Representação de Programas
Operações Aritméticas e Lógicas

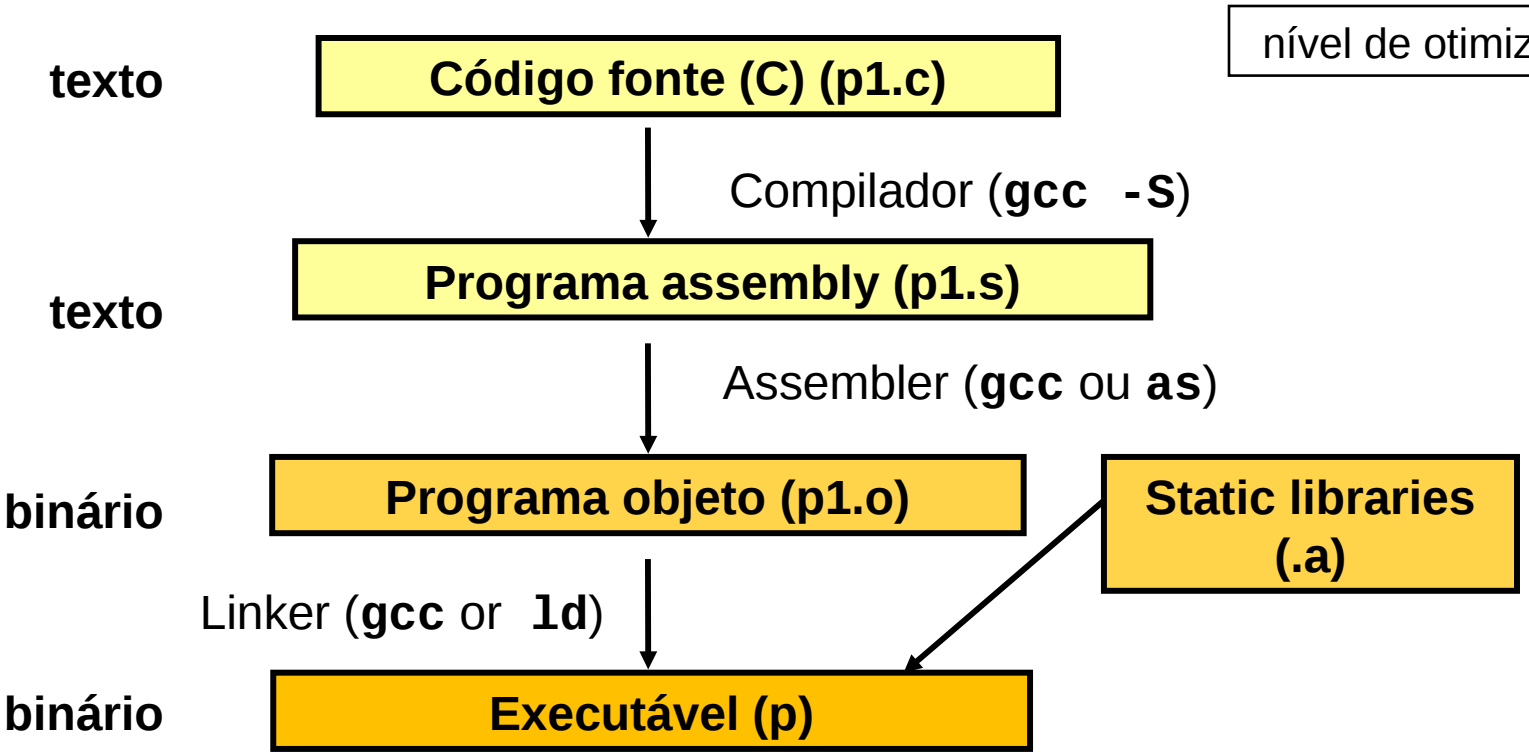
Representação de Programas

- Computadores executam código de máquina
  - sequências de bytes que codificam operações que manipulam dados, gerenciam memória, realizam E/S, ...
- Um compilador gera o código de um programa conforme
  - o conjunto de instruções da máquina alvo (IA32)
  - as regras estabelecidas pela linguagem de programação (C)
  - as convenções seguidas pelo sistema operacional (Linux)
- O compilador gcc gera como saída do passo de compilação um programa em linguagem de montagem
  - representação textual do programa
  - entrada para o passo de montagem (assembler)

Geração de um executável

- Arquivo fonte **`p1.c`**
- Comando de compilação: **`gcc -O1 -o p p1.c`**

nível de otimização



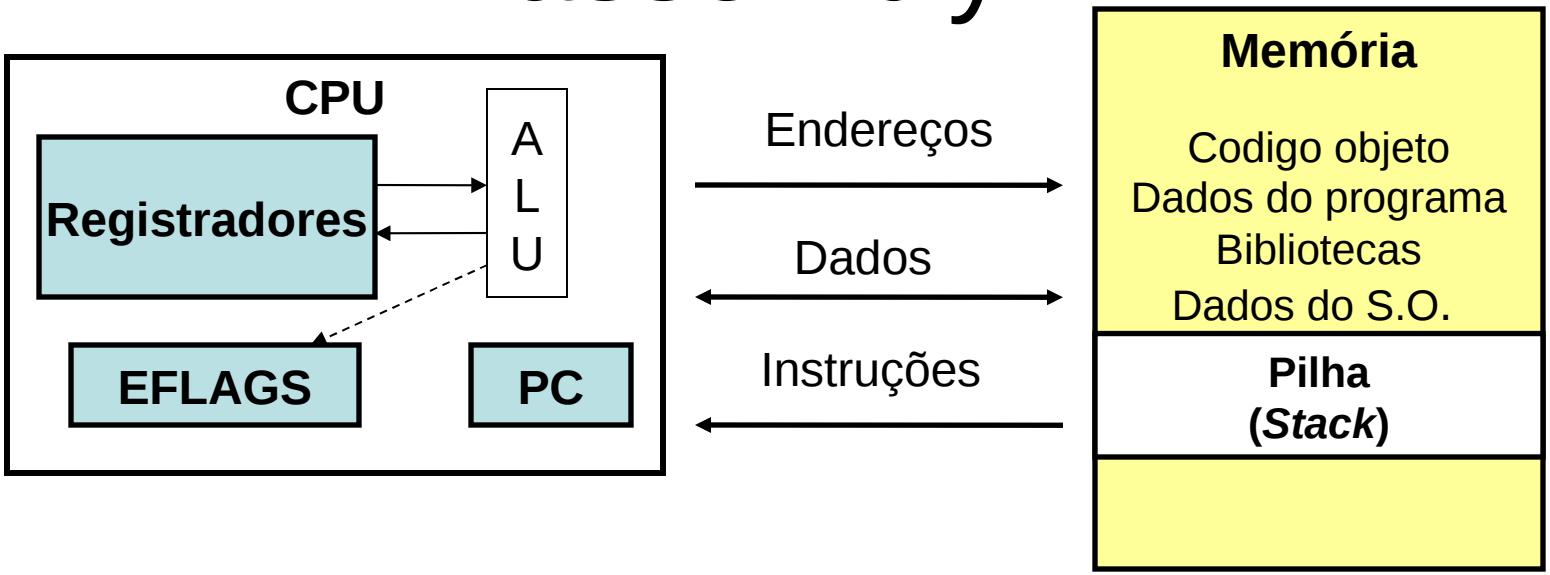
Níveis de otimização

- O tipo (nível) de otimização utilizado pode tornar difícil entender a relação entre o código gerado e o programa original
- Um compilador pode:
  - reordenar instruções
  - eliminar de código desnecessário
  - substituir operações mais lentas por operações mais rápidas
  - eliminar recursão
- Atualmente os compiladores geram código **pelo menos** tão eficiente quanto o de um programador experiente
  - escrever código X entender o código gerado pelo compilador

Linguagem de Montagem

- Muito próxima à linguagem de máquina
- Instruções de máquina executam operações bem simples
  - operações aritméticas/lógicas
  - transferência de dados entre memória e registradores
  - controle do fluxo de execução (desvios)
- Tipos de dados básicos
  - valores inteiros de 1, 2, ou 4 bytes
  - endereços
  - dados em ponto flutuante (4 ou 8 bytes)
- Não existem tipos de dados estruturados (arrays ou structs)
  - dados básicos alocados de forma contígua na memória

Visão do programador assembly



Estado da CPU

- PC: endereço da próxima instrução (%eip)
- Registradores
  - valores inteiros, endereços
- Registrador de condição (EFLAGS)
  - status da última operação aritmética/lógica (overflow?, zero?,…)

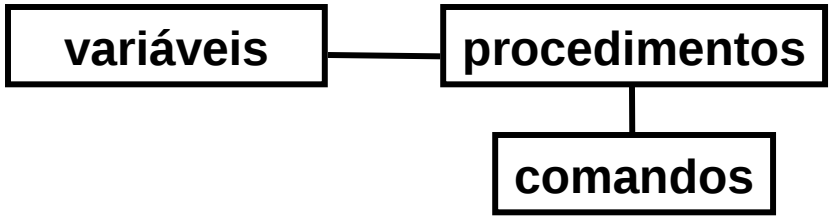
Memória “plana”

- Array de bytes endereçáveis
- Contém código, dados e pilha
- Pilha usada para tratar chamada a procedimentos

Representação de programas

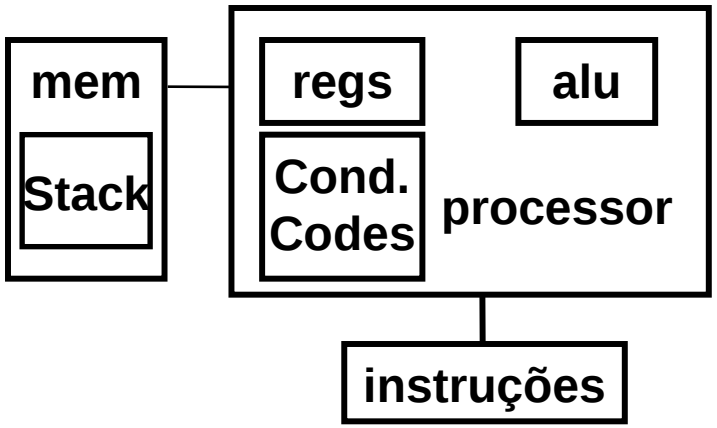
Modelos de máquinas **Dados**

Ling. C



1) `Char`
2) `short`
2) `int`, `float`
3) `double`
4) `struct`, array
5) pointer

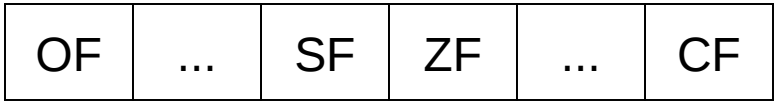
Assembly



1) Byte
2) 2-byte word
2) 4-byte lword
3) 8-byte quad word
4) Alocação contígua de bytes
5) Endereço do byte inicial

Registrador EFLAGS

- Seus bits indicam a ocorrência de diferentes eventos

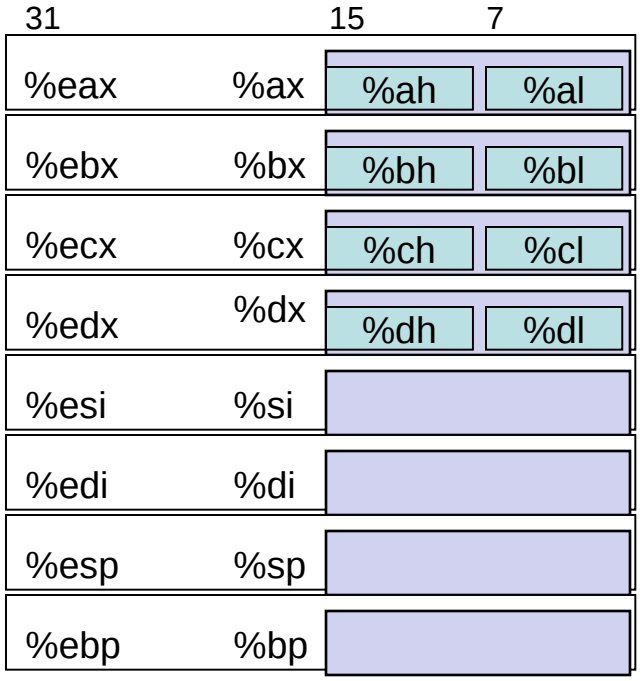


  - OF (Overflow flag) — overflow (negativo ou positivo)
  - SZ (Sign flag) — resultado < 0
  - ZF (Zero flag) — resultado == 0
  - CF (Carry flag) — unsigned overflow

- Os bits de EFLAGS são consultados por instruções de desvio condicional

Registradores

- A CPU IA32 tem 8 registradores de 32 bits
  - valores inteiros e endereços (ponteiros)
- 6 registradores são de propósito geral
  - algumas instruções usam registradores específicos
  - convenções estabelecidas pela linguagem
- **%ebp** e **%esp** tem ponteiros para a pilha
  - uso de acordo com as convenções para a gerência da pilha
- Todos podem ser acessados como 16 bits (w) ou 32 bits (l)
  - os 4 primeiros registradores permitem acesso independente aos 2 bytes menos significativos (b)



Modos de endereçamento

- A maioria das instruções tem um ou mais operandos

  `movl` *fonte*, *destino*

  - constantes (fonte), ou conteúdo de registrador/memória

- Modos de endereçamento (especificação de operandos)
  - Modo imediato (constante): **`$0x0000aabb, $1234`**
  - Modo registrador (conteúdo de registrador) : **`%eax, %ax, %al`**
  - Modo memória (registrador como ponteiro)
    - direto por registrador (sempre 32 bits!): **`(%ebx)`**, **`(%esi)`**
    - base-deslocamento (soma de offset): **`8(%ebp)`** → soma 8 ao end. em %ebp
  - Modo direto (“nome” de um símbolo/variável): **`mov var, %ebx`**
    - **ATENÇÃO** → $var é o **endereço** de var ! (&var)

Combinações de operandos

	Fonte	Destino	Exemplo	Equiv. em C
movl	*Imed*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

Não é possível fazer transferência direta memória-memória com uma única instrução !!!

Tamanho dos operandos

- Sufixo da instrução indica o tamanho do(s) operando(s)
  - long (l), word (w), byte (b)
  - termos (long - l, word - w) são “herança” da arquitetura 16-bit

- b - byte (8 bits)
  - **`movb $0, (%eax)`**
- w - word (2 bytes = 16 bits)
  - **`movw $0, (%eax)`**
- l - long (4 bytes = 32 bits)
  - **`movl $0, (%eax)`**

Grupos de Instruções

- Movimentação de dados
- Aritmética inteira
- Instruções lógicas
- Instruções de controle de fluxo
- Instruções em ponto flutuante

Movimentação de Dados

Instrução	Efeito	Descrição
mov S, D	**S → D**	**Move**
movb	**move byte**	
movw	**move word (16)**	
movl	**move long (32)**	
movs S, D	**signExtend(S) → D**	**Move with sign extension**
movsbw	**move signed-ext byte to word**	
movsbl	**move signed-ext byte to long**	
movswl	**move signed-ext word to long**	
movz S, D	**zeroExtend(S) → D**	**Move with zero extension**
movzbw	**move zero-ext byte to word**	
movzbl	**move zero-ext byte to long**	
movzwl	**move zero-ext word to long**	

Exemplos

- movl $0x4050, %eax - imm--reg
- movw %bx, %ax - reg--reg
- movb $-17, 1(%eax) - imm--mem

- Movimentação com extensão
  - assuma que %dh = CD e %eax = 98765432

  - movb %dh, %al → %eax = 987654CD
  - movsbl %dh, %eax → %eax = FFFFFFCD
  - movzbl %dh, %eax → %eax = 000000CD

Exercício (tradução)

- Observe o seguinte trecho em C:

```
src_t v;
dst_t *p
*p = (dest_t) v;
```

Assuma que o valor de v está armazenado em porção de %eax (%ax, %al) e o ponteiro p em %edx. Para as combinações de src_t e dst_t indica a operação de movimentação adequada:

src_t	**dst_t**	**instrução**
int	int	`movl %eax, (%edx)`
char	int	**`movsbl %al,(%edx)`**
char	unsigned	**`movsbl %al,(%edx)`**
unsigned char	int	**`movzbl %al,(%edx)`**
int	char	**`movb %al, (%edx)`**
unsigned	int	**`movl %eax,(%edx)`**

Operações aritméticas e lógicas

Instrução	Efeito	Descrição
incs **D**	**D + 1 → D**	Incremento (reg ou mem)
decs **D**	**D - 1 → D**	Decremento (reg ou mem)
negs **D**	**- D → D**	Negação (compl 2)
nots **D**	**~D → D**	NOT (lógico, bit a bit)
adds **S, D**	**D + S → D**	ADD
subs **S, D**	**D - S → D**	Subtract
imuls **S, D**	**D * S → D**	Signed Multiply
ands **S, D**	**D & S → D**	AND
ors **S, D**	**D | S → D**	OR
xors **S, D**	**D ^ S → D**	XOR

Exercício

- Assuma os seguintes valores armazenados em memória e registradores:

Endereço	Valor	Registrador	Valor
0x100	0xFF	%eax	0x100
0x104	0xAB	%ecx	0x01
0x108	0x13	%edx	0x03

Indique o efeito das instruções abaixo (destino e valor)

Instrução	**Destino**	**Valor**
`addl %ecx,(%eax)`	**Mem em 0x100**	**0x100**
`subl %edx, 4(%eax)`	**Mem em 0x104**	**0xA8**
`subl %edx, %eax`	**%eax**	**0xFD**
`incl 8(%eax)`	**Mem em 0x108**	**0x14**

Operações de deslocamento

Instrução	Efeito	Descrição
sals **k, D**	**D << k → D**	Left shift (preenche com 0)
shls **k, D**	**D << k → D**	Left shift (preenche com 0)
sars **k, D**	**D >> k → D**	Arithmetic Right shift (ext sinal)
shrs **k, D**	**D >> k → D**	Logical Right shift (preenche com 0)

- k de 0 a 31 (codificação em 1 byte)
  - valor imediato ou conteúdo de %cl

Exercício

- Suponha que queremos gerar código para o trecho em C:

```
int x = ...;     // armazenado em 8(%ebp)
int n = ...;     // armazenado em 12(%ebp)
x <<= 2;
x >>= n;
```

Complete o código *assembly* que representa este trecho em *assembly*

`movl 8(%ebp), %eax`
`shll $2, %eax`
`movl 12(%ebp), %ecx`
`sarl  %cl, %eax`

Exemplo

```
int arith(int x, int y, int z) { // x em %ebp + 8
                                 // y em %ebp + 12
                                 // z em %ebp + 16

  int t1 = x^y;     // movl 12(%ebp),%eax → get y
                    // xorl 8(%ebp),%eax  → x xor y (t1 em %eax)
  int t2 = t1 >> 3; // sarl $3,%eax       → t1 >> 3 (t2 em %eax)
  int t3 = ~t2;     // notl %eax          → ~t2 (t3 em %eax)
  int t4 = t3 – z;  // subl 16(%ebp),%eax  → t3–z (t4 em %eax)
  return t4;
}
```

- Note que o compilador empregou otimização, mantendo variáveis locais em registrador (um único!)

Operação de divisão

Instrução	Efeito	Descrição
idivl S	**%eax = %edx:%eax / S (quociente)** **%edx = %edx:%eax % S (resto)**	Divisão (quociente e resto)

- Há variantes da operação para divisores de tamanho diferente (byte, word)
  - divisor byte → dividendo obtido em %ax

    quociente armazenado em %al e resto armazenado em %ah
  - divisor word → dividendo obtido em %dx:%ax, quociente armazenado em %al e resto armazenado em %dx

Exemplos de divisão

- Divisão com sinal: valores inteiros (x e y) nas posições 8(%ebp) e 12(%ebp)

```
movl   8(%ebp),%edx      → obtém x em %edx
movl   %edx,%eax  → copia x para %eax
sarl   $31,%edx   → estende o sinal de x
idivl  12(%ebp)   → x / y: quoc em %eax, resto em %edx
```

- Divisão unsigned:

```
movl   8(%ebp),%eax      → obtém x em %edx
xorl   %edx,%edx         → zera %edx (movl $0, %edx)
idivl  12(%ebp)   → x / y: quoc em %eax, resto em %edx
```

Declaração de Dados

- Assembler do gcc permite declaração de variáveis de vários tipos (na seção de dados .data).
- Exemplos:

```
vet: .byte  48, 0b00110000, 0x30, ‘0’
s1: .string “o resultado eh de %d\n”
```