“Computadores são como os deuses do velho testamento; cheios de regras e sem piedade alguma.” (Joseph Campbell)

Comparações e branches em x86-64

Paulo Ricardo Lisboa de Almeida

UDESC
UNIVERSIDADE DO ESTADO DE SANTA CATARINA

O registrador rflags

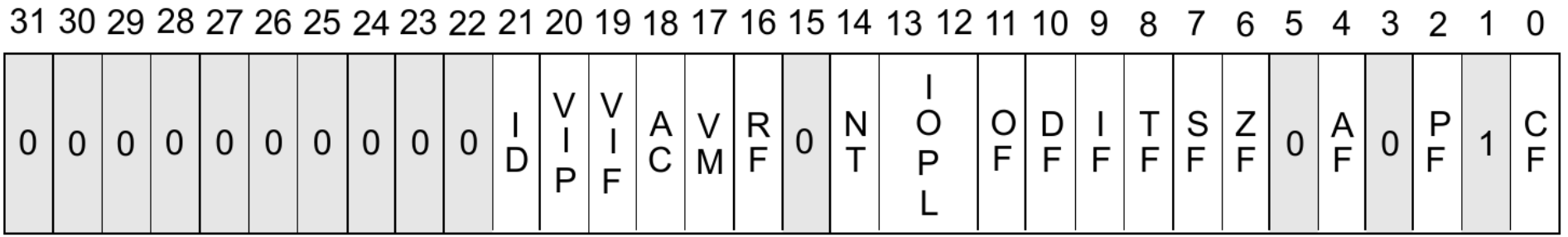
- Os resultados de comparações, *overflows*, *carries*, … são armazenados em um registrador especial, chamado **rflags**
  - **rflags** é um registrador de 64 bits
  - Extensão do registrador **eflags**, que possuía 32 bits
    - Os 32 bits extras de rflags não são usados, e são marcados apenas como “reservados” nos manuais da Intel
    - Sendo assim, você pode ler os manuais do **eflags** sem problemas

O registrador rflags

Itens com valor fixo são reservados

O ***rflags*** tem 32 bits extras quando comparado ao eflags, mas nenhum é utilizado (reservados)

Bit	Nome	Descrição
0	CF	Carry Flag
2	PF	Parity Flag
4	AF	Auxiliary Carry Flag
6	ZF	Zero Flag
7	SF	Sign Flag
8	TF	Trap Flag
9	IF	Interrupt Enable Flag
10	DF	Direction Flag
11	OF	Overflow Flag
...		



eflags - Intel® 64 and IA-32 Architectures Software Developer's Manual.

UDESC

Comparações com cmp

- Podemos utilizar a instrução **cmp** para realizar comparações
  cmp[bwlq] operando1, operando2
  - Onde
    - Operando1 e 2 podem ser um registrador, um endereço de memória ou um imediato
      - Somente um dos dois pode ser um endereço de memória
      - Somente um dos dois pode ser um imediato
    - **Os operandos não são alterados**
    - O resultado é armazenado em rflags
  - A comparação é feita **subtraindo-se os valores**
    - Manual Intel
  - Flags afetadas
    - CF, OF, SF, ZF, AF e PF

Comparações com test

- A instrução **test** funciona da seguinte forma
  test[bwlq] operando1, operando2
  - Onde
    - Operando1 e 2 podem ser um registrador, um endereço de memória ou um imediato
      - Somente um dos dois pode ser um endereço de memória
      - Somente um dos dois pode ser um imediato
    - **Os operandos não são alterados**
    - O resultado é armazenado em rflags
  - A comparação é feita realizando-se um **AND bit a bit entre os operandos**
    - Manual Intel
  - Flags afetadas
    - SF, ZF, and PF

Realizando Jumps Condicionais

- Branches condicionais são realizados através de instruções jcc
  - Jump if condition is met
  - Verifica o estado do registrador rflags para decidir se o salto será realizado ou não
- Formato

  jcc LABEL
- A seguir, apenas algumas instruções de jump condicional
  - Para uma lista exaustiva, veja *jcc* no manual Intel

Jumps Condicionais

Mnemônico	Instrução	Condição
ja	jump if above	(CF = 0) E (ZF = 0)
jae	jump if above or equal	CF = 0
jb	jump if below	CF = 1
jbe	jump if below or equal	(CF = 1) OU (ZF = 1)
jc	jump if carry	CF = 1
je	jump if equal	ZF = 1
jg	jump if greater	(ZF = 0) E (SF = OF)
jge	jump if greater or equal	SF = OF
jl	jump if less	SF != OF
jle	jump if less or equal	(ZF = 1) OU (SF != OF)
jne	jump if not equal	ZF = 0
jz	jump if zero	ZF = 1
jnz	jump if not zero	ZF = 0
…	…	...

UDESC

Alguns detalhes

- *jump if above* e *jump if greater* são similares
  - Above (acima) se refere a uma comparação sem sinal
  - Greater (maior que) considera que os valores comparados possuem sinal
    - Complemento de dois
- O mesmo é válido para bellow (abaixo) e less (menor)
  - Bellow para comparação sem sinal, e less para comparação com sinal

Alguns detalhes

- Note que je e jz **têm o mesmo comportamento**
  - O mesmo é válido para jne e jnz
- Nesse caso, utilize a instrução que faz mais sentido
  - Ambas levam ao mesmo resultado
  - Mas uma dá a entender que você está checando uma igualdade
  - A outra dá a entender que você precisa verificar se um resultado é 0

Mnemônico	**Instrução**	**Condição**
…	…	…
je	jump if equal	ZF = 1
…	…	…
jz	jump if zero	ZF = 1
…	…	…

Jumps condicionais

- Internamente, os jumps condicionais do x86-64 funcionam como os branches do MIPS
  - O salto é relativo ao PC (*Instruction Pointer* – *IP* no x86-64)

Questão de Ordem

- **A ordem dos operandos é importante!**
- Exemplo

```
cmpq %rax, %rbx    #compare rax com rbx
jae LABEL          #salte para LABEL se rax está acima de rbx
movb $0x123, %al   #mova o valor para o byte mais baixo de a (al)
…
```

- Então

```
cmpq %rax, %rbx
```

  - **Não é o mesmo que**

```
cmpq %rbx, %rax
```

- **Na dúvida, teste com o GDB ou leia os manuais**

Saltos incondicionais

- A instrução jmp (jump) é utilizada para saltos **incondicionais**
  - jmp LABEL
    - Salte para o LABEL
    - Exemplo: jmp SAIDA
    - O mesmo que um j SAIDA em MIPS
  - jmp *REGISTRADOR
    - Salte para o endereço armazenado no REGISTRADOR
      - O asterisco denota “O endereço em”
      - Outras instruções utilizam ( ) para denotar isso, mas o jmp é diferente
    - Exemplo: jmp *%rax
    - O mesmo que jr $ra em MIPS
  - jmp *MEMORIA
    - Salte para o endereço armazenado na posição de memória MEMORIA
    - Exemplo: jmp *ENDERECO_DO_PONTEIRO

Exemplo

- Vamos criar o seguinte programa em assembly do x86-64

```
#include <unistd.h>

int main(void){
	char *strOla = "Ola\n";
	while (*strOla != '\0'){
		write(STDOUT_FILENO, strOla, 1);
		strOla++;
	}
	return 0;
}
```

loop

- Crie um programa chamado loop.s

```
#Constantes
        .equ STDOUT,1
#Posições na STACK
        .equ ptrOla,-8  #local do ponteiro para STR_OLA na pilha
        .equ localSize,-16

        .section .rodata
STR_OLA:
        .string "Ola\n"

        .text
        .globl main
        .type main,@function
main:
        pushq %rbp
        movq %rsp,%rbp
        addq $localSize,%rsp      #final do prólogo
```

Copia o endereço inicial de STR_OLA para a pilha

```
movq $STR_OLA,%rsi    #carregar o end. base de STR_OLA para rsi
movq %rsi, ptrOla(%rbp) #armazenar o endereço de STR_OLA na pilha
```

UDESC

loop

Carrega o ponteiro que foi salvo na pilha para rsi

```
while:
    movq ptrOla(%rbp),%rsi #carregar a posição atual para rsi
    cmpb $0,(%rsi)         #o valor na posição de memória é nulo?
    je fimLoop             #salta se igual. Checa flag Z em rflags

    movl $1,%edx           #imprimir 1 caractere
    movq $STDOUT, %rdi     #id. arquivo STDOUT em rdi
    call write             #chama função do C

    incl ptrOla(%rbp)      #somar um no endereço armazenado na pilha
    jmp while              #salto incondicional
fimLoop:                   #label de saída
    movl $0,%eax           #return 0
    movq %rbp, %rsp
    popq %rbp              #fim do epílogo
    ret                    #retornar ao chamador
```

Compara o byte (cmp**b**) que está no endereço apontado por %rsi com zero. Os parêntesis ( ) indicam “o endereço apontado por”

Poderíamos usar add para somar 1, mas incl já faz isso. Vai carregar o ponteiro da pilha, somar 1, e depois salvar na pilha novamente, tudo em uma instrução.

UDESC

Endereçamento

- Alguns modos de endereçamento vistos até agora
  - Instrução %registrador
    - Acessar o valor armazenado no registrador
  - Instrução $imediato
    - Utilizar o imediato como valor
  - Instrução (%registrador)
    - Acessar o endereço de memória armazenado no registrador
  - Instrução offset(%registrador)
    - Adicionar o offset no endereço armazenado no registrador e acessar
  - j *%registrador
    - Utilizado em jumps para saltar para o endereço armazenado no registrador
  - j *Memória
    - Utilizado em jumps para saltar para o endereço armazenado em Memória

GDB - Comandos

- O comando **x** imprime o conteúdo de uma posição de memória
  - Veja detalhes em visualgdb.com/gdbreference/commands/x
  - Exemplos:
    - **x/dg -8+($rbp)**
      - Exibir em a palavra de 64 bits (g – giant word) no formato decimal armazenada no endereço apontado pelo registrador rbp – 8
    - **x/cb $esi**
      - Exibir o byte (b) no formato de char armazenado no endereço apontado pelo registrador esi
- Verificar o conteúdo de *eflags*
  - **i r eflags**
  - **p/t $eflags**
    - Para exibir em binário
    - Onde **p** é o comando print
      - Veja detalhes e formatos em visualgdb.com/gdbreference/commands/print

Exercícios

1. Rode o programa anterior passo a passo e analise com o GDB.
   - Insira um breakpoint na instrução de comparação
   - Verifique a cada passo o char armazenado na memória sendo comparado, e o resultado da comparação aramzenado em eflags

Exercícios

2.Faça as seguintes alterações no programa

- Leia uma a string do teclado
  - Considere que a string tem no máximo 15 caracteres
  - Utilize read via syscall
    - read lê até o limite de especificado, ou até o usuário entrar com ‘\n’ (enter)
    - A quantidade de caracteres lido pelo read é retornado em rax
      - Utilize esse valor como critério de parada
- Converta todos os caracteres entre [a-z] para maiúsculo. Os demais caracteres devem permanecer inalterados.
- Escreva os caracteres (já convertidos) na tela utilizando uma **única** syscall para write
- Monte com o GAS (*as*) e faça a linkedição com o GNU Linker (*ld*)
  - Não utilize o *GCC*
  - Não se esqueça de terminar seu programa via syscall
- Analise o seu programa rodando com o GDB
- Compare sua resposta com a disponibilizada no Moodle
- **Dica**
  - Sempre salve os registradores relevantes na pilha antes de uma chamada de função
    - Não temos como garantir se os valores serão restaurados
  - Uma syscall no Linux preserva o conteúdo dos registradores, **exceto rax**
- **Submeta sua solução no Moodle**

Exercícios

3. Analise a forma que foi feita a verificação se o caractere está entre [a-z] na solução do exercício 2 disponibilizada no Moodle (a sua solução provavelmente é parecida). Essa é uma construção em "curto circuito" para condicionais com "*AND*"
   - Exemplo: if(a>b **&&** a >50 **&&** a != x)
   - Quando a primeira condição não é satisfeita, o resultado é automaticamente falso, e as demais condições não são avaliadas.
     - Veja que é isso o que acontece no código disponibilizado.
   - Devido a essa construção, a linguagem C por exemplo garante que condições não são avaliadas desnecessariamente
   - Um raciocínio similar é aplicado para "ou's"
   - **Veja a seção 10.2.1 de Plantz(2011)**

Referências

- Bob Plantz. **Introduction to Computer Organization: A Guide to X86-64 Assembly Language and GNU/Linux.** 2011.
- **Intel® 64 and IA-32 Architectures Software Developer's Manual.** Intel, 2019.
- D. Patterson; J. Henessy. **Organização e Projeto de Computadores**: **A Interface Hardware / Software.** 4a Edição. Elsevier Brasil, 2014.
- STALLINGS, W. **Arquitetura e Organização de Computadores.** 9 ed. Prentice Hall. São Paulo, 2012.
- M. Matz, J. Hubička, A. Jaeger, M. Mitchell. **System V Application Binary Interface AMD64 Architecture Processor Supplement.** 2014.