Software Básico

Tradução de Mecanismos de Controle

Fluxo de Execução

- Ciclo da CPU: *fetch-decode-execute*
  - CPU busca automaticamente a instrução seguinte à executada anteriormente
- Registrador RIP contém o endereço da próxima instrução a ser executada

1150	`MOVQ L1, %rdx`
1158	`MOVL (%rdx),%ecx`
115b	`SUBL VG, %ecx`
1162	`ANDL $1, %ecx`
1165	`ADDL %eax, %ecx`
1167	`ADDL (%rsi), %ecx`

Fluxo de Execução

- No entanto, para processar um array de 500 posições, não vamos repetir 500 vezes um grupo de instruções
- Instruções de desvio de controle permitem alterar esse fluxo

1150	MOVQ L1, %rdx
1158	MOVL (%rdx),%ecx
115b	SUBL VG, %ecx
1162	ANDL $1, %ecx
1165	ADDL %eax, %ecx
1167	ADDL (%rsi), %ecx
1169	JZ 115b

Fluxo de Execução

- No entanto, para processar um array de 500 posições, não vamos repetir 500 vezes um grupo de instruções
- Instruções de desvio de controle permitem alterar esse fluxo
- Essas instruções mudam o valor de RIP

Estruturas de Controle

- Em linguagens como C:
  - “if-then-else” e “switch-case” permitem a execução condicional de código
  - “while”, “for” e “do-while” permitem a execução de código zero, uma ou várias vezes

```
if (d < a) {
  c = a - d;
} else {
  c = d - a;
}
```

```
while (a < 10) {
  ...
  a++;
}
```

Estruturas de Controle

- Em Assembly, temos mecanismos básicos para obter o mesmo resultado
  - Mas nem sempre tão simples como em C
- Temos instruções para
  - Testar resultado de operações aritméticas e lógicas
  - Desviar (alterar o fluxo de controle do programa) conforme esse resultado

Registrador RFLAGS

- O registrador RFLAGS possui vários bits (flags) usados pelo mecanismo de controle de fluxo

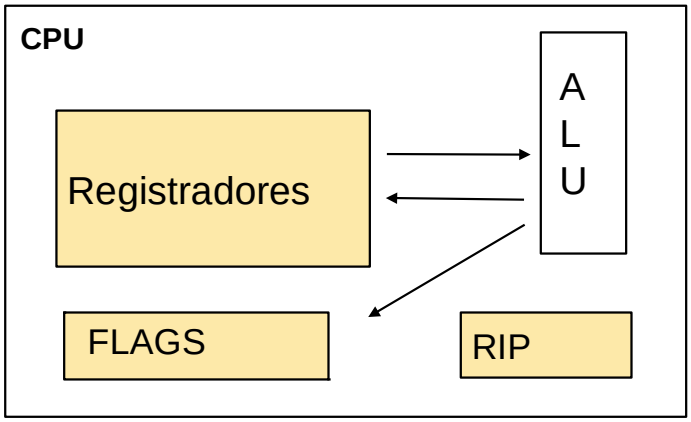
FLAG		SIGNIFICADO
CF	Carry Flag	Última operação gerou um bit “vai-um”
ZF	Zero Flag	Última operação resultou em zero
SF	Signal Flag	Última operação resultou em um valor negativo
OF	Overflow Flag	Última operação resultou em overflow

Registrador RFLAGS

- O registrador RFLAGS possui vários bits (flags) usados pelo mecanismo de controle de fluxo
- Os bits são atualizados automaticamente de acordo com o resultado das operações lógicas, aritméticas ou de comparação

FLAG		SIGNIFICADO
CF	Carry Flag	Última operação gerou um bit “vai-um”
ZF	Zero Flag	Última operação resultou em zero
SF	Signal Flag	Última operação resultou em um valor negativo
OF	Overflow Flag	Última operação resultou em overflow

Registrador RFLAGS



Registrador RFLAGS

- O registrador RFLAGS possui vários bits (flags) usados pelo mecanismo de controle de fluxo
- Os bits são atualizados automaticamente de acordo com o resultado das operações lógicas, aritméticas ou de comparação
- Combinações desses bits podem ser usadas como condições para alterar o fluxo de controle decisão sobre qual a próxima instrução a ser executada

Comparação de Dados

- Instrução “cmp” compara a fonte$_1$ com a fonte$_2$ e ativa as flags de acordo com o resultado

```
cmp   fonte2, fonte1    /* f1 - f2 */
```

- Os valores das fontes não são alterados

```
cmpl %eax, %ebx     /* %ebx - %eax  */
cmpb %al, (%rbx)    /* (%rbx) - %al */
cmpq %rsi, x        /* x - %si      */
```

Teste de Dados

- Instrução “test” compara a fonte$_1$ com a fonte$_2$ e ativa as flags de acordo com o resultado

```
test   fonte2, fonte1    /* f1 & f2 */
```

- Os valores das fontes não são alterados

```
testl %eax, %ebx     /* %ebx & %eax  */
testb %al, (%rbx)    /* (%rbx) & %al */
testq %rsi, x        /* x & %si      */
```

Desvio Incondicional

- Semelhante ao “goto” de C
- Muda o fluxo de execução imediatamente para um label
  - Não leva em conta as flags do RFLAGS
- Sintaxe

```
jmp label
```

```
L1:
  addl %eax, %ebx
  subl %ebx, %ecx
  …
  jmp L1
```

Desvio Condicional

- Muda o fluxo de execução de acordo com as flags do RFLAGS
  - Ou seja, de acordo com as comparações e testes
- Temos que tomar cuidados com o tipo (signed ou unsigned) de dado que estão envolvidos nas condições

Desvio Condicional

INSTRUÇÃO	SINÔNIMO	DESCRIÇÃO
`je` ***Label***	`jz`	equal / zero
`jne` ***Label***	`jnz`	not equal / not zero
`js` ***Label***		negative
`jns` ***Label***		non negative
`jg` ***Label***	`jnle`	> (greater)
`jge` ***Label***	`jnl`	>= (greater or equal)
`jl` ***Label***	`jnge`	< (less)
`jle` ***Label***	`jng`	<= (less or equal)

Comparação *signed* !!!

Desvio Condicional

INSTRUÇÃO	SINÔNIMO	DESCRIÇÃO
`je` ***Label***	`jz`	equal / zero
`jne` ***Label***	`jnz`	not equal / not zero
`ja` ***Label***	`jnbe`	> (above)
`jae` ***Label***	`jnb`	>= (above or equal)
`jb` ***Label***	`jnae`	< (below)
`jbe` ***Label***	`jna`	<= (below or equal)

Comparação *unsigned* !!!

Exemplo

Linguagem C

```
int a = 10;
int b = 5;

if (a > b) goto fim;

…

fim:
```

Assembly

```
movl $10, %eax
movl $5,  %ebx

cmpl %ebx, %eax
jg   fim
…

fim:
```

Exemplo

Linguagem C

```
int a = 10;
int b = 5;

if (a > b) goto fim;

…

fim:
```

Assembly

```
movl $10, %eax
movl $5,  %ebx

cmpl %ebx, %eax
jg   fim
…

fim:
```

Fonte$_1$ é maior que Fonte$_2$???

Exemplo

Linguagem C

```
int a = 10;
int b = 5;

if (a > b) goto fim;

…

fim:
```

Assembly

```
movl $10, %eax
movl $5,  %ebx

cmpl %ebx, %eax
jg   fim
…

fim:
```


%eax é maior que %ebx ???

Exemplo

Linguagem C

```
unsigned int a = 10;
unsigned int b = 5;

if (a > b) goto fim;

…

fim:
```

Assembly

```
movl $10, %eax
movl $5,  %ebx

cmpl %ebx, %eax
ja   fim
…

fim:
```

A interpretação se um número é signed ou unsigned está na operação usada e não no padrão de bits do número

Mapeando as Estruturas de Controle para Assembly

Estrutura Condicional “if”

- Bloco do “if” só é executado se expressão de teste for verdadeira

```
if ( <expr> ) {
    // bloco
}

// Continuação do programa
```

Estrutura Condicional “if”

```
if ( <expr> ) {
    // bloco
}

// Continuação
```

```
  if (! <expr> ) goto depois_if;

  // bloco

depois_if:
  // Continuação
```

Estrutura Condicional “if”

```
if ( a > b ) {
   a = a + b;
}

printf("%d\n", a);
```

```
  if (! (a > b) ) goto depois_if;

  a = a + b

depois_if:
  printf("%d\n", a);
```

Estrutura Condicional “if”

```
if ( a > b ) {
   a = a + b;
}

printf("%d\n", a);
```

```
  if (a <= b) goto depois_if;

  a = a + b

depois_if:
  printf("%d\n", a);
```

Estrutura Condicional “if”

```
if (a <= b) goto depois_if;

a = a + b

depois_if:
printf("%d\n", a);
```

Teste

```
  movl a, %eax
  movl b, %ebx

  cmpl %ebx, %eax
  jle  depois_if

  addl %ebx, %eax

depois_if:
  call printf
```

Estrutura Condicional “if”

```
if (a <= b) goto depois_if;

a = a + b

depois_if:
printf("%d\n", a);
```

```
  movl a, %eax
  movl b, %ebx

  cmpl %ebx, %eax
  jle  depois_if

  addl %ebx, %eax

depois_if:
  call printf
```

Corpo

Estrutura Condicional “if-then-else”

```
if ( <expr> ) {
  // bloco if
}
else {
  // bloco else
}

// continuação
```

```
   if (! <expr> ) goto blc_else;

   // bloco if
   goto depois_if;

blc_else:
   // bloco else

depois_if:
   // continuação
```

Estrutura Condicional “if-then-else”

```
a = 10;
b = 5;

if ( a < b )
  c = a - b
else
  c = b - a

printf("%d", c);
```

```
   movl $10, %eax
   movl $5,  %ebx

   cmpl %ebx, %eax
   jge  blc_else

   movl %eax, %ecx
   subl %ebx, %ecx
   jmp  depois_if
blc_else:
   movl %ebx, %ecx
   subl %eax, %ecx
depois_if:
   call printf
```

Estrutura de Repetição “while”

- Bloco do “while” executa enquanto a expressão de teste for verdadeira

```
while (<expr>) {

  // bloco

}

// continuação
```

Estrutura de Repetição “while”

- Bloco do “while” executa enquanto a expressão de teste for verdadeira

```
while (<expr>) {

  // bloco

}

// continuação
```

```
loop:
   if (! <expr>) goto depois_while;

   // bloco while

   goto loop

depois_while:
   // continuação
```

Estrutura de Repetição “while”

```
i = 0; sum = 0;

while (i < 10) {
  sum += i;
  i++;
}

printf("%d", sum);
```

```
i = 0; sum = 0;
loop:
   if (i >= 10) goto after;
   sum += i;
   i++;
   goto loop;
after:
   printf("%d", sum);
```

Estrutura de Repetição “while”

```
i = 0; sum = 0;
loop:
   if (i >= 10) goto after;
   sum += i;
   i++;
   goto loop;
after:
   printf("%d", sum);
```

```
   movl $0, %ecx   # i
   movl $0, %eax   # sum
loop:
   cmpl $10, %ecx
   jge  after
   addl %ecx, %eax
   incl %ecx
   jmp  loop
after:
   call printf
```

Estrutura de Repetição “for”

- <init> é executada uma vez, antes do loop
- <test> é executada antes de cada iteração
- <update> é executada no fim de cada iteração

```
for (<init>; <test>; <update>) {
   // bloco for
}

// continuação
```

Estrutura de Repetição “for”

- <init> é executada uma vez, antes do loop
- <test> é executada antes de cada iteração
- <update> é executada no fim de cada iteração

```
for (<init>;<test>;<update>) {
   // bloco for
}

// continuação
```

```
<init>
while (<test>) {
   // bloco for
   <update>
}
// continuação
```

Estrutura de Repetição “for”

- <init> é executada uma vez, antes do loop
- <test> é executada antes de cada iteração
- <update> é executada no fim de cada iteração

```
for (<init>;<test>;<update>) {
   // bloco for
}

// continuação
```

```
  <init>
loop:
   if (!<test>) goto after;
   // bloco for
   <update>
   goto loop;
after:
   // continuação
```

Estrutura de Repetição “for”

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

printf("%d", sum);
```

```
  movl $0, %eax   # sum
  movl $0, %ecx   # i
loop:
  cmpl $10, %ecx
  jge  after

  addl %ecx, %eax

  incl %ecx
  jmp  loop
after:
  # call printf
```

Operadores Lógicos
AND e OR

Operadores AND e OR

- Não existe operadores lógicos AND e OR em Assembly
- O conceito de *curto-circuito* deve ser implementado pelo próprio programador

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

c = d;
```

Operadores AND e OR

- Não existe operadores lógicos AND e OR em Assembly
- O conceito de *curto-circuito* deve ser implementado pelo próprio programador

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

c = d;
```

```
  if (a == b) goto bloco;
  if (c >= d) goto after;
bloco:
  a = c;
after:
  c = d;
```

Operadores AND e OR

```
  if (a == b) goto bloco;
  if (c >= d) goto after;
bloco:
  a = c;
after:
  c = d;
```

```
  cmpl %ebx, %eax
  je   bloco
  cmpl %edx, %ecx
  jge  after

bloco:
  movl %ecx, %eax

after:
  movl %edx, %ecx
```

Operadores AND e OR

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

c = d;
```

```
  if (a != b) goto after;
  if (c >= d) goto after;
  a = c;
after:
  c = d;
```

Operadores AND e OR

```
  if (a != b) goto after;
  if (c >= d) goto after;
  a = c;
after:
  c = d;
```

```
  cmpl %ebx, %eax
  jne  after
  cmpl %edx, %ecx
  jge  after

  movl %ecx, %eax

after:
  movl %edx, %ecx
```