ESTRUTURAS DE CONTROLO

TeSP de Aplicações Móveis

André Martins Pereira

IPCA
INSTITUTO POLITÉCNICO
DO CÁVADO E DO AVE
ESCOLA SUPERIOR DE TECNOLOGIA

FLUXO DE EXECUÇÃO

- Até agora foi visto que o fluxo de execução de instruções é normalmente sequencial
    - CPUs podem alterar essa ordem, *out-of-order execution*, mas trata-se de um mecanismo de mais baixo nível ao que está a ser estudado
    - Função é inicializada, instruções são executadas por ordem, uma única vez, e resultados são guardados/retornados

FLUXO DE EXECUÇÃO

- Em linguagens de alto nível é possível alterar esta ordem de execução
  - Execução de estruturas de controlo
    - Instruções *if*
  - Execução de estruturas de repetição
    - Ciclos *for*, *while*, *do-while*...
  - Gestão de exceções
    - Normalmente apenas suportado em linguagens por objectos

FLUXO DE EXECUÇÃO

- Como é que essas alterações são refletidas no *assembly*?
  - Temos um registo que indica a próxima instrução a ser executada...
  - Existem instruções específicas para alterar o seu valor
    - De modo condicional ou incondicional
    - Por um valor absoluto ou relativo
  - Instruções com propósitos diferentes
    - *jmp, jge, ...*
    - *call, ret*
    - ...

CODIFICAÇÃO DE CONDIÇÕES

- O resultado de uma condição é guardado em registos específicos
  - Tem apenas 1 bit
  - Servem diferentes propósitos
    - CF – Carry Flag
    - SF – Sign Flag
    - ZF – Zero Flag
    - OF – Overflow Flag

CODIFICAÇÃO DE CONDIÇÕES

- Estes registos (denominados de Flags) podem ser implicitamente ou explicitamente alterados
  - Um *addl* altera implicitamente as 4 flags
  - Um *cmpl s2, s1* faz uma comparação equivalente a *s2 – s1* e altera explicitamente as 4 flags
  - Um *testl s2, s1* faz uma comparação *bit-wise s2&s1* e altera explicitamente as 4 flags

UTILIZAÇÃO DAS FLAGS

- Podem ser colocadas em registos de 8 bits
- Podem ser usadas por instruções específicas

(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 (1) *IF-THEN-ELSE*

```
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*

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:
```

```
# edx = x
# eax = y
# compare x : y
# if <, goto then_statement
# edx = x - y
# return value = edx
# goto done
# then_statement:
# return value = y - x
# done:
```

AJProença, Sistemas de Computação,

6

ANÁLISE DE UM EXEMPLO (2) *DO-WHILE LOOP*

- Passem o código para uma versão em pseudo-código com a instrução *goto*
  - **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 (2) *DO-WHILE LOOP*

Utilização dos registos		
Registo	Variável	Valor inicial
%ecx	i	0
%esi	n	n (argumento)
%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   # loop:
    movl  %edx,%ebx          # t = val + nval
    movl  %eax,%edx          # val = nval
    incl  %ecx               # nval = t
    cmpl  %esi,%ecx          # i++
    jl    .L2                # compare i : n
    movl  %ebx,%eax          # if <, goto loop
                             # para devolver val
```

ANÁLISE DE UM EXEMPLO (3) *WHILE LOOP*

```c
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

```c
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 (3) *WHILE LOOP*

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

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

ANÁLISE DE UM EXEMPLO (4) *FOR LOOP*

O compilador gera código equivalente à versão *do-while* com *goto*

```
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;
}
```

```
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;
}
```