Programando em Assembly

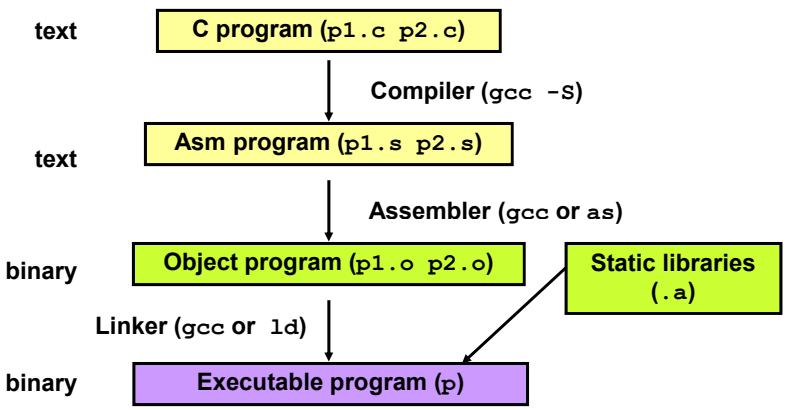
- precisa-se saber exatamente como interpretar & gerenciar a memória e como usar instruções de baixo nível para o processamento
- Não existem tipos e variáveis (apenas bytes na memória)
- não tem-se o auxílio da verificação de tipos do compilador, que ajudam a detectar vários erros
- código assembly é altamente dependente da máquina -> perde-se a portabilidade de uma linguagem de alto nível

Geração de Código objeto

– Arquivos fonte `p1.c p2.c`

– Comando de compilação: `gcc -O p1.c p2.c -o p`

• Otimizações (`-O`)



Diferenças entre assemblers

Formato Intel/Microsoft

```
lea  eax,[ecx+ecx*2]
sub  esp,8
cmp  dword ptr [ebp-8],0
mov  eax,dword ptr [eax*4+100h]
```

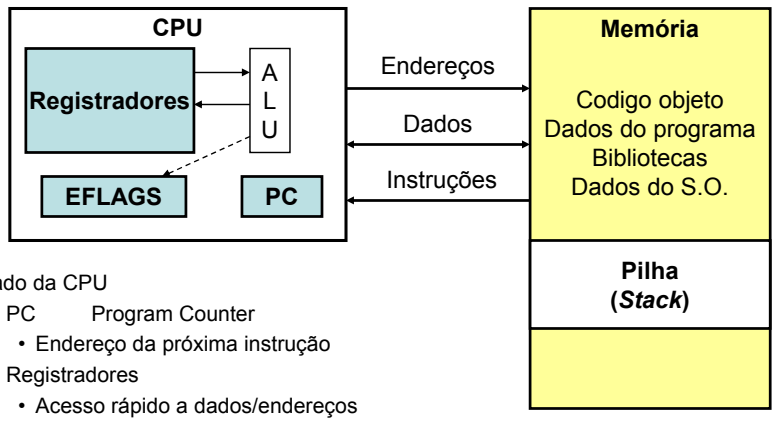
Formato GAS/Gnu

```
leal (%ecx,%ecx,2),%eax
subl $8,%esp
cmpl $0,-8(%ebp)
movl $0x100(,%eax,4),%eax
```

Diferenças Intel/Microsoft X GAS

- Operandos em ordem contrária
  `mov` Dest, Src `movl` Src, Dest
- Constantes não precedidas pelo '$', hexadecimais com 'h' no final
  `100h` `$0x100`
- Tamanho do operando indicado por operando ou sufixo de instrução
  `sub` `subl`
- entre outros…

Visão do programador Assembly



- Estado da CPU
  - PC Program Counter
    - Endereço da próxima instrução
  - Registradores
    - Acesso rápido a dados/endereços
  - Registrador EFLAGS (Códigos de condição)
    - Armazena informação sobre resultado da última operação artimética (overflow?, zero?, negativo?)
    - Usados para desvios condicionais

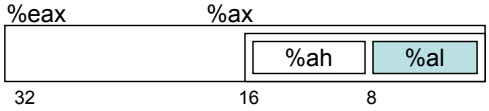
- Memória "plana"
  - Array de bytes endereçáveis
  - Contém código, dados e pilha
  - Pilha usada para gerenciar chamada a procedimentos

Características do Assembly

- Tipos de dados básicos
  - "integer" de 1, 2, ou 4 bytes
    - Valores int, short, long int, char
    - Endereços (equivalente a untyped pointer)
  - Dados em ponto flutuante de 4, 8, ou 10 bytes
  - Não existem tipos de dados complexos como arrays ou structs
    - Apenas bytes alocados de forma contígua na memória
- Operações primitivas
  - Funções aritméticas sobre o conteúdo de registradores ou dados na memória
  - Transferência de dados (mov) entre memória e registradores
    - Carregar dado de memória para registrador
    - Armazenar dado em registrador para memória
  - Controle de fluxo de execução
    - Desvios (jump) incondicionais
    - Desvios condicionais (dependendo de um bit em EFLAGS)

Registradores de Propósito Geral

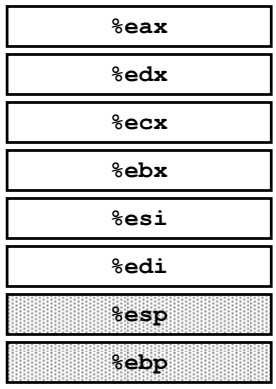
- São usados para armazenamento temporário de dados e endereços da memória (uso: %nome)
- Capacidade: 32 bits (=word do IA32)
- Todos podem armazenar tb. dados menores:



- `%esp` e `%ebp` reservados para uso especial ("p" de ponteiro)
- Exemplos:

```
movl %edx, %eax
incl  %ecx
addl  %eax, $0xff
```

%edx → %eax
%ecx+1 → %ecx
%eax+255 → %eax



O Registrador EFLAGS

Seus bits indicam a ocorrência de eventos (bit carry) ou o resultado da última operação aritmética, p.ex. t = a+b:

OF	SF	ZF	CF	PF	...

- OF (Overflow flag)=1 se houve overflow (negativo ou positivo) de representação com sinal
- SZ (Sign flag)=1 se t < 0
- ZF (Zero flag)=1 se t ==0
- CF (Carry flag)=1 se houve overflow de unsigned
- PF (Parity flag)=1 se o byte menos significativo possui o número par de bits 1

Os bits de EFLAGS são consultados por instruções de desvio condicional

Transferência de bytes

- Instrução mais frequente

`movl` *fonte*, *destino*

- Operando *fonte* pode ser:
  - Conteúdo de registrador — *modo registrador*
  - Conteúdo de memória apontado por registrador — *modo memória*
  - Constante — *modo imediato*
  - Símbolo — *modo direto*
  - Combinações dos anteriores
- Operando *destino* pode ser:
  - Registrador
  - Memória apontada por registrador
  - Combinações dos anteriores

Obs:

Operando em modo memória é caracterizado pelo uso de parenteses "( )"

Exemplo: `movl (%ebx), %ecx` // *Mem(ebx)* → R_{ecx}

Combinações de operandos de **movl**

	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;`

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

Exemplos

- Modo registrador

  `movl %eax, %ecx` // $R_{eax} \rightarrow R_{ecx}$
- Modo imediato

  `movl $0x100, %eax` // *const* $\rightarrow R_{eax}$
- Modo memória forma: *base (reg)*

  `movl %eax, 16(%ecx)` // $R_{eax} \rightarrow Mem[16+R_{ecx}]$

  `movl %eax, (%ecx)` // $R_{eax} \rightarrow Mem[R_{ecx}]$
- Modo direto

  `movl table, %ebx` // $Mem[table] \rightarrow R_{ebx}$

Todas as formas de operandos

Type	Form	Operand Value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$\mathbf{E}_a$	$Reg[\mathbf{E}_a]$	Register
Memory	Imm	$Mem[Imm]$	Absolute
Memory	$(\mathbf{E}_a)$	$Mem[Reg[\mathbf{E}_a]]$	Indirect
Memory	$Imm\,(\mathbf{E}_b)$	$Mem[Imm + Reg[\mathbf{E}_b]]$	Base + Displacement
Memory	$(\mathbf{E}_b, \mathbf{E}_i)$	$Mem[Reg[\mathbf{E}_b] + Reg[\mathbf{E}_i]]$	Indexed
Memory	$Imm\,(\mathbf{E}_b, \mathbf{E}_i)$	$Mem[Imm + Reg[\mathbf{E}_b] + Reg[\mathbf{E}_i]]$	Indexed
Memory	$(, \mathbf{E}_i, s)$	$Mem[Reg[\mathbf{E}_i] \cdot s]$	Scaled Indexed
Memory	$Imm\,(, \mathbf{E}_i, s)$	$Mem[Imm + Reg[\mathbf{E}_i] \cdot s]$	Scaled Indexed
Memory	$(\mathbf{E}_b, \mathbf{E}_i, s)$	$Mem[Reg[\mathbf{E}_b] + Reg[\mathbf{E}_i] \cdot s]$	Scaled Indexed
Memory	$Imm\,(\mathbf{E}_b, \mathbf{E}_i, s)$	$Mem[Imm + Reg[\mathbf{E}_b] + Reg[\mathbf{E}_i] \cdot s]$	Scaled Indexed

Figura 3.3 (p.137)

Sufixos

- Para muitas instruções existem várias variantes, definidas pelo seu sufixo (e.g. **`movl, addl, decl`**)
- Sufixo indica o tamanho do operando (resultado é armazenado alinhado com o byte menos significativo)

`b` = tamanho é de 1 byte

```
movb $0, (%eax)          //0 → Mem[R_eax]
```

`w` = tamanho é de 2 bytes

```
movw $0, (%eax)
```

`l` = tamanho de 4 bytes

```
movl $0, (%eax)
```

Grupos de Instruções

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

Exemplos: Aritmética Inteira & Instruções Lógicas

- Instruções de dois operandos

Formato		Efeito		
`addl`	*Src*,*Dest*	*Dest* = *Dest* + *Src*		
`subl`	*Src*,*Dest*	*Dest* = *Dest* − *Src*		
`imull`	*Src*,*Dest*	*Dest* = *Dest* * *Src*		
`sall`	*Src*,*Dest*	*Dest* = *Dest* << *Src*		
`sarl`	*Src*,*Dest*	*Dest* = *Dest* >> *Src*		aritmético
`shrl`	*Src*,*Dest*	*Dest* = *Dest* >> *Src*		lógico
`xorl`	*Src*,*Dest*	*Dest* = *Dest* ^ *Src*		*xor bit-a-bit*
`andl`	*Src*,*Dest*	*Dest* = *Dest* & *Src*		*and bit-a-bit*
`orl`	*Src*,*Dest*	*Dest* = *Dest* | *Src*		*or bit-a-bit*

Exemplos: Aritmética Inteira & Instruções Lógicas

- Instruções de um operando

Formato	Efeito
`incl` *Dest*	*Dest* + 1 →*Dest*
`decl` *Dest*	*Dest* – 1 →*Dest*
`negl` *Dest*	– *Dest* →*Dest*
`notl` *Dest*	~ *Dest* →*Dest*

Declaração de Dados

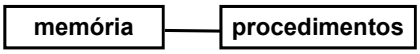
- Assembler do gcc permite declaração de variáveis de vários tipos (na seção de dados .data).
- Exemplos:

```
vet: .byte   48, 0b00110000, 0x30, '0'
s1: .string "o resultado eh de %d\n"
```

Resumo: Máquinas abstratas

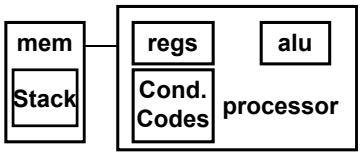
Modelos de máquinas **Dados**

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 word
3) 8-byte long word
4) Alocação contígua de bytes
5) Endereço do byte inicial

Exercício

- Tentem entender...

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp              Inicialização
    movl %esp,%ebp          (ignorar)
    pushl %ebx

    movl 12(%ebp),%ecx      } yp
    movl 8(%ebp),%edx       } xp
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx      Finalização
    movl %ebp,%esp          (ignorar)
    popl %ebp
    ret
```