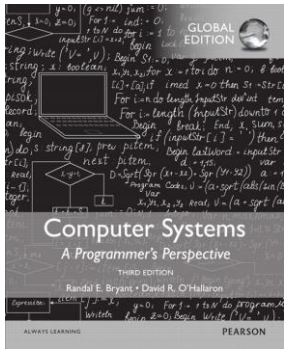


X86-64
Intel architecture (AT&T syntax)

Bib: ComputerSystems: A Programmer's Perspective
x86-64 Machine-Level Programming (adenda ao cap. 3 do livro anterior)

Programação em Sistemas Computacionais

João Pedro Patriarca (jpatri@cc.isel.ipl.pt, joao.patriarca@isel.pt), Gabinete F.0.23 do edifício F

ISEL, ADEETC, LEIC

Agenda

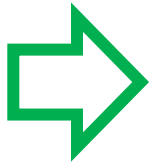
- Estruturas programáticas em *assembly*:
  - Instruções de decisão: *if*, *if-then-else* , *?:* (operador ternário)
  - Instruções de ciclo: *do-while*, *while*, *for*
  - Instrução de múltipla decisão: *switch-case*

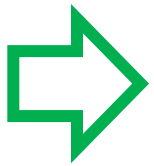
Instruções de decisão

IF e IF-THEN-ELSE

```
int if_then_test(int v) {
    if (v < 0) v = -v;
    return v;
}


int if_then_else_test(int v) {
    if (v >= 10) v += 1;
    else v = -v;
    return v;
}
```





```
if_then_test:
    mov     %edi, %eax
    test    %eax, %eax
    jns     .L0_if_test1
    neg     %eax
.L0_if_test1:
    ret

if_then_else_test:
    mov     %edi, %eax
    cmp     $10, %eax
    jl      .L0_if_test2
    add     $1, %eax
    jmp     .L1_if_test2
.L0_if_test2:
    neg     %eax
.L1_if_test2:
    ret
```

- Por norma, o salto condicional é implementado com a relação inversa

Instruções de decisão

Operador ternário

```
int ternary_op_test(int v) {
    return v >= -1 && v <= 1 ?
               0 :
               v*4;
}
```

```
ternary_op_test:
    lea     (,%edi,4), %eax
    cmp     $-1, %edi
    jl      .L0_ternary_op_test
    cmp     $1, %edi
    jg      .L0_ternary_op_test
    xor     %eax, %eax
.L0_ternary_op_test:
    ret
```

- Transforma a instrução de dois casos para um caso, produzindo inicialmente o resultado da cláusula *else* e alterando-o apenas se o resultado da comparação for verdadeiro

Instruções de ciclo
WHILE

```
int while_test(uint v,
               uint idx,
               uint len) {
    ...
    v >>= idx;
    int cnt = 0;
    while (len > 0) {
        cnt += v & 1;
        len -= 1;
        v >>= 1;
    }
    return cnt;
}
```

```
while_test:
    ...
    mov     %sil, %cl
    shr     %cl, %edi
    xor     %eax, %eax
    jmp     .L1_while_test
.L2_while_test:
    mov     %edi, %ecx
    and     $1, %ecx
    add     %ecx, %eax
    dec     %dx
    shr     $1, %edi
.L1_while_test:
    test    %dx, %dx
    jnz     .L2_while_test
.L0_while_test:
    ret
```

%sil = idx = %cl
%edi = v

%edi = v = %ecx

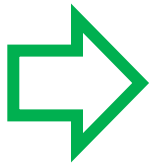
%dx = len

- Para reduzir o número de saltos por iteração, a expressão de controlo é movida para o final do *while*

Instruções de ciclo

DO-WHILE

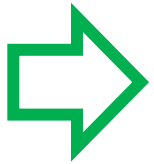
```
int do_while_test_c(ulong v) {
    int cnt = 0, cnt_tmp;
    do {
        cnt_tmp = 0;
        while (v & 1) {
            cnt_tmp += 1;
            v >>= 1;
        }
        if (cnt_tmp > cnt)
            cnt = cnt_tmp;
        while (v != 0 &&
               (v & 1) == 0)
            v >>= 1;
    } while (v != 0);
    return cnt;
}
```



```
do_while_test:
    xor     %eax, %eax # cnt = 0  %eax = return
.L5_do_while_test:
    xor     %edx, %edx # cnt_tmp = 0
    jmp     .L0_do_while_test
.L1_do_while_test:
    inc     %edx
    shr     $1, %rdi
.L0_do_while_test:
    test    $1, %rdi    %rdi = v
    jnz     .L1_do_while_test
    cmp     %edx, %eax  %edx = cnt_tmp %eax=cnt
    jge     .L2_do_while_test
    mov     %edx, %eax
    jmp     .L2_do_while_test
.L3_do_while_test:
    shr     $1, %rdi
.L2_do_while_test:
    test    %rdi, %rdi
    jz     .L4_do_while_test
    test    $1, %rdi
    jz      .L3_do_while_test
.L4_do_while_test:
    test    %rdi, %rdi
    jnz     .L5_do_while_test
    ret
```

Instruções de ciclo
FOR

```
int for_test(int v[],
             int vsize) {
    int sum = 0;
    for (int i = 0;
             i < vsize;
             i++)
        sum += v[i];
    return sum;
}
```

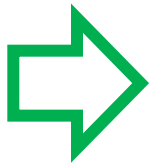


```
for_test:
    xor     %eax, %eax # sum = 0
    xor     %rdx, %rdx # i = 0
    jmp     .L0_for_test
.L1_for_test:
    add     (%rdi, %rdx, 4), %eax
    inc     %rdx        %rdi=v[] %rdx=i %eax=sum
.L0_for_test:
    cmp     %esi, %edx  %esi=vsize %edx=i
    jl      .L1_for_test
    ret
```

Instrução de múltipla decisão

SWITCH-CASE com tabela de cases

```c
int switch_case_test_c(
    Operation op,
    int a,
    int b)
{
    int r;
    switch(op) {
        case add: r = a+b; break;
        case sub: r = a-b; break;
        case mul: r = a*b; break;
        case div: r = a/b; break;
        case mod: r = a%b; break;
        default: r = 0;
    }
    return r;
}
```



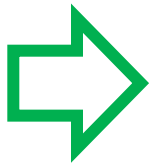
```
switch_case_test_v2:
    cmp     $4, %edi  %edi=op
    jg      .switch_end
    mov     %esi, %eax  %esi=a %eax=ret
    movabs  $table_cases, %rcx  %rcx tem 64bits
    jmp     *(%rcx, %rdi, 8)
.case_add:  ...
    jmp     .switch_end
.case_sub:  ...
    jmp     .switch_end
.case_mul:  ...
    jmp     .switch_end
.case_div:  ...
    jmp     .switch_end
.case_mod:  ...
.switch_end:
    ret

    .section .rodata
table_cases:
    .quad    .case_add, .case_sub,
             .case_mul, .case_div,
             .case_mod
```

Instrução de múltipla decisão

SWITCH-CASE com tabela de JMPs

```c
int switch_case_test_c(
    Operation op,
    int a,
    int b)
{
    int r;
    switch(op) {
        case add: r = a+b; break;
        case sub: r = a-b; break;
        case mul: r = a*b; break;
        case div: r = a/b; break;
        case mod: r = a%b; break;
        default: r = 0;
    }
    return r;
}
```



```asm
switch_case_test:
    cmp     $4, %edi
    jg      .switch_end
    movabs  $.table_jmp, %rcx
    lea     (%rcx, %rdi, 2), %rcx
    jmp     *%rcx
.table_jmp:
    jmp     .case_add
    jmp     .case_sub
    jmp     .case_mul
    jmp     .case_div
    jmp     .case_mod
.case_add: ...
    jmp     .switch_end
.case_sub: ...
    jmp     .switch_end
.case_mul: ...
    jmp     .switch_end
.case_div: ...
    jmp     .switch_end
.case_mod: ...
.switch_end:
    ret
```