Fluxo de Execução em Assembly

- A forma natural de execução de um programa é sequencial: CPU busca instruções na memória em endereços sequenciais
- *Instruções de desvio* de assembly servem para quebrar a execução sequencial (i.e. pular para outro ponto do código, como em um “goto”)
- O ponto é marcado com um rótulo (label)
- Existem desvios *condicionais* e *incondicionais* (“jump” no Pentium, ou “branch”)

Como implementar essas estruturas de controle?

```
if (a==b) c=d;
d=a+c;
```

```
if (a>=b) {
  c=(a-b);
}
else {
  c=(b-a);
}
```

```
while (a<=b) {
  ...
  a++;
}
d=a+c
```

```
for (i=0; i< 10; i++){
  Sum+=sum+i;
}
```

Jump Condicional

```
addl%eax, %ebx
jne label          # se %ebx != 0  vai para label
```

- A condição se refere sempre ao resultado da instrução anterior
- No Pentium, a maioria das instruções registram características de seus resultados nos bits/flags de EFLAGS
- Os desvios condicionais operam sobre o valor destes flags

```
cmpl %eax, %ebx    # compara %ebx com %eax
jle label          # se %ebx <= %eax  vai para label
```

- Obs: note a inversão dos operandos!

Flags de Condição (1/2)

`CF` Carry Flag   `SF` Sign Flag

`ZF` Zero Flag   `OF` Overflow Flag

`PF` Parity Flag

Setados implicitamente por operadores aritméticos, p.ex.

addl f, d

Equivale: `d = f +` d_{ant}

– CF=1 se houver carry do bit mais significativo(overflow de unsigned): exemplo `jc label`

– ZF=1 se `d == 0` exemplo `jz label`

– SF=1 se `d < 0` exemplo `js label`

– OF=1 se houver overflow em compl.-2

– `(f>0 && `d_{ant}` >0 && d<0) || (f<0 && `d_{ant}` <0 && d>=0)` usado em `jg, jl, jge,…`

Flags de condição (2/2)

- Instrução de comparação:

  `cmpl` *b,a*

  – equivale a `a-b` sem atribuição do resultado
  – CF=1 se carry do bit mais significativo
    - Para comparação de unsigned
  – ZF=1 se `a == b` exemplo `jz label`
  – SF=1 se `(a-b) < 0` exemplo `js label`
  – OF=1 se houver overflow em complemento a dois (com sinal)
    - `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

Todos os Desvios

- Instruções tipo: **`jx label`**
  – Jump dependendo dos flags de condição

jX	condição	Descrição
`jmp`	`1`	**Unconditional**
`je`	`ZF`	**Equal / Zero**
`jne`	`~ZF`	**Not Equal / Not Zero**
`js`	`SF`	**Negative**
`jns`	`~SF`	**Nonnegative**
`jg`	`~(SF^OF)&~ZF`	**Greater (Signed)**
`jge`	`~(SF^OF)`	**Greater or Equal (Signed)**
`jl`	`(SF^OF)`	**Less (Signed)**
`jle`	`(SF^OF)|ZF`	**Less or Equal (Signed)**
`ja`	`~CF&~ZF`	**Above (unsigned)**
`jb`	`CF`	**Below (unsigned)**

Traduzindo “if-else”

```
if (test-expr)
    then-statement;
  else
    else-statement;
```

Esquema geral

```
t= test-expr;
if (t)
goto true;
       else-statement
goto done;
true:
       then-statement
done:
```

```
d = 16;
a = 10;
if (d < a) c= a - d;
else c = d -a;
```

Caso Específico

```
       movl $0x10, %edx
       movl $0xA, %eax
       cmpl %eax, %edx
       jl .L3
       subl %eax, %edx
       movl %edx, %ecx
       jmp .L5
.L3:          /* true */
       subl %edx, %eax
       movl %edx, %ecx
.L5:          /* done*/
```

Traduzindo “if” puro

Quando não há parte else, deve-se negar a condição de teste.

```
if (test-expr)
    then-statement;
```

Esquema geral

```
t= test-expr;
if (!t)
goto done;
       then-statement
done:
```

```
if (a==b) {
       c= d;
}
d = a+c;
```

Caso Específico

```
       cmpl %eax, %ebx
       jne .L5
       movl %edx, %ecx
.L5:          /* done*/
       movl %eax, %edx
       addl %ecx, %edx
```

Traduzindo loops “do-while”

Código C

```
int fat (int x) {
int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

Versão goto

```
int fat_goto  (int x) {
int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

Faça desvio para trás para continuar iterando (enquanto condição do while é satisfeita)

Traduzindo loops “do-while”

Versão goto

```
int fat_goto  (int x)
{
int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

```
_fat_goto:
   pushl %ebp            # Setup
   movl %esp,%ebp        # Setup
   movl $1,%eax          # eax = 1
   movl 8(%ebp),%edx     # edx = x

L11:
   imull %edx,%eax       # result *= x
   decl %edx             # x--
   cmpl $1,%edx          # Compare x : 1
   jg L11                # if > goto loop

   movl %ebp,%esp        # Finish
   popl %ebp             # Finish
   ret                   # Finish
```

Obs: O valor de retorno é o último valor armazenado em %eax antes de `ret`.

Diferenças “do-while” X “while”

Código C

```
do
  Body
while (Test);
```

Versão goto

```
loop:
  Body
  if (Test)
  goto loop
```

Código C

```
while (Test){
  Body
}
```

Versão goto

```
loop:
   If (!Test)
   goto after
   Body
   goto loop
after:
```

Tradução do “while”: um exemplo

```
while (a<=b ){
  …
     a++;
}
d=a+c;
```

```
loop:
  cmpl %ebx, %eax
  jg depois  /* se a>b */
  …
  incl %eax
  jmp loop
depois:
  movl %ecx, %edx
  addl %eax, %edx
```

Convertendo “while” como “do-while”

```
while (Test)
  Body
```

Outro esquema geral para implementar “while”

Versão do-while

```
  if (!Test)
    goto done;
  do
    Body
  while(Test);
done:
```

Versão c/ goto

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

Traduzindo o “for”

```
for (Init; Test; Update )

    Body
```

Versão “while”

```
Init;
while (Test ) {
    Body
    Update ;
}
```

Versão “do-while”

```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update ;
  } while (Test)
done:
```

Versão goto

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update ;
  if (Test)
    goto loop;
done:
```

Avaliando condições complexas

- A condição ***test*** pode conter operadores lógicos
  - Exemplo: ((c>a) || (a ==1) && (d < b)))
- Expressão é avaliada da esquerda p/ direita
- Em C, e outras linguagens de alto nível, a avaliação é interrompida assim que o resultado é conhecido (“*curto circuito”*)
  - (x || y) se x resulta em true, não executa y
  - (x && y) se x resulta em false, não executa y
- Isto é refletido no código Assembly gerado

```
if ((a==b) || (c<d))
{
  a = c;      //L1
}
  c = d;      //Lfim
```

```
    cmp %ebx, %eax
    je L1
    cmp %edx, %ecx
    jge Lfim
L1:  movl %ecx, %eax
Lfim: movl %edx, %ecx
```

Desvios indiretos

- É possível também fazer um desvio para o endereço contido em um registrador, ou na memória (sintaxe: prefixo ‘*’)

`jmp *%ebx` destino: conteúdo de ebx

`jmp *(%ebx)` destino: conteúdo de M[ebx]

`jmp *.tab(%ebx)` destino: “ de M[tab+ebx]

Isto é usado na tradução do “switch” de C, onde monta-se uma ***tabela de jump***, contendo os endereços dos desvios.

Uso da Tabela Jump para switch

Switch

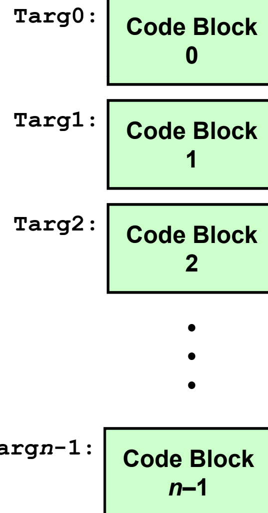
```
switch(op) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Tabela jump

jtab:

Targ0
Targ1
Targ2
• • •
Targ*n*-1

Blocos & endereços



Tradução:

- op vira o índice na tabela

```
target = jtab[op];
goto *target;
```

```
movl op, %ebx
jmp *.jtab(%ebx)
```