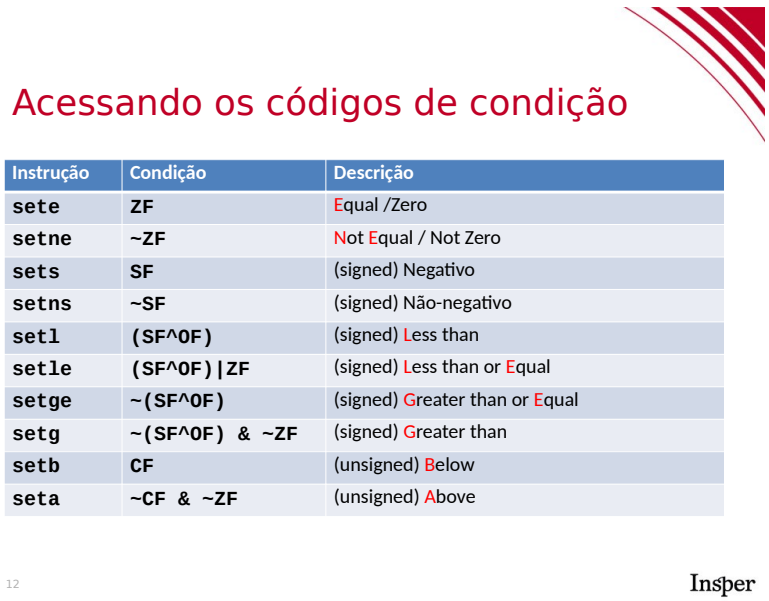
Insper

Engenharia reversa 2: Condicionais

Igor Montagner

Parte 1 - Expressões booleanas

Vimos na expositiva que toda operação aritmética preenche as flags `CF`, `ZF`, `SF` e `OF` e que podemos usar estas flags para montar expressões booleanas com as instruções `set*`. A tabela abaixo mostra as instruções responsáveis cada tipo de expressão booleana.



Acessando os códigos de condição

Instrução	Condição	Descrição
sete	ZF	Equal /Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	(signed) Negativo
setns	~SF	(signed) Não-negativo
setl	(SF^OF)	(signed) Less than
setle	(SF^OF)|ZF	(signed) Less than or Equal
setge	~(SF^OF)	(signed) Greater than or Equal
setg	~(SF^OF) & ~ZF	(signed) Greater than
setb	CF	(unsigned) Below
seta	~CF & ~ZF	(unsigned) Above

12 Insper

Figure 1: Tabela de set para operações booleanas

Também vimos que podemos preencher estas flags usando as instruções `cmp` e `test`, que executam operações aritméticas (subtração e E bit a bit) mas não guardam o resultado.

Vamos analisar o código assembly da seguinte função:

```
int igual(int a, int b) {
    return a == b;
}
```

```
0000000000000000 <igual>:
0:   39 f7                   cmp    %esi,%edi
2:   0f 94 c0                sete   %al
5:   0f b6 c0                movzbl %al,%eax
8:   c3                      retq
```

A comparação `a == b` é feita primeiro executando `cmp` entre os argumentos `%edi` e `%esi` e depois usando `sete` (*set equal*) para atribuir 1 `%al` se `%edi == %esi` e 0 caso contrário.

Por fim, temos a instrução `movzbl`, que faz o cast de `char` (`%al`) para `int` (`%eax`). Lembre-se que as instruções `set*` só modificam os primeiros 8 bits de `%eax`. O restante continua com o valor antigo. Usamos `movzbl` para extender o número em `%al` para ocupar todo `%eax`.

As instruções de conversão de tipos são bastante frequentes em Assembly, principalmente para expandir valores colocados em registradores menores para registradores maiores. Elas seguem a seguinte regra:

`MOVtsd`

- `t` pode ser `z` para tipos unsigned (completando com **z**eros) e `s` para tipos signed (completando com o bit de **s**inal).
- `s` é o tamanho do registrador fonte seguindo a notação `b` para 1 byte, `w` para 2 bytes, `l` para 4 bytes e `q` para 8 bytes.
- `d` é o tamanho do registrador destino, seguinto a mesma notação acima.

Por exemplo, a instrução `MOVZWQ` converte um `unsigned short` para um `unsigned long`. Conversões de 4 para 8 bytes muitas vezes são feitas com a instrução `cltq`, que extende (com sinal) `%eax` para `%rax`. Uma boa referência é este site da Oracle.

Vamos agora praticar. Nos 3 exercícios abaixo temos funções que avaliam uma (ou mais) expressões booleanas entre seus argumentos e retornam o resultado.

Exercício 1: Reconstrua a função `ex1` a partir do código assembly abaixo.

```
0000000000000000 <ex1>:
   0:   83 ff 0a                cmp    $0xa,%edi
   3:   0f 9f c0                setg   %al
   6:   0f b6 c0                movzbl %al,%eax
   9:   c3                      retq
```

1. Qual é o tamanho do argumento de `ex1`? Ele é `signed` ou `unsigned`?

2. Coloque sua tradução abaixo. Valide sua solução com o professor ou com algum colega que já validou sua solução.

Exercício 2: Reconstrua a função `ex2` a partir do código assembly abaixo

```
0000000000000000 <ex2>:
   0:   48 39 f7                cmp    %rsi,%rdi
   3:   0f 96 c0                setbe  %al
   6:   0f b6 c0                movzbl %al,%eax
   9:   c3                      retq
```

1. Qual é o tamanho dos argumentos de `ex2`? Ele é `signed` ou `unsigned`?

2. Coloque sua tradução abaixo. Valide sua solução com o professor ou com algum colega que já validou sua solução.

Exercício 3: Reconstrua a função `ex3` a partir do código assembly abaixo.

```
0000000000000000 <ex3>:
   0:   48 39 f7                cmp    %rsi,%rdi
   3:   0f 9f c0                setg   %al
   6:   48 85 f6                test   %rsi,%rsi
   9:   0f 9f c2                setg   %dl
   c:   21 d0                   and    %edx,%eax
   e:   0f b6 c0                movzbl %al,%eax
  11:   c3                      retq
```

1. Qual é o tamanho dos argumentos de `ex3`? Ele é `signed` ou `unsigned`?

2. Coloque sua tradução abaixo. Valide sua solução com o professor ou com algum colega.

Parte 2 - Condicionais

Vimos na segunda parte expositiva que Assembly possui apenas instruções de pulos condicionais (`j*` onde `*` representa uma comparação usando as mesmas abreviações de `set*`) e não condicionais (`jmp`). Vimos também que a combinação destas instruções com `cmp` e `test` é equivalente à dupla de comandos

```
if (cond-booleana) {
    goto label;
}
```

A tabela abaixo

Figure 2: Tabela de set para saltos condicionais

Vamos agora fazer um exemplo guiado. Analisaremos o seguinte código:

```
0000000000000000 <eh_par>:
0:   40 f6 c7 01             test   $0x1,%dil
4:   74 06                   je     c <eh_par+0xc>
6:   b8 00 00 00 00          mov    $0x0,%eax
b:   c3                      retq
c:   b8 01 00 00 00          mov    $0x1,%eax
11:  c3                      retq
```

Pares de instruções `test-j*` ou `cmp-j*` são comumente usadas para representar a construção `if-goto`.

- O nome da função dá uma dica de seu valor de retorno. Você consegue entender o porquê `test $1, %dil` faz isto?

Vamos agora traduzir a função `eh_par` para **gotoC**. As linhas `0-4` são transformadas em um par `if-goto`. O restante são instruções que já conhecemos.

```
int eh_par(long a) {
    if (a & 1 == 0) goto if1;

    return 0;

    if1:
    return 1;
}
```

Tiramos então o `goto` e levando em conta sua resposta no item anterior, ficamos com o seguinte código. Note que precisamos negar a comparação feita no código anterior!

```
int eh_par(long a) {
    if (a % 2 != 0) {
        return 0;
    }
    return 1;
}
```

Podemos observar duas coisas no código assembly gerado:

1. O código que estava dentro do `if` foi colocado após o código que estava fora do if! O compilador pode mudar a ordem dos nossos blocos de código se for conveniente (para ele, não para nós).
2. A construção `test-j*` e `cmp-j*` pode ser mapeada diretamente para `if-goto`. Porém, reconstruir um código legível requer, muitas vezes, mudar código de lugar.

Vamos agora praticar com alguns exercícios simples:

Exercício 4: veja o código abaixo

```
0000000000000000 <fun4>:
0:   48 85 ff                test   %rdi,%rdi
3:   7e 0a                   jle    f <fun4+0xf>
5:   b8 02 00 00 00          mov    $0x2,%eax
a:   48 0f af c6             imul   %rsi,%rax
e:   c3                      retq
f:   b8 01 00 00 00          mov    $0x1,%eax
14:  eb f4                   jmp    a <fun4+0xa>
```

1. Qual expressão booleana é testada?

2. Faça a tradução desta função para **gotoC**.

3. Transforme o código acima em *C* legível.

Exercício 5: Veja o código da função abaixo.

```
0000000000000000 <ex5>:
0:   48 85 ff                test   %rdi,%rdi
3:   0f 9f c2                setg   %dl
6:   48 85 f6                test   %rsi,%rsi
9:   0f 9e c0                setle  %al
c:   84 c2                   test   %al,%dl
e:   75 05                   jne    15 <ex5+0x15>
10:  48 8d 46 fe             lea    -0x2(%rsi),%rax
14:  c3                      retq
15:  48 8d 47 05             lea    0x5(%rdi),%rax
19:  c3                      retq
```

1. Qual são as expressões booleanas testadas? (Dica: são 3, assim como no exercício 3).

2. Faça uma tradução para **gotoC**.

3. Transforme seu código acima para *C* legível.

Veremos agora um exemplo `if/else`:

```
int exemplo2(long a, long b) {
    long c;
    if (a >= 5 && b <= 0) {
        c = a + b;
    } else {
        c = a - b;
    }
    return c;
}
```

Seu assembly correspondente, quando compilado com `gcc -Og -c` é

```
0000000000000000 <exemplo2>:
0:   48 83 ff 04             cmp    $0x4,%rdi
4:   0f 9f c2                setg   %dl
7:   48 85 f6                test   %rsi,%rsi
a:   0f 9e c0                setle  %al
d:   84 c2                   test   %al,%dl
f:   75 07                   jne    18 <exemplo2+0x18>
11:  48 89 f8                mov    %rdi,%rax
14:  48 29 f0                sub    %rsi,%rax
17:  c3                      retq
18:  48 8d 04 37             lea    (%rdi,%rsi,1),%rax
1c:  c3                      retq
```

Primeiramente, notamos que a função recebe dois argumentos (pois só utiliza `%rdi` e `%rsi`) e que ambos são tratados como `long`. Vamos então às expressões booleanas. Existem três expressões boolenas:

1. `cmp`-`setg`(linhas `0-4`) compara `%rdi` com `4` e seta `%dl=1` se `%rdi>4` (**g**reater)
2. `test-setle` (linhas `7-a`) compara `%rsi` com `0` e seta `%al=1` se `%rsi<=0` (**l**ess or **e**qual).
3. `test`(linha `d`) entre `%dl` e `%al`. O resultado não é armazenado.

Logo abaixo do último `test` temos um `jne` (linha `f`), acrônimo para **j**ump if **n**ot **e**qual. Ou seja, fazemos o jump se `%dl && %al` for verdadeiro.

Logo em seguida temos instruções aritméticas, que já estudamos nos últimos handouts. Assim como vimos nos slides, vamos converter este código para **gotoC** primeiro.

Assim como fizemos nos exercícios de 1 a 3, criaremos uma variável para as expressões booleanas 1 e 2 e substituiremos as instruções `test-jne` (linhas `d-f`) por um par `if-goto`. Veja abaixo:

```
int exemplo2(long a, long b) {
    long retval;
    int expr1 = a > 4;
    int expr2 = b <= 0;
    if (expr1 && expr2) goto if1;

    retval = a;
    retval -= b;
    return retval;

    if1:
    retval = a + b;
    return retval;
}
```

Podemos então melhorar tornar este código mais legível, resultando no seguinte:

```
int exemplo2(long a, long b) {
    if (a > 4 && b <= 0) {
        return a+b;
    } else {
        return a-b;
    }
}
```

Duas coisas importantes podem ser vistas neste código

1. As comparações não são exatamente iguais (`a>4` e `a>=5`), mas são equivalentes.
2. O compilador pode trocar a ordem do `if/else` e colocar o `else` primeiro no Assembly gerado. Isto não altera o resultado da função, mas pode ser confuso de início.

Exercício 6: O exercício abaixo usa `if-else`.

```
0000000000000000 <ex6>:
0:   48 39 f7                cmp    %rsi,%rdi
3:   7e 03                   jle    8 <ex6+0x8>
5:   48 89 fe                mov    %rdi,%rsi
8:   48 85 ff                test   %rdi,%rdi
b:   7e 03                   jle    10 <ex6+0x10>
d:   48 f7 de                neg    %rsi
10:  89 f0                   mov    %esi,%eax
12:  c3                      retq
```

1. Traduza o código acima para **gotoC**.

2. Faça uma versão legível do código acima.