

Análise do Instruction Set Architecture (3)



Estrutura do tema ISA do IA-32

1. Desenvolvimento de programas no IA-32 em Linux
2. Acesso a operandos e operações
- 3. Suporte a estruturas de controlo**
4. Suporte à invocação/regresso de funções
5. Análise comparativa: IA-32 vs. x86-64 e RISC (MIPS e ARM)
6. Acesso e manipulação de dados estruturados

Alteração do fluxo de execução de instruções



- **Por omissão, as instruções são sempre executadas sequencialmente, i.e., uma após outra** (em HLL & em ling. máq.)
- **Em HLL o fluxo de instruções poderá ser alterado:**
 - na execução de estruturas de controlo (*nestes slides...*)
 - na invocação / regresso de funções (*mais adiante...*)
 - na ocorrência de exceções / interrupções (*mais adiante?*)
- **Em ling. máq. isso traduz-se na alteração do IP, de modo incondicional / condicional, por um valor absoluto / relativo**
 - **jump / branch / skip** (no IA-32 apenas **jmp**)
 - **call** (com salvaguarda do endereço de regresso) e **ret**
 - em exceções / interrupções . . .

Instruções de controlo de fluxo no IA-32



`jmp Label %eip ← Label` Unconditional jump

`je Label` Jump if Zero/Equal

`js Label` Jump if Negative

`jg Label` Jump if Greater (signed >)

`jge Label` Jump if Greater or equal (signed ≥)

`ja Label` Jump if Above (unsigned >)

`jb Label` Jump if Below (unsigned <)

`call Label pushl %eip; %eip ← Label` Procedure call

`ret` `popl %eip` Procedure return

Estruturas de controlo de uma linguagem imperativa



Estruturas de controlo em C

– *if-else statement*

Estrutura geral:

```
...  
    if (condição)  
        expressão_1;  
    else  
        expressão_2;  
...
```

Exemplo:

```
int absdiff(int x, int y)  
{  
    if (x < y)  
        return y - x;  
    else  
        return x - y;  
}
```

– *do-while statement*

– *while statement*

– *for loop*

– *switch statement*

Assembly:

"condição":

expressão Booleana, V ou F

"if condição statement":

salte se V para ...

Codificação das condições no IA-32

(para utilização posterior)



- **Condições (V/F) codificadas a partir de registos de 1 bit -> *Flags***

ZF *Zero Flag* SF *Sign Flag*
OF *Overflow Flag* CF *Carry Flag*

- **As *Flags* podem ser implícita ou explicitamente alteradas:**

- implicitamente, por operações aritméticas/lógicas; *exemplo:*

addl *Src, Dest* **Equivalente em C:** $a = a + b$
Flags afetadas: ZF SF OF CF

- explicitamente, por instruções de comparação e teste

cmpl *Src2, Src1* **Equivalente em C...** apenas calcula $Src1 - Src2$
Flags afetadas: ZF SF OF CF
testl *Src2, Src1* **Equivalente em C...** apenas calcula $Src1 \& Src2$
Flags afetadas: ZF SF OF CF

Utilização das Flags no IA-32



A condição codificada a partir das *Flags* pode ser:

- Colocada diretamente num registo de 8 bits (V/F) **ou...**

`setcc Dest` *Dest:* %al %ah %dl %dh %ch %cl %bh %bl

Nota: não altera restantes 3 *bytes* do reg de 32 bits; usada normal/ com `movzbl`

- Usada numa instrução de salto condicional:

`jcc Label` *Label:* endereço destino ou distância para destino

Códigos de condição (*cc*):

<code>(set/j) cc</code>	Descrição	Flags
<code>(set/j) e, z</code>	<i>Equal, zero</i>	ZF
<code>(set/j) ne</code>	<i>Not Equal</i>	~ZF
<code>(set/j) s</code>	<i>Sign (-)</i>	SF
<code>(set/j) ns</code>	<i>Not Sign (-)</i>	~SF

<code>(set/j) g</code>	$> (c/ sinal)$	$\sim(SF \wedge OF) \& \sim ZF$
<code>(set/j) ge</code>	$\geq (c/ sinal)$	$\sim(SF \wedge OF)$
<code>(set/j) l</code>	$< (c/ sinal)$	$(SF \wedge OF)$
<code>(set/j) le</code>	$\leq (c/ sinal)$	$(SF \wedge OF) ZF$
<code>(set/j) a</code>	$> (s/ sinal)$	$\sim CF \& \sim ZF$
<code>(set/j) b</code>	$< (s/ sinal)$	CF

if-then-else statement (1)



Análise de um exemplo

```
int absdiff(int x, int y)
{
    if (x < y)
        return y - x;
    else
        return x - y;
}
```

C original

Corpo {

```
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jl   .L3
    subl %eax, %edx
    movl %edx, %eax
    jmp  .L5
.L3:
    subl %edx, %eax
.L5:
```

```
int goto_diff(int x, int y)
{
    int rval;
    if (x < y)
        goto then_statement;
    rval = x - y;
    goto done;
then_statement:
    rval = y - x;
done:
    return rval;
}
```

Versão goto

```
# edx = x
# eax = y
# compare x : y (~ x-y)
# if x<y, goto then_statement
# compute x - y
# return the value (x - y)
# goto done
# then_statement:
# return the value (y - x)
# done:
```

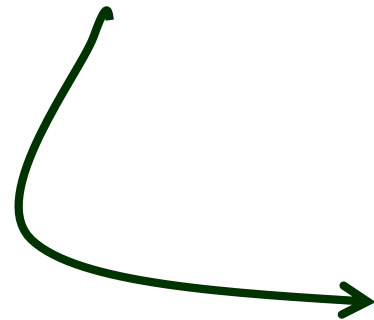
if-then-else statement (2)



Generalização

```
if (expressão_de_teste)  
    then_statement  
else  
    else_statement
```

Forma genérica em C



```
cond = expressão_de_teste  
if (cond)  
    goto true;  
else_statement  
goto done;  
true:  
    then_statement  
done:
```

**Versão com *goto*, ou
assembly com sintaxe C**

if-then-else statement (3)



Generalização alternativa

```
if (expressão_de_teste)  
    then_statement  
else  
    else_statement
```

Forma genérica em C

```
cond = expressão_de_teste  
if (~cond)  
    goto else;  
then_statement  
goto done;  
else:  
    else_statement  
done:
```

**Versão com *goto*, ou
assembly com sintaxe C**

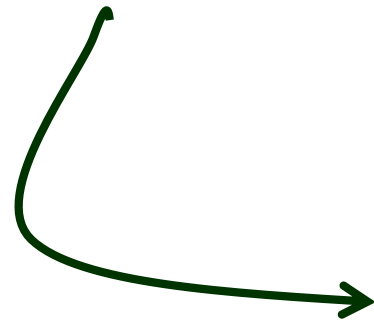
if-then-else statement (4)



Generalização alternativa (*sem else*)

```
if (expressão_de_teste)
    then_statement
else
    else_statement
```

Forma genérica em C



```
cond = expressão_de_teste
if (~cond)
    goto done;
then_statement
goto done;
else:
    else_statement
done:
```

**Versão com *goto*, ou
assembly com sintaxe C**

do-while statement (1)



Generalização

```
do  
    body_statement  
while (expressão_de_teste) ;
```

Forma genérica em C

```
loop:  
    body_statement  
    cond = expressão_de_teste  
    if (cond)  
        goto loop;
```

Versão com *goto*, ou
assembly com sintaxe C

do-while statement (2)



Análise de um exemplo

– série de Fibonacci:

$$F_1 = F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2}, \quad n \geq 3$$

```
int fib_dw(int n)
{
    int i = 0;
    int val = 0;
    int nval = 1;

    do {
        int t = val + nval;
        val = nval;
        nval = t;
        i++;
    } while (i < n);

    return val;
}
```

C original

```
int fib_dw_goto(int n)
{
    int i = 0;
    int val = 0;
    int nval = 1;

loop:
    int t = val + nval;
    val = nval;
    nval = t;
    i++;
    if (i < n);
        goto loop;
    return val;
}
```

Versão com goto

do-while statement (3)



Análise de um exemplo – série de Fibonacci

Utilização dos registos		
Variável	Registo	Valor inicial
n	%esi	n (argumento)
i	%ecx	0
val	%ebx	0
nval	%edx	1
t	%eax	1

Corpo
(loop)

.L2:

```
leal (%edx,%ebx),%eax
movl %edx,%ebx
movl %eax,%edx
incl %ecx
cmpl %esi,%ecx
jl .L2
movl %ebx,%eax
```

```
int fib_dw_goto(int n)
{
    int i = 0;
    int val = 0;
    int nval = 1;

loop:
    int t = val + nval;
    val = nval;
    nval = t;
    i++;
    if (i<n);
        goto loop;
    return val;
}
```

Versão goto

loop:

```
# t = val + nval
# val = nval
# nval = t
# i++
# compare i : n
# if i<n, goto loop
# para devolver val
```

while statement (1)



Generalização

```
while (expressão_de_teste)  
    body_statement
```

Forma genérica em C

```
loop:  
    cond = expressão_de_teste  
    if (! cond)  
        goto done;  
    body_statement  
    goto loop;  
done:
```

Versão com *goto*

```
if (! expressão_de_teste)  
    goto done;  
do  
    body_statement  
    while (expressão_de_teste) ;  
done:
```

Conversão *while* em *do-while*

```
cond = expressão_de_teste  
if (! cond)  
    goto done;  
loop:  
    body_statement  
    cond = expressão_de_teste  
    if (cond)  
        goto loop;  
done:
```

Versão *do-while* com *goto*

while statement (2)



Análise de um exemplo

– série de Fibonacci

```
int fib_w(int n)
{
    int i = 1;
    int val = 1;
    int nval = 1;

    while (i < n) {
        int t = val + nval;
        val = nval;
        nval = t;
        i++;
    }

    return val;
}
```

C original

```
int fib_w_goto(int n)
{
    int i = 1;
    int val = 1;
    int nval = 1;

    if (i ≥ n);
        goto done;

loop:
    int t = val + nval;
    val = nval;
    nval = t;
    i++;
    if (i < n);
        goto loop;
done:
    return val;
}
```

Versão do-while com goto

while statement (3)



Análise de um exemplo – série de Fibonacci

Utilização dos registos		
Variável	Registo	Valor inicial
n	%esi	n
i	%ecx	1
val	%ebx	1
nval	%edx	1
t	%eax	2

```
int fib_w_goto(int n)
{
    (...)

    if (i ≥ n);
        goto done;

loop:
    (...)
    if (i < n);
        goto loop;
done:
    return val;
}
```

**Versão
do-while
com goto**

Corpo {

```
(...)
    cmp1 %esi,%ecx
    jge .L7
.L5:
    (...)
    cmp1 %esi,%ecx
    jl .L5
.L7:
    mov1 %ebx,%eax
```

esi=n, i=val=nval=1
compare i : n
if i ≥ n, goto done
loop:
compare i : n
if i < n, goto loop
done:
return val

**Nota: Código
gerado com
gcc -O1 -S**

for loop (1)



Generalização

```
for (expr_inic; expr_test; update)  
  body_statement
```

Forma genérica em C

```
expr_inic ;  
while (expr_test) {  
  body_statement  
  update ;  
}
```

Conversão
for em
while

```
expr_inic ;  
if (! expr_test)  
  goto done;  
do {  
  body_statement  
  update ;  
} while (expr_test) ;  
done:
```

Conversão
para
do-while

```
expr_inic ;  
cond = expr_test ;  
if (! cond)  
  goto done;  
loop:  
  body_statement  
  update ;  
  cond = expr_test ;  
  if (cond)  
    goto loop;  
done:
```

Versão
do-while
com goto

for loop (2)



Análise de um exemplo

– série de Fibonacci

```
int fib_f(int n)
{
    int i;
    int val = 1;
    int nval = 1;

    for (i=1; i<n; i++) {
        int t = val + nval;
        val = nval;
        nval = t;
    }

    return val;
}
```

C original

```
int fib_f_goto(int n)
{
    int val = 1;
    int nval = 1;

    int i = 1;
    if (i >= n)
        goto done;

loop:
    int t = val + nval;
    val = nval;
    nval = t;
    i++;
    if (i < n)
        goto loop;
done:
    return val;
}
```

Versão do-while com goto

Nota: gcc gera mesmo código...

switch statement



**"Salto" com escolha múltipla;
alternativas de implementação:**

- Sequência de `if-then-else` *statements*
- Com saltos "indiretos": endereços especificados numa tabela de salto (*jump table*)