende 32 bits para 64 bits
`idivl S`	%edx ← %edx:%eax mod S (resto)	Divisão com sinal
	%eax ← %edx:%eax ÷ S (quociente)	
`divl S`	%edx ← %edx:%eax mod S (resto)	Divisão sem sinal
	%eax ← %edx:%eax ÷ 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 overfow 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	
`cmpl`		Compare double word	
TEST	S_2, S_1	S_1 & S_2	Test
`testb`		Test byte	
`testw`		Test word	
`testl`		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
`sete`	*D*	`setz`	*D* ← ZF	Equal / zero
`setne`	*D*	`setnz`	*D* ← ~ZF	Not equal / not zero
`sets`	*D*		*D* ← SF	Negative
`setns`	*D*		*D* ← ~SF	Nonnegative
`setg`	*D*	`setnle`	*D* ← ~(SF ^ OF) & ~ZF	Greater (signed >)
`setge`	*D*	`setnl`	*D* ← ~(SF ^ OF)	Greater or equal (signed >=)
`setl`	*D*	`setnge`	*D* ← SF ^ OF	Less (signed <)
`setle`	*D*	`setng`	*D* ← (SF ^ OF) | ZF	Less or equal (signed <=)
`seta`	*D*	`setnbe`	*D* ← ~CF & ~ZF	Above (unsigned >)
`setae`	*D*	`setnb`	*D* ← ~CF	Above or equal (unsigned >=)
`setb`	*D*	`setnae`	*D* ← CF	Below (unsigned <)
`setbe`	*D*	`setna`	*D* ← CF | 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	*Label*		1	Direct jump
jmp	**Operand*		1	Indirect jump
je	*Label*	jz	ZF	Equal / zero
jne	*Label*	jnz	~ZF	Not equal / not zero
js	*Label*		SF	Negative
jns	*Label*		~SF	Nonnegative
jg	*Label*	jnle	~(SF ^ OF) & ~ZF	Greater (signed >)
jge	*Label*	jnl	~(SF ^ OF)	Greater or equal (signed >=)
jl	*Label*	jnge	SF ^ OF	Less (signed <)
jle	*Label*	jng	(SF ^ OF) | ZF	Less or equal (signed <=)
ja	*Label*	jnbe	~CF & ~ZF	Above (unsigned >)
jae	*Label*	jnb	~CF	Above or equal (unsigned >=)
jb	*Label*	jnae	CF	Below (unsigned <)
jbe	*Label*	jna	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): ~ZF (diferente/não-zero)
  ▪ js: SF (negativo)
  ▪ jns: ~SF (não negativo)
  ▪ jg (ou jnle): ~(SF^OF) & ~ZF (maior com sinal)ª
  ▪ jge (ou jnl): ~(SF^ OF) (maior ou igual com sinal)
  ▪ jl (ou jnge): SF^OF (menor com sinal)
  ▪ jle (ou jng): (SF^OF) | ZF (menor ou igual com sinal)
  ª^ é a operação ou-exclusivo (XOR)
  ▪ ja (ou jnbe): ~CF & ~ZF (acima sem sinal)
  ▪ jae (ou jnb): ~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   }
```

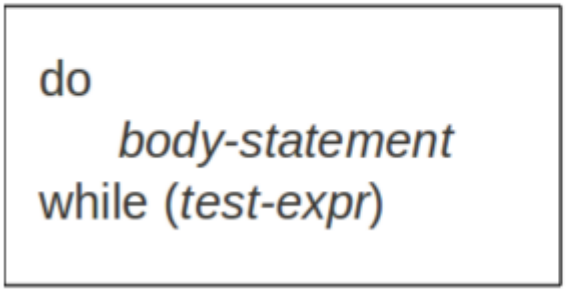
(b) Equivalent goto version

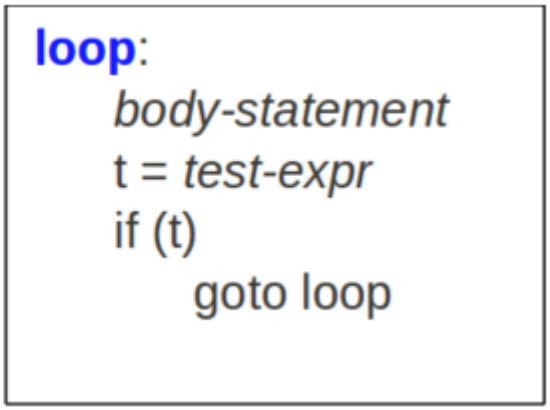
```
1   int gotodiff(int x, int y) {
2       int result;
3       if (x >= y)
4           goto x_ge_y;
5       result =  y - x;
6       goto done;
7    x_ge_y:
8       result = x - y;
9    done:
10      return result;
11  }
```

(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





(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   }
```

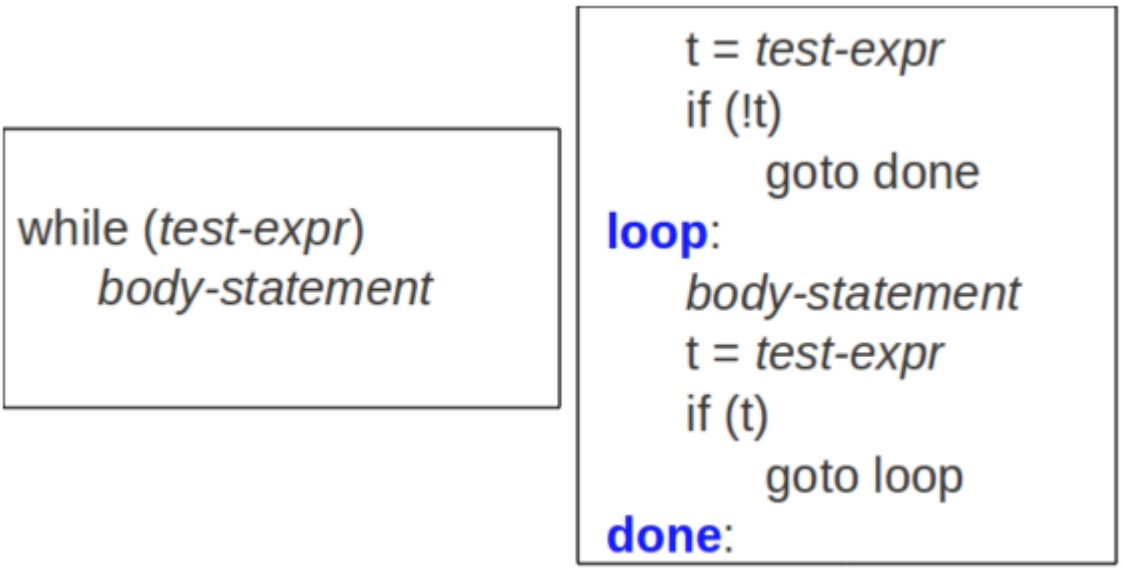
(b) Register usage

Register	Variable	Initially
%eax	result	1
%edx	n	*n*

(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



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

Switch Case

Switch em C

```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

O compilador usa uma técnica chamada **Tabela de Salto** (Jump Table), um **array de endereços de memória**. Cada elemento do array "aponta" para o início do bloco de código de um `case` específico.

- Quando o código do `switch` é executado, ele não compara o valor de `n` com cada `case`. Em vez disso, ele usa o valor de `n` para **calcular um índice** para acessar essa tabela.
- Por exemplo, se os casos são 100, 101, 102, o compilador pode fazer `indice = n - 100`.
  O programa então pega o endereço que está na posição `tabela[indice]` e **salta (Jumps)** diretamente para o bloco de código correto, sem precisar fazer nenhuma comparação.

```
    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
```

- Tabela de salto

```
1       .section        .rodata
2       .align 4        Align address to multiple of 4
3     .L7:
4       .long   .L3     Case 100: loc_A
5       .long   .L2     Case 101: loc_def
6       .long   .L4     Case 102: loc_B
7       .long   .L5     Case 103: loc_C
8       .long   .L6     Case 104: loc_D
9       .long   .L2     Case 105: loc_def
10      .long   .L6     Case 106: loc_D
```

Aula 11 - Implementação de subrotinas na arquitetura IA32

Uma chamada de subrotina requer:

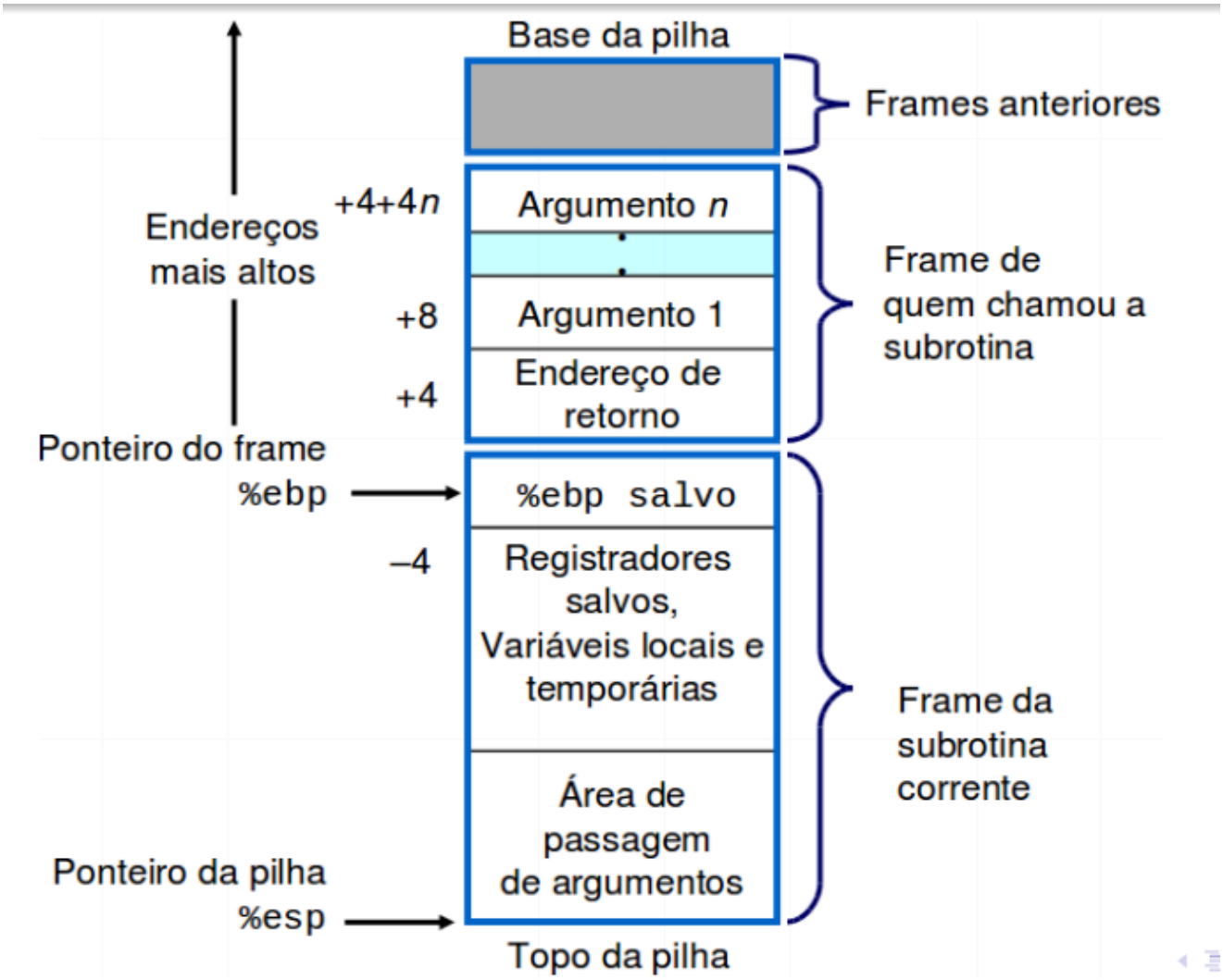
- Passagem de valores (parâmetros de entrada e de saída /retorno)
- transferência de controle (desvio da execução para outra parte do programa)
- alocação /desalocação de espaço de memória para as variáveis locais
  Para que os procedimentos funcionem corretamente (uma função `A` chama uma função `B`, que chama `C`, e depois todas retornam na ordem inversa), o programa precisa de um mecanismo para gerenciar o estado de cada chamada de função. Cada função precisa de seu próprio espaço de memória para:
- Armazenar argumentos que não couberam nos registradores.
- Guardar variáveis locais.
- Salvar o endereço de retorno (para onde o programa deve voltar quando a função terminar).

A porção da pilha alocada para uma chamada particular de subrotina é chamada de **stack frame** (ou registro de ativação)

A estrutura de dados que gerencia tudo isso é a **pilha em tempo de execução** (geralmente chamada apenas de **stack**). Ela é uma região de memória que funciona no princípio **LIFO (Last-In, First-Out)**, ou "o último a entrar é o primeiro a sair".

Analogia Clássica: Pense em uma pilha de pratos. Você só pode colocar um prato novo no **topo** da pilha. E quando vai tirar um prato, você só pode pegar o que está no **topo**. Você não consegue tirar um prato do meio da pilha sem antes remover todos os que estão acima dele. A pilha do computador funciona da mesma forma.

Conceitos-Chave



1. Direção de Crescimento da Pilha (Stack Growth)

- Este é um dos pontos mais importantes e contraintuitivos. Na arquitetura x86-64 (usada em praticamente todos os computadores modernos), a pilha **cresce para baixo**.
- Isso significa que, à medida que novos dados são adicionados à pilha, ela se expande em direção a **endereços de memória mais baixos**.
- Isso confunde muita gente. Em vez de a pilha crescer para endereços de memória maiores (como `0x100`, `0x104`, `0x108`...), ela cresce para **endereços menores** (como `0x108`, `0x104`, `0x100`...).

- **"Adicionar"** algo na pilha (`push`) significa **diminuir** o endereço do topo.
- **"Remover"** algo da pilha (`pop`) significa **aumentar** o endereço do topo.

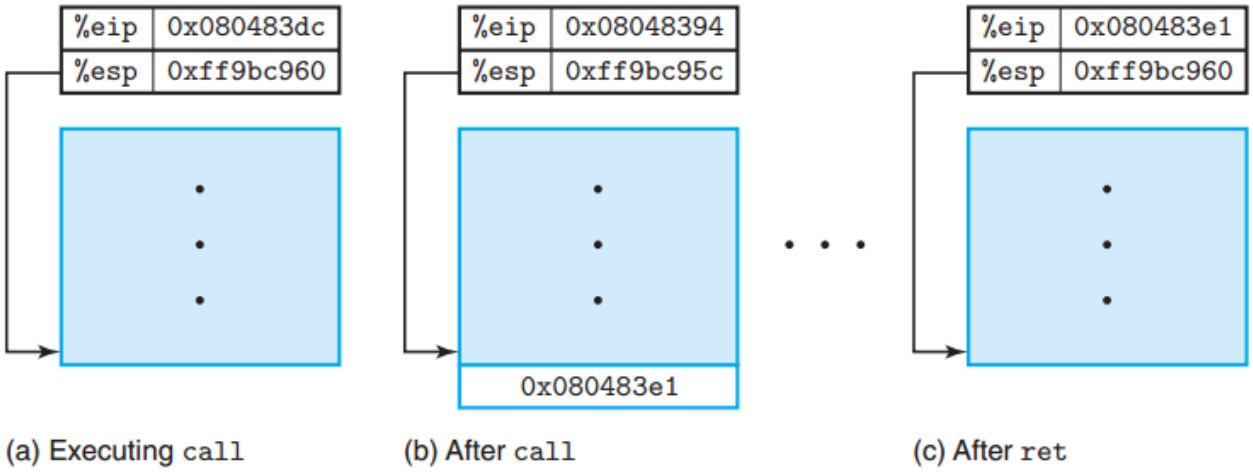
2. Os Ponteiros da Pilha: `%esp` e `%ebp`
O processador usa dois registradores especiais para gerenciar a pilha. Entender a diferença entre eles é a chave:

- `%esp` (Stack Pointer - Ponteiro da Pilha): **Sempre aponta para o topo exato da pilha**. É um ponteiro dinâmico, que muda de valor constantemente toda vez que colocamos (`push`) ou tiramos (`pop`) algo da pilha, ou quando alocamos espaço para variáveis locais. É a "ponta viva" da pilha.
- `%ebp` (Base Pointer / Frame Pointer - Ponteiro de Base): **Serve como uma referência fixa** para a "área de trabalho" (o stack frame) da função que está *executando no momento*. Enquanto `%esp` se move, `%ebp` fica parado durante a execução de uma função. Isso torna muito mais fácil para o computador encontrar os argumentos e as variáveis locais da função, pois eles estarão sempre a uma distância fixa de `%ebp`.

A Instrução `call`

A instrução `call` tem como alvo o endereço da função que está sendo chamada. O livro explica que o efeito da instrução `call` é duplo:

- **Empurrar o endereço de retorno na pilha:** O processador calcula o endereço da instrução que vem *imediatamente depois* da própria instrução `call` e empurra esse endereço de 64 bits na pilha.
- **Pular para o início da função chamada:** O contador de programa (PC) é alterado para o endereço da primeira instrução da função que está sendo chamada.



- **Antes da chamada (a):** O programa está prestes a executar a instrução `call` no endereço `0x080483dc`. A próxima instrução está em `0x080483e1`.
- **Depois da chamada (b):** A instrução `call` empurra o endereço de retorno `0x080483e1` na pilha e, em seguida, pula para o início da função `sum` no endereço `0x08048394`.

O livro também menciona que a instrução `call` pode ter duas formas:

- **Direta:** `call Label` - Onde o alvo é um rótulo (o nome da função). O endereço do alvo é codificado de forma relativa ao endereço da instrução seguinte (PC-relative).
- **Indireta:** `call *Operando` - Onde o alvo é lido de um registrador ou de uma posição de memória. É assim que ponteiros para funções em C são implementados.

A Instrução `ret`

A instrução `ret` é mais simples. Sua função é retornar o controle para a função que fez a chamada. Para isso, ela:

- **Desempilha o endereço de retorno:** A instrução `ret` remove ("pop") o endereço que está no topo da pilha.
- **Pula para esse endereço:** Ela altera o contador de programa (PC) para o endereço que acabou de ser removido da pilha.

Na **Figura (c)**, mostra a instrução `ret` da função `sum` sendo executada:

- Ela remove o endereço `0x080483e1` do topo da pilha e pula para lá, fazendo com que a execução da função `main` continue exatamente do ponto onde parou.

A Instrução `leave` (Auxiliar)

O livro também menciona a instrução `leave` nesta seção. Ela é usada para preparar a pilha para o retorno. É um atalho para as duas seguintes operações:

1. `movl %ebp, %esp` : Libera todo o espaço do stack frame atual, fazendo o ponteiro da pilha (`%esp`) apontar para a base (`%ebp`).
2. `popl %ebp` : Restaura o `%ebp` da função chamadora, que estava salvo na pilha.

Após a instrução

`leave`, o topo da pilha conterá exatamente o endereço de retorno, pronto para a instrução `ret`.

Registradores "Caller-Save" (Salvos pela Função Chamadora)

- **Registradores:** `%eax`, `%ecx`, `%edx`.
- **A Regra:** A função `callee` (a que é chamada) pode usar esses registradores livremente, sem se preocupar em salvar o valor que estava neles. Ela pode sobrescrevê-los à vontade.
- **A Responsabilidade:** Se a função `caller` (a que chama) se importa com o valor de um desses registradores, é **responsabilidade dela** salvar esse valor (geralmente na sua própria área na pilha) *antes* de fazer a chamada para outra função.
- **Analogia:** Pense nestes registradores como "rascunhos" em um quadro branco compartilhado. Se você vai pedir para um colega usar o quadro, você assume que ele vai apagar a área de rascunho. Se havia algo importante lá, você precisa copiar para o seu caderno antes.

2. Registradores "Callee-Save" (Salvos pela Função Chamada)

- **Registradores:** `%ebx`, `%esi`, `%edi`.
- **A Regra:** A função `caller` pode chamar a `callee` esperando que os valores nesses registradores permaneçam intactos após o retorno da `callee`.
- **A Responsabilidade:** Se a função `callee` precisar usar um desses registradores, é **responsabilidade dela** primeiro salvar o valor original

na sua própria área na pilha, usar o registrador, e depois restaurar o valor original *antes* de retornar para a `caller`.

- **Analogia:** Pense nestes como "quadros de status de projeto" no mesmo quadro branco. A regra do escritório é: se você precisar usar um desses quadros, primeiro tire uma foto dele (salve na pilha), faça seu trabalho, e depois restaure o conteúdo original a partir da foto antes de sair. Assim, quem te pediu ajuda pode confiar que o status do projeto não foi alterado.

Registradores Especiais Os registradores `%ebp` (frame pointer) e `%esp` (stack pointer) também são considerados *callee-save*, pois a função `callee` deve restaurá-los ao seu estado original antes de retornar.

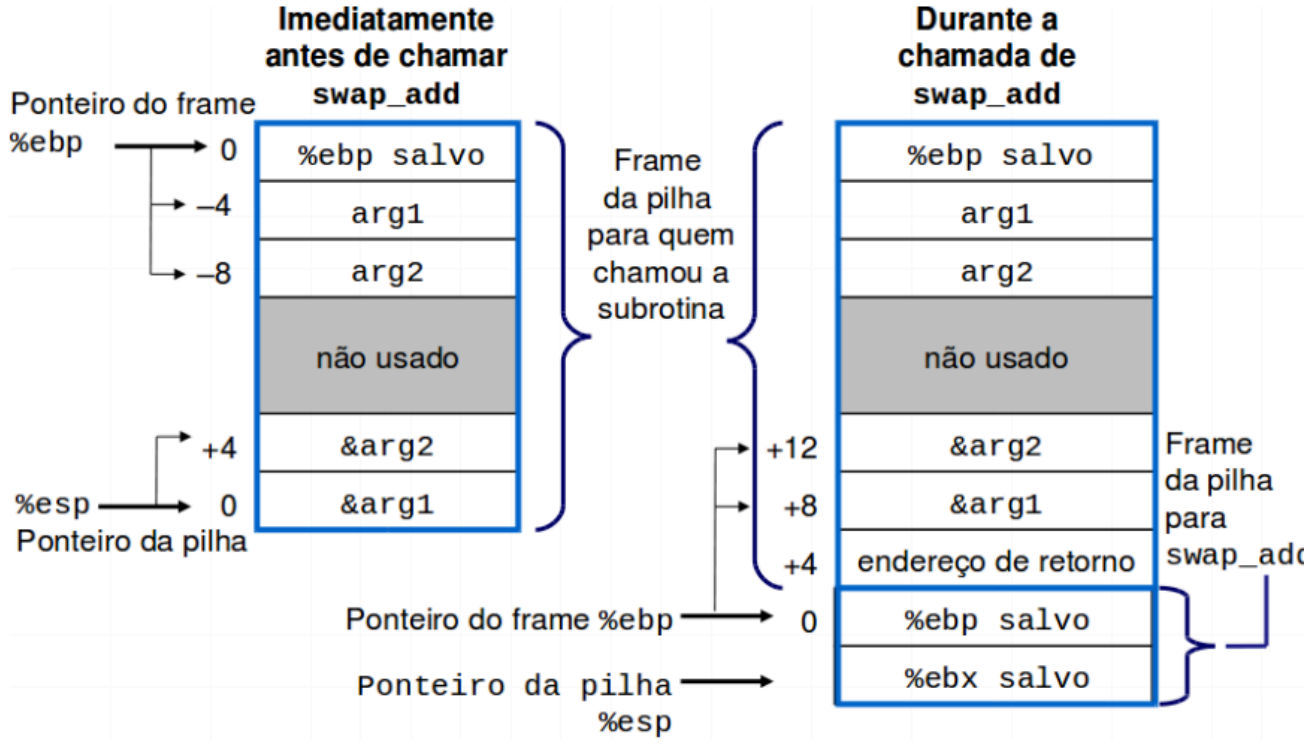
```
1   int swap_add(int *xp, int *yp)
2   {
3       int x = *xp;
4       int y = *yp;
5
6       *xp = y;
7       *yp = x;
8       return x + y;
9   }
10
11  int caller()
12  {
13      int arg1 = 534;
14      int arg2 = 1057;
15      int sum = swap_add(&arg1, &arg2);
16      int diff = arg1 - arg2;
17
18      return sum * diff;
19  }
```

Passo 1: A função `caller` se prepara para chamar `swap_add`

1. **Cria seu Stack Frame:** A `caller` cria seu próprio frame na pilha para suas variáveis locais (`arg1` e `arg2`).
2. **Prepara os Argumentos:** Ela precisa passar os *endereços* de `arg1` e `arg2` para `swap_add`. Para isso, ela usa a instrução `leal` para calcular os endereços e depois os empurra (`pushl`) na pilha. Os argumentos são colocados na pilha em ordem inversa (primeiro o endereço de `arg2`, depois o de `arg1`).

3. **Faz a Chamada:** A `caller` executa a instrução `call swap_add`. Isso automaticamente empurra o endereço de retorno para a pilha e pula para o código de `swap_add`.

Neste ponto, a pilha está assim:



Passo 2: A função `swap_add` é executada

4. **Setup (Configuração do Frame):**
   - Salva o `%ebp` antigo da `caller` na pilha (`pushl %ebp`).
   - Define seu próprio frame (`movl %esp, %ebp`).
   - Percebe que vai usar o registrador `%ebx`. Como `%ebx` é *callee-save*, ela **salva o valor antigo de `%ebx` na pilha** (`pushl %ebx`) para poder restaurá-lo mais tarde. Isso segue a convenção da seção 3.7.3.
5. **Corpo da Função:**
   - Acessa seus argumentos. Agora, os ponteiros `xp` e `yp` estão em `%ebp+8` e `%ebp+12`, respectivamente, pois estão no frame da `caller`.
   - Executa a lógica de troca (swap) e soma, usando os registradores `%eax`, `%ecx`, `%edx` e `%ebx` para os valores temporários.
   - Coloca o resultado da soma (`x + y`) no registrador `%eax`, pois, por convenção, é o registrador usado para retornar valores inteiros.
6. **Finish (Finalização):**

- Restaura o valor original de `%ebx` que ela tinha salvo (`popl %ebx`).
- Restaura o ponteiro de base da `caller` (`popl %ebp`).
- Retorna com a instrução `ret`, que pega o endereço de retorno do topo da pilha e pula para lá.

Passo 3: A função `caller` continua

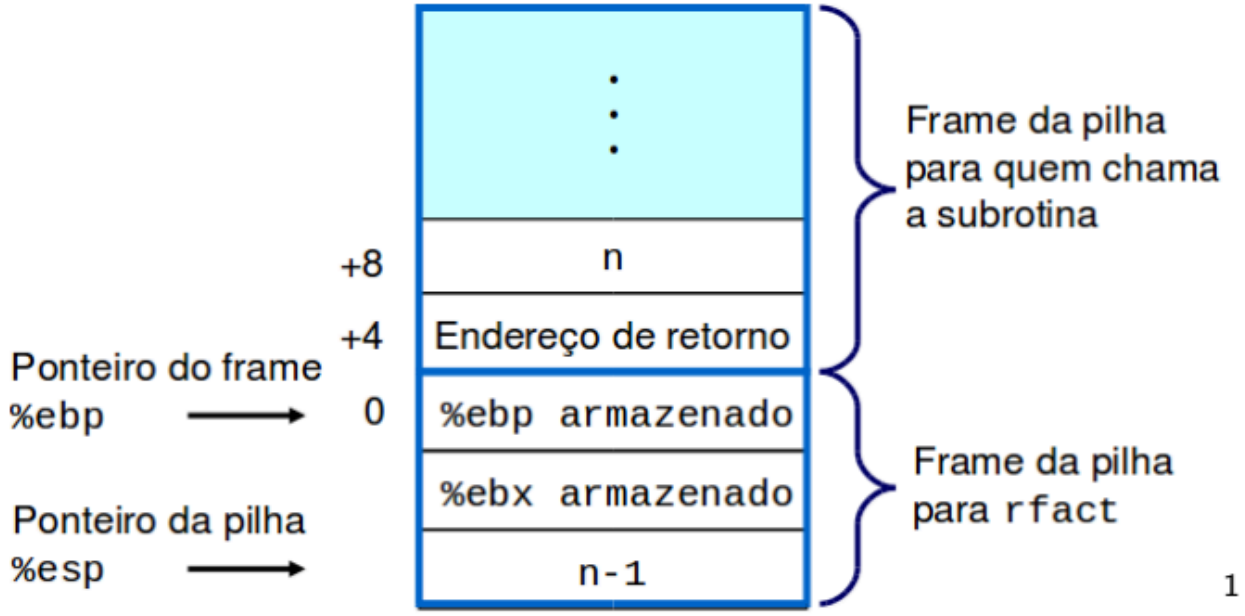
- A execução continua na instrução logo após o `call`. O valor de retorno de `swap_add` (a soma) está agora disponível em `%eax`.
- A `caller` usa esse valor para seus cálculos finais, limpa sua própria pilha e retorna.

Aula 12 - Subrotinas recursivas e manipulação de vetores na arquitetura IA32

Sub-rotinas Recursivas (Seção 3.7.5)

Uma função que chama a si mesma é tratada da mesma forma que qualquer outra chamada de função. A disciplina de pilha que gerencia os *stack frames* funciona perfeitamente para a recursão.

- **Frames de Pilha Separados:** Cada chamada recursiva recebe seu próprio *stack frame* na pilha. Isso garante que as variáveis locais de uma chamada não interfiram com as de outra. Por exemplo, em uma função fatorial recursiva `rfact(n)`, a variável `n` de uma chamada é independente da variável `n` de outra chamada.



- **Estado Salvo:** Antes de fazer a chamada recursiva, o estado atual é salvo. O endereço de retorno é empurrado para a pilha, como em qualquer chamada. Se for necessário preservar o valor de um registrador (como o argumento `n`) para ser usado após o retorno da chamada recursiva, ele deve ser salvo, seja na pilha ou em um registrador *callee-save* (como o `%ebx`, conforme mostrado no exemplo do livro).
- **Retorno:** Quando a condição de base é atingida (ex: `n <= 1`), a função retorna um valor. A instrução `ret` pega o endereço de retorno do topo da pilha, e a execução volta para a chamada anterior. O valor de retorno (geralmente em `%eax`) é então usado pela função chamadora para continuar seu cálculo (por exemplo, `n * rfact(n-1)`).

Em resumo, a pilha gerencia o estado de cada chamada de forma isolada, permitindo que a recursão funcione corretamente sem que as chamadas interfiram umas com as outras

Manipulação de Vetores (Seções 3.8.1 a 3.8.5)

Princípios Básicos de Vetores (Arrays)

- **Alocação Contígua:** Quando você declara um vetor em C, como `T A[N];`, o compilador aloca um bloco **contínuo** de memória para ele. Este bloco tem um tamanho total de `N * L` bytes, onde `L` é o tamanho em bytes do tipo de dado `T` (por exemplo, `L=4` para um `int`).
- **Identificador do Vetor como Ponteiro:** O nome do vetor, `A` neste caso, funciona como um ponteiro para o primeiro elemento do vetor. O valor deste ponteiro é o endereço de memória onde o primeiro byte do vetor está armazenado.
- **Acesso aos Elementos:** Para acessar um elemento específico, como `A[i]`, o computador calcula seu endereço usando a seguinte fórmula: `endereço_de_A + L * i`. Isso significa que para encontrar o i-ésimo elemento, o programa começa no endereço base do vetor e avança `L * i` bytes.
- **Eficiência no Nível de Máquina:** A arquitetura IA32 (e x86-64) possui modos de endereçamento de memória que tornam esse cálculo muito eficiente. A seção destaca a instrução de "endereçamento indexado com escala". Por exemplo, a instrução `movl (%edx,%ecx,4),%eax` realiza o cálculo completo `endereço_base + i * 4` e busca o valor da memória, tudo em uma única operação.

- `%edx` conteria o endereço base do vetor.
- `%ecx` conteria o índice `i`.
- `4` é o fator de escala (o tamanho de um `int`), que corresponde ao `L` na fórmula.

Aritmética com Ponteiros (Seção 3.8.2)

A linguagem C permite realizar operações aritméticas com ponteiros, o que está intimamente ligado ao acesso a vetores.

- **Ponteiros e Vetores:** O nome de um vetor em C pode ser tratado como um ponteiro para seu primeiro elemento. A expressão `A[i]` é idêntica a `*(A+i)`.
- **Escalonamento:** A principal característica da aritmética de ponteiros é o escalonamento. Se `p` é um ponteiro para um tipo de dado `T` que ocupa `L` bytes, a expressão `p+i` calcula o endereço `xp + L * i`, onde `xp` é o endereço inicial do ponteiro. Isso garante que `p+i` aponte para o i-ésimo elemento do tipo `T` a partir de `p`.
- **Tradução para Assembly:** O compilador traduz essas operações em instruções de máquina. Para calcular o *endereço* de um elemento (como em `&A[i]`), ele frequentemente usa a instrução `leal`. Para acessar o *valor* de um elemento (como em `A[i]`), ele primeiro calcula o endereço e depois usa uma instrução de movimento de dados, como `movl`.

Vetores Aninhados (Vetores Multidimensionais) (Seção 3.8.3)

Vetores aninhados, como

`int A[5][3]`, são armazenados na memória de forma contígua em uma ordem conhecida como **"row-major order"** (ordem por linha). Isso significa que todos os elementos da linha 0 são armazenados primeiro, seguidos por todos os elementos da linha 1, e assim por diante.

Para acessar o elemento

`D[i][j]` de um vetor declarado como `T D[R][C]`, o compilador gera código para calcular seu endereço de memória com a seguinte fórmula:

```
&D[i][j] = 0xD + L * (C * i + j)
```

Onde

`0xD` é o endereço inicial do vetor, `L` é o tamanho do tipo de dado `T`, `R` é o número de linhas e `C` é o número de colunas. O compilador otimiza o cálculo

`(C * i + j)` usando instruções de deslocamento (shift) e soma em vez de uma multiplicação, que é mais lenta.

Vetores de Tamanho Fixo vs. Variável (Seções 3.8.4 e 3.8.5)

- **Vetores de Tamanho Fixo:** Quando as dimensões de um vetor são constantes conhecidas em tempo de compilação (ex: `#define N 16`), o compilador pode realizar otimizações significativas. Ele pode, por exemplo, gerar código de laço altamente eficiente que calcula os endereços usando apenas deslocamentos e somas, evitando multiplicações. No exemplo de multiplicação de matrizes do livro (`fix_prod_ele`), o compilador otimiza o acesso criando ponteiros para o início das linhas e colunas relevantes e simplesmente os incrementa com um passo fixo a cada iteração, resultando em um código de laço interno muito rápido.
- **Vetores de Tamanho Variável:** Este recurso, introduzido no padrão ISO C99, permite que as dimensões de um vetor sejam determinadas em tempo de execução. Por exemplo, `int A[n][n]`, onde `n` é um parâmetro de uma função.
  - **Cálculo de Endereço:** Para acessar elementos desses vetores, o compilador não pode mais usar apenas deslocamentos, pois a dimensão (o número de colunas) não é uma constante. Ele precisa gerar uma **instrução de multiplicação** (`imull`) para calcular o deslocamento da linha (ex: `n * i`). Isso pode tornar o acesso um pouco mais lento em comparação com vetores de tamanho fixo.
  - **Otimizações:** Apesar disso, o compilador ainda pode otimizar laços que percorrem esses vetores, de forma semelhante ao que faz com vetores de tamanho fixo, mas os cálculos podem envolver o uso de mais registradores ou até mesmo o "derramamento de registradores" (*register spilling*), onde um valor (como a dimensão `n`) precisa ser lido da pilha a cada iteração se não houver registradores suficientes.

Aula 13 - Implementação de estruturas de dados heterogêneas

Implementação de Estruturas de Dados Heterogêneas

A linguagem C oferece duas maneiras de agrupar dados de tipos diferentes: `struct` (estruturas) e `union` (uniões). A forma como o compilador as traduz para o código de máquina é bastante diferente.

Implementação de Estruturas de Dados Heterogêneas

A linguagem C oferece duas maneiras de agrupar dados de tipos diferentes: `struct` (estruturas) e `union` (uniões). A forma como o compilador as traduz para o código de máquina é bastante diferente.

Structs (Estruturas) - Seção 3.9.1

Uma `struct` agrupa múltiplos objetos, possivelmente de tipos diferentes, em uma única unidade.

- **Alocação de Memória:** Todos os campos de uma `struct` são armazenados em uma região **contígua** de memória. Um ponteiro para uma `struct` aponta para o endereço do seu primeiro byte.
- **Acesso aos Campos:** O compilador calcula o deslocamento (offset) em bytes de cada campo a partir do início da estrutura. Para acessar um campo, como `r->i`, o código de máquina simplesmente soma o endereço base da estrutura (contido no ponteiro `r`) com o deslocamento do campo `i`. Esse cálculo de deslocamento é feito em tempo de compilação; o código de máquina final não sabe os nomes dos campos, apenas seus deslocamentos.
- **Alinhamento de Dados (Seção 3.9.3):** Para melhorar a performance, muitos sistemas exigem que os dados estejam "alinhados". Por exemplo, em um sistema Linux IA32, um `int` (4 bytes) deve estar em um endereço de memória que seja múltiplo de 4. Para garantir isso, o compilador pode precisar:
  - **Inserir Espaçamento Interno:** Adicionar "buracos" ou preenchimento (*padding*) entre os campos para garantir que cada campo comece em um endereço alinhado. Por exemplo, após um `char` (1 byte), o compilador pode inserir 3 bytes de preenchimento antes do próximo `int`.

- **Inserir Espaçamento no Final:** Adicionar preenchimento no final da estrutura para garantir que, em um vetor de estruturas, cada elemento do vetor comece em um endereço alinhado.

Devido a esse alinhamento, o tamanho total de uma `struct` pode ser maior do que a simples soma dos tamanhos de seus campos.

Unions (Uniões) - Seção 3.9.2

Uma `union` é uma forma de permitir que um único objeto de dados seja referenciado usando múltiplos tipos diferentes.

- **Alocação de Memória:** Diferente de uma `struct`, todos os campos de uma `union` **compartilham o mesmo bloco de memória**. Eles começam no mesmo endereço.
- **Tamanho da Union:** O tamanho total de uma `union` é determinado pelo tamanho do seu **maior campo**.
- **Casos de Uso:**
  1. **Economia de Memória:** São úteis quando se sabe que dois ou mais campos são mutuamente exclusivos (nunca serão usados ao mesmo tempo). Assim, eles podem compartilhar o mesmo espaço. O livro dá um exemplo de um nó de árvore binária que é ou um nó interno (com ponteiros para filhos) ou um nó folha (com dados), mas nunca ambos.
  2. **Acessar Padrões de Bits:** `Unions` são uma maneira de contornar o sistema de tipos do C para acessar a representação em bits de um tipo de dado como se fosse outro. Por exemplo, você pode armazenar um valor como

     `float` e depois ler seus bits como um `unsigned int` para ver sua representação IEEE 754.
- **Cuidado com a Ordem dos Bytes:** Como os campos compartilham memória, a forma como os dados são interpretados pode depender da ordem dos bytes da máquina (little-endian vs. big-endian), especialmente se os campos tiverem tamanhos diferentes.

Aula 14 - Combinando código assembly com programas C

Embora os compiladores C sejam muito bons em gerar código eficiente, há situações em que o programador precisa de controle em nível de

máquina, algo que a linguagem C não oferece diretamente. Isso pode ser necessário para acessar registradores específicos, usar instruções que o compilador não gera ou escrever código de sistema de baixo nível.

Existem duas abordagens para integrar código de montagem (Assembly) com C

Escrever funções separadas em código assembly e ligar com código C

- Respeitar as convenções para passagem de argumento e uso de registradores seguidas pelo compilador C

Para compilar basta fazer: `gcc teste.c soma.s`

```
#include <stdio.h>

int soma (int, int);

int main (void) {
   printf("%d\n", soma(-3, 5));
   return 0;
}
```

```
.global  soma

soma:
     pushl   %ebp
     movl    %esp, %ebp
     movl    8(%ebp), %eax
     addl    12(%ebp), %eax
     popl    %ebp
     ret
```

Principais Regras a Seguir:

- **Convenção de Registradores:** A função Assembly deve preservar os valores dos registradores *callee-save* (`%ebx`, `%esi`, `%edi`). Antes de usá-los, ela precisa salvar seus valores na pilha e restaurá-los antes de retornar. Os registradores

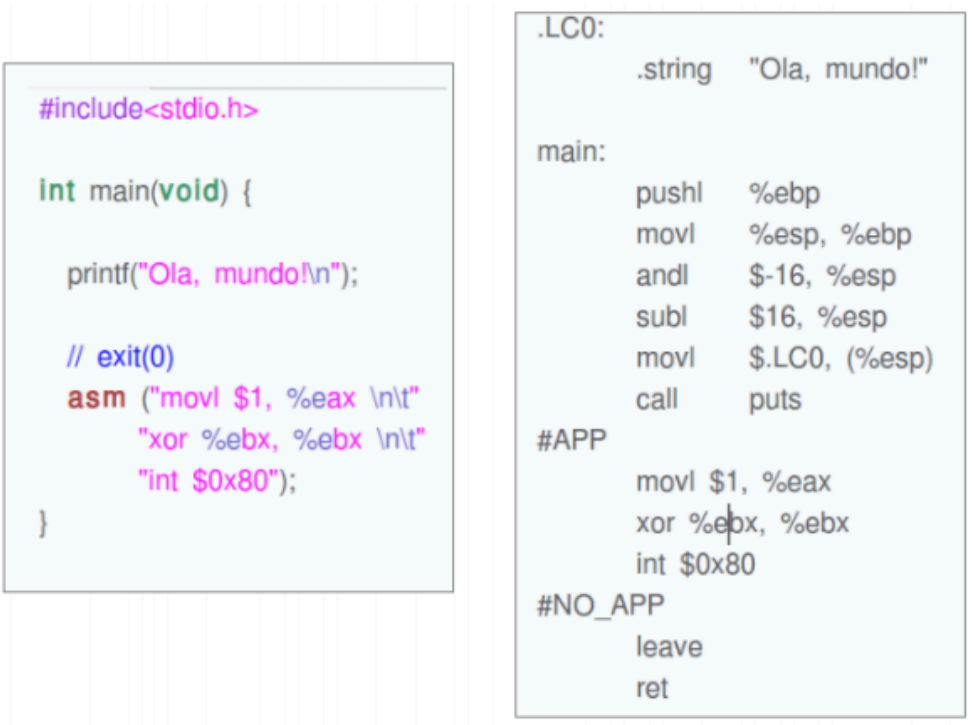
  caller-save (`%eax`, `%ecx`, `%edx`) podem ser modificados livremente.
- **Transferência de Argumentos:** A função Assembly acessa seus argumentos na pilha. Os argumentos são encontrados em deslocamentos positivos a partir do ponteiro de base (

  `%ebp`), começando em `%ebp+8`.
- **Retornando um Valor:** O valor de retorno da função (se for um inteiro ou um ponteiro) deve ser colocado no registrador `%eax` antes da instrução `ret`.
- **Gerenciamento da Pilha (*Stack*):** A função deve criar seu próprio *stack frame* no início (setup) e desalocá-lo antes de retornar (finish), garantindo que a pilha volte ao seu estado original.

Embutir código de montagem num programa em C, enxertando-o diretamente no código gerado pelo GCC

- Usa a diretiva especial provida pelo GCC chamada `asm` (“inline assembly”)

  “inline assembly”: usa a função: `asm (codigo-assembly);`



```
#include<stdio.h>

int main(void) {

  printf("Ola, mundo!\n");

  // exit(0)
  asm ("movl $1, %eax \n\t"
       "xor %ebx, %ebx \n\t"
       "int $0x80");
}
```

```
.LC0:
        .string  "Ola, mundo!"

main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    puts
#APP
        movl $1, %eax
        xor %ebx, %ebx
        int $0x80
#NO_APP
        leave
        ret
```

  A diretiva `asm` é específica para GCC e incompatível com outros compiladores

A forma básica é: `asm("instruções de montagem");`

No entanto, a forma estendida é mais útil, pois permite interagir com variáveis C:

`asm("instruções" : "operandos de saída" : "operandos de entrada" : "registradores afetados");`

Componentes Principais:

- **Instruções de Montagem:** Uma string contendo uma ou mais instruções Assembly. Registradores são referenciados com `%%` em vez de `%` (ex: `%%eax`). Os operandos que se conectam com as variáveis C são representados por placeholders como `%0`, `%1`, etc..
- **Operandos de Saída:** Lista as variáveis C que receberão os resultados da computação em Assembly. É preciso especificar uma "restrição"

que informa ao compilador como o valor será fornecido (ex: em um registrador, na memória).
- **Operandos de Entrada:** Lista as expressões ou variáveis C que servirão de entrada para o código Assembly, também com restrições.
- **Registradores Afetados (*Clobbers*):** Informa ao compilador quais registradores foram modificados pelo código Assembly embutido. Isso é crucial para que o compilador não assuma que o valor de um registrador foi preservado quando na verdade não foi. A string `"memory"` pode ser usada para informar que a memória foi alterada, forçando o compilador a salvar e restaurar valores se necessário.

Vantagens e Desvantagens:

- **Vantagem:** O compilador gerencia a maior parte do trabalho pesado, como salvar/restaurar registradores e alocar variáveis C para registradores ou memória. O programador pode se concentrar no pequeno trecho de código em nível de máquina que é realmente necessário.
- **Desvantagem:** A sintaxe é complexa e propensa a erros. O documento alerta que é fácil cometer erros que só se manifestam em tempo de execução, sendo muito difíceis de depurar.

Aula 15 - Referências a Memória fora dos limites e estouro de buffer

Esta seção do livro (iniciando na página 256) explica uma das vulnerabilidades de segurança mais perigosas e comuns em programas C. A causa fundamental reside na combinação de duas características da linguagem e do sistema:

1. A linguagem C

   não realiza verificação de limites (bounds checking) em acessos a vetores (arrays).
2. O estado de controle do programa, como o endereço de retorno e os valores salvos de registradores, é armazenado na mesma

   pilha (stack) que as variáveis locais, incluindo os vetores.

Essa combinação torna os programas vulneráveis a um ataque conhecido como **estouro de buffer (buffer overflow)**.

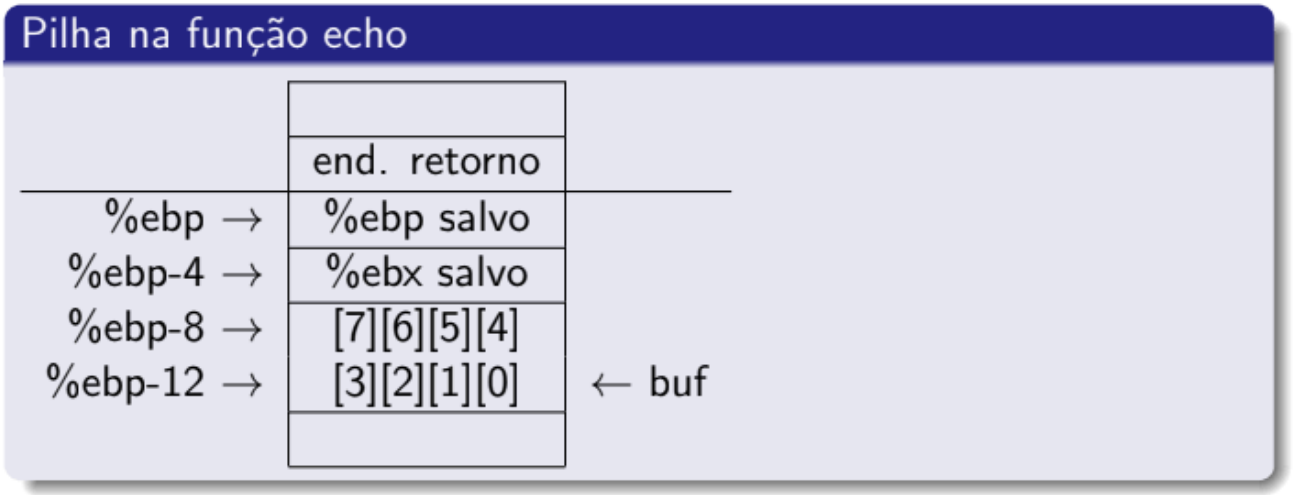
O Mecanismo do Estouro de Buffer

Um estouro de buffer ocorre quando o programa escreve dados em um vetor (o buffer) na pilha, mas os dados de entrada são maiores do que o espaço alocado para o buffer. Como não há verificação de limites, os bytes extras sobrescrevem as áreas adjacentes da pilha.

O livro ilustra isso com uma função

`echo` que usa a função de biblioteca `gets` para ler uma entrada do usuário e armazená-la em um buffer local. A função

`gets` é extremamente perigosa porque continua lendo da entrada até encontrar um caractere de nova linha (`\n`), sem se importar com o tamanho do buffer de destino.

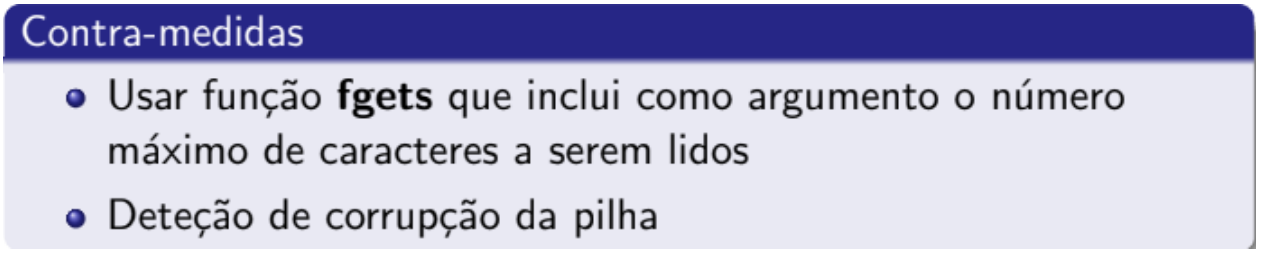


Caracteres lidos	Corrupção adicional
0-7	nenhuma
8-11	%ebx salvo
12-15	%ebp salvo
16-19	endereço de retorno
20+	estado salvo no registro do chamador

A figura mostra a organização da pilha para a função `echo`. Podemos ver que o buffer `buf` está localizado na pilha logo abaixo do estado salvo (registradores salvos e o endereço de retorno). Se um usuário digitar uma string maior que o buffer, os dados excedentes irão sobrescrever sequencialmente:

1. **Registradores salvos:** Os valores de registradores *callee-save* (como `%ebp` e `%ebx`) que foram salvos no início da função serão corrompidos.

2. **Endereço de Retorno:** Se a string for longa o suficiente, ela sobrescreverá o endereço de retorno que a instrução `call` salvou na pilha.



A Consequência: Execução de Código Malicioso

O maior perigo ocorre quando o endereço de retorno é corrompido. Quando a função tentar retornar usando a instrução

`ret`, em vez de pular de volta para a função chamadora, ela pulará para o endereço corrompido que foi inserido pelo atacante.

Isso abre a porta para ataques graves. Um atacante pode criar uma string de entrada que contém:

- **Código de Exploração (*Exploit Code*):** Uma sequência de bytes que corresponde a instruções de máquina maliciosas.
- **Um Endereço de Retorno Falso:** Um endereço que aponta para o início do código de exploração, que agora está na pilha.

Quando a função vulnerável retorna, o processador executa o código do atacante em vez de continuar a execução normal do programa. Esse código pode, por exemplo, iniciar um

shell (interpretador de comandos), dando ao atacante controle total sobre o sistema. O famoso "Internet Worm" de 1988 usou um ataque de estouro de buffer no daemon `finger` para se espalhar.

Outras Funções Perigosas: O livro alerta que outras funções da biblioteca C padrão, como `strcpy`, `strcat` e `sprintf`, também são perigosas porque podem causar estouros de buffer ao não verificarem o tamanho do buffer de destino.

Aula 16 - Fluxo de controle com exceções

O **Fluxo de Controle com Exceções (ECF)** é o mecanismo fundamental que os sistemas operacionais utilizam para reagir a eventos do sistema (como erros de divisão por zero ou chegada de um pacote de rede) e

para implementar funcionalidades essenciais como system calls e multitarefa.

Tratamento de Exceções

O processo de lidar com uma exceção envolve uma cooperação próxima entre o hardware e o software do sistema operacional:

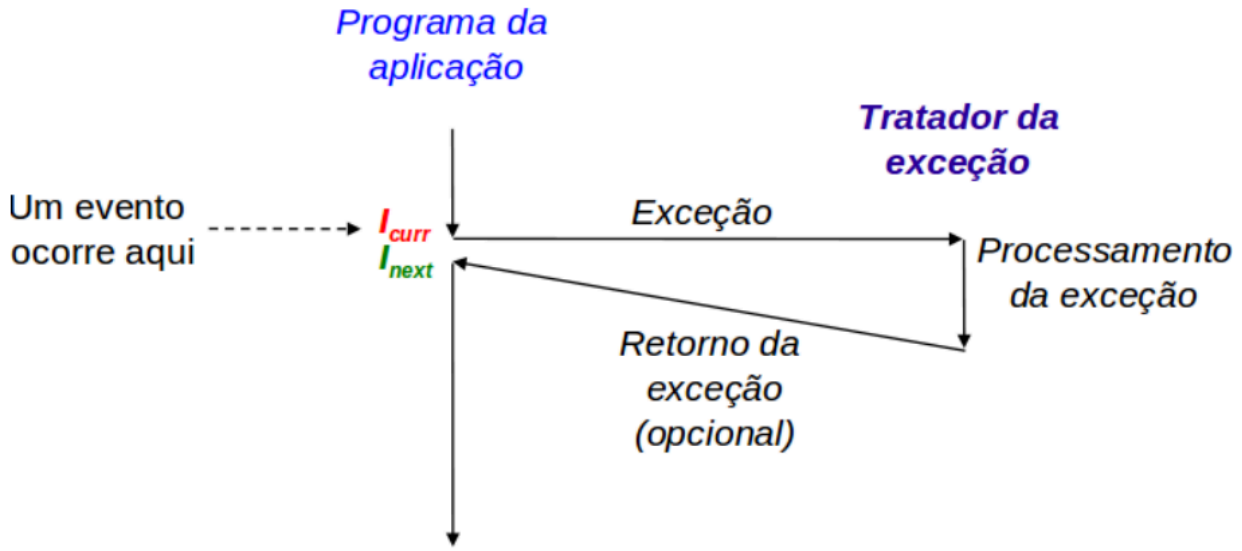
1. **Evento e Número da Exceção:** Quando um evento ocorre, o processador lhe atribui um número de exceção único, que identifica o tipo do evento.
2. **Tabela de Exceções:** Na inicialização do sistema, o sistema operacional cria uma tabela de pulo (jump table) chamada **tabela de exceções**.

   Cada entrada `k` desta tabela contém o endereço da rotina que tratará a exceção de número `k`.
3. **Transferência de Controle:** O processador usa o número da exceção como um índice para encontrar o endereço do **handler de exceção** apropriado na tabela de exceções e transfere o controle para ele. Essa transferência é semelhante a uma chamada de procedimento, mas com diferenças importantes:

   ```
   - O endereço de retorno (da instrução atual ou da próxima) é colocado na pilha.

   - O processador também salva na pilha parte do seu estado (como o registrador de flags).

   - A execução muda do **modo usuário** para o **modo kernel**, permitindo que o handler de exceção tenha acesso privilegiado aos recursos do sistema.
   ```



Após o handler terminar seu processamento, ele pode:

- Retornar o controle para a instrução que causou a exceção (`Icurr`).
- Retornar o controle para a próxima instrução (`Inext`).
- Abortar o programa.

Classificação de Exceções

As exceções são classificadas em quatro tipos principais, conforme detalhado no slide "Classificação de Exceções":

- **Interrupção (*Interrupt*):**
  - **Causa:** Um sinal de um dispositivo de I/O externo ao processador (ex: chegada de um pacote de rede, clique do mouse).
  - **Sincronia:** É **assíncrona**, pois não está relacionada à execução de uma instrução específica.
  - **Retorno:** Sempre retorna para a *próxima* instrução do fluxo interrompido.
- **Trap:**
  - **Causa:** É um evento **intencional** resultante da execução de uma instrução, como uma *system call* (chamada de sistema).
  - **Sincronia:** É **síncrona**.
  - **Retorno:** Sempre retorna para a *próxima* instrução.
- **Falha (*Fault*):**
  - **Causa:** Um erro potencialmente recuperável (ex: falta de página na memória, ou *page fault*).
  - **Sincronia:** É **síncrona**.
  - **Retorno:** Pode retornar para a instrução *atual* para reexecutá-la (se o erro foi corrigido) ou pode abortar o programa.

- **Abort:**
  - **Causa:** Um erro fatal e irrecuperável, geralmente de hardware (ex: erro de paridade na memória).
  - **Sincronia:** É **síncrona**.
  - **Retorno:** **Nunca** retorna para o programa que causou o erro; o programa é finalizado.

Exceções nos Sistemas Linux/IA32

O slide "Exceções nos Sistemas Linux/IA32" fornece exemplos concretos dessas classes:

- **Falhas e Aborts:**
  - **Erro de Divisão (Exceção 0):** Ocorre ao dividir por zero. É uma falha.
  - **Falha de Proteção Geral (Exceção 13):** Ocorre por uma referência de memória inválida, como escrever em uma seção de código somente leitura. É comumente reportada como "Segmentation fault".
  - **Falha de Página (*Page Fault*, Exceção 14):** Uma falha recuperável que é fundamental para a memória virtual.
  - **Verificação de Máquina (*Machine Check*, Exceção 18):** Um abort causado por um erro de hardware fatal.
- **Chamadas de Sistema (*System Calls*):**
  - No Linux/IA32, as *system calls* são implementadas como um **trap** através da instrução `int $0x80`, que dispara a exceção de número 128 (0x80). Isso fornece uma interface controlada entre os programas do usuário e o kernel do sistema operacional.

Aula 17: Programas em execução

Programas em Execução

Um **processo** é a abstração que o sistema operacional fornece para um programa em execução. Ele consiste no código do programa, dados, pilha, registradores, contador de programa (PC) e outras informações de estado do sistema, o que é chamado de

contexto do processo.

O processo fornece ao programa duas abstrações fundamentais: um fluxo de controle independente e um espaço de endereçamento privado.

Fluxo de Controle Lógico

- Um processo tem a ilusão de que possui o uso exclusivo da CPU.
- A sequência de valores do contador de programa (PC) que correspondem às instruções do programa é chamada de **fluxo de controle lógico**.
- Na realidade, vários processos rodam concorrentemente, com seus fluxos de controle sendo intercalados pelo sistema operacional. Esse processo é chamado de **multitarefa** (*multitasking*) e é implementado através de um mecanismo chamado **troca de contexto**.
- **Fluxos Concorrentes:** São fluxos que se sobrepõem no tempo. Se eles executam em diferentes núcleos (cores) de processador, são chamados de **fluxos paralelos**.

Espaço de Endereçamento Privado

- Um processo também tem a ilusão de que possui o uso exclusivo da memória do sistema.
- Cada processo possui seu próprio **espaço de endereçamento virtual privado**, o que significa que um processo não pode ler ou escrever na memória de outro processo sem permissão explícita.
- A estrutura desse espaço de endereçamento é consistente entre os processos, geralmente com uma área para o código do programa, dados, heap e pilha do usuário, além de uma área reservada para o kernel do sistema operacional.

Modo Usuário e Modo Kernel

Para garantir que um processo não interfira com outros ou com o próprio sistema operacional, os processadores usam um

bit de modo.

- **Modo Kernel (ou Supervisor):** Quando o bit de modo está ativado, o processo está rodando em modo kernel. Nesse modo, ele pode executar qualquer instrução da máquina e acessar qualquer endereço de memória.
- **Modo Usuário:** Quando o bit de modo está desativado, o processo está em modo usuário e tem acesso restrito. Ele não pode executar

instruções privilegiadas (como parar o processador) nem acessar diretamente a área de memória do kernel.
- A única forma de um processo passar do modo usuário para o modo kernel é através de uma **exceção**, como uma interrupção, falha ou trap (chamada de sistema).

Trocas de Contexto

- A troca de contexto é o mecanismo do kernel que permite a multitarefa. O **contexto** é toda a informação de estado que o kernel precisa para reiniciar um processo que foi interrompido (preempted).
- Uma **troca de contexto** envolve:
  1. Salvar o contexto do processo atual.
  2. Restaurar o contexto salvo de um processo anteriormente interrompido.
  3. Passar o controle para esse novo processo restaurado.
- Essa troca é realizada pelo **agendador (*scheduler*)** do kernel e pode ocorrer quando um processo faz uma chamada de sistema que bloqueia (como `read`) ou quando ocorre uma interrupção de tempo (*timer interrupt*).

Criando e Terminando Processos no Linux

O Unix fornece chamadas de sistema para controlar processos.

- **Estados de um Processo:** Um processo pode estar em um de três estados:
  - **Executando (*Running*):** Está na CPU ou aguardando para ser executado.
  - **Parado (*Stopped*):** Sua execução foi suspensa e não será agendado para rodar até receber um sinal `SIGCONT`.
  - **Terminado (*Terminated*):** O processo parou permanentemente.
- **Função `exit`:** Um processo termina sua execução chamando a função `exit`.
- **Função `fork`:** Um processo pai cria um novo processo filho chamando a função `fork`.
  - `fork` é chamada uma vez, mas retorna duas vezes: no pai, retorna o PID do filho; no filho, retorna `0`.
  - O filho recebe uma cópia idêntica, porém **separada**, do espaço de endereçamento virtual do pai, mas herda os descritores de

arquivos abertos.

- **Reaproveitamento (*Reaping*) e Zumbis:** Quando um processo termina, ele não é removido completamente do sistema até que seja "reaproveitado" (*reaped*) por seu processo pai. Um processo terminado mas ainda não reaproveitado é chamado de **zumbi** (*zombie*).
- **Função `waitpid`:** O pai espera pelo término de seus filhos usando a função `waitpid`. Isso permite que o pai obtenha o status de saída do filho e que o kernel remova o processo zumbi do sistema.

Aula 20 : Ligação e carga de programas

Ligação (Linking) e Carga (Loading)

Ligação (Linking) é o processo de coletar e combinar várias partes de código e dados (provenientes de diferentes arquivos objeto) em um único arquivo que pode ser executado. Este arquivo único é chamado de

arquivo objeto executável.

A principal vantagem da ligação é que ela permite a **compilação separada**. Em vez de escrever um programa inteiro em um único arquivo gigante, podemos dividi-lo em módulos menores e mais fáceis de gerenciar. Quando alteramos um módulo, apenas ele precisa ser recompilado antes de ser "re-ligado" ao restante do programa.

O processo de ligação pode ocorrer em diferentes momentos:

- **Em tempo de compilação:** Ligação estática, que é o foco principal.
- **Em tempo de carga:** Quando o programa é carregado na memória.
- **Em tempo de execução:** Ligação dinâmica, usada por bibliotecas compartilhadas.

Carga (Loading) é o processo que vem depois da ligação estática. Quando você executa um programa no shell, o

carregador (*loader*) do sistema operacional copia o código e os dados do arquivo executável para a memória e, em seguida, transfere o controle do processador para o programa para que ele comece a ser executado. O slide "Carregando Arquivos Objeto Executáveis" mostra como o programa é organizado na memória virtual do Linux, com seções para código (

`.text`), dados (`.data`, `.bss`), heap, bibliotecas compartilhadas e a pilha (*stack*).

Arquivos Objeto

O processo de ligação trabalha com três tipos de

arquivos objeto:

1. **Arquivo Objeto Relocável:**
   - Contém código e dados em um formato que pode ser combinado com outros arquivos relocáveis.
   - É o resultado da compilação de um arquivo de código-fonte (ex: `.c` -> `.o`).
2. **Arquivo Objeto Executável:**
   - Contém código e dados em um formato que pode ser carregado diretamente na memória para execução.
   - É o resultado final do processo de ligação.
3. **Arquivo Objeto Compartilhado (*Shared Object*):**
   - Um tipo especial de arquivo relocável que é carregado na memória e ligado dinamicamente, seja em tempo de carga ou em tempo de execução. São as famosas bibliotecas compartilhadas (ex: `.so` no Linux, `.dll` no Windows).

O slide "Formato de um Arquivo Objeto Relocável ELF" mostra a estrutura de um arquivo `.o` típico em sistemas Unix/Linux. Ele é composto por várias seções, incluindo:

- **Cabeçalho ELF:** Contém metadados sobre o arquivo (tipo de máquina, tamanho, etc.).
- **.text:** O código de máquina compilado do programa.
- **.rodata:** Dados somente leitura, como strings de formatação para `printf`.
- **.data:** Variáveis globais e estáticas que são inicializadas.
- **.bss:** Variáveis globais e estáticas que **não** são inicializadas. Esta seção é um placeholder e não ocupa espaço no arquivo objeto, economizando espaço em disco.
- **.symtab:** A **tabela de símbolos**, que guarda informações sobre funções e variáveis globais que o módulo define e referencia.
- **.rel.text** e **.rel.data:** As **entradas de realocação**. Elas contêm informações que dizem ao linker como modificar as referências a

símbolos globais e funções externas quando o arquivo for ligado a outros.

Aula 21 - Ligação estática