DCC510 – PROGRAMAÇÃO EM BAIXO NÍVEL

AULA 3

Carlos Bruno Oliveira Lopes

Engenheiro de Computação

Mestre em Ciência da Computação

Linguagem Assembly

Exibe o valor de rax

- Programa que exibi o valor de `rax` em formato hexadecimal.

```
section .data
codes:
     db '0123456789ABCDEF'

section .text
global _start
_start:
     ; number 1122... in hexadecimal format
     mov rax, 0x1122334455667788

     mov rdi, 1
     mov rdx, 1
     mov rcx, 64
     ; Each 4 bits should be output as one hexadecimal digit
     ; Use shift and bitwise AND to isolate them
     ; the result is the offset in 'codes' array
.loop:
     push rax
     sub  rcx, 4
     ; cl is a register, smallest part of rcx
     ; rax -- eax -- ax -- ah + al
     ; rcx -- ecx -- cx -- ch + cl
     sar rax, cl
     and rax, 0xf
     lea rsi, [codes + rax]
     mov rax, 1
```

Linguagem Assembly

Programa Exibe o valor de rax

- No programa é realizado um deslocamento no valor de `rax` e na sua operação de AND lógico com a máscara `0xF`, para transformar o número todo em um dos seus dígitos hexadecimal;
  - Cada dígito é um número de 0 a 15;
  - Esses números são utilizados como um índice que são somados ao endereço do rótulo `codes` com objetivo de obter o caractere de representação.
    - Ex.: Se `rax = 0x4A`, então deve usar o índice `0x4 = 4`$_{10}$ e `0xA = 10`$_{10}$. Portanto, o primeiro nos dará um caractere ‘4’ cujo código é `0x34`. O segundo resultará no caractere ‘a’ cujo código é `0x61`.

Linguagem Assembly

Rótulos locais

- Rótulo iniciado com . (ponto) são locais
  - `.loop:`
- O último rótulo global usado sem ponto é uma base para todos os rótulos locais subsequentes (até que ocorra o próximo rótulo global)
  - O nome completo do rótulo `.loop` é `_start.loop`
  - Esse nome pode ser usado para endereçá-lo de qualquer ponto do programa, mesmo após a ocorrência de outros rótulos globais

Linguagem Assembly

Endereçamento Relativo

- São endereçamentos mais complexos de memória

Observe o trecho de código

```
lea rsi, [code + rax]
```

- Colchetes indicam endereçamento indireto, e o endereço é escrito dentro deles

  `mov rsi, rax` copia rax para rsi

  `mov rsi, [rax]` copia o conteúdo da memória (8 bytes sequenciais) começando no endereço armazenado em `rax`, para rsi

Linguagem Assembly

Endereçamento Relativo

- `lea` e `mov` são instruções com objetivos similares, mas com uma diferença sutil.
  - `lea` (load effective address) permite calcular o endereço de uma célula de memória e armazená-lo em algum lugar

Linguagem Assembly

Endereçamento Relativo

- Exemplo do que `lea` e `mov` fazem:

```
; rsi <- endereço do rótulo ‘codes’, um número
mov rsi, codes

; rsi <- conteúdo da memória, começando no endereço ‘codes’
; 8 bytes consecutivos são obtidos porque o tamanho de rsi é de 8 bytes
mov rsi, [codes]

; rsi <- endereço de ‘codes’
; neste caso, é equivalente a mov rsi, codes
; em geral, o endereço pode conter vários componentes
lea rsi, [codes]
```

Linguagem Assembly

Endereçamento Relativo

- Exemplo do que `lea` e `mov` fazem:

```
; rsi <- conteúdo da memória, começando em (codes + rax)
mov rsi, [codes + rax]

; rsi <- codes + rax
; equivalente à combinação:
; -- mov rsi, codes
; -- add rsi, rax
; Não é possível fazer isto com um único mov!
lea rsi, [codes + rax]
```

Linguagem Assembly

Ordem de execução

- Todos os comandos são executados de modo consecutivo, exceto quando há instruções jump especiais: condicionais e incondicionais.
  - Instrução jump incondicional

    ```
    jmp addr
    ```

  - Ela substitui o equivalente a

    ```
    mov rip, addr*
    ```

    (* essa ação não é implementada pela arquitetura Intel)

    → RIP (Instruction Pointer Register)

Linguagem Assembly

Ordem de execução

- Jumps condicionais dependem do conteúdo do registrador `rflags`.
  - Instrução condicional `jz address`
    - Realiza jump para o endereço somente se a flag zero estiver ativa
    - Em geral, as instruções `test` ou `cmp` são usada para configurar as flags, em conjunto com uma instrução jump condicional
  - Instrução `cmp`
    - Subtrai o segundo operando do primeiro;
      - Ela não armazena o resultado em lugar nenhum, mas ativa as flags* apropriadas com base no resultado.
        - Se o resultado for zero, então os operandos de destino e origem são iguais.
        - Se o resultado for um número negativo, então o operando de destino é maior que o de origem.
        - Se o resultado for um número positivo, então o operando de destino é menor que o de origem.
  - Instrução `test`
    - Faz o mesmo, mas, utiliza o AND lógico no lugar da subtração.

*Veja as páginas 79-82 do Intel® 64 and IA-32 Architectures Software Developer's Manual

Linguagem Assembly

Ordem de execução

- Outros comandos de jump:
  1. **`ja`** (*jump if above*, isto é, jump se acima) / `jb` (*jump if below*, isto é, jump se abaixo) para um jump depois de uma comparação de números sem sinal com cmp.
  2. **`jg`** (*jump if greater*, isto é, jump se maior) / `jl` (*jump if less*, isto é, jump se menor) para comparação com sinal.
  3. **`jae`** (*jump if above or equal*, isto é, jump se maior ou igual), **`jle`** (*jump if less or equal*, isto é, jump se menor ou igual) e similar.

Linguagem Assembly

Ordem de execução

- Exemplo que incorpora a escrita de 1 em `rbx` se `rax` < 42, e 0 caso contrário.

```
    cmp rax, 42   ;subtrai rax de 42
    jl yes        ;se rax < 42, negativo vai para yes (executa)
    mov rbx, 0       ;caso contrário
    jmp ex        ;vá para ex (excuta as instruções em ex)
yes:
    mov rbx, 1
ex:
```

Linguagem Assembly – Questões

1. Pesquise sobre as instruções de Jump Condicional na arquitetura intel 64, e crie uma tabela com as colunas o nome da instrução (mnemônica), sua condição (flag states) e sua descrição.
2. Pesquise quais são as flags de status e crie uma tabela com coluna sigla, nome e sua descrição.
3. Descreva o trecho de código abaixo esta fazendo.

```
mov rax, -1
mov rdx, 2

cmp rax, rdx
jg location
ja location

cmp rax, rdx
je location
jne location
```

Linguagem Assembly

Chamadas de função

- Rotinas permitem isolar parte da lógica de um programa e usá-la como se fosse uma caixa-preta
  - Prover abstração
  - E permite criar sistema mais complexos por meio do encapsulamento de algoritmos
- Instrução `call <endereço>`
  - Usado para fazer as chamadas de rotinas (funções/procedimentos)
  - Ela faz basicamente:

```
push rip
jmp <endereço>
```

    - O endereço rip é armazenado na pilha sendo chamado de **endereço de retorno**.

Linguagem Assembly

Chamadas de função

- Qualquer função pode aceitar um número ilimitado de argumentos.
  - Os sei primeiros argumentos são passados em:
    - `rdi`, `rsi`, `rdx`, `rcx`, `r8` e `r9`, respectivamente.
  - O restante é passado na pilha na ordem inversa
- Instrução `ret`
  - Indica o final da função.
  - Sua semântica é equivalente a `pop rip`

Linguagem Assembly

Chamadas de função

- O mecanismo de `call` e `ret` só funciona quando o estado da pilha é cuidadosamente administrado
  - Não deve-se chamar ret, a menos que a pilha esteja exatamente no mesmo estado em que se encontrava quando a função iniciou
  - Senão, o processador utilizará o que estiver no topo da pilha como endereço de retorno e o usará como o novo conteúdo do `rip`

Linguagem Assembly

Chamadas de função

- Há dois grupos de registradores:
  - **Registradores callee-saved** (salvos por quem é chamado)
    - Devem ser restaurados pelo procedimento sendo chamado
    - Se houver necessidade de modifica-los, eles deverão ser alterados de volta

      `rbx`, `rbp`, `rsp`, `r12-r15` (total de sete registradores)
  - **Registradores caller-saved** (salvos por quem chama)
    - Devem ser salvos antes de chamar uma função e restaurados depois
    - Não é necessário salvá-los e restaurá-los se seus valores não forem importantes depois
    - Todos os demais registradores são caller-saved

Linguagem Assembly

Chamadas de função

- Há dois grupos de registradores:
  - **Registradores callee-saved** (salvos por quem é chamado)
  - **Registradores caller-saved** (salvos por quem chama)
- Essas duas categorias são uma convenção e devem ser seguidos pelo programador fazendo:
  - Salvando e restaurando os registradores callee-saved;
  - Estar ciente de que os registradores caller-saved podem ser alterados durante a execução da função

Linguagem Assembly

Chamadas de função

- **Bugs comum**: Um erro comum consiste em não salvar registradores caller-saved antes de call e usá-los após o retorno da função. Lembre-se de que:
  1. Se alterar `rbx`, `rbp`, `rsp`, `r12-r15`, você deve alterá-los de volta!
  2. Se for necessário que qualquer outro registrador sobreviva a uma chamada de função, salve-o por conta própria antes da chamada!
- Funções podem **retornar um valor**
  - Ex.: uma função que recebe um número como argumento e retorna o seu quadrado
- Qualquer valor de retorno de uma função é **salvo no acumulador (`rax`)** antes que função termine a sua execução
  - Caso seja necessário retornar dois valores, pode-se usar o `rdx` para o segundo valor

Linguagem Assembly

Chamadas de função

- Resumindo a chamada de uma função:
  - Salve todos os registradores caller-saved que você queira que sobrevivam a uma chamada de função (use `push` para isso).
  - Armazene os argumentos nos registradores relevantes (`rdi`, `rsi` etc.).
  - Chame a função usando `call`.
  - Depois que a função retornar, `rax` armazenará o valor de retorno.
  - Restaure os registradores caller-saved armazenados antes da chamada da função.

Linguagem Assembly

Chamadas de função

- Observações:
  - Algumas chamadas de sistema retornam valores.
  - Não use os registradores rbp e rsp. Eles são implicitamente usados durante a execução com um ponteiro. (Como o rsp é usado como um ponteiro de pilha)

Linguagem Assembly

Chamadas de função

- Programa Print_Call
  - Programa que exibi em formato hexadecimal e pula uma linha usando duas chamadas de função.

```
section .data

newline_char: db 10
codes: db '0123456789abcdef'

section .text
global _start

print_newline:
  mov rax, 1              ; identificador da syscall 'write'
  mov rdi, 1              ; descritor do arquivo stdout
  mov rsi, newline_char ; local de onde os dados são obtidos
  mov rdx, 1              ; quantidade de bytes a ser escrita
  syscall
 ret
print_hex:
  mov rax, rdi

  mov rdi, 1
  mov rdx, 1
  mov rcx, 64         ; até que ponto estamos deslocando rax ?

iterate:
  push rax            ; Salva o valor inicial de rax
```

Linguagem Assembly

Chamadas de função

- Programa Print_Call

```
  push rcx  ; syscall alterará rcx
  syscall   ; rax = 1 (31) - o identificador de write,
            ; rdi = 1 para stdout,
            ; rsi = o endereço de um caractere; veja linha 30

  pop rcx

  pop rax   ; ^ veja a linha 25 ^
  test rcx, rcx  ; rcx = 0 quando todos os digitos forem                    ; mostrados
  jnz iterate

 ret

_start:
  mov rdi, 0x1122334455667788
  call print_hex
  call print_newline

     mov rax, 60
     xor rdi, rdi
     syscall
```

Linguagem Assembly – dados

- Olhe o resultado do trecho de código que imprime dados

```
section .data
demo1: dq 0x112233445566ff
demo2: db 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0xff

section .text

_start:
    mov rdi, [demo1]
    call print_hex
    call print_newline

    mov rdi, [demo2]
    call print_hex
    call print_newline

    mov rax, 60
    xor rdi, rdi
    syscall
```

```
$ ./endianness
112233445566ff
ff7766554433221 1
```

Linguagem Assembly – dados

```
$ ./endianness
1122334455667ff
ff7766554433221 1
```

- Na saída mostrada na segunda linha, podemos observar que a impressão na saída foi inversa!
- Por que?
  - Isso ocorre porque diferentes o processadores têm convenções distintas quanto ao modo como os bytes são armazenados:
    - Números **big endian** com vários bytes são armazenados na memória começando pelos bytes mais significativos.
    - Números **little endian** com vários bytes são armazenados na memória começando pelos bytes menos significativos.

Linguagem Assembly – dados

```
$ ./endianness
1122334455667ff
ff7766554433211
```

- Os bits em cada byte são armazenados de forma direta, porém os bytes são armazenados do menos significativo para o mais significativo
- Isso se aplica somente às operações em memória
- Nos registradores, os bytes são armazenados na ordem natural

Linguagem Assembly – dados

Strings

- Caracteres são codificados usando a tabela ASCII
  - Um código é atribuído a cada caractere
  - Uma string é uma sequência de códigos de caracteres
- O problema que temos, é quanto ao tamanho da string, pois, não temos nada que informe ou determine o seu tamanho, por isso:
  1. Strings começam com seu tamanho explícito

     ```
     db 27, 'Selling England by Pound'
     ```
  2. Um caractere especial indica o término da string. Tradicionalmente, o código zero é usado. Essa strings são conhecidas como **terminadas com nulo**.

     ```
     db 'Selling England by Pound', 0
     ```

Linguagem Assembly

Ponteiros e diferentes tipos de endereçamento

- Ponteiros são endereços de células de memória
  - Eles podem ser armazenados na memória ou em registradores
  - O tamanho do ponteiro é de 8 bytes
  - Os dados ocupam várias células de memória (vários endereços consecutivos)
- Ponteiros não armazenam informações quanto ao tamanho dos dados que eles apontam.

Linguagem Assembly

Ponteiros e diferentes tipos de endereçamento

- Portanto, devemos fornecer o tamanho explicitamente, conforme o exemplo abaixo:

```
section .data
test: dq -1

section .text
  mov byte[test],  1 ;1
  mov word[test],  1 ;2
  mov dword[test], 1 ;4
  mov qword[test], 1 ;8
```

Linguagem Assembly

Ponteiros e diferentes tipos de endereçamento

- Codificação dos operandos nas instruções:
  1. **DE IMEDIATO**: Uma instrução está, ela mesma, contida na memória. Os operandos, de alguma forma, fazem parte dela; essas partes têm endereços próprios.

     Ex.: movendo o número 10 para rax

     ```
     mov rax, 10
     ```

Linguagem Assembly

Ponteiros e diferentes tipos de endereçamento

- Codificação dos operandos nas instruções:
  1. **DE IMEDIATO**: Uma instrução está, ela mesma, contida na memória. Os operandos, de alguma forma, fazem parte dela; essas partes têm endereços próprios.
     Ex.: movendo o número 10 para rax
     ```
     mov rax, 10
     ```
  2. **POR MEIO DE UM REGISTRADOR**:
     Ex.: instrução transfere o valor de `rbx` para `rax`
     ```
     mov rax, rbx
     ```
  3. **POR ENDERECAMENTO DIRETO DA MEMÓRIA**:
     Ex.01: instrução transfere 8 bytes, começando no décimo endereço, para `rax`
     ```
     mov rax, [10]
     ```
     Ex.02: endereço obtido a partir de registrador
     ```
     mov r9, 10
     mov rax, [r9]
     ```

Linguagem Assembly

Ponteiros e diferentes tipos de endereçamento

- Codificação dos operandos nas instruções:
  4. **INDEXADO POR UMA BASE COM ESCALA E DESLOCAMENTO**: O enderenço é calculado com base nos componentes:

$$\text{Endereço} = \text{base} + \text{índice} * \text{escala} + \text{deslocamento}$$

  - *base* é imediata ou está em um registrador;
  - *escala* só pode ser imediata e é igual a 1, 2, 4 ou 8;
  - *índice* é imediato ou está em um registrador;
  - *deslocamento* é sempre imediato

Ex.:

```
mov rax, [rbx + 4* rcx + 9]
mov rax, [4*r9]
mov rdx, [rax + rbx]
lea rax, [rbx + rbx * 4] ; rax = rbx * 5
add r8,  [9 + rbx*8 + 7]
```

Linguagem Assembly

Função para calcular o tamanho de uma string terminada com nulo

- Vamos escrever um programa Assembly que simule o comando de shell `false`:

```
> true
> echo $?
0
> false
> echo $?
1
```

- Como não temos uma rotina para exibir algo na saída-padrão.
- A única maneira de exibir um valor é devolvê-lo como um código de saída é usando a chamada de sistema exit.

Linguagem Assembly

Programa `false.asm`

```
global _start

section .text
_start:
    mov rdi, 0
    mov rax, 60
    syscall
```

Linguagem Assembly

Programa tamanho de string

- Calcula o tamanho de uma string
- Arquivo: **strlen.asm**

```
global _start
section .data
test_string: db "abcdef", 0

section .text
strlen:          ; por convenção, o primeiro e único argumento
...
  mov rax, 60
  syscall
```

Linguagem Assembly

Programa tamanho de string alternativo:

- Consegue identificar algum bug? E quando ele podem ocorrer?

```
global _start

section .data
test_string: db "abcdef", 0

section .text

strlen:
  ;falta um zerar o registrador xor r13, r13

.loop:
  cmp byte [rdi+r13], 0
  je .end
  inc r13
  jmp .loop

.end:
  mov rax, r13
  ret

_start:
  mov rdi, test_string
  call strlen
  mov rdi, rax

  mov rax, 60
  syscall
```

Linguagem Assembly – Questões

1. Qual é a conexão entre rax, eax, ax, ah e al?
2. Como podemos trabalhar com uma pilha de hardware? Descreva as instruções que podem ser usadas.
3. Liste os registradores callee-saved.
4. Liste os registradores caller-saved.
5. Qual é o significado do registrador rip?
6. O que é a flag SF?
7. O que é a flag ZF?

Linguagem Assembly – Questões

8. Descreva os efeitos das instruções a seguir:
   - sar
   - shr
   - xor
   - cmp
   - mov
   - inc, dec
   - add
   - imul, mul
   - sub
   - idiv, div
   - push, pop
   - call, ret

Linguagem Assembly – Questões

9. O que é um código de retorno de um programa?
10. O que é um rótulo (label)? Ele tem um tamanho?

RESUMO

- Rótulos Locais
- Endereçamento relativo
- Ordem de execução
- Chamadas de função
- Exemplos de código em assembly