# Variáveis, Registradores e Movimentação de Dados

João Marcelo Uchôa de Alencar

14 de março de 2023

Introdução

Primeiro Programa

Declaração de Variáveis

Dados Imediatos

Registradores

Movimentação de Dados

Dados de Caracteres

Erros

Programa Completo

Resumo

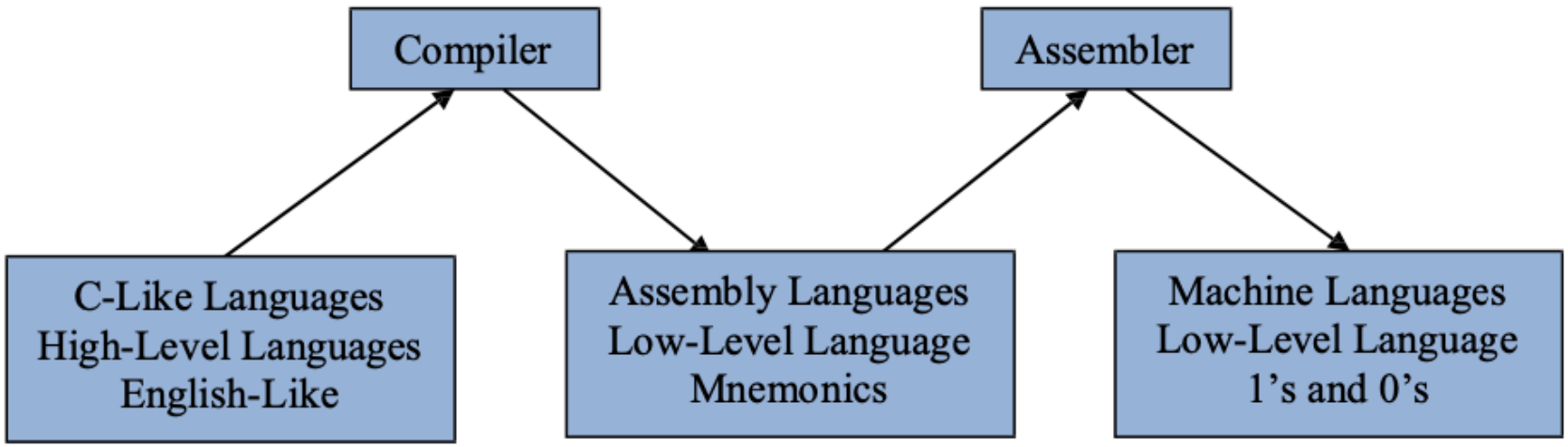
# Introdução

- C, C++ e Java tem similaridades com linguagens naturais, o que facilita a compreensão de programas.
- Uma linguagem de alto nível pode ser convertida em várias linguagens de baixo nível:
  - Tradutores.
  - Compiladores.
- A linguagem nativa de mais baixo nível de um computador é conhecida como **liguagem de máquina**:
  - É específica de cada arquitetura.
  - Composta por sequências de 0s e 1s.

# Linguagem de Montagem

- Programar em linguagem de máquina é tedioso e erros são facilmente cometidos.
- Usamos **linguagem de montagem** ou *assembly*:
  - Abreviaturas para as instruções.
  - Nomes de variáveis para localizações de memória.
  - Relação uma para uma entre instruções de montagem e instruções de máquina.
- A vantagem em relação a linguagens de alto nível:
  - Aprendizado dos detalhes de uma arquitetura.
  - Possibilidade de escrever programas eficientes tanto em tempo de execução quanto consumo de memória.
- Compiladores convertem uma linguagem de alto nível para uma linguagem de baixo nível.

# Compiladores e Montadores



# Montadores

- MASM (*Microsoft Assembler*) para plataformas Windows.
- NASM (*Netwide Assembler*) para várias plataformas.

Tem mais semelhanças do que diferenças. Vamos tentar mostrar os exemplos nas duas soluções, começando pela primeira.

# Primeiro Programa

Entrada e saída não é trivial em *assembly*, por isso vamos começar com algo mais simples do que um "Olá Mundo".

```
int main(){
        int num1,num2;
        num1= 5;
        num2= num1;
        return 0;
}
```

Considerando o programa em C acima...

# Primeiro Programa

```
.686
.model flat, c
.stack 100h .data
num1 sdword ? ; first number
num2 sdword ? ; second number
.code
main proc
     mov num1,5 ; initialize num1 with 5
     mov eax,num1 ; load eax with contents of num1
     mov num2,eax ; store eax in num2
     ret
main endp
     end
```

# Diretivas e Instruções

- Diretivas são comandos para o montador:
  - O *.686* no início do programa indica ao montador que o código deve executar em um processador 686 (Pentium Pro ou mais novo).
  - *.model flat* indica o modo protegido de memória, com endereços de 32 *bits* e no máximo 4GB de RAM.
  - *.stack* indica o tamanho da pilha em hexadecimal. Mas na frente vamos ver a finalidade da pilha.
  - *proc* indica que o nome do procedimento (função) é *main*.
  - *main endp* indica o fim do procedimento e *end* indica o fim do programa.
- Instruções são comandos para a CPU:
  - *mov* movimenta dados.
  - *ret* retorna o controle do procedimento.
- Operadores também informam ao montador o que fazer com determinada instrução.

# Rótulos, Código de Operação, Operandos e Comentários

| Rótulo | Código de Operação | Operando | Comentário |
|---|---|---|---|
| | .data | | |
| num1 | sword | ? | ; first number |
| num2 | sword | ? | ; second number |
| | .code | | |
| main | proc | | |
| | mov | num1, 5 | ; initialize num1 with 5 |
| | mov | eax, num1 | ; load eax with ... |
| | mov | num2, eax | ; store eax in num2 |
| | ret | | |

# Declaração de Variáveis

- O segmento *.data* no exemplo acima declara duas variáveis, *num1* e *num2*.
- Regras para nomeação de variáveis:
  - Devem começar por uma letra, seguida de números ou dígitos.
  - Símbolos especiais como , _ e $ são permitidos, mas é melhor evitar.
  - *CAT* e *cat* são duas variáveis diferentes.
  - Tamanho máximo de 247 caracteres.
- No exemplo, o código de operação com a diretiva *sdword* significa *signal double word*, um valor de 32 *bits*.
- O símbolo de interrogação indica que o montador não deve inicializar a variável. Se for colocado um número, será o valor inicial da variável. Em C, *int num3 = 5;* equivale a *num3 sdword 5* em *assembly*.

# Declaração de Variáveis

| Tipo | Número de *bits* | Faixa (inclusiva) |
|---|---|---|
| sdword | 32 | –2,147,483,648 até +2,147,483,647 |
| dword | 32 | 0 até +4,294,967,295 |
| sword | 16 | –32,768 até +32,767 |
| word | 16 | 0 até +65,535 |
| sbyte | 8 | –128 até +127 |
| byte | 8 | 0 até +255 |

# Dados Imediatos

- Se variável não for inicializada no segmento *.data*, como pode receber um valor?
- A instrução *mov* (mover, inglês *move*) transporta informação do operando à direita, chamado origem, para o operando na esquerda, chamado destino.
- Tipos de operandos:
  - Constante, chamada de *imm* (do inglês, *immediate*).
  - Memória, chamada de *mem* (do inglês, *memory*).
  - Registrador, chamada de *reg* (do inglês, *register*).

# Instruções *mov*

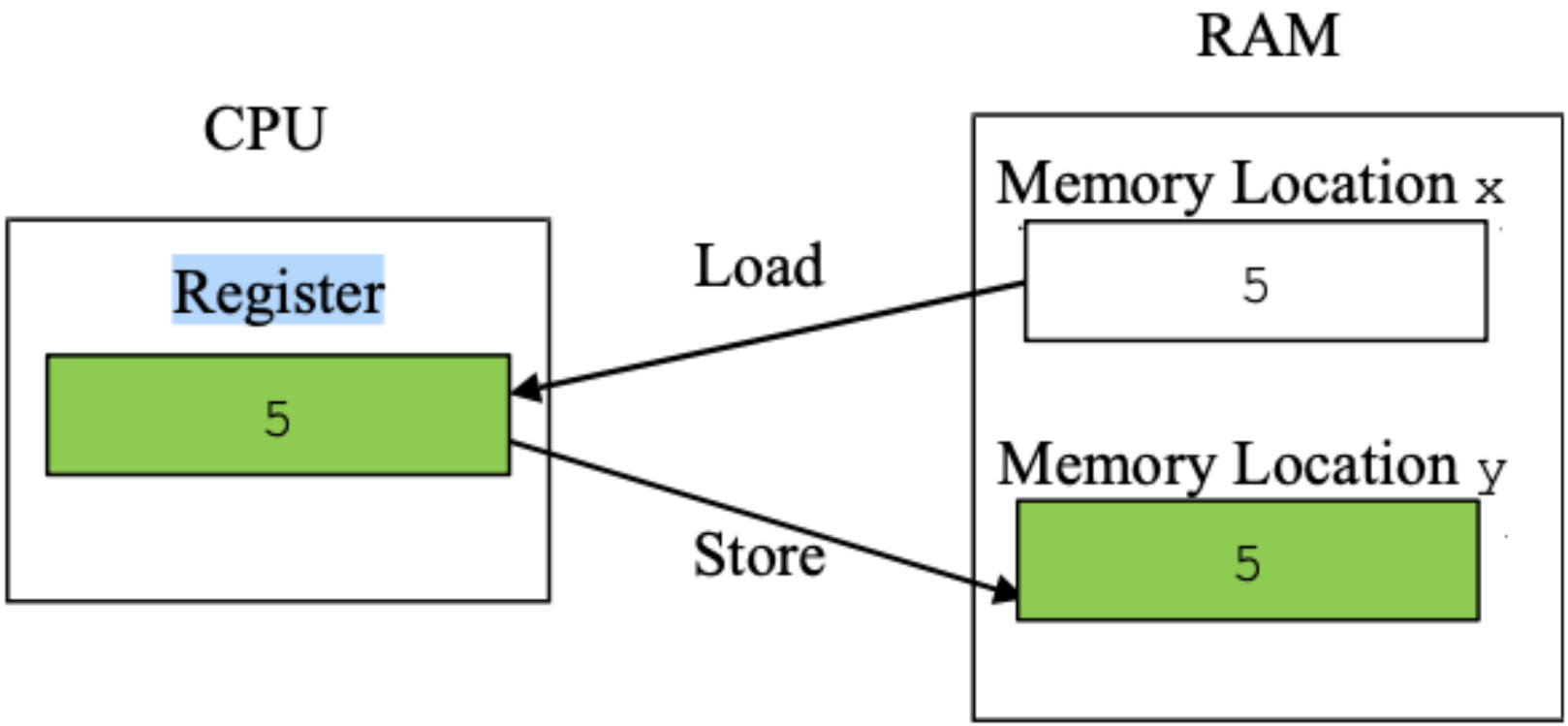
| Instrução | Significado |
|---|---|
| mov mem,imm | mover constante para memória |
| mov reg,mem | mover conteúdo da memória para registrador |
| mov mem,reg | mover conteúdo do registrador para memória |
| mov reg,imm | mover constante para registrador |
| mov reg,reg | mover conteúdo entre registradores |

A atribuição *num1 = 5;* em C equivale a *mov num1, 5* em *assembly*. A variável *num1* é uma **localização de memória** declarada e 5 é uma **constante ou valor imediato**.

# Registradores

- Como mover dados de um endereço de memória para outro?
- Em C/C++/Java, a instrução *y=x* não copia diretamente o conteúdo do endereço *&x* para o endereço *&y*.
- Em geral, o dado em *&x* é copiado para a CPU, que depois armazena em *&y*.
- O passo intermediário dentro da CPU faz uso de um registrador, memória de alta velocidade:
  1. Uma operação de carga (*load*) carrega o conteúdo de *&x* em um registrador.
  2. Uma operação de armazenamento (*store*) armazena o conteúdo do registrador em *&y*.
- Enquanto alguns computadores possuem operações *load* e *store* independentes, em processadores Intel podemos fazer tudo com *move*.

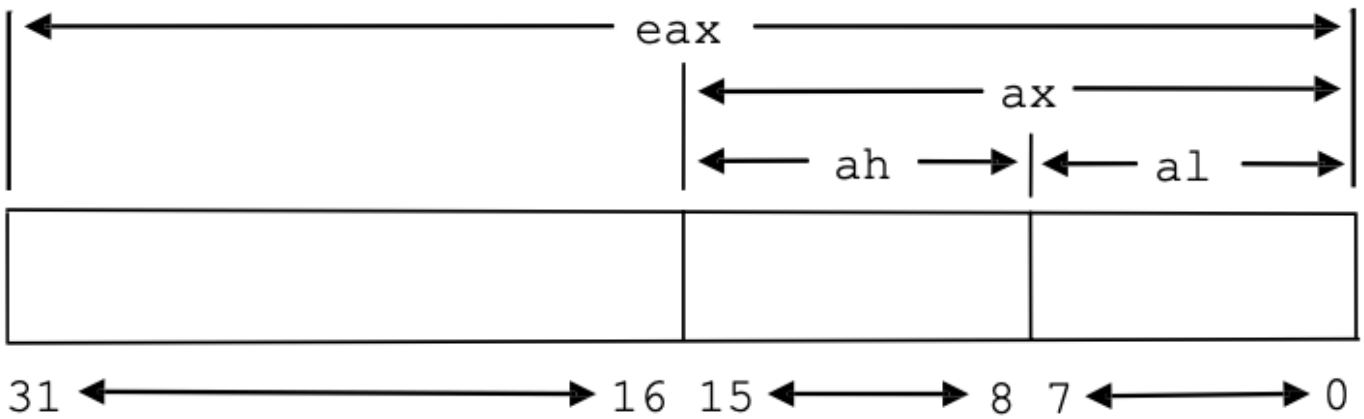
# Registradores



# Registradores

- Os registradores acessíveis aos programadores de linguagem de montagem são chamados **registradores de propósito geral**.
- Na sua versão original, a arquitetura Intel era de 16 *bits* (anos 70), portanto os registadores tinha comprimento de 16 *bits*:
  - *ax*, *bx*, *cx* e *dx*.
- O processador 386 (anos 80) passou a utilizar registradores de 32 *bits*:
  - *eax*, *ebx*, *ecx* e *edx*.
- AMD e Intel, ao introduzirem arquiteturas 64 *bits* (anos 2000), adotaram nova nomenclatura:
  - *rax*, *rbx*, *rcx* e *rdx*.
  - Vamos deixar para ver a arquitetura de 64 *bits* mais adiante.

# Registradores




A mesma divisão vale para os registradores *b*, *c* e *d*.

# Registradores

- As letras *a*, *b*, *c* e *d* indicam propósitos especiais:
  - *eax* costuma ser usado como acumulador em operações aritméticas.
  - *ebx* é o registrador base para opções de *array*.
  - *ecx* atua como contador para laços.
  - *edx* é registrador de dados.
- Outros registradores importantes:
  - *ebp* e *esp* são para gestão da pilha, o acesso é indireto.
  - *esi* e *edi* são registradores de índices para *arrays* e cadeias de caracteres.
  - *cs*, *ds* e *ss* são registradores de 16 *bits* que apontam para os segmentos de código, dados e pilha.
  - *es*, *fs* e *gs* também são usados para apontar para segmentos de dados.
  - *eip* é o contador de programa e *eflags* indica o *status* da CPU.

# Resumo dos Registradores de 32 *bits*

| Registrador | Nome | Equivalentes 16/8 *bits* | Descrição |
|---|---|---|---|
| eax | Acumulador | ax, ah, al | Lógica e artimética |
| ebx | Base | bx, bh, bl | Arrays |
| ecx | Contador | cx, ch, cl | Laços |
| edx | Dados | dx, dh, dl | Aritmética |
| esi | Índice Origem | si | Strings e arrays |
| edi | Índice Destino | di | Strings e arrays |
| esp | Ponteiro da Pilha | sp | Topo da Pilha |
| ebp | Base da Pilha | bp | Base da Pilha |
| eip | Contador de Programa | ip | Próxima Instrução |
| eflags | Flags | flags | Status e Controle |

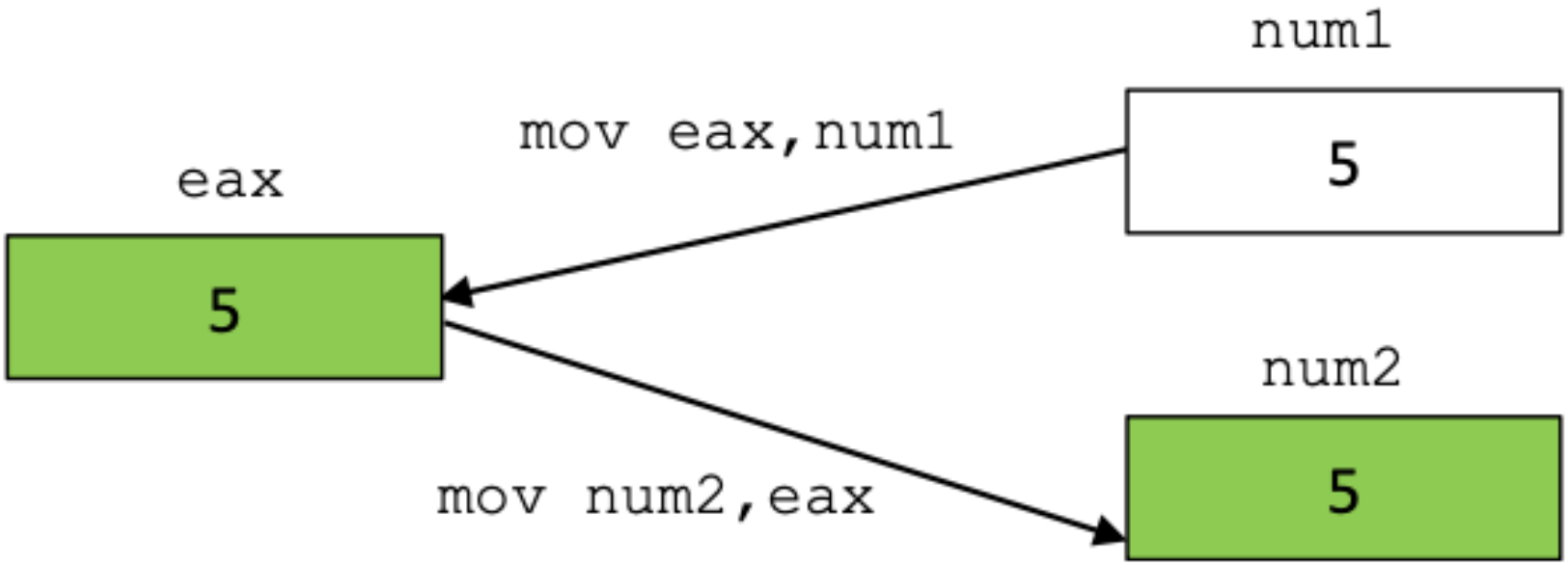
# Movimentação de Dados

- Não existe a opção *mov mem, mem*.
- É um padrão que se repete para a maioria das instruções.

```
; num2 = num1
mov eax, num1 ; carrega em eax o conteúdo de &num1
mov num2, eax ; armazena o conteúdo de eax em &num2
```

- Outras arquiteturas, além da Intel, adotam essa abordagem.
- Você consegue imaginar por que?

# Movimentação de Dados



# Movimentação de Dados

- Registradores são muito mais rápidos que memória principal.
- O ideal seria programar apenas usando os registradores.
- Entretanto, o número de registradores é limitado.
- Alguns ainda tem uso específico:
  - *ebx* usado para indexar um *array*...
  - *ecx* usado para controlar um laço *for*...
  - *edx* usado para uma multiplicação...
  - resta apenas *eax*.
- É papel do programador gerenciar o uso dos registradores.
- Programar usando variáveis na memória torna o código mais legível.

# Dados de Caracteres

- Dados de caracteres também podem ser declarados.
- Por exemplo, em C/++:

```
char grade1;
char grade2='A';
```

- *grade1* não foi inicializada. *grade2* tem o caractere *A*.
- Em *assembly*:

```
grade1 byte ?
grade2 byte 'A'
```

- *Strings* de *bytes*:

```
grades byte 'A', 'B', 'C' ; individualmente
name byte 'Abe' ; cadeia
```

# Dados de Caracteres

- Assim como em C, terminamos *strings* com 0.

```
name byte 'Abe', 0
```

- Perceba que o tamanho do caractere é *byte*.
- Até agora, usamos *mov* com *sdword*.
- Considere o trecho abaixo em C:

```
char letter1, letter2;
letter1 = 'A';
letter2 = letter1;
```

- Como faríamos em *assembly*?

# Dados de Caracteres

```
                .data
letter1         byte            ?
letter2         byte            ?
                .code
                ; letter1 = 'A'
                mov letter1, 'A' ; armazena 'A' em letter1
                ; letter2 = letter1
                mov al, letter1  ; carrega em al o
                                 ; valor de letter1
                mov letter2, al  ; armazena al em letter2
```

- São usados registradores de 8 *bits*.
- Veremos instruções específicas de *strings* mais adiante.

# Erros

- Erros de sintaxe: *move* no lugar de *mov*.
- Erros de tempo de execução: divisão por zero.
- Erros de lógica de programação.

Não são problemas apenas do *assembly*, ocorrem em todas as linguagens.

# Programa Completo

- Vamos ver mais adiante como fazer entrada e saída em *assembly*.
- É um processo mais complexo que nas outras linguagens.
- Envolve a invocação de chamadas de sistemas.
- *Assembly inline* ou embutido em C:
  - Usar a diretiva *__asm* para embarcar código *assembly* em programas C.
  - Podemos utilizar *printf*, *scanf*, *cout*, *cin*, etc para fazer E/S.

# Programa Completo

```
#include <stdio.h>
int main(){
    int num1,num2;
    num1 = 5;
    num2 = num1;
    printf("%s%d n","The answer is: ",num2);
    return 0;
}
```

# Programa Completo

```
#include <stdio.h>
int main(){
    int num1,num2;
    num1 = 5;
    __asm {
        mov eax,num1
        mov num2,eax
    }
    printf("%s%d n","The answer is: ",num2);
    return 0;
}
```

# Resumo

- Diretivas dizem ao montador o que fazer e instruções dizem ao processador o que fazer.
- Um *byte* são 8 *bits*, uma *word* são 16 *bits*, uma *double word* - *dword* são 32 *bits* e uma *quad word* - *qword* são 64 *bits*.
- Os quatros registradores de propósito geral são *eax*, *ebx*, *ecx* e *edx*.
- Dados imediatos são dados que aparecem com operandos.

# Resumo

- A instrução *mov* não pode mover dados de uma localização de memória para outra localização de memória.
- Em geral, um nome de variável começa com uma letra e é seguida por uma combinação de letras e números. O uso de caracteres especiais deve ser evitado.
- Inteiros são *sdword*, caracteres são *byte*.
- *Assembly inline* é útil para testar segmentos de código.
- Mensagens de erro são exibidas para problemas em sintaxe e execução, mas não para problemas na lógica do programa.