Introdução ao Assembly

Arquitetura e Organização de Computadores

Prof. Lucas de Oliveira Teixeira

UEM

Introdução

Introdução

- O Assembly ou linguagem de montagem é uma linguagem de baixo nível para programação de computadores.
- As suas características são inerentes ao hardware, logo o programador precisa conhecer os detalhes da estrutura da máquina.

Introdução

- O Assembly é atrelada à arquitetura de uma certa CPU. Cada família de processador tem seu próprio Assembly (Exemplo: x86, ARM, SPARC, MIPS)
- Não é uma linguagem portável, ao contrário da maioria das linguagens de alto nível (Ex.: Java, Python).

Introdução

- As primeiras linguagens Assembly surgiram na década de 50.
- É a segunda geração de linguagens de programação, depois do código binário.
- A segunda geração visou libertar os programadores de dificuldades de lembrar códigos numéricos e calcular endereços.

Introdução

- Assembly foi muito usada para várias aplicações até os anos 80.
- Porém, foi substituída pelas linguagens de alto nível, por causa da necessidade de aumento da produtividade de software.
- Por exemplo, o módulo de orientação da nave espacial Apolo 11 foi escrito inteiramente em Assembly (https://github.com/chrislgarry/Apollo-11).

Introdução

- Atualmente, ela é usada em sistemas que necessitem de performance crítica (sistemas de tempo real), drivers de sistema e sistemas embarcados de baixo nível.

Assembly

Assembly

- Um código Assembly é texto, logo precisa ser traduzido em algo que a máquina entenda (binário).
- O Assembler faz essa traduzido (vamos usar um programa chamado NASM em Linux 64 bits, https://www.nasm.us/).
- O Assembler traduz Assembly para binário:

```
1 MOV al, 061h
```

- Se transforma em:

```
1 10110000 01100001
```

Assembly

- O processador trabalha com diferentes tamanhos de dados para executar as instruções.
- Byte possui 8 bits.
- Word possui 16 bits.
- Dword possui 32 bits.
- Qword possui 64 bits.

Assembly

- Em Assembly é comum representar os números na forma binária, octal, hexadecimal ou decimal.
- O octal e hexadecimal são utilizados pois representam números binários grandes usando poucos símbolos. Cada dígito octal corresponde a três binários. Cada dígito hexadecimal corresponde a quatro binários.
- O decimal utiliza diferentes representações dependendo do processador (exemplo, BCD, excesso de 3, Johnson, ...).

Assembly

- **Decimal:** Prefixo **0d**; ou, sufixo **d**. Ambos opcionais.
- **Binário:** Prefixo **0b**; ou, sufixo **b**.
- **Octal:** Prefixo **0o** ou **0q**; ou, sufixo **o** ou **q**.
- **Hexadecimal:** Prefixo **0x**; ou, sufixo **h**.

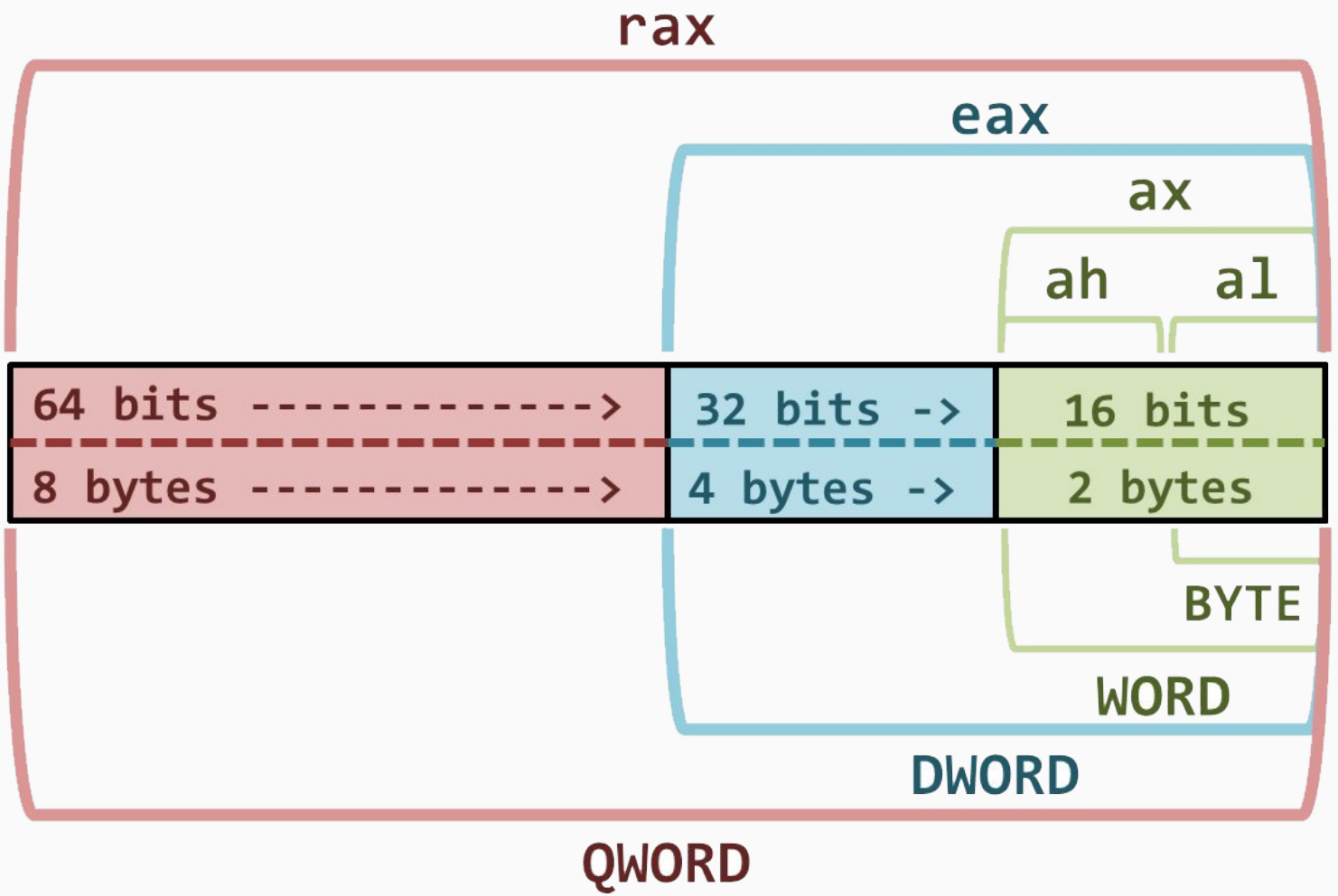
Registradores

Registradores

- São áreas especiais dentro do processador que podem armazenar dados para a realização de operações.
- Seu acesso pelo processador é praticamente instantâneo.
- Nos processadores da Intel existem 8 registradores de uso geral: RAX, RBX, RCX, RDX, RSI, RDI, RSP e RBP.

Registradores

- Nos registradores de uso geral (exceto RSI e RDI) é permitido usar quatro modos de acesso diferentes:



Registradores

- RAX: Acumulador, operações aritméticas e para guardar resultados.
- RBX: Base, armazenar dados em geral.
- RCX: Contador, usado como contador principalmente em loops.
- RDX: Dados, guardar o endereço de uma variável que está na memória.
- Os registradores RSP e RBP devem ser usados preferencialmente para trabalhar com a pilha.
- Os registradores RSI e RDI devem ser usados preferencialmente para trabalhar com agregados de dados.

Registradores

- Os registradores RSP (Stack pointer) e RBP (Base ou frame pointer) são registradores de manipulação da pilha.
- Usados quando procedimentos ou funções são chamadas.
- Por exemplo:
  - RSP guarda a referência para o topo da pilha (começo da função).
  - RBP é usado para andar pela pilha (acessar parâmetros ou variáveis locais).

Registradores

- Os registradores RSI (Source index) e RDI (Destination index) são registradores de deslocamento.
- Usados para movimentação de dados em massa para ou da memória.
- Por exemplo:
  - RSI guarda o endereço fonte do início de um vetor.
  - RDI guarda o endereço destino do inicio.
  - RCX guarda o tamanho do vetor.

Registradores

Flags:

- Registradores especiais que armazenam o estado do processador depois de uma operação.
- Podemos saber se:
  - Se dois valores são iguais;
  - Se um valor é maior que outro;
  - Se um valor é negativo;
  - Se a operação anterior causou um carry out;
  - Se a operação anterior causou um overflow

Registradores

Flags:

- **Carry:** Indicador de carry out.
- **Parity:** Indicador de número par de 1's na última operação.
- **Zero:** Indicador de zero na última operação.
- **Sign:** Indicador de resultado negativo na última operação.
- **Overflow:** Indicador de overflow na última operação.

Instruções

Instruções

Movimentação de dados:

```
1  mov destino, fonte
2
3  mov ax, 200
4  mov ebx, 02Fh
5  mov eax, ecx
6  mov eax, [100]
7  mov eax, [esi]
```

Instruções

Aritméticas:

```
1  add destino, fonte  ;  Adição: destino = destino + fonte
2  sub destino, fonte  ;  Subtração: destino = destino — fonte
3
4  mul fonte ;  Multiplicação:  eax = eax * fonte
5  div  fonte  ;  Divisão:  eax = eax / fonte
6
7  inc  operando ; Incremento: operando = operando + 1
8  dec operando ; Decremento: operando = operando — 1
```

Instruções

Por exemplo, transforme o seguinte código C em Assembly:

```
1  int a, b, c;
2  a = 5;
3  b = 10;
4
5  c = ((a + 5) * b) - 10
```

Instruções

```
1  mov rax, 5
2  mov rbx, 10
3  add rax, 5
4  mul rbx
5  sub rax, 10
6  mov rcx, rax
```

- Considerando que, eax representa a variável A, ebx representa a variável B e ecx representa a variável C.

Instruções

Lógicas:

```
1 not operando ; operando = not operando
2 and destino, fonte
3 or destino, fonte
4 xor destino, fonte
```

Instruções

Comparação:

```
1 cmp operando1, operando2
2 cmp rax, 0x8
```

- Esta instrução altera bits do registrador de flags!

Instruções

Salto incondicional:

```
1 jmp local
2 jmp [100]
3 jmp [rax]
```

- Sempre muda o fluxo de execução.
- O local pode ser um label textual definido no próprio código Assembly.
- Usado para saltar para funções ou pular blocos que não devem ser executados.

Instruções

Salto condicional:

```
1  jz  local
2  jg  [100]
3  jne [rax]
```

- Só muda o fluxo de execução se a condição for verdadeira.

Instruções

- **JE:** Salta se igual.
- **JNE:** Salta se diferente.
- **JG:** Salta se maior.
- **JGE:** Salta se maior ou igual.
- **JB:** Salta se menor.
- **JBE:** Salta se menor ou igual.

Instruções

Por exemplo, transforme o seguinte código C em Assembly (assuma que as variáveis utilizadas foram declaradas e inicializadas apropriadamente):

```
1  b = 1;
2  if (a > 5) {
3     b += 2;
4  }
5  b += 1;
```

Instruções

```
1  mov rax, 1
2  cmp rbx, 5
3  jbe endif
4  add rax, 2
5  endif:
6  add rax, 1
```

- Considerando que, eax representa a variável b e ebx representa a variável A.

Instruções

Por exemplo, transforme o seguinte código C em Assembly (assuma que as variáveis utilizadas foram declaradas e inicializadas apropriadamente):

```
1  b = 2
2  if (a == b) {
3    c = a * b;
4  } else {
5    c = a - b;
6  }
7  c += 1
```

Instruções

```
1  mov rbx, 2
2  cmp rax, rbx
3  jne else
4  mul rbx
5  mov rcx, rax
6  jmp endif
7  else:
8  sub rax, rbx
9  mov rcx, rax
10 endif:
11 add rcx, 1
```

Pilha

Pilha

- A pilha é uma área de dados existente na memória na qual seu programa pode armazenar dados temporariamente.
- Todos os programas fazem uso da pilha em tempo de execução. Nas linguagens de alto nível não é preciso se preocupar com o funcionamento da pilha.
- Em Assembly o programador precisa saber trabalhar com a pilha, pois ela é uma ferramenta importante.
- Importante: A pilha cresce ao contrário, ou seja, do maior valor para o menor.

Pilha

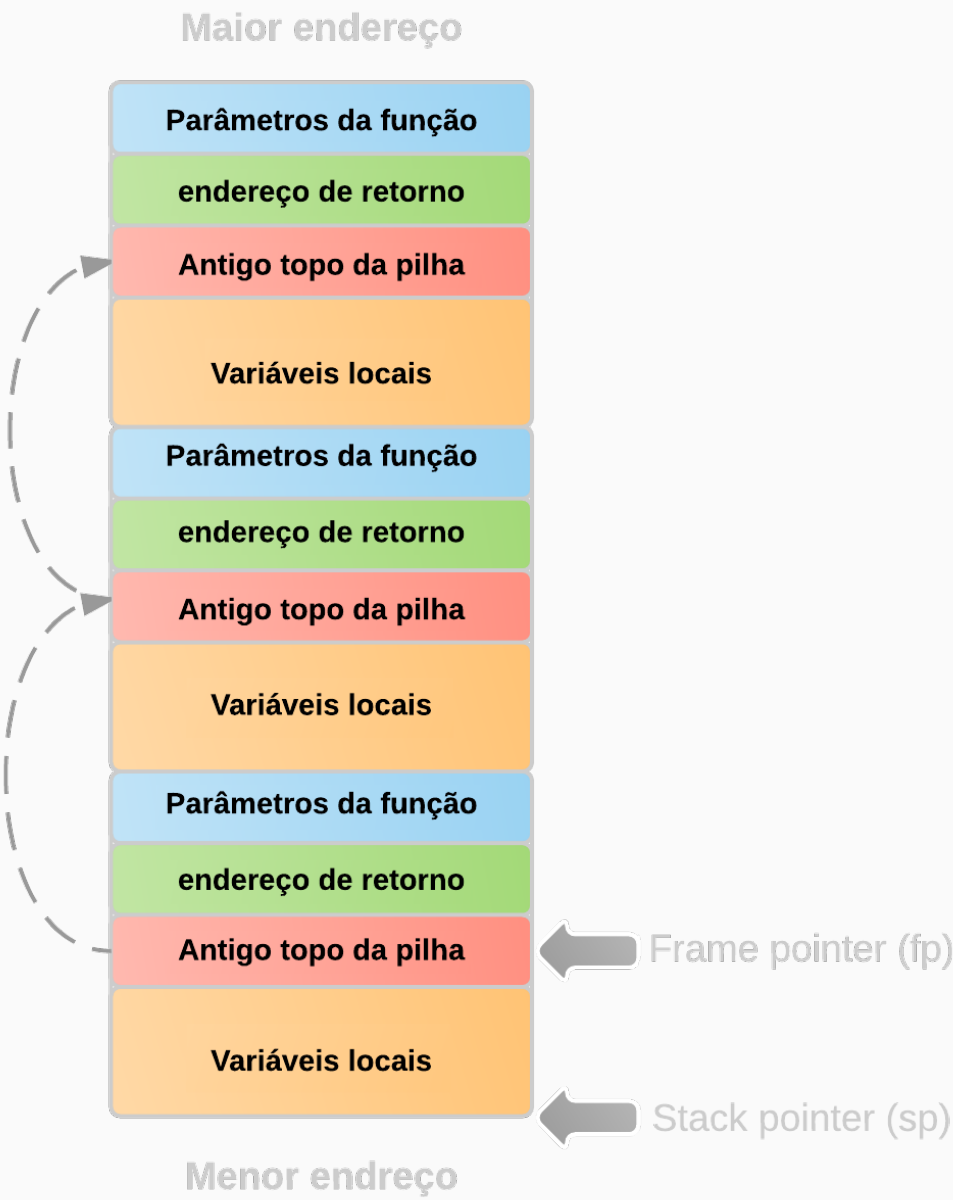
Funcionalidades:

- Variáveis locais.
- Passar parâmetros para funções.
- Preservar valores de registradores para execução de funções.
- Preservar dados da memória.
- Transferir dados sem usar registradores.
- Manter o histórico das chamadas de funções para retornar.

Pilha

- Para manipular a pilha vamos usar dois registradores: RBP (base ou frame pointer) e RSP (stack pointer).
- Ambos os registradores possuem valores diferentes para cada procedimento, por isso as variáveis locais de uma função não são alteradas durante as chamadas (mesmo em funções recursivas).
- O RBP sempre aponta para o início da região da memória do procedimento atual e é usado para acessar os parâmetros e variáveis locais do procedimento.
- O RSP acompanha o crescimento da pilha, ou seja, ele aumenta conforme o procedimento aloca mais variáveis locais. Isso é importante porque se algum procedimento for chamado, os dados precisam ser preservados e o RBP precisa ser alterado para o início da nova região de memória.

Pilha


Maior endereço
Parâmetros da função
endereço de retorno
Antigo topo da pilha
Variáveis locais
Parâmetros da função
endereço de retorno
Antigo topo da pilha
Variáveis locais
Parâmetros da função
endereço de retorno
Antigo topo da pilha
Variáveis locais
Frame pointer (fp)
Stack pointer (sp)
Menor endreço

Pilha

- Empilhar um valor significa subtrair a quantidade de bytes respectiva de RSP e depois mover o valor para a posição de memória que RSP aponta.

```
1 push rax
```

- Desempilhar um valor significa mover o valor para a posição de memória que RSP aponta para outro lugar e adicionar a quantidade de bytes respectiva de RSP.

```
1 pop rbx
```

Pilha

Procedimentos:

- A pilha é bastante usada para a chamada de procedimentos para armazenar parâmetros e variáveis locais.

Pilha

Procedimentos:

- Os parâmetros são empilhados.
- O endereço de retorno é empilhado. A instrução call faz isso automaticamente.
- O valor anterior de RBP é empilhado (RBP é a base da pilha, ele é usado para acessar os parâmetros ou as variáveis locais) e é atualizado para RSP, isso demarca o início da região de memória do procedimento local.
- As variáveis locais são empilhadas.
- A rotina é executada.
- O valor de retorno é preparado.
- Restaura ESP e RBP.
- Retorna pra quem chamou. A instrução ret automaticamente pega o endereço de retorno na pilha.

Pilha

Procedimentos:

```
1   call  nomeProcedimento
2    ...
3   nomeProcedimento:
4      push rbp
5      mov rbp, rsp
6
7       Instrucoes...
8
9      mov rsp, rbp
10     pop rbp
11     ret
```

Pilha

Por exemplo, transforme o código C abaixo em Assembly:

```
1  int somatorio(int n) {
2    int soma = 0;
3    int i;
4    for (i = 0; i < n; i++)
5      soma = soma + i;
6    return soma;
7  }
```

Pilha

Por exemplo, transforme o código C abaixo em Assembly:

```
1  push 32
2  call somatorio
3  add rsp, 8 ; altera rsp para retirar 32 da pilha
4   ...
5  somatorio:
6     push rbp
7     mov rbp, rsp
8
9     sub rsp, 16 ; guarda espaço na pilha para as variaveis locais
10    mov qword [rbp-8], 0 ; varivel soma
11    mov qword [rbp-16], 0 ; variavel i
12    mov rax, 0 ; copia da variavel em i em registrador
13    jmp l2
```

Pilha

Por exemplo, transforme o código C abaixo em Assembly:

```
1   l1 :
2   add [rbp-8], rax ; soma = soma + i
3   inc rax ; incrementa i
4   mov [rbp-16], rax ; salva copia de i na pilha
5
6   l2 :
7   cmp rax, [rbp]+16 ; compara o valor atual de i com o parametro n
8   jl l1 ; se i < n, a repeticao continua
9
10  mov rax, [rbp]-8 ; atribui soma para rax
11
12  mov rsp, rbp
13  pop rbp
14  ret
```

Seções

Seções

- O código Assembly é dividido em seções para separar instruções e diferentes regiões de memória.
- Por enquanto, vamos usar as seções .data e .text

Seções

Seção .data:

- É usada para declarar variáveis constantes, logo elas não mudam no decorrer do programa.

```
1 section .data
2     mensagem db "Hello world!"
3     msglength equ 12
4     fmt1 db "Numero = %d", 10, 0
```

Seções

Seção .text:

- É usada para declarar as instruções do programa.

```
1  section  .text
2  global  main
3  main:
4     mov rax, 10
5     add rax,  5
```

Interrupções

Interrupções

- Interrupções são chamadas ao Sistema Operacional requisitando um serviço de baixo nível.
- O nome interrupção é utilizado porque de fato o processador tem sua atividade atual interrompida quando recebe este tipo de sinal.
- Quando acontece, o processador salva o contexto do programa atual e executa o código relativo àquela interrupção.
- Após a execução do código, o processador retorna ao programa que estava anteriormente.

Interrupções

- Em nível de código Assembly, devemos chamar interrupções para a realização de algumas operações, como: impressão de caracteres, finalização do programa, entre outros.
- Para se chamar uma interrupção é feito o seguinte processo:
  - Coloca-se o número da interrupção no registrador EAX.
  - Coloca-se os argumentos requeridos pela interrupção nos devidos registradores.
  - Chama-se a instrução de interrupção (INT).
  - Caso tenha resultado, geralmente será retornado em EAX

Interrupções

Por exemplo, chamado do sistema para encerrar o programa:

```
1 mov rax, 1
2 mov rbx, 0
3 int 80h
```

- Nos exemplos anteriores, usamos a instrução ret que faz exatamente a mesma coisa.

Exemplo completo

Exemplo completo

```
1  extern printf
2
3  section .data
4     fmt1 db "Register = %d", 10, 0
5
6  section .text
7  global main
8  main:
9
10    push 6
11    call somatorio
12    add rsp, 8
13    jmp end
```

Exemplo completo

```
1  somatorio:
2      push rbp
3      mov rbp, rsp
4      sub rsp, 16
5      mov qword [rbp-8], 0
6      mov qword [rbp-16], 0
7      mov rax, 0
8      jmp l2
9      l1:
10     add [rbp-8], rax
11     inc rax
12     mov [rbp-16], rax
13     l2:
14     cmp rax, [rbp+16]
15     jl l1
16     mov rax, [rbp-8]
17     mov rsp, rbp
18     pop rbp
19     ret
```

Exemplo completo

```
1  end:
2  mov rsi, rax
3  mov rdi, fmt1
4  xor rax, rax
5  call printf
6
7  ret
```

- Compilar e linkar usando: `nasm -felf64 exemplo.asm -o exemplo.o && gcc -o exemplo exemplo.o -no-pie`