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, x86-64 e MIPS (RISC)
6. Acesso e manipulação de dados estruturados

- **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 (adiante...)
  - 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 . . .

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 do 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: Se argumento **`x`** colocado em **`%edx`** e argumento **`y`** colocado em **`%eax`**, para implementar a estrutura de controlo **`if-else`** como fazer ?

- **Condições codificadas em registos de 1 bit -> *Flag***

  `ZF` *Zero Flag* `SF` *Sign Flag*

  `OF` *Overflow Flag* `CF` *Carry Flag*

- **As *Flags* podem ser implicita ou explicitamente alteradas:**
  - implicitamente, por operações aritméticas/lógicas

    `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`

A informação das *Flags* pode ser:

- **Colocada diretamente num de 8 registos de 8 bits;** **ou...**

  `set`$_{cc}$ *Dest* *Dest*: `%al %ah %dl %dh %ch %cl %bh %bl`

  Nota: não altera restantes 3 *bytes*; usada normal/ com `movzbl`
- **Usada numa instrução de salto condicional:**

  `j`$_{cc}$ *Label* *Label*: endereço destino **ou** distância para destino

Códigos de condição (`cc`):

(set/j)$_{cc}$	Descrição	*Flags*
(set/j)**e**	*Equal*	**ZF**
(set/j)**ne**	*Not Equal*	**~ZF**
(set/j)**s**	*Sign (-)*	**SF**
(set/j)**ns**	*Not Sign (-)*	**~SF**

(set/j)**g**	*> (c/ sinal)*	**~(SF^OF)& ~ZF**
(set/j)**ge**	*>= (c/ sinal)*	**~(SF^OF)**
(set/j)**l**	*< (c/ sinal)*	**(SF^OF)**
(set/j)**le**	*<= (c/ sinal)*	**(SF^OF)|ZF**
(set/j)**a**	*> (s/ sinal)*	**~CF&~ZF**
(set/j)**b**	*< (s/ sinal)*	**CF**

Análise de um exemplo

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

C original

```
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*

```
    movl 8(%ebp),%edx      # edx = x
    movl 12(%ebp),%eax     # eax = y
    cmpl %eax,%edx         # compare x : y (≈ x-y)
    jl   .L3               # if x<y, goto then_statement
    subl %eax,%edx         # compute x - y
    movl %edx,%eax         # return the value (x - y)
    jmp  .L5               # goto done
.L3:                       # then_statement:
    subl %edx,%eax         # return the value (y - x)
.L5:                       # done:
```

Corpo

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

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

Generalização alternativa

```
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

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

Análise de um exemplo

– **série de Fibonacci:** $F_1 = F_2 = 1$

$F_n = F_{n-1} + F_{n-2}$, $n>=3$

```
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*

Análise de um exemplo
– série de Fibonacci

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

```
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*

Corpo (*loop*)

```
.L2:
    leal (%edx,%ebx),%eax   # t = val + nval
    movl %edx,%ebx          # val = nval
    movl %eax,%edx          # nval = t
    incl %ecx               # i++
    cmpl %esi,%ecx          # compare i : n
    jl   .L2                # if i<n, goto loop
    movl %ebx,%eax          # para devolver val
```


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*

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*

Análise de um exemplo
– série de Fibonacci

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

```
    (…)
    cmpl %esi,%ecx      # esi=n, i=val=nval=1
    jge  .L7            # compare i : n
.L5:                    # if i>=n, goto done
    (…)                 # loop:
    cmpl %esi,%ecx      # compare i : n
    jl   .L5            # if i<n, goto loop
.L7:                    # done:
    movl %ebx,%eax      # 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*)