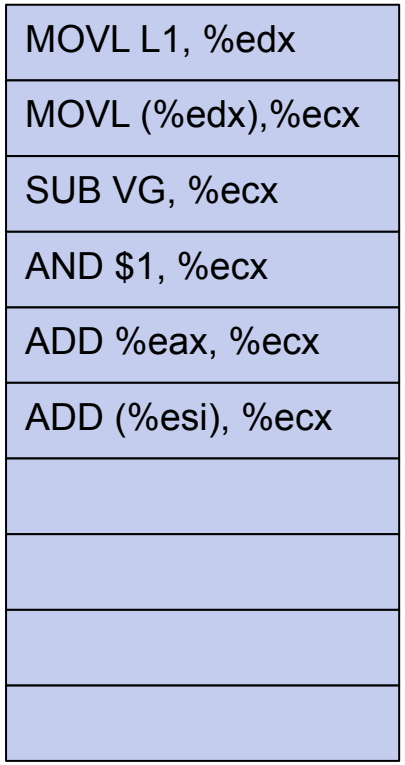
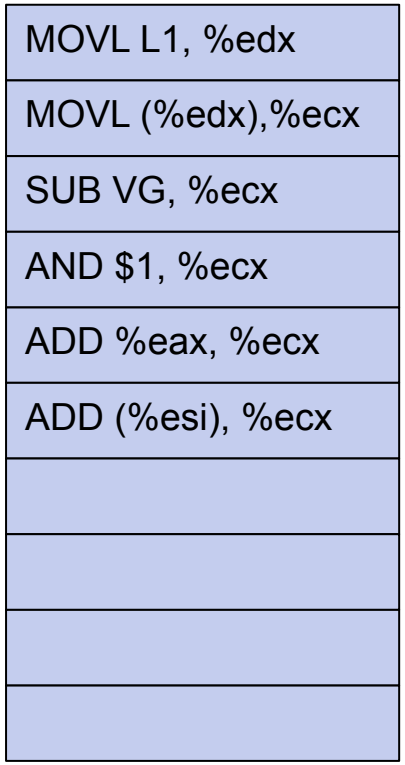
Assembly – Tradução de Mecanismos de Controle

Fluxo de Execução



fetch-decode-execute
busca automaticamente a instrução seguinte à executada anteriormente

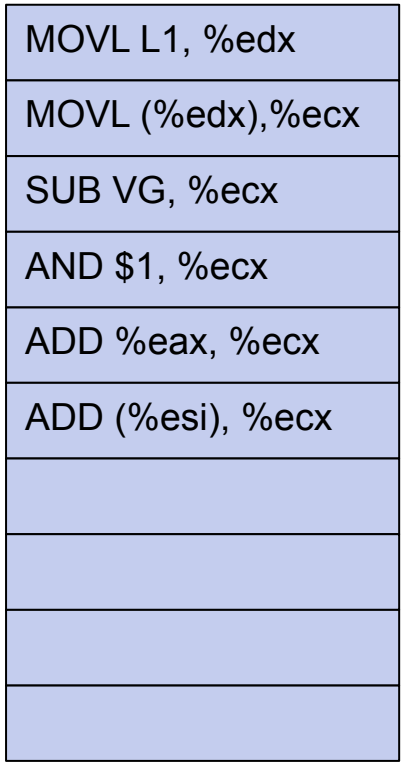
Fluxo de Execução



fetch-decode-execute
busca automaticamente a instrução seguinte à executada anteriormente

... mas então vamos repetir 500 vezes um grupo de instruções quando quisermos tratar um array de 500 posições?

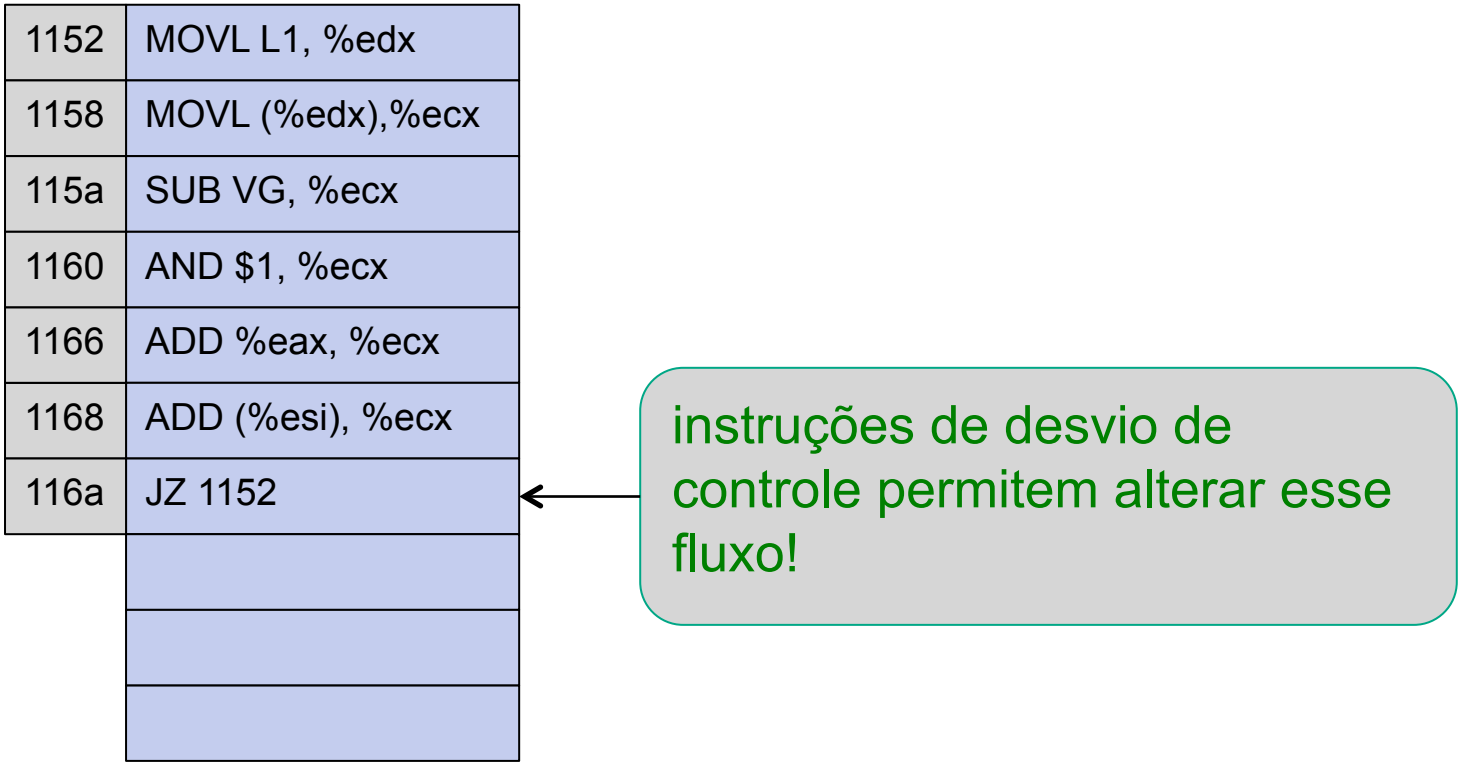
Fluxo de Execução



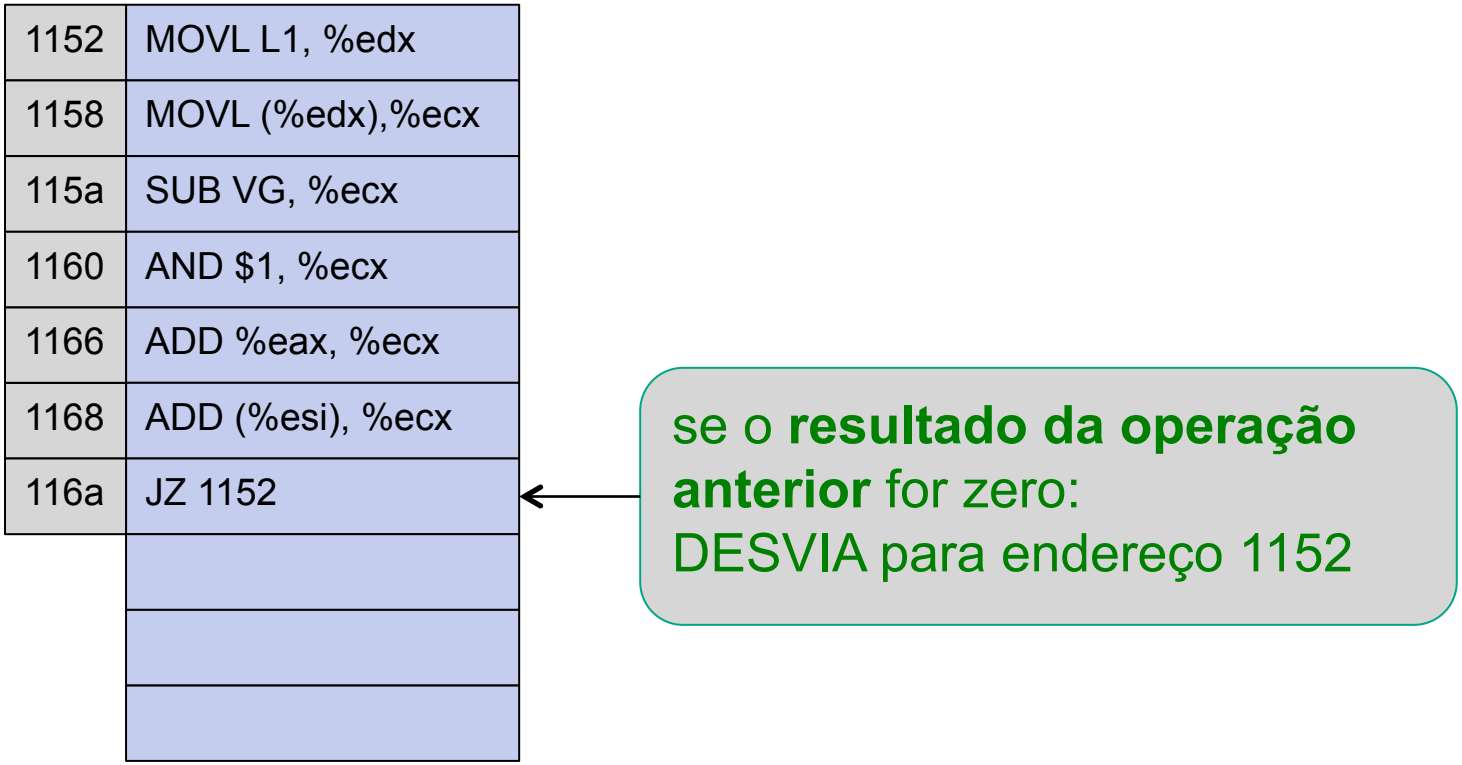
fetch-decode-execute
busca automaticamente a instrução seguinte à executada anteriormente

instruções de desvio de controle permitem alterar esse fluxo!

Fluxo de Execução



Fluxo de Execução



Fluxo de Execução em Assembly

- Em C:
  - a sequência de operações executada depende do resultado de testes aplicados aos dados: execução condicional

```
if (d < a)
  c= a - d;
else
  c = d - a;
while (a<=b ){
  …
  a++;
}
...
```

Fluxo de Execução em Assembly

- em C:
  - a sequência de operações executada depende do resultado de testes aplicados aos dados: execução condicional

- na linguagem de máquina também, mas com mecanismos bem mais limitados:
  - testar resultado de operações aritméticas e lógicas
  - *desviar* (alterar o fluxo de controle do programa) conforme esse resultado

Testes sobre Dados

- A CPU mantém um conjunto de bits (EFLAGS) que descreve o resultado da última operação aritmética/ lógica realizada
  - ZF, SF, CF, OF

- Combinações desses bits podem ser usadas como condições para alterar o fluxo de controle
  - decisão sobre qual a próxima instrução a ser executada

Instruções de Desvio

- Podem alterar a sequência de execução (“goto”)
- O destino do desvio é indicado no código assembly por um “label simbólico”
  - traduzido pelo assembler + linker para um endereço de memória

```
L1:
    cmpl   %ebx, %ebx
    jz     L2      ➔ condicional
    ...
    jmp    L1      ➔ incondicional
L2:
    xorl       %eax, %eax
```

Desvios Condicionais

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)

Desvios Condicionais

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!

Desvios Condicionais

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!

Mecanismos de controle: ‘if’

- Suponha o código C:

```
if (a==b)
  c=d;      ➔   suponha a em %eax, b em %ebx,
                          c em %ecx, d em %edx
d=a+c;
```

Mecanismos de controle: ‘if’

- Suponha o código C:

```
if (a==b) c=d;   ➔ %eax, %ebx, %ecx, %edx
d=a+c;
```

- Esse código pode ser traduzido da seguinte forma:

```
      cmpl %eax, %ebx
      jne depois_if    ➔ condição é negativa do if!
      movl %edx, %ecx
depois_if:
      movl %eax, %edx
      addl %ecx, %edx
```

Exemplo do Laboratório

- Impressão dos números pares de um array

```
  movl  $S2, %ecx

L1:
   movl  (%ecx), %eax
   cmpl  $0, %eax
   je      L3
   movl  %eax, %edx
   andl  $0x01, %edx    / * ímpar ? */
   jnz     L2
   ...
   call printf
  ...
L2:
   addl    $4, %ecx  /* ecx += 4; */
   jmp    L1

L3:
```

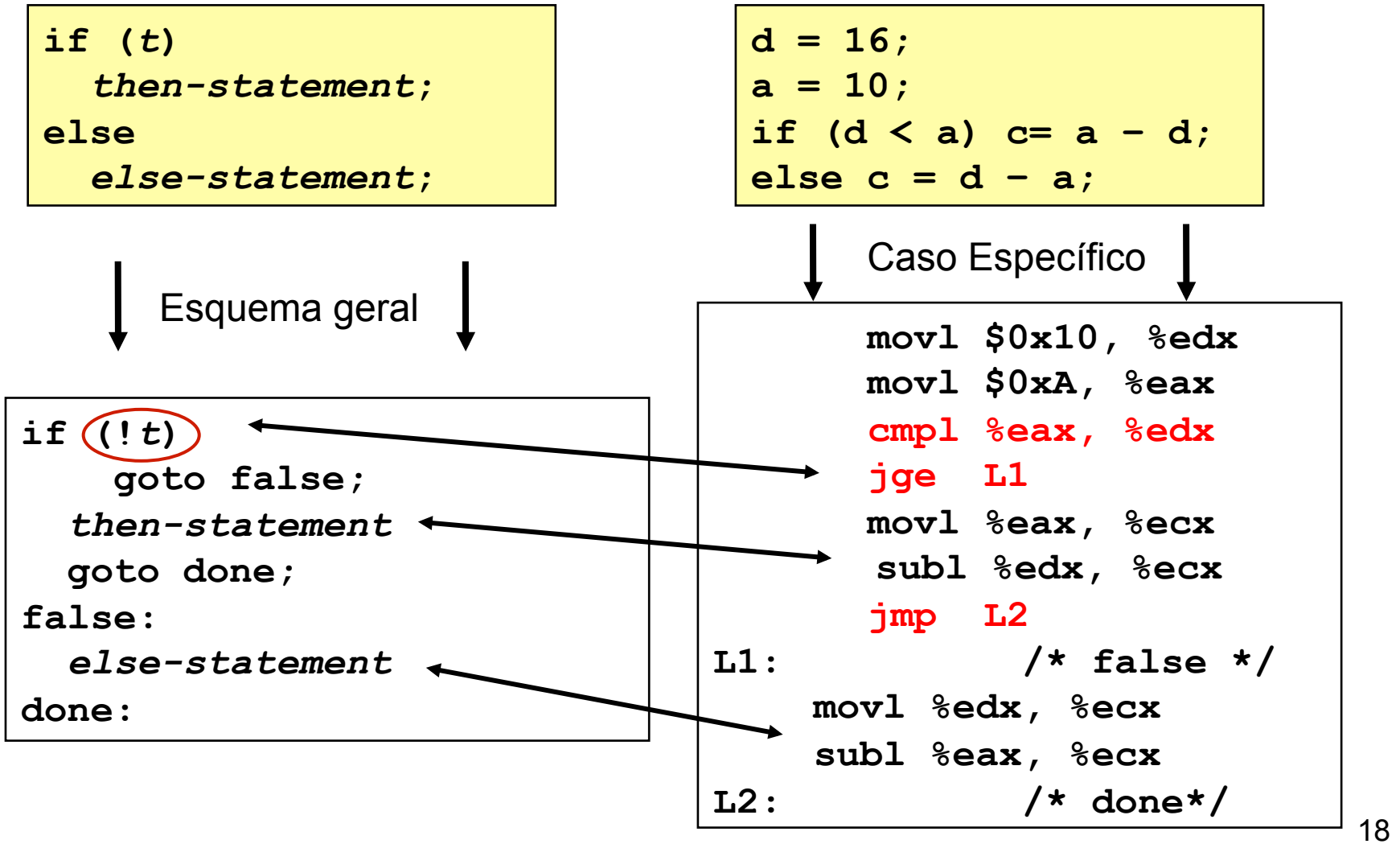
```
if (a % 2 == 0)
   printf(…);
```

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

Traduzindo “if-else”

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

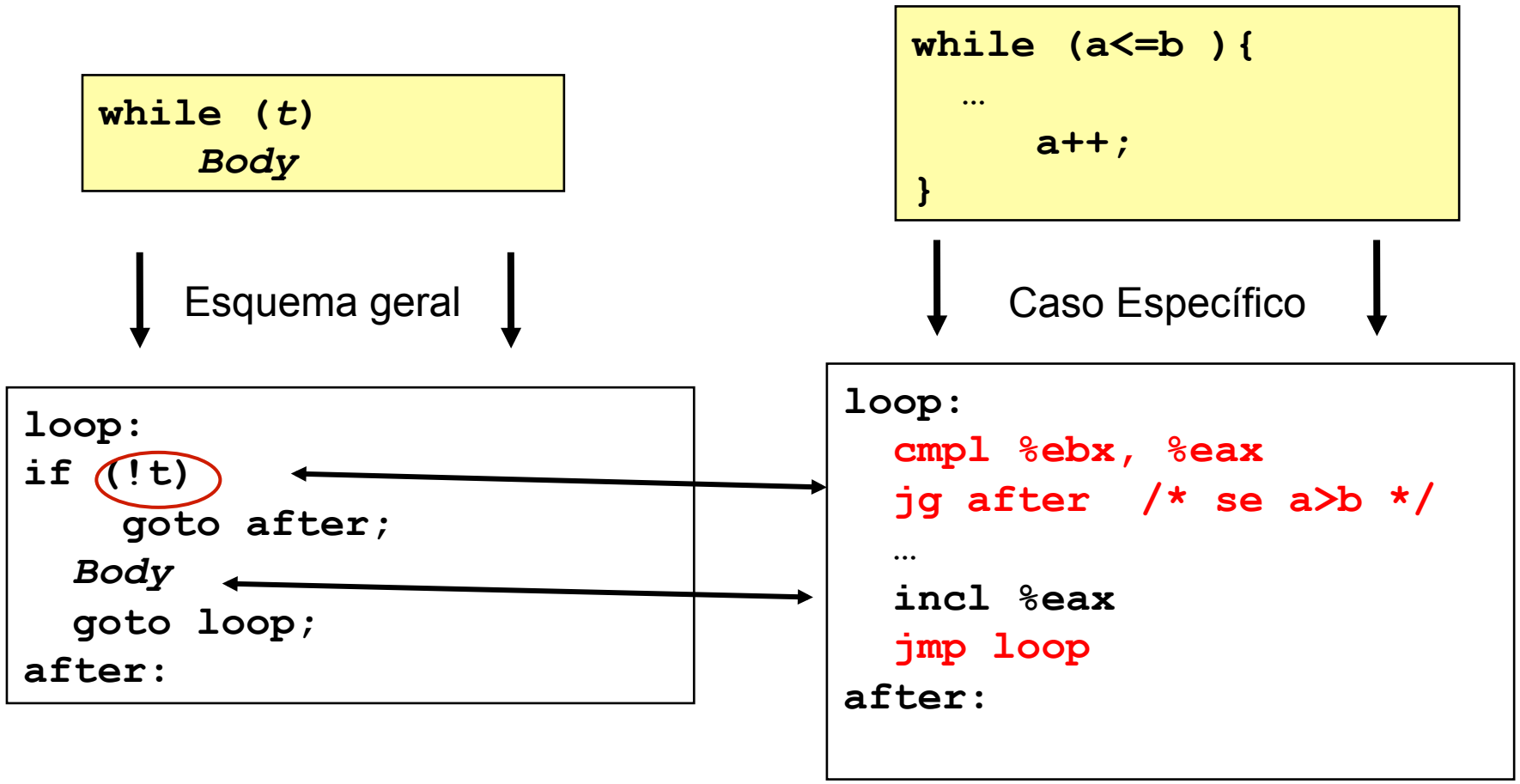
Traduzindo “if-else”



Traduzindo “while”

```
while (a<=b ){
   …
   a++;
}
```

Traduzindo “while”

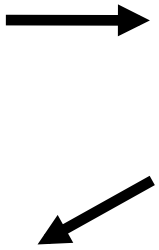


Traduzindo o “for”

```
for (Init; Test; Update)
    Body
```

Traduzindo o “for”

```
for (Init; Test; Update)
    Body
```



The Goto statement considered harmful

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

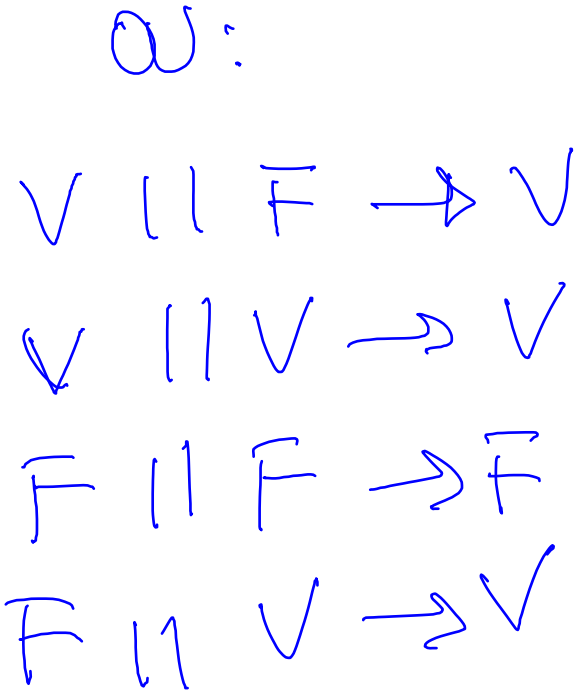
```
  Init;
loop:
  if (!Test)
    goto after;
  Body;
  Update;
  goto loop;
after:
```

instrução que "seta" flags

desvio condicional

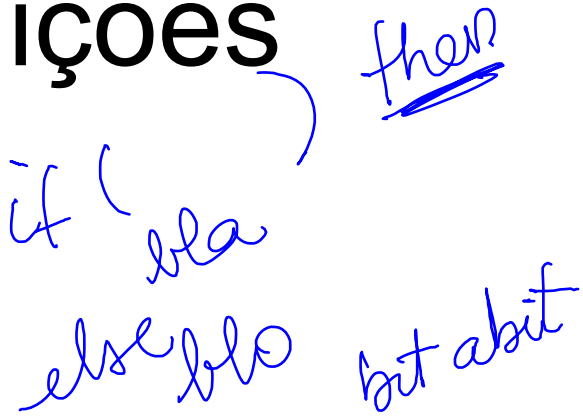
jump

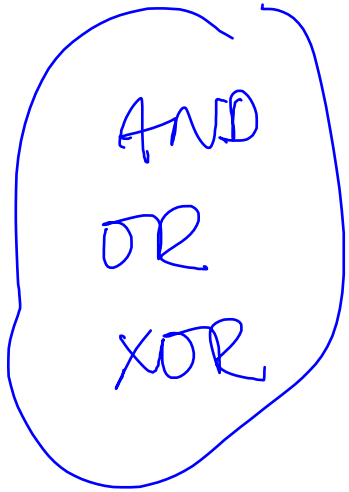
curto circuito em condições lógicas




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

não existem operadores *and* e *or* lógicos nas linguagens de máquina!





curto circuito em condições lógicas

&&

F && ? → F

V && V → V

V && F → F

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

if((a != NULL) &&
a → idade > 10)

tem que testar cada uma delas e construir caminhos diferentes no código!

- em C, como em outras linguagens, avaliação é interrompida quando já se conhece o resultado

A && B
se A é falso...
A || B
se B é verdadeiro...

curto circuito em condições lógicas

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

Avaliando condições com “curto circuito”

- A condição de teste em uma construção de controle pode conter operadores lógicos
  - Exemplo: ((c>a) || ((a ==1) && (d < b)))

- 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 avalia y
  - (x && y) se x resulta em false, não avalia y
- Isto é refletido no código Assembly gerado

Exemplo de curto circuito

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

Exemplo de curto circuito

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



```
  cmp %ebx, %eax
  je L1
  cmp %edx, %ecx
  jge L2
L1:
  movl %ecx, %eax
L2:
  movl %edx, %ecx
```