[MAC0211] Laboratório de Programação I
Aula 4
Linguagem de Montagem

Alair Pereira do Lago

DCC–IME–USP

5 de março de 2015

Aula passada – arquitetura da família x86

Registradores de propósito geral

- A (acumulador)
- B (base)
- C (contador)
- D (dados)
- processador 8086 (16 bits): AX (AH,AL), BX (BH,BL), CX (CH,CL), DX (DH,DL), SP, BP, SI, DI
- processador 80386 (32 bits): EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI
- processador Intel x86-64 e AMD64 (64 bits): RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8–15

Aula passada – Instrução para transferência de dados: **MOV**

Copia o valor do segundo operando no primeiro operando.
O conteúdo do segundo operando permanece inalterado.

Formatos

- **MOV** ***reg,reg/mem/const***
- **MOV** ***mem,reg/const***

Operandos

- *reg* – um registrador de propósito geral
- *mem* – posição de memória (pode ser indicada por meio de uma constante, como `[1000]`, `ou por meio de um registrador, como [EBX]`)
- *const* – valor *constante*

Aula passada – Instrução para transferência de dados: **MOV**

Exemplos

Correto		**Incorreto**		**Problema**
MOV	AH,-14	MOV	AL,999	; 999 não cabe em 8 bits
MOV	AX,36H			
MOV	AL,'A'	MOV	EBX,DX	; não possuem o mesmo
MOV	EAX,EBX			; tamanho
MOV	BX,1000			
MOV	AX,[EBX]			
MOV	AX,[1000]			
MOV	AX,[1000+EBX]			
MOV	[1000],AX	MOV	[1000],[EBX]	; não há MOV direto
MOV	[1000],36H			; entre memórias

Aula passada – Considerações sobre o uso de memória como operando

Casos de não ambiguidade no tamanho

Acontecem quando a instrução envolve um operando do tipo *mem* e outro do tipo *reg*.
Neste caso, o número de palavras manipuladas na memória é determinado pelo tamanho de *reg*.
Exemplo: a instrução

MOV AX, [1000]

copia 2 palavras da memória (posições 1000 e 1001) porque o registrador AX é de 16 bits.

Aula passada – Considerações sobre o uso de memória como operando

Casos de ambiguidade no tamanho

Acontecem quando a instrução envolve um operando do tipo *mem* e outro do tipo *const*. Exemplo:

```
MOV [EBX], 5
```

Neste caso, o número de palavras manipuladas na memória pode ser determinado de duas maneiras:

1. a arquitetura do processador determina a quantidade de bits *default* (16 bits, 32 bits, 64 bits)
2. uso de notação para determinar o quantidade de bytes manipulados. Exemplo:

```
MOV   BYTE [EBX],5    ; BYTE para designar 8 bits
MOV   WORD [EBX],5    ; WORD para designar 16 bits
MOV   DWORD [EBX],5   ; DWORD para designar 32 bits
```

Aula passada – Um "parênteses": convenções de notação

Soluções para problemas de ambiguidade

- Problema-exemplo 1: **50** pode ser um número em notação decimal ou hexadecimal
- Solução: usar sufixos que determinam o sistema de numeração. Por exemplo, **50D** designa um número decimal, enquanto **50H** é hexadecimal (**10B** é binário)
- Problema-exemplo2 (consequência da solução anterior): **AH, BH, CH e DH** designam números hexadecimais, mas também são nomes de registradores
- Solução: na linguagem de montagem, fazer com que todos os números hexadecimais sejam também iniciados por um dígito em $0, 1, \ldots, 9$ [1]. Por exemplo, **0AH** designa o número hexadecimal A e não o registrador AH

[1] Na linguagem C, números hexadecimais são precedidos por "0x"

Aula passada – Instrução para troca de dados: **XCGH**

Troca os valores dos operandos (ou seja, faz o primeiro receber o valor do segundo e o segundo receber o valor do primeiro).
Os operandos precisam ser do mesmo tamanho.

Formatos

- **XCGH** *reg,reg/mem*
- **XCGH** *mem,reg*

Exemplos

```
XCHG   AH,BL
XCHG   AH,[BL]
XCHG   [EBX],AH
```

Aula passada – Instruções aritméticas – soma: **ADD**

Soma o valor do segundo operando ao valor do primeiro, armazenando o resultado no primeiro operando.
O valor do segundo operando permanece inalterado.

Formato

- **ADD** ***reg,reg/mem/const***

Exemplos

```
ADD     BL,10       ; BL <- BL + 10
ADD     BL,AL       ; BL <- BL + AL
ADD     BL,[1000]       ; BL <- BL + [1000]
```

Aula passada – Instruções aritméticas – subtração: **SUB**

Subtrai o valor do segundo operando do valor do primeiro, armazenando o resultado no primeiro operando.
O valor do segundo operando permanece inalterado.

Formato

- **SUB** ***reg,reg|mem|const***

Exemplos

```
SUB     BL,10       ; BL <- BL - 10
SUB     BL,AL       ; BL <- BL - AL
SUB     BL,[1000]       ; BL <- BL - [1000]
```

Aula passada – Instruções aritméticas – incremento e decremento: **INC** e **DEC**

Incrementa ou decrementa o valor do operando em 1.

Formato

- **INC** ***reg/mem***
- **DEC** ***reg/mem***

Exemplos

INC CX $\longleftrightarrow$ ADD CX, 1
DEC CX $\longleftrightarrow$ SUB CX, 1

Instruções aritméticas – multiplicação: **MUL**

Formato

Não tem o mesmo formato que as operações aritméticas anteriores porque a multiplicação pode gerar um número que tem até o dobro de bits que os operandos.
MUL é válida apenas para a multiplicação de números sem sinal.

- **MUL** ***reg/mem***
  Se o operando tem 8 bits, por exemplo,

  MUL BH

  então o comando equivale a

  AX <– AL × BH

  Ou seja, o operando é sempre multiplicado pelo valor em AL e o resultado é armazenado em AX.

Instruções aritméticas – multiplicação: **MUL**

Formato

- **MUL** ***reg/mem***

  Se o operando tem 16 bits, por exemplo,

  MUL BX

  então o comando equivale a

  DX:AX <– AX × BX

  Ou seja, o operando é sempre multiplicado pelo valor em AX e o resultado de 32 bits é armazenado em 2 registradores de 16 bits: os 16 primeiros bits em AX e os 16 últimos em DX.

Instruções aritméticas – multiplicação: **MUL**

Formato

- **MUL** ***reg/mem***

  Se o operando tem 32 bits, por exemplo,

  MUL EBX

  então o comando equivale a

  EDX:EAX <– EAX × EBX

  Ou seja, o operando é sempre multiplicado pelo valor em EAX e o resultado de 64 bits é armazenado em 2 registradores de 32 bits: os 32 primeiros bits em EAX e os 16 últimos em EDX.

Obs.: O MUL não pode ser usado com um valor constante. Por exemplo, o comando a seguir é inválido: MOV 7

Instruções aritméticas – divisão inteira: **DIV**

Formato

Funciona de forma inversa ao MUL.
DIV é válida apenas para a divisão de números inteiros sem sinal.

- **DIV** ***reg/mem***

  Por exemplo,

  DIV BH

  divide o valor em AX pelo valor em BH, armazenando o quociente em AL e o resto em AH

Divisor	**Dividendo**	**Resto**	**Quociente**
32 bits	EDX:EAX	EDX	EAX
16 bits	DX:AX	DX	AX
8 bits	AX	AH	AL

Instruções aritméticas – divisão: **DIV**

Situações que geram exceção:

- divisão por zero
- transbordamento (*overflow*) – ocorre quando o resto gerado na divisão não cabe no registrador. Exemplo:

```
MOV   AX,1024
MOV   BH,2
DIV   BH
```

Quociente deveria ser armazenado em AL, mas 512 ocupa no mínimo 10 bits!

Instruções aritméticas – divisão e multiplicação envolvendo números com sinal: **IMUL** e **IDIV**

Funcionam de modo análogo aos comandos DIV e MUL, mas podem ser aplicados a números com sinal.

Formato

- **IMUL** ***reg/mem***
- **IDIV** ***reg/mem***

Instruções lógicas: **AND**, **OR**, **NOT**

O resultado é armazenado no primeiro operando.

Formato

- **AND** *reg,reg/mem/const* ou **AND** *mem,reg/const*
- **OR** *reg,reg/mem/const* ou **OR** *mem,reg/const*
- **XOR** *reg,reg/mem/const* ou **XOR** *mem,reg/const*
- **NOT** *reg/mem* **; inverte os bits**

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

NOT	0	1
	1	0

Exemplos

```
AND   AX,BX   |   OR   CX,5Fh   |   NOT   AX
```

"Truques" com números binários

A operações lógicas podem ser usadas para:

- "resetar"/limpar (= atribuir zero a) bits
- "setar" (= atribuir 1 a) bits
- inverter bits
- examinar bits

Para "setar" um bit

Exemplo: setar o 3º bit menos significativo do AH.

```
OR   AH, 00000100B
```

Para "resetar" um bit

Exemplo: resetar o 3º bit menos significativo do AH.

```
AND   AH,11111011B
```

"Truques" com números binários

Para inverter bits específicos

Exemplo: Inverter o quarto bit mais significativo do AX.

```
XOR   AX,1000H
```

Para examinar bits específicos

Exemplo: determinar o valor do quarto bit mais significativo do AX.

```
AND   AX,1000H
```

Se o resultado da operação for zero, o bit desejado vale 0. Senão, o bit vale 1.

"Truques" com números binários

Para zerar um registrador

Exemplo: zerar o registrador ECX.

```
XOR   ECX,ECX
```

Para verificar se um registrador é nulo

Exemplo: verificar se ECX é nulo.

```
OR   ECX,ECX
```

Obs.: se o registrado for nulo, então a *flag* zero é setada.

Instrução para trocar sinal – **NEG**

Gera o Complemento 2 do operando e armazena-o no próprio operando (ou seja, troca o sinal do operando).

Formato

- **NEG** ***reg/mem***

Exemplo

```
NEG  EAX   ⟷   NOT  EAX
               ADD  EAX,1
```

Instruções para a transferência de controle
Salto incondicional – **JMP**

Transfere a execução para o endereço especificado pelo rótulo

Formato:

- **JMP** ***rot***

Exemplo de programa

```
          :
inicio:   MOV   AX,5
          ADD   AX,AX
          :
          JMP   inicio
          :
```

Instrução para comparação – CMP

Compara o valor do primeiro operando com o valor do segundo.

Formato:

- **CMP** ***reg,reg/mem/const***

Resultado da comparação é armazenado em uma *flag*.

Exemplos

```
CMP   AX,5
CMP   CX,[EBX]
```

Instruções para saltos condicionais

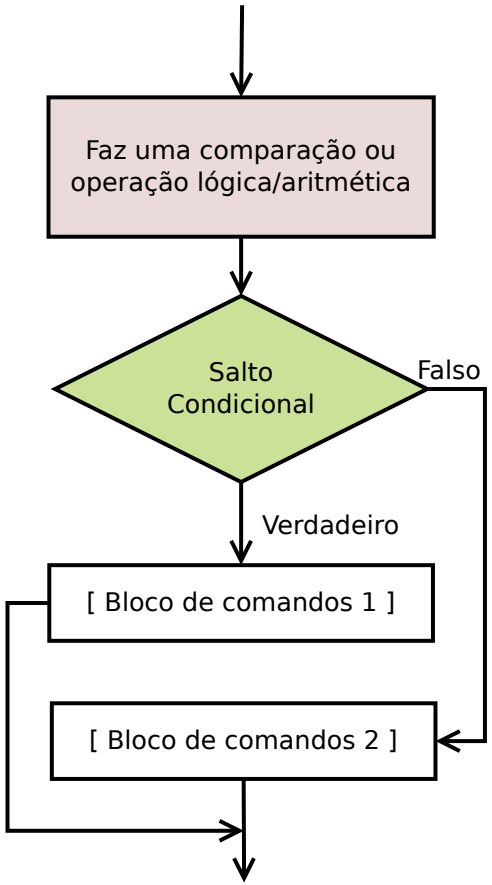
Variações:

- **JE** – *jump if equal* (salta se é igual)
- **JNE** – *jump if equal* (salta se não é igual)
- **JG** – *jump if greater* (salta se é maior)
- **JGE** – *jump if greater or equal* (salta se é maior ou igual)
- **JNG** – *jump if not greater* (salta se não é maior)
- **JNGE** – *jump if not greater or equal* (salta se não é maior ou igual)
- **JL** – *jump if less* (salta se é menor)
- **JLE** – *jump if less or equal* (salta se é menor ou igual)
- **JNL** – *jump if not less* (salta se não é menor)
- **JNLE** – *jump if less or equal* (salta se não é menor ou igual)

Esses saltos consideram o resultado da última comparação realizada.
Importante: esses saltos consideram que a comparação envolveu números com sinal (*signed*).

Estrutura de um comando "if-else"

se (expressão) então «bloco 1» senão «bloco 2» fim;



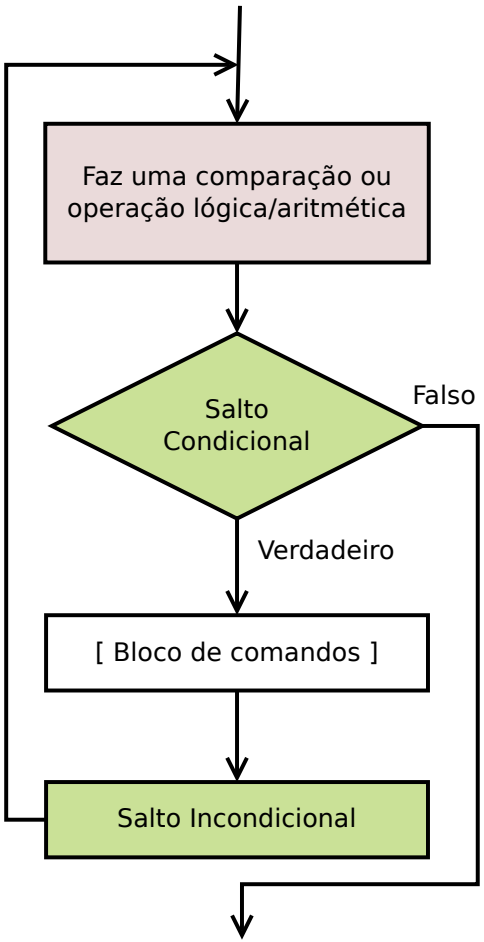
Exemplo de implementação em assembly

se (cont<15) então «bloco 1» senão «bloco 2»;

```
            :
            CMP   CX,15
            JNL   senao     ; se contador >= 15
                            ; então vai para o bloco 2
            :               ; bloco 1 de comandos
            JMP   fimse
            :
senao:      :               ; bloco 2 de comandos
            :
fimse:      :               ; instruções depois do SE
```

Estrutura de um comando "while"

enquanto (expressão) faça «bloco de comandos» fim;



Exemplo de implementação em assembly

while (cont<15) faça «bloco de comandos» fim;

```
            :
            MOV     CX,0            ; inicializa o contador
inicio:     CMP     CX,15
            JGE     fim             ; se contador >= 15,
                                    ; sai do laço
            :                       ; bloco de comandos
            INC     CX              ; incrementa o contador
            JMP     inicio          ; vai para o início do laço
fim:        MOV     AX,[EBX]        ; 1ª instrução fora do laço
            :
```

Instruções para saltos condicionais – JZ e JNZ

Variações:

- **JZ** – *jump if zero* (salta se é nulo)
- **JNZ** – *jump if not zero* (salta se não é nulo)

Esses saltos consideram o resultado da última operação aritmética ou lógica realizada.

Exemplo de programa

```
        MOV   CX, 5     ; laço será executado 5 vezes
inicio: ⋮
        ⋮               ; bloco de comandos do laço
        DEC   CX        ; contador <- contador - 1
        JNZ   inicio
```

Instruções para saltos condicionais (versão *unsigned*)

Esses saltos consideram o resultado da última comparação realizada.
Consideram também que a comparação envolveu números sem sinal (*unsigned*).

Variações:

- **JA** – *jump if above* (salta se é maior)
- **JAE** – *jump if above or equal* (salta se é maior ou igual)
- **JNA** – *jump if not above* (salta se não é maior)
- **JNAE** – *jump if not above or equal* (salta se não é maior ou igual)
- **JB** – *jump if below* (salta se é menor)
- **JBE** – *jump if below or equal* (salta se é menor ou igual)
- **JNB** – *jump if not below* (salta se não é menor)
- **JNBE** – *jump if below or equal* (salta se não é menor ou igual)

Chamadas ao sistema operacional

Chamadas ao sistema (= *system calls*, ou somente *syscalls*)

- Forma por meio da qual programas solicitam serviços ao núcleo do SO
- Exemplos de serviços: operações para leitura e escrita em arquivos, criação e execução de novos processos, etc.

Chamadas ao sistema – como fazê-las em *assembly*?

- colocar número da chamada ao sistema em EAX
- colocar 3 primeiros argumentos em EBX, ECX, EDX (mais ESI e EDI se necessário)
- gerar a interrupção de chamada ao sistema (instrução INT 0x80)
- quando há valor de retorno, ele é colocado em EAX

Montadores

GCC Inline Assembly

- Suporte à arquitetura x86 bastante satisfatório
- Possibilita que código em linguagem de máquina seja inserido em programas em C
- Usa o GAS

GAS – *GNU Assembler*

- Por padrão, segue a sintaxe da AT&T (e não a da Intel, usada pela maioria dos montadores). Mas, em suas versões mais novas, aceita também a sintaxe da Intel
- Plataformas: Unix-like, Windows, DOS, OS/2
- Parte do pacote `binutils` do Linux
- Nome do executável: `gas` ou simplesmente `as`

Montadores

NASM – *Netwide Assembler*

- Bastante usado (confiável para o desenvolvimento de aplicações de grande porte, de uso comercial e industrial)
- Plataformas: Windows, Linux, Mac OS X, DOS, OS/2
- Instalação: pacote `nasm` do Linux

```
$ sudo apt-get install nasm
```

"Hello, world!" para NASM (versão 32 bits) – hello.asm

```
global _start       ; exporta para o ligador (ld) o ponto de entrada

section .text
_start:

    ; sys_write(stdout, mensagem, tamanho)

    mov eax, 4            ; chamada de sistema sys_write
    mov ebx, 1            ; stdout
    mov ecx, mensagem     ; endereço da mensagem
    mov edx, tamanho      ; tamanho da string de mensagem
    int 80h               ; chamada ao núcleo (kernel)

    ; sys_exit(return_code)

    mov eax, 1          ; chamada de sistema sys_exit
    mov ebx, 0          ; retorna 0 (sucesso)
    int 80h             ; chamada ao núcleo (kernel)

section .data
mensagem: db 'Hello, world!',0x0A    ; mensagem e quebra de linha
tamanho:  equ $ - mensagem           ; tamanho da mensagem
```

"Hello, world!" para GAS (versão 32 bits) – hello.S

```
.text

   .global  _start     # exporta para o ligador (ld) o ponto de entrada

_start:

    # sys_write(stdout, mensagem, tamanho)
    movl    $4, %eax              # chamada de sistema sys_write
    movl    $1, %ebx              # stdout
    movl    $mensagem, %ecx       # endereço da mensagem
    movl    $tamanho, %edx        # tamanho da string de mensagem
    int     $0x80                 # chamada ao núcleo (kernel)

    # sys_exit(codigo_retorno)
    movl    $1, %eax      # chamada de sistema sys_exit
    movl    $0, %ebx      # retorna 0 (sucesso)
    int     $0x80         # chamada ao núcleo (kernel)

.data
mensagem:
    .ascii  "Hello, world!\n"     # mensagem e quebra de linha
    tamanho =   . - mensagem      # tamanho da mensagem
```

Geração do executável

Passo 1 – Geração do código objeto

- Usando NASM, em um computador de 32 bits:

```
$ nasm -f elf32 hello.asm
```

- Usando NASM, em um computador de 64 bits:

```
$ nasm -f elf64 hello.asm
```

- Usando o GAS:

```
$ as -o hello.o hello.S
```

Os comandos acima gerarão um arquivo `hello.o`.

Geração do executável

Passo 2 – Ligação (geração do código de máquina)

```
$ ld -s -o hello hello.o
```

O comando acima gerará o arquivo executável `hello` .

Estrutura de um programa em linguagem de montagem

Seções

- `.text` – onde fica o código-fonte; é uma seção só para leitura
- `.data` – onde fica os dados/variáveis
- `.bss` – onde fica os dados/variáveis não inicializados

Bibliografia e materiais recomendados

- Capítulos 3, 4 e 6 do livro *Linux Assembly Language Programming*, de B. Neveln
- Livro *The Art of Assembly Language Programming*, de R. Hyde
  `http://cs.smith.edu/~thiebaut/ArtOfAssembly/artofasm.html`
- *The Netwide Assembler* – NASM
  `http://www.nasm.us/`
- *GNU Assembler* – GAS
  `http://sourceware.org/binutils/docs-2.23/as/index.html`
- *Linux assemblers: A comparison of GAS and NASM*
  `http://www.ibm.com/developerworks/linux/library/l-gas-nasm/index.html`
- Tabela de chamadas ao sistema no Linux
  `http://www.ime.usp.br/~kon/MAC211/syscalls.html`

Cenas dos próximos capítulos...

- Declaração de variáveis e constantes
- Mais exemplos de programas

Apêndice: "Hello, world!" para NASM (versão 64 bits)

```
global _start          ; exporta para o ligador (ld) o ponto de entrada

section .text
_start:

    ; sys_write(stdout, mensagem, tamanho)

    mov rax, 1             ; chamada de sistema sys_write
    mov rdi, 1             ; stdout
    mov rsi, mensagem      ; endereço da mensagem
    mov rdx, tamanho       ; tamanho da string de mensagem
    syscall                ; chamada ao núcleo (kernel)

    ; sys_exit(return_code)

    mov rax, 60            ; chamada de sistema sys_exit
    mov rdi, 0             ; retorna 0 (sucesso)
    syscall                ; chamada ao núcleo (kernel)

section .data
mensagem: db 'Hello, world!',0x0A     ; mensagem e quebra de linha
tamanho:  equ $ - mensagem            ; tamanho da mensagem
```