MICROPROCESSADORES E MICROCONTROLADORES

FUNÇÕES

Funções em C são fundamentais para a criação de código estruturado e para o reuso deste código.

FUNÇÕES

Funções em C são fundamentais para a criação de código estruturado e para o reuso deste código.

Imagine a função como um "espião" que arruma um plano secreto, adquire os recursos necessários, faz a tarefa, apaga seus "rastros" e retorna ao ponto de origem com o resultado desejado.

FUNÇÕES

Uma sub-rotina em Assembly é o equivalente da função em C. A chamada à sub-rotina segue estes passos:

1. Coloque parâmetros aonde a sub-rotina possa acessa-los.
2. Transfira o controle para a sub-rotina.
3. Separe memória para a sub-rotina.
4. Realize a tarefa da sub-rotina.
5. Coloque o resultado em um lugar acessível a quem chamou a sub-rotina.
6. Retorne ao ponto de origem.

FUNÇÕES

Uma sub-rotina em Assembly é o equivalente da função em C. A chamada à sub-rotina segue estes passos:

1. Coloque parâmetros aonde a sub-rotina possa acessa-los.
2. Transfira o controle à sub-rotina.
3. Coloque o resultado em um lugar acessível a quem chamou a sub-rotina.
6. Retorne ao ponto de origem.

Use os registradores R15, R14, R13 e R12 (nesta ordem) para passar os parâmetros à sub-rotina.

FUNÇÕES

Uma sub-rotina em Assembly é o equivalente da função em C. A chamada à sub-rotina segue estes passos:

1. Coloque parâmetros aonde a sub-rotina possa acessa-los.
2. Transfira o controle para a sub-rotina.

Use a instrução *call NomeSubRotina*, que automaticamente salva o endereço de retorno na pilha.

FUNÇÕES

Uma sub-rotina em Assembly é o equivalente da função em C. A chamada à sub-rotina segue estes passos:

1. Coloque parâmetros aonde a sub-rotina possa acessa-los.
2. Transfira o controle para a sub-rotina.
3. Separe memória para a sub-rotina.
6. Retorne ao ponto de origem.

Use os registradores e a memória *sem afetar o funcionamento do resto do programa.*

FUNÇÕES

Uma sub-rotina em Assembly é o equivalente da função em C. A chamada à sub-rotina segue estes passos:

1. Coloque parâmetros aonde a sub-rotina possa acessa-los.
2. Transfira o controle para a sub-rotina.
3. Separe memória para a sub-rotina.

Os registradores R4-R11 podem estar sendo utilizados em outra parte do código, portanto devem ser guardados na pilha antes de serem usados.

vel

FUNÇÕES

Uma sub-rotina em Assembly é o equivalente da função em C. A chamada à sub-rotina segue estes passos:

1. Coloque parâmetros aonde a sub-rotina possa acessa-los
5. Coloque o resultado em um lugar acessível a quem chamou a sub-rotina.
6. Retorne ao ponto de origem.

Use o registrador R15 para retornar valores.

FUNÇÕES

Uma sub-rotina em Assembly é o equivalente da função em C. A chamada à sub-rotina segue estes passos:

1. Coloque parâmetros aonde a sub-rotina possa acessa-los.
2. Transfira o controle para a sub-rotina.
3. Separe memória para a sub-rotina.

Use a instrução *ret* para retornar ao ponto aonde a sub-rotina foi chamada.

6. Retorne ao ponto de origem.

FUNÇÕES

Como a seguinte função em C é compilada para o MSP430?

```
int Func_Exemplo(int g, int h, int i, int j)
{
    return (g+h-i-j);
}
```

FUNÇÕES

Como a seguinte função em C é compilada para o MSP430?

```
int Func_Exemplo(int g, int h, int i, int j)
{
    return (g+h-i-j);
}
```

```
Func_Exemplo:
add.w R14, R15      ; R15 = g, R14 = h
sub.w R13, R15      ; R13 = i
sub.w R12, R15      ; R12 = j
ret                 ; Retorne aonde a função foi chamada
```

FUNÇÕES

Como a seguinte função em C é compilada para o MSP430?

```
int Func_Exemplo(int g, int h, int i, int j)
{
    return (g+h-i-j);
}
```

```
Func_Exemplo:
add.w R14, R15      ; R15 = g, R14 = h
```

Em outra parte do código, atribuiu-se o valor de g para o registrador R15, e h para R14.

ão foi chamada

FUNÇÕES

Como a seguinte função em C é compilada para o MSP430?

```
int Func_Exemplo(int g, int h, int i, int j)
{
    return (g+h-i-j);
}
```

Em outra parte do código, atribuiu-se o valor de i para o registrador R13, e j para R12.

```
sub.w R13, R15      ; R13 = i
sub.w R12, R15      ; R12 = j
ret                 ; Retorne aonde a função foi chamada
```

FUNÇÕES

Como a seguinte função em C é compilada para o MSP430?

```
int Func_Exemplo(int g, int h, int i, int j)
{
    return (g+h-i-j);
}
```

```
Func_Exemplo:
```

Retorno para a outra parte do código que chamou a função.

```
ret                ; Retorne aonde a função foi chamada
```

FUNÇÕES

Como a seguinte função em C é compilada para o MSP430?

```
int Func_Exemplo(int g, int h, int i, int j)
{
    return (g+h-i-j);
}
```

```
Func_Exemplo:
ret                 ; Retorne aonde a função foi chamada
```

Na outra parte do código que chamou a função, o resultado deverá ser lido em R15.

FUNÇÕES

Como a seguinte função em C é compilada para o MSP430?

```
int Func_Exemplo(int g, int h, int i, int j)
{
    return (g+h-i-j);
}
```

O endereço de retorno foi guardado na pilha quando foi feita a chamada *call Func_Exemplo*

```
ret                 ; Retorne aonde a função foi chamada
```

Como a seguinte função em C é compilada para o MSP430?

```
int Func_Exemplo(int g, int h, int i, int j)
{
    int f = (g+h) - (i+j);
    return f;
}
```

Como a seguinte função em C é compilada para o MSP430?

```
int Func_Exemplo(int g, int h, int i, int j)
{
    int f = (g+h) - (i+j);
    return f;
}
```

```
Func_Exemplo:
push.w R4          ; Guarde R4 na pilha
push.w R5          ; Guarde R5 na pilha
mov.w R15, R4      ; R4 = g
add.w R14, R4      ; R4 = g+h
mov.w R13, R5      ; R5 = i
add.w R12, R5      ; R5 = i+j
sub.w R5, R4       ; R4 = f = (g+h) - (i+j)
mov.w R4, R15      ; Retorne f por R15
pop.w R5           ; Recupere R5 na pilha
pop.w R4           ; Recupere R4 na pilha
ret
```

Como a seguinte função em C é compilada para o MSP430?

```
int Func_Exemplo(int g, int h, int i, int j)
{
    int f = (g+h) - (i+j);
    return f;
}
```

```
Func_Exemplo:
push.w R4          ; Guarde R4 na pilha
push.w R5          ; Guarde R5 na pilha
pop.w R4           ; Recupere R4 na pilha
ret
```

Foi definida uma variável *f* na função, e teremos também cálculos intermediários para fazer (*g+h* e *i+j*).

Deve-se separar dois registradores para isso (no caso, R4 e R5).

Como a seguinte função em C é compilada para o MSP430?

```
int Func_Exemplo(int g, int h, int i, int j)
{
    int f = (g+h) - (i+j);
    return f;
}
```

```
Func_Exemplo:
push.w R4          ; Guarde R4 na pilha
push.w R5          ; Guarde R5 na pilha
                   ; Retorne f por R15
pop.w R5           ; Recupere R5 na pilha
pop.w R4           ; Recupere R4 na pilha
ret
```

Como R4 e R5 podem estar sendo usados em outra parte do código, deve-se guardar seus valores atuais na pilha.

Como a seguinte função em C é compilada para o MSP430?

```
int Func_Exemplo(int g, int h, int i, int j)
{
    int f = (g+h) - (i+j);
    return f;
}
```

```
Func_Exemplo:
push.w R4          ; Guarde R4 na pilha
push.w R5          ; Guarde R5 na pilha
```

A pilha é uma região especial da memória previamente separada.

A última posição da pilha é apontada pelo registrador $sp (*stack pointer*).

A pilha é preenchida do topo para baixo.

Como a seguinte função em C é compilada para o MSP430?

```
int Func_Exemplo(int g, int h, int i, int j)
{
    int f = (g+h) - (i+j);
    return f;
}
```

```
Func_Exemplo:
push.w R4          ; Guarde R4 na pilha
push.w R5          ; Guarde R5 na pilha
```

Depois de usar R4 e R5, deve-se recuperar da pilha seus valores antigos.

O *stack pointer* é automaticamente atualizado pelas instruções *push* e *pop*.

```
pop.w R5           ; Recupere R5 na pilha
pop.w R4           ; Recupere R4 na pilha
ret
```

Como a seguinte função em C é compilada para o MSP430?

```
int Func_Exemplo(int g, int h, int i, int j)
{
    int f = (g+h) - (i+j);
    return f;
}
```

```
Func_Exemplo:
push.w R4                ; Guarde R4 na pilha
push.w R5
mov.w R15, R4
add.w R14, R4
mov.w R13, R5
add.w R12, R5
sub.w R5, R4
mov.w R4, R15
pop.w R5
pop.w R4                 ; Recupere R4 na pilha
ret
```

O resto do código segue a mesma lógica do exemplo anterior.

FUNÇÕES

Como fazer multiplicações em Assembly com o MSP430?

Não existem instruções de multiplicação e de divisão na sua CPU.

Alguns modelos possuem periféricos específicos (*Hardware Multiplier*).

Outra opção é criar uma subrotina para isso.

FUNÇÕES

```
int MULT_unsigned(unsigned int a, unsigned int b)
{
   if(b==0) return 0;
   else
      return a+MULT_unsigned(a, b-1));
}
```

```
int MULT_unsigned(unsigned int a, unsigned int b)
{
   if(b==0) return 0;
   else
      return a+MULT_unsigned(a, b-1));
}
```

```
MULT_unsigned:
tst.w R14              ; b==0 ?
jnz MULT_unsigned_else    ; Se b não é zero, vá para o else
clr.w R15              ; return 0
ret
MULT_unsigned_else:
push.w R15             ; Guarde a na pilha
dec.w R14              ; b--
call MULT_unsigned     ; Calcule a*(b-1)
pop.w R14              ; Recupere a na pilha
add.w R14, R15         ; return a + a*(b-1)
ret
```

```
int MULT_unsigned(unsigned int a, unsigned int b)
{
   if(b==0) return 0;
   else
      return a+MULT_unsigned(a, b-1));
}
```

```
MULT_unsigned:
tst.w R14                ; b==0 ?
jnz MULT_unsigned_else   ; Se b não é zero, vá para o else
clr.w R15                ; return 0
ret

                         ; Se b==0, retorne o valor 0.

                         a na pilha

                         ; Calcule a*(b-1)
pop.w R14                ; Recupere a na pilha
add.w R14, R15           ; return a + a*(b-1)
ret
```

```
int MULT_unsigned(unsigned int a, unsigned int b)
{
   if(b==0) return 0;
   else
      return a+MULT_unsigned(a, b-1));
}
```

MULT_unsigned:

Se *b>0*, guarde o valor de R15 (a variável *a*) na pilha, pois vamos chamar *MULT_unsigned( )* novamente, sobrescrevendo R15.

```
MULT_unsigned_else:
push.w R15          ; Guarde a na pilha
dec.w R14           ; b--
call MULT_unsigned  ; Calcule a*(b-1)
pop.w R14           ; Recupere a na pilha
add.w R14, R15      ; return a + a*(b-1)
ret
```

```
int MULT_unsigned(unsigned int a, unsigned int b)
{
   if(b==0) return 0;
   else
      return a+MULT_unsigned(a, b-1));
}
```

```
MULT_unsigned:
tst.w R14              ; b==0 ?
                                                    else
```

A instrução *push.w R15* não altera o valor de R15. Vamos decrementar R14 para chamar *MULT_unsigned(a, b-1)*

```
dec.w R14              ; b--
call MULT_unsigned     ; Calcule a*(b-1)
pop.w R14              ; Recupere a na pilha
add.w R14, R15         ; return a + a*(b-1)
ret
```

```
int MULT_unsigned(unsigned int a, unsigned int b)
{
   if(b==0) return 0;
   else
      return a+MULT_unsigned(a, b-1));
}
```

```
MULT_unsigned:
tst.w R14              ; b==0 ?
```

a o else

R15 agora guarda o resultado de *MULT_unsigned(a, b-1)*

Vamos então somar a R15 a variável *a*, que guardamos na pilha, e retornar da subrotina.

```
pop.w R14              ; Recupere a na pilha
add.w R14, R15         ; return a + a*(b-1)
ret
```

```
int MULT_signed(int a, int b)
{
    if(b<0){
        a = -a;
        b = -b;
    }
    return MULT_unsigned(a,b);
}
```

```
int MULT_signed(int a, int b)
{
    if(b<0){
        a = -a;
        b = -b;
    }
    return MULT_unsigned(a,b);
}
```

```
MULT_signed:
tst.w R14                   ; b<0 ?
jge MULT_signed_else        ; Se b>=0, não é necessário trocar sinais
inv.w R15                   ; a = -a usando complemento de 2
inc.w R15
inv.w R14                   ; b = -b usando complemento de 2
inc.w R14
MULT_signed_else:
call MULT_unsigned
ret                     ;  Retorne da subrotina
```

```
int MULT_signed(int a, int b)
{
    if(b<0){
        a = -a;
        b = -b;
    }
    return MULT_unsigned(a,b);
}
```

MULT_signed:

```
tst.w R14          ; b<0 ?
jge MULT_signed_else ; Se b>=0, não é necessário trocar sinais
inv.w R15          ; a = -a usando complemento de 2
inc.w R15
inv.w R14          ; b = -b usando complemento de 2
inc.w R14
```

Se *b<0*, inverta os sinais de *a* e *b* usando complemento de dois

subrotina

```
int MULT_signed(int a, int b)
{
    if(b<0){
        a = -a;
        b = -b;
    }
    return MULT_unsigned(a,b);
}
```

MULT_signed:

```
tst.w R14                   ; b<0 ?
jge MULT_signed_else        ; Se b>=0, não é necessário trocar sinais
inv.w R15                   ; a = -a usando complemento de 2
```

Reaproveite a subrotina criada anteriormente

ando complemento de 2

```
MULT_signed_else:
call MULT_unsigned
ret                         ; Retorne da subrotina
```

FUNÇÕES

Como a seguinte função em C é compilada para o MSP430?

```
int Fatorial(int n)
{
    if(n<2) return 1;
    else return n*Fatorial(n-1);
}
```

```
int Fatorial(int n)
{
    if(n<2) return 1;
    else return n*Fatorial(n-1);
}
```

```
Fatorial:
cmp #2, R15          ;Testar se n<2
jge L1               ; Se N>=2, pule para L1
mov.w #1, R15        ; return 1;
ret                  ; Retorne para onde a função foi chamada
L1: push.w R15       ; Guarde n (R15) na pilha
dec.w R15            ; Calcule n-1 para
call Fatorial        ; chamar Fatorial(n-1)
pop.w R14            ; Recupere n na pilha
call MULT_signed     ; Calcular n*Fatorial(n-1)
ret                  ; Retorne para onde a função foi chamada
```

```
int Fatorial(int n)
{
    if(n<2) return 1;
    else return n*Fatorial(n-1);
```

A função *Fatorial( )* usa a mesma lógica da função *MULT_unsigned( )*: a recursividade de funções.

Ela também faz uma chamada a a outra subrotina, assim como a subrotina *MULT_signed*

```
Fatorial:
cmp #2, R15
jge L1
mov.w #1, R
ret                                  chamada
L1: push.w R
dec.w R15
call Fatorial          ; chamar Fatorial(n-1)
pop.w R14              ; Recupere n na pilha
call MULT_signed       ; Calcular n*Fatorial(n-1)
ret                    ; Retorne para onde a função foi chamada
```

OPERAÇÕES COM BYTES

Como a seguinte função em C é compilada para o MSP430?

```
int strcpy(char x[ ], char y[ ])
{
    int i = 0;
    while((x[i]=y[i]) != '\0') i++;
}
```

```
strcpy: push.w R4      ; R4 = i
push.w R5              ; R5 será temporário
push.w R6              ; R6 será temporário
clr.w R4               ; R4 = i = 0
strcpy_while: mov.w R14, R5  ; R5 = y (endereço do vetor y[ ])
add.w R4, R5                 ; R5 = y+i (endereço de y[i])
mov.w R15, R6                ; R6 = x (endereço do vetor x[ ])
add.w R4, R6                 ; R6 = x+i (endereço de x[i])
mov.b 0(R5), 0(R6)           ; x[i] = y[i]
tst.w 0(R6)                  ; x[i] == 0? ('\0' vale 0 em ASCII)
jz strcpy_end                ; Se for, saia do while( )
inc.w R4                     ; i++;
jmp strcpy_while             ;Volte para o começo do while( )
strcpy_end: pop.w R6     ; Fim da função: recuperar R6 na pilha
pop.w R5                 ; Fim da função: recuperar R5 na pilha
pop.w R4                 ; Fim da função: recuperar R4 na pilha
ret                      ; Retornar
```

```
int strcpy(char x[ ], char y[ ])
{
    int i = 0;
    while((x[i]=y[i]) != '\0') i++;
}
```

```
strcpy: push.w R4       ; R4 = i
push.w R5               ; R5 será temporário
push.w R6               ; R6 será temporário
```

Guardar R4, R5 e R6 na pilha.

R4 será a variável *i*

R5 e R6 serão variáveis temporárias.

```
int strcpy(char x[ ], char y[ ])
{
    int i = 0;
    while((x[i]=y[i]) != '\0') i++;
}
```

```
                                ereço do vetor y[ ])
                                ndereço de y[i])
                                ereço do vetor x[ ])
                                ndereço de x[i])

                                \0' vale 0 em ASCII)
jz strcpy_end                   ; Se for, saia do while( )
inc.w R4                        ; i++;
jmp strcpy_while                ;Volte para o começo do while( )
strcpy_end: pop.w R6    ; Fim da função: recuperar R6 na pilha
pop.w R5                ; Fim da função: recuperar R5 na pilha
pop.w R4                ; Fim da função: recuperar R4 na pilha
ret                     ; Retornar
```

```
strcpy: push.w R4       ; R4 = i
push.w R5               ; R5 será temporário
push.w R6               ; R6 será temporário
clr.w R4                ; R4 = i = 0
                    R14, R5  ; R5 = y (endereço do vetor y[ ])
                             ; R5 = y+i (endereço de y[i])
                             ; R6 = x (endereço do vetor x[ ])
add.w R4, R6                 ; R6 = x+i (endereço de x[i])
mov.b 0(R5), 0(R6)           ; x[i] = y[i]
tst.w 0(R6)                  ; x[i] == 0? ('\0' vale 0 em ASCII)
jz strcpy_end                ; Se for, saia do while( )
inc.w R4                     ; i++;
jmp strcpy_while             ;Volte para o começo do while( )
strcpy_end: pop.w R6    ; Fim da função: recuperar R6 na pilha
pop.w R5                ; Fim da função: recuperar R5 na pilha
pop.w R4                ; Fim da função: recuperar R4 na pilha
ret                     ; Retornar
```

```
int strcpy(char x[ ], char y[ ])
{
    int i = 0;
    while((x[i]=y[i]) != '\0') i++;
}
```

***i* = 0**

```
strcpy: push.w R4      ; R4 = i
push.w R5              ; R5 será temporário
push.w R6              ; R6 será temporário
clr.w R4               ; R4 = i = 0
strcpy_while: mov.w R14, R5  ; R5 = y (endereço do vetor y[ ])
add.w R4, R5                 ; R5 = y+i (endereço de y[i])
                             ; ...dereço do vetor x[ ])
                             ; ...ndereço de x[i])
                             ; '\0' vale 0 em ASCII)
                             ; ...do while( )
jmp strcpy_while             ; Volte para o começo do while( )
strcpy_end: pop.w R6   ; Fim da função: recuperar R6 na pilha
pop.w R5               ; Fim da função: recuperar R5 na pilha
pop.w R4               ; Fim da função: recuperar R4 na pilha
ret                    ; Retornar
```

```
int strcpy(char x[ ], char y[ ])
{
    int i = 0;
    while((x[i]=y[i]) != '\0') i++;
}
```

Precisamos ler *y[i]*. Para isso, precisamos pegar o endereço inicial de *y*, que veio pelo registrador R14, e andar *i* casas.

```
strcpy: push.w R4       ; R4 = i
push.w R5               ; R5 será temporário
push.w R6               ; R6 será temporário
clr.w R4                ; R4 = i = 0
strcpy_while: mov.w R14, R5  ; R5 = y (endereço do vetor y[ ])
add.w R4, R5                 ; R5 = y+i (endereço de y[i])
                             ; ...dereço do vetor x[ ])
                             ; ...ndereço de x[i])
                             ; '\0' vale 0 em ASCII)
                             ; ...do while( )
jmp strcpy_while             ; Volte para o começo do while( )
strcpy_end: pop.w R6    ; Fim da função: recuperar R6 na pilha
pop.w R5                ; Fim da função: recuperar R5 na pilha
pop.w R4                ; Fim da função: recuperar R4 na pilha
ret                     ; Retornar
```

```
int strcpy(char x[ ], char y[ ])
{
    int i = 0;
    while((x[i]=y[i]) != '\0') i++;
}
```

Precisamos ler *y[i]*. Para isso, precisamos pegar o endereço inicial de *y*, que veio pelo registrador R14, e andar *i* casas.

```
strcpy: push.w R4      ; R4 = i
push.w R5              ; R5 será temporário
push.w R6              ; R6 será temporário
clr.w R4               ; R4 = i = 0
strcpy_while: mov.w R14, R5  ; R5 = y (endereço do vetor y[ ])
add.w R4, R5                 ; R5 = y+i (endereço de y[i])
                             ; ereço do vetor x[ ])
                             ; ndereço de x[i])
                             ; '\0' vale 0 em ASCII)
                             ; do while( )
jmp strcpy_while             ; Volte para o começo do while( )
strcpy_end: pop.w R6   ; Fim da função: recuperar R6 na pilha
pop.w R5               ; Fim da função: recuperar R5 na pilha
pop.w R4               ; Fim da função: recuperar R4 na pilha
ret                    ; Retornar
```

```
int strcpy(char x[ ], char y[ ])
{
    int i = 0;
    while((x[i]= y[i]) != '\0') i++;
}
```

Precisamos ler *y[i]*. Para isso, precisamos pegar o endereço inicial de *y*, que veio pelo registrador R14, e andar *i* casas.

```
strcpy: push.w R4       ; R4 = i
push.w R5               ; R5 será temporário
push.w R6               ; R6 será temporário
clr.w R4                ; R4 = i = 0
strcpy_while: mov.w R14, R5  ; R5 = y (endereço do vetor y[ ])
add.w R4, R5            ; R5 = y+i (endereço de y[i])
mov.w R15, R6           ; R6 = x (endereço do vetor x[ ])
add.w R4, R6            ; R6 = x+i (endereço de x[i])
                        ; x[i] = y[i]
                        ; x[i] == 0? ('\0' vale 0 em ASCII)
                        ; Se for, saia do while( )
                        ; i++;
jmp strcpy_while        ;Volte para o começo do while( )
strcpy_end: pop.w R6    ; Fim da função: recuperar R6 na pilha
pop.w R5                ; Fim da função: recuperar R5 na pilha
pop.w R4                ; Fim da função: recuperar R4 na pilha
ret                     ; Retornar
```

```
int strcpy(char x[ ], char y[ ])
{
    int i = 0;
    while((x[i]=y[i]) != '\0') i++;
}
```

Precisamos escrever em *x[i]*.
O raciocínio é o mesmo.

```
strcpy: push.w R4      ; R4 = i
push.w R5              ; R5 será temporário
push.w R6              ; R6 será temporário
clr.w R4               ; R4 = i = 0
strcpy_while: mov.w R14, R5  ; R5 = y (endereço do vetor y[ ])
add.w R4, R5                 ; R5 = y+i (endereço de y[i])
mov.w R15, R6                ; R6 = x (endereço do vetor x[ ])
add.w R4, R6                 ; R6 = x+i (endereço de x[i])
mov.b 0(R5), 0(R6)           ; x[i] = y[i]
                             ] == 0? ('\0' vale 0 em ASCII)
                               for, saia do while( )
                             -;
                             te para o começo do while( )
                               função: recuperar R6 na pilha
                               função: recuperar R5 na pilha
                               função: recuperar R4 na pilha
                             ar
```

```
int strcpy(char x[ ], char y[ ])
{
    int i = 0;
    while((x[i]=y[i]) != '\0') i++;
}
```

R5 contém o endereço de *y[i]*

O operador 0(R5) quer dizer "*use como endereço o valor em R5 somado de 0 bytes*"

O mesmo vale para 0(R6)

```
strcpy: push.w R4      ; R4 = i
push.w R5              ; R5 será temporário
push.w R6              ; R6 será temporário
clr.w R4               ; R4 = i = 0
strcpy_while: mov.w R14, R5  ; R5 = y (endereço do vetor y[ ])
add.w R4, R5                 ; R5 = y+i (endereço de y[i])
mov.w R15, R6                ; R6 = x (endereço do vetor x[ ])
add.w R4, R6                 ; R6 = x+i (endereço de x[i])
mov.b 0(R5), 0(R6)           ; x[i] = y[i]
                             0' vale 0 em ASCII)
                             o while( )

                             começo do while( )
                             cuperar R6 na pilha
pop.w R5               ; Fim da função: recuperar R5 na pilha
pop.w R4               ; Fim da função: recuperar R4 na pilha
ret                    ; Retornar
```

```
int strcpy(char x[ ], char y[ ])
{
    int i = 0;
    while((x[i]=y[i]) != '\0') i++;
}
```

Ou seja, *mov.b 0(R5), 0(R6)* quer dizer "*Busque o* byte *no endereço R5+0, e guarde-o no endereço R6+0*"

```
strcpy: push.w R4      ; R4 = i
push.w R5              ; R5 será temporário
push.w R6              ; R6 será temporário
clr.w R4               ; R4 = i = 0
strcpy_while: mov.w R14, R5  ; R5 = y (endereço do vetor y[ ])
add.w R4, R5                 ; R5 = y+i (endereço de y[i])
mov.w R15, R6                ; R6 = x (endereço do vetor x[ ])
add.w R4, R6                 ; R6 = x+i (endereço de x[i])
mov.b 0(R5), 0(R6)           ; x[i] = y[i]
                                0' vale 0 em ASCII)
                             o while( )

                             começo do while( )
                             cuperar R6 na pilha
pop.w R5               ; Fim da função: recuperar R5 na pilha
pop.w R4               ; Fim da função: recuperar R4 na pilha
ret                    ; Retornar
```

```
int strcpy(char x[ ], char y[ ])
{
    int i = 0;
    while((x[i]=y[i]) != '\0') i++;
}
```

Aqui, usamos *mov.b* ao invés de *mov.w* porque o vetor é do tipo char (1 byte)

```
strcpy: push.w R4       ; R4 = i
push.w R5               ; R5 será temporário
push.w R6               ; R6 será temporário
clr.w R4                ; R4 = i = 0
strcpy_while: mov.w R14, R5  ; R5 = y (endereço do vetor y[ ])
add.w R4, R5                 ; R5 = y+i (endereço de y[i])
mov.w R15, R6                ; R6 = x (endereço do vetor x[ ])
add.w R4, R6                 ; R6 = x+i (endereço de x[i])
mov.b 0(R5), 0(R6)           ; x[i] = y[i]
tst.w 0(R6)                  ; x[i] == 0? ('\0' vale 0 em ASCII)
jz strcpy_end                ; Se for, saia do while( )

                                ra o começo do while( )
                             ão: recuperar R6 na pilha
                             ão: recuperar R5 na pilha
                             ão: recuperar R4 na pilha
ret                          ; Retornar
```

```
int strcpy(char x[ ], char y[ ])
{
    int i = 0;
    while((x[i]=y[i]) != '\0') i++;
}
```

Se o valor copiado de *y[i]* para *x[i]* for igual a 0, é porque encontramos um caracter '\0' (0 em ASCII), e podemos sair do while( )

```
strcpy: push.w R4      ; R4 = i
push.w R5              ; R5 será temporário
push.w R6              ; R6 será temporário
clr.w R4               ; R4 = i = 0
strcpy_while: mov.w R14, R5  ; R5 = y (endereço do vetor y[ ])
add.w R4, R5                 ; R5 = y+i (endereço de y[i])
                             (endereço do vetor x[ ])
                             +i (endereço de x[i])
                             y[i]
                             0? ('\0' vale 0 em ASCII)
                             saia do while( )
inc.w R4                     ; i++;
jmp strcpy_while             ;Volte para o começo do while( )
strcpy_end: pop.w R6   ; Fim da função: recuperar R6 na pilha
pop.w R5               ; Fim da função: recuperar R5 na pilha
pop.w R4               ; Fim da função: recuperar R4 na pilha
ret                    ; Retornar
```

```
int strcpy(char x[ ], char y[ ])
{
    int i = 0;
    while((x[i]=y[i]) != '\0') i++;
}
```

Se não encontramos um caracter '\0', incrementamos *i* em um byte (porque o vetor é do tipo char) e voltamos para o início do while( )

```
strcpy: push.w R4      ; R4 = i
push.w R5              ; R5 será temporário
push.w R6              ; R6 será temporário
clr.w R4               ; R4 = i = 0
strcpy_while: mov.w R14, R5  ; R5 = y (endereço do vetor y[ ])
add.w R4, R5                 ; R5 = y+i (endereço de y[i])
mov.w R15, R6                ; R6 = x (endereço do vetor x[ ])
add.w R4, R6                 ; R6 = x+i (endereço de x[i])
                             = y[i]
                             == 0? ('\0' vale 0 em ASCII)
                             for, saia do while( )
                             ;
                             e para o começo do while( )
strcpy_end: pop.w R6         ; Fim da função: recuperar R6 na pilha
pop.w R5                     ; Fim da função: recuperar R5 na pilha
pop.w R4                     ; Fim da função: recuperar R4 na pilha
ret                          ; Retornar
```

```
int strcpy(char x[ ], char y[ ])
{
    int i = 0;
    while((x[i]=y[i]) != '\0') i++;
}
```

Se encontramos um caracter '\0', recuperamos R4, R5 e R6 da pilha e saímos da subrotina

```
int strcpy(char x[ ], char y[ ])
{
    while((*x)=(*y)) != '\0')
    {
        x++;
        y++;
    }
}
```

```
strcpy: mov.b 0(R14), 0(R15)  ; (*x) = (*y)
tst.w 0(R15)                  ; (*x) == '\0'?
jz strcpy_end                 ; Se for, saia do while( )
inc.w R15                     ; x++
inc.w R14                     ; y++
jmp strcpy                    ;Volte para o começo do while( )
strcpy_end: ret               ; Retornar
```

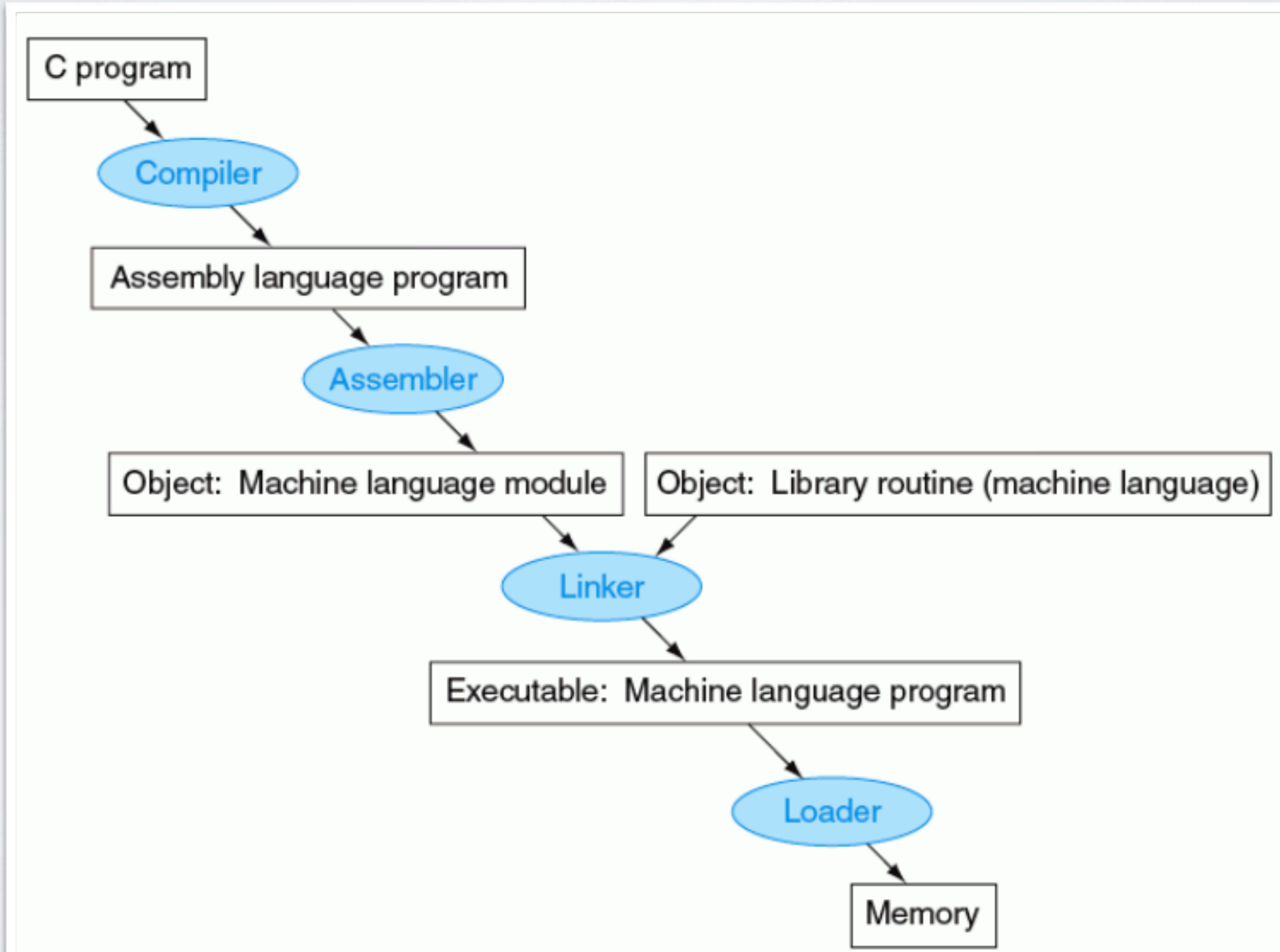
```
int strcpy(char x[ ], char y[ ])
{
    while((*x)=(*y)) != '\0')
    {
        x++;
        y++;
    }
}
```

Se retirarmos a variável *i*, o código fica bem mais simples

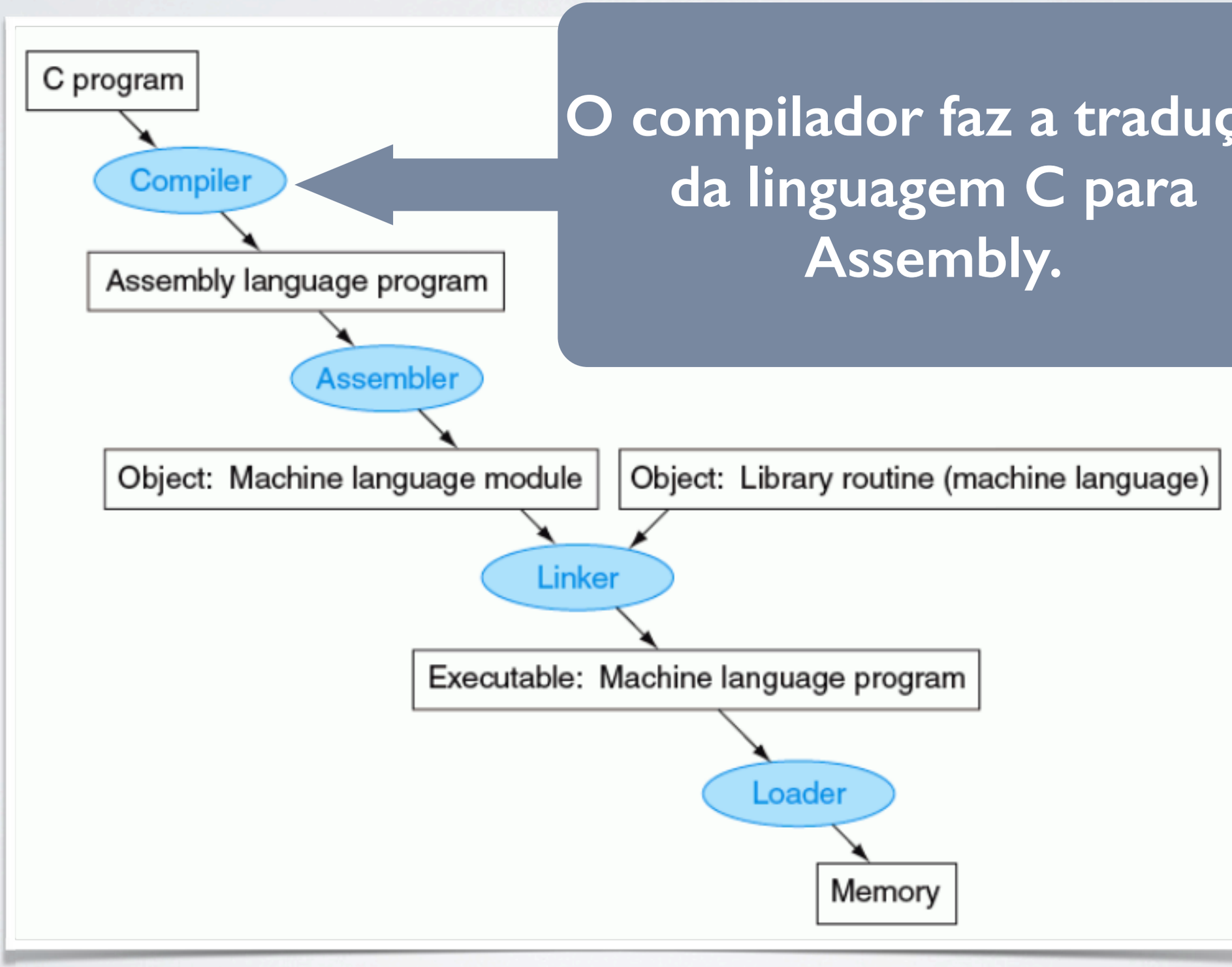
Agora, só incrementamos os ponteiros para os vetores

```
strcpy: mov.b 0(R14), 0(R15)  ; (*x) = (*y)
tst.w 0(R15)                  ; (*x) == '\0'?
jz strcpy_end                 ; Se for, saia do while( )
inc.w R15                     ; x++
inc.w R14                     ; y++
jmp strcpy                    ;Volte para o começo do while( )
strcpy_end: ret               ; Retornar
```

TRADUÇÃO

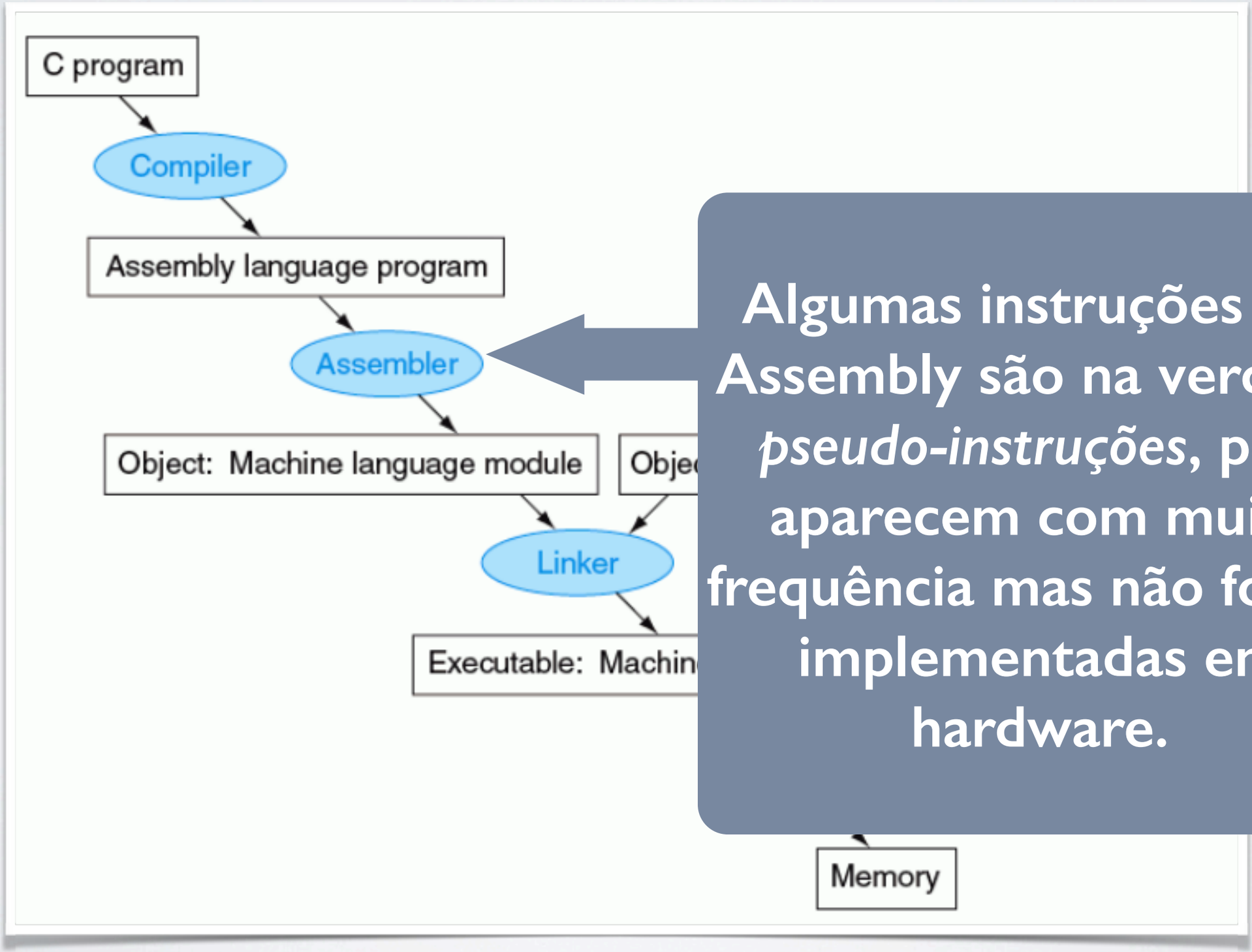

C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

TRADUÇÃO



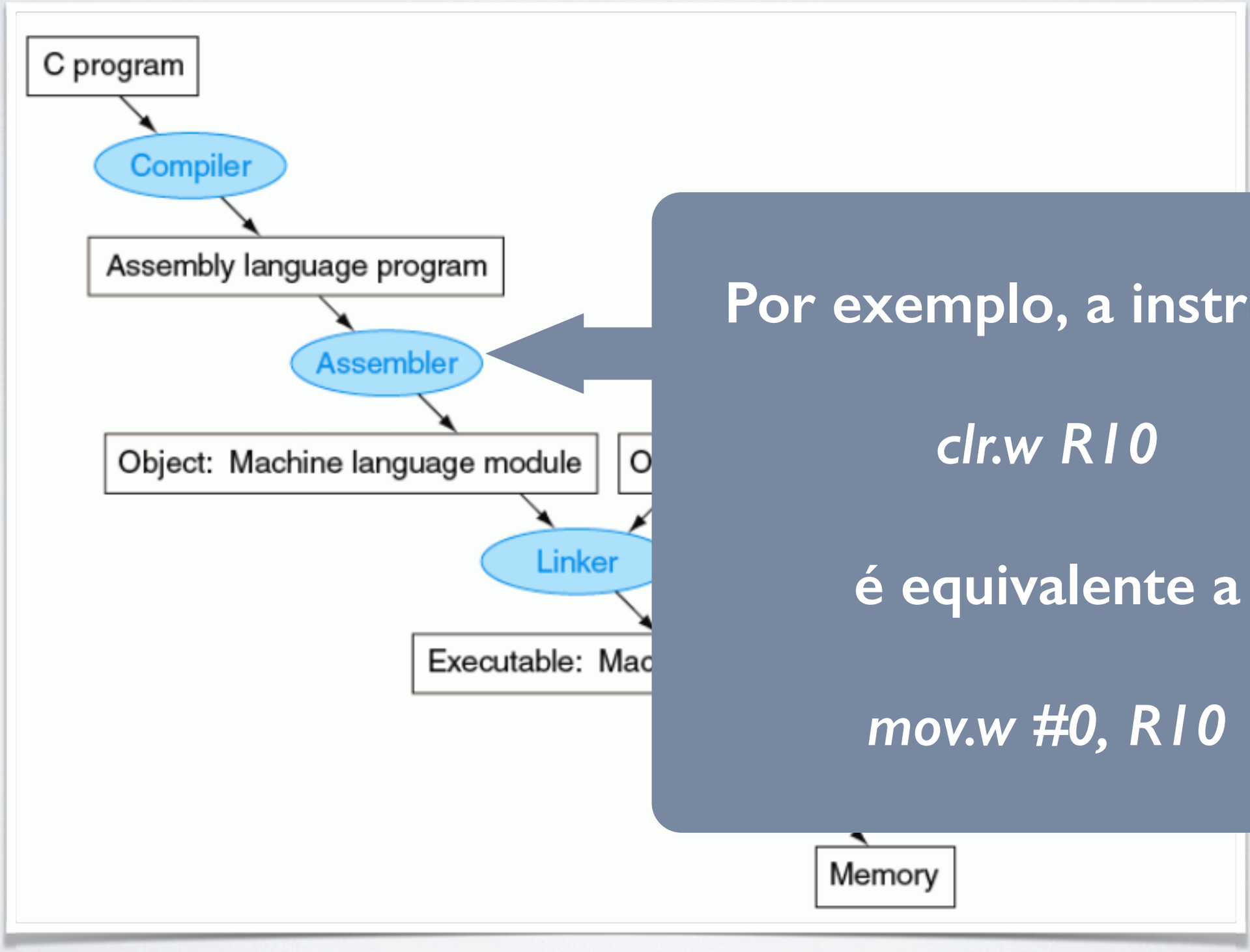
O compilador faz a tradução da linguagem C para Assembly.

TRADUÇÃO



Algumas instruções em Assembly são na verdade *pseudo-instruções*, pois aparecem com muita frequência mas não foram implementadas em hardware.

TRADUÇÃO



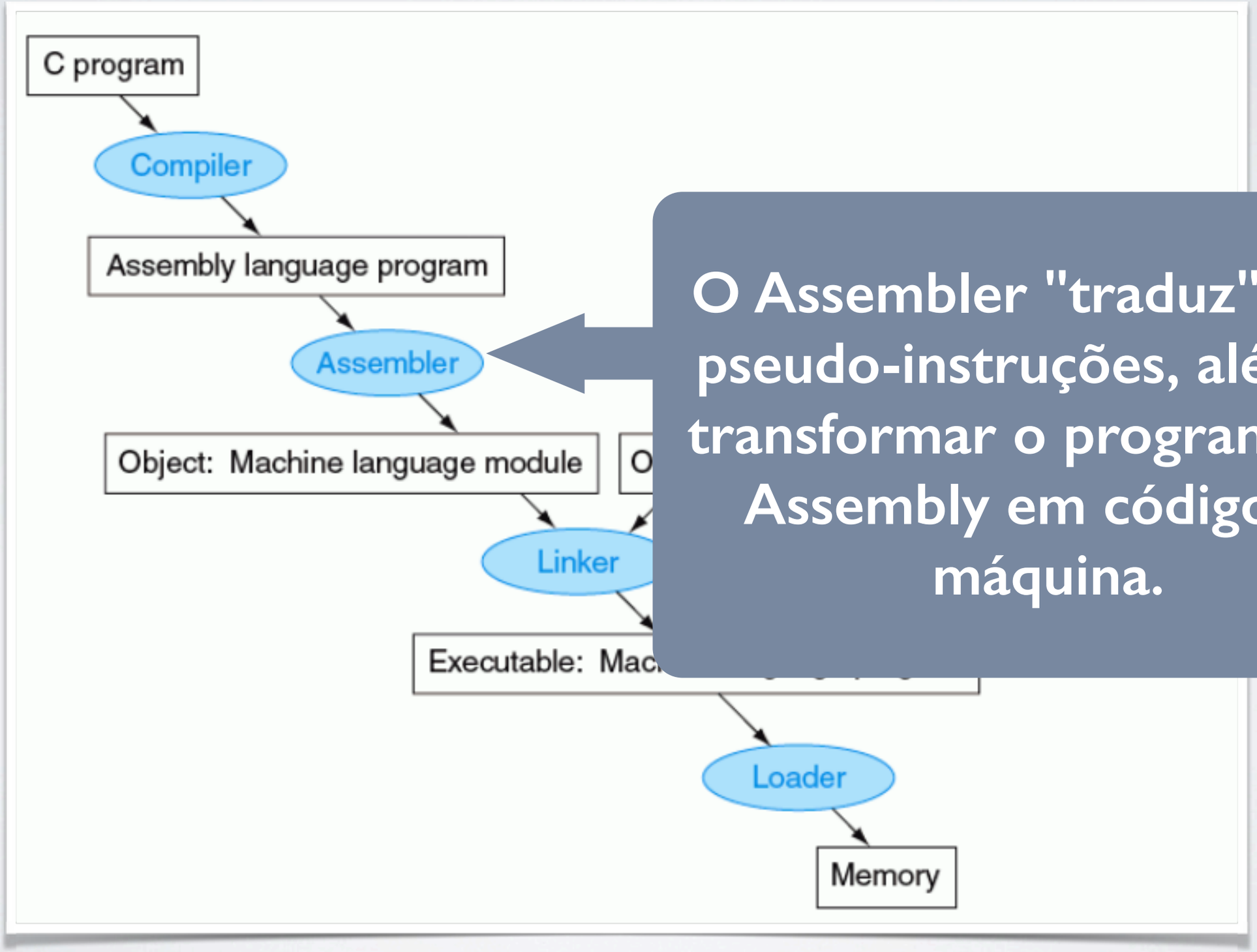
Por exemplo, a instrução

clr.w R10

é equivalente a

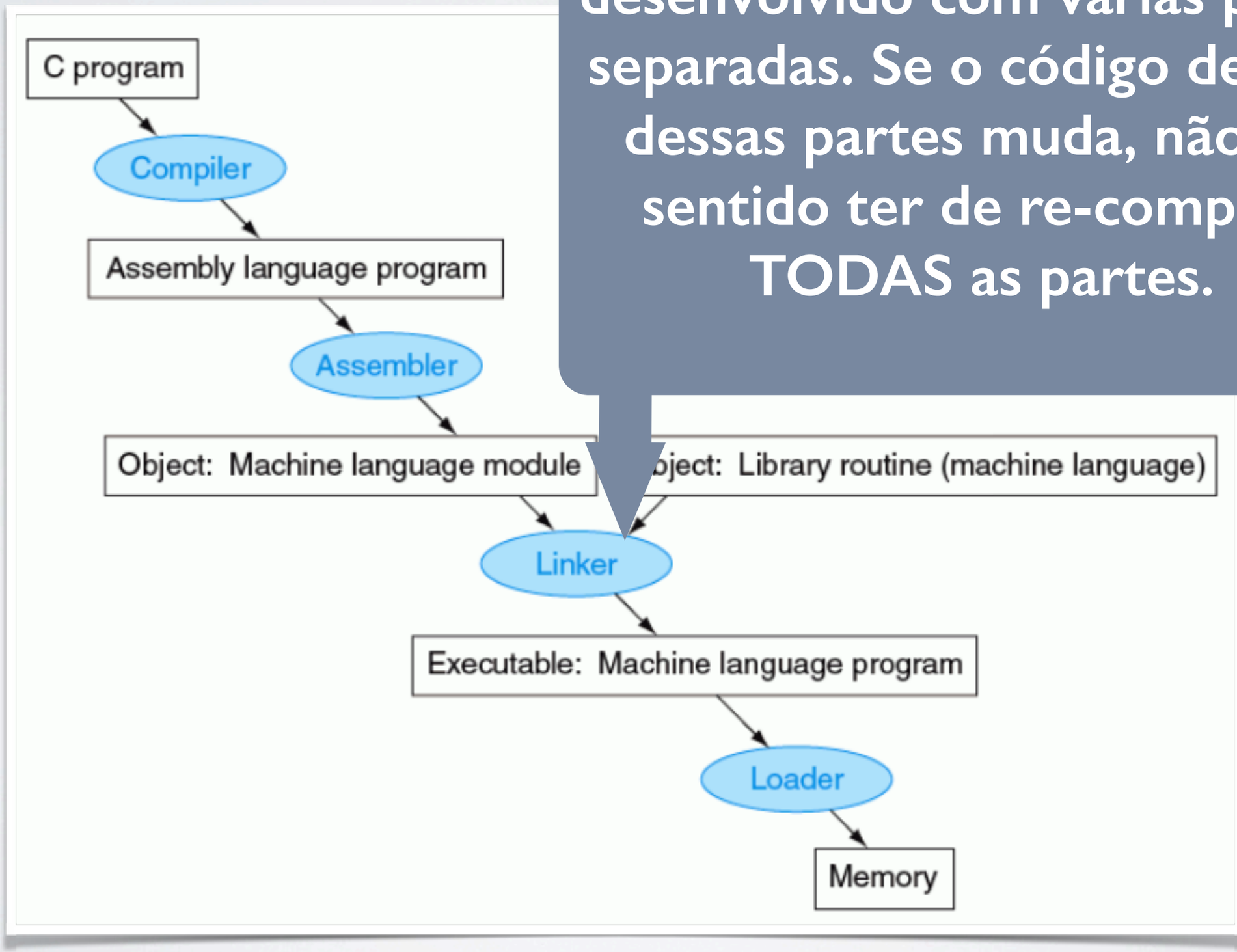
mov.w #0, R10

TRADUÇÃO



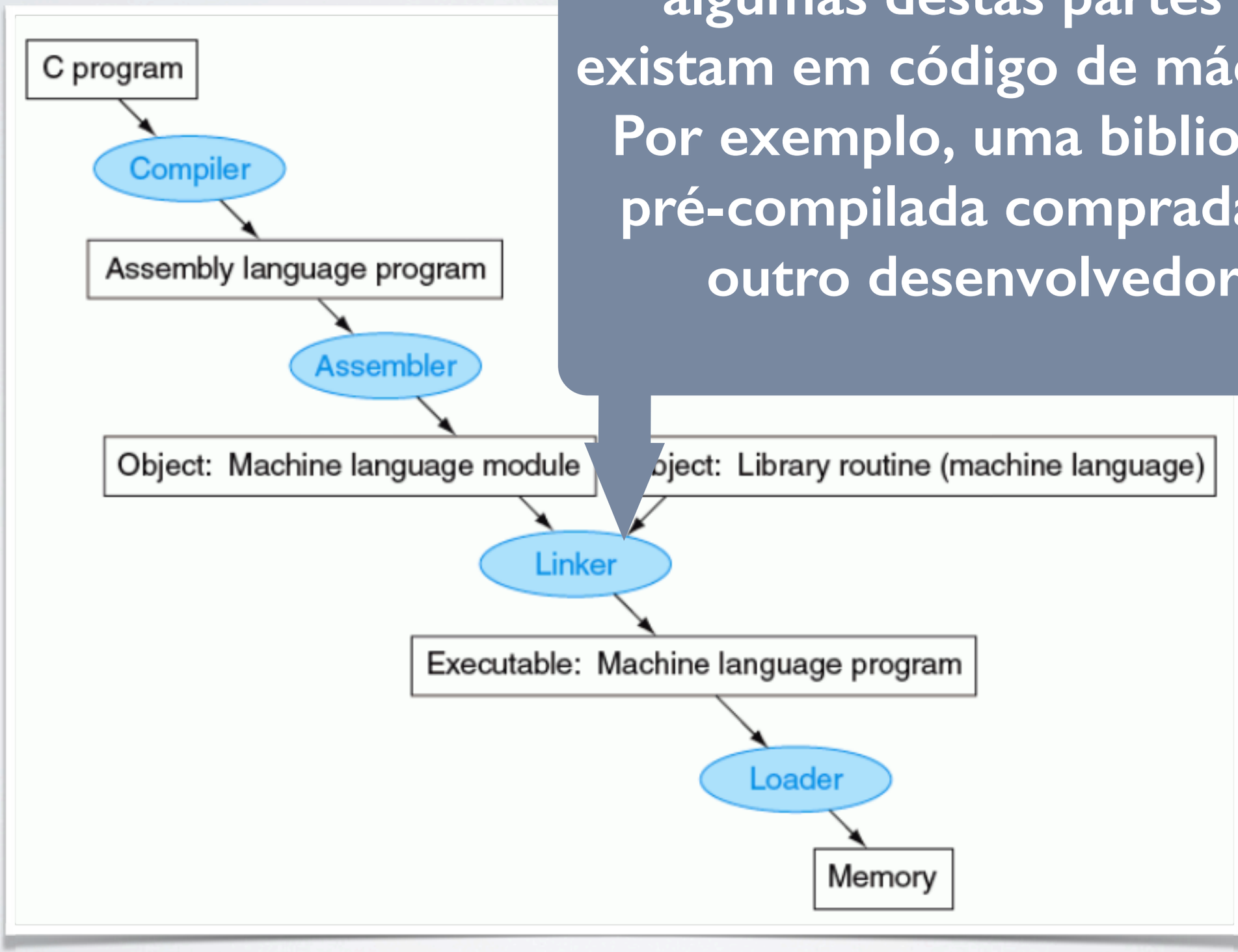
O Assembler "traduz" estas pseudo-instruções, além de transformar o programa em Assembly em código de máquina.

TRAD



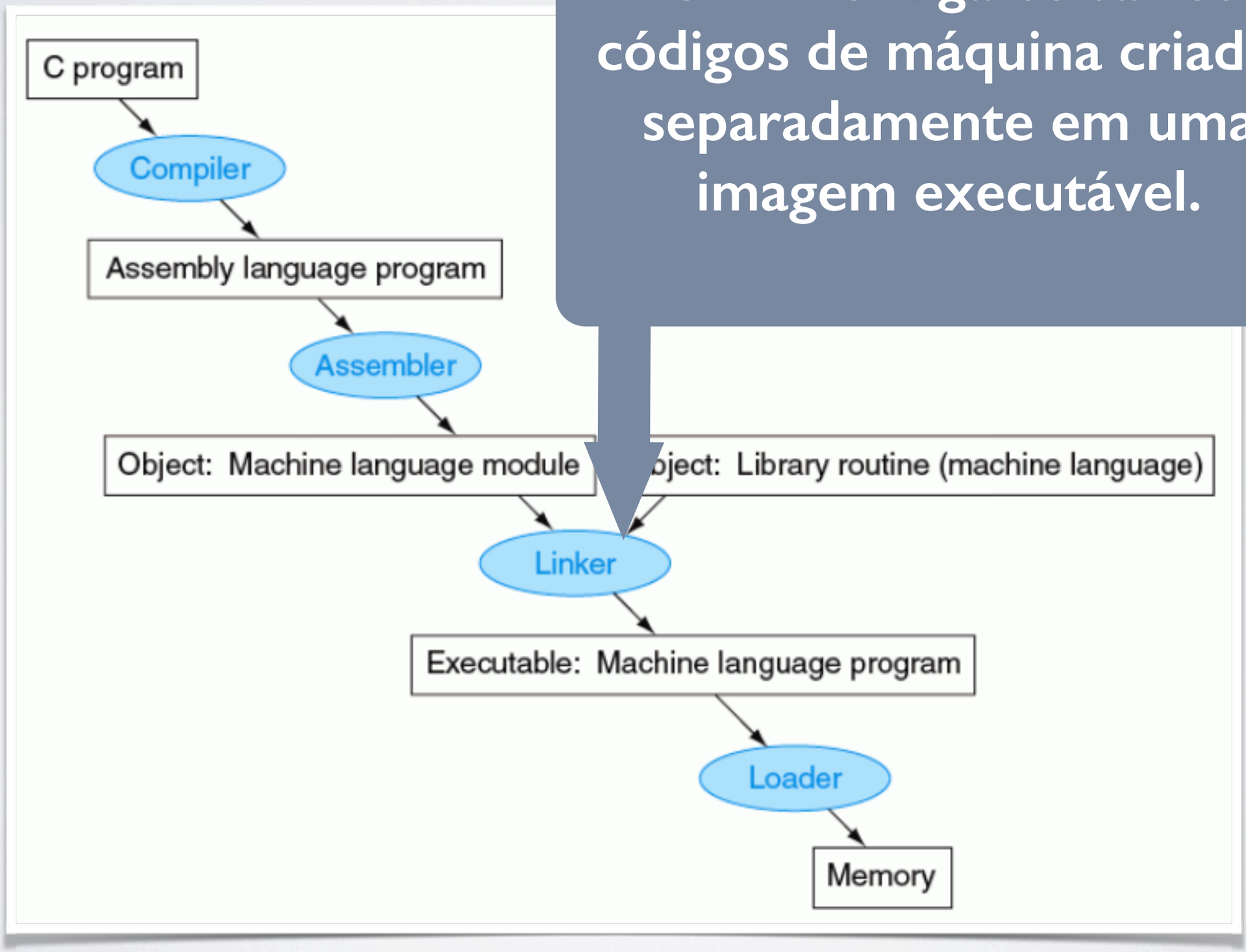
Imagine um programa desenvolvido com várias partes separadas. Se o código de uma dessas partes muda, não faz sentido ter de re-compilar TODAS as partes.

TRAD



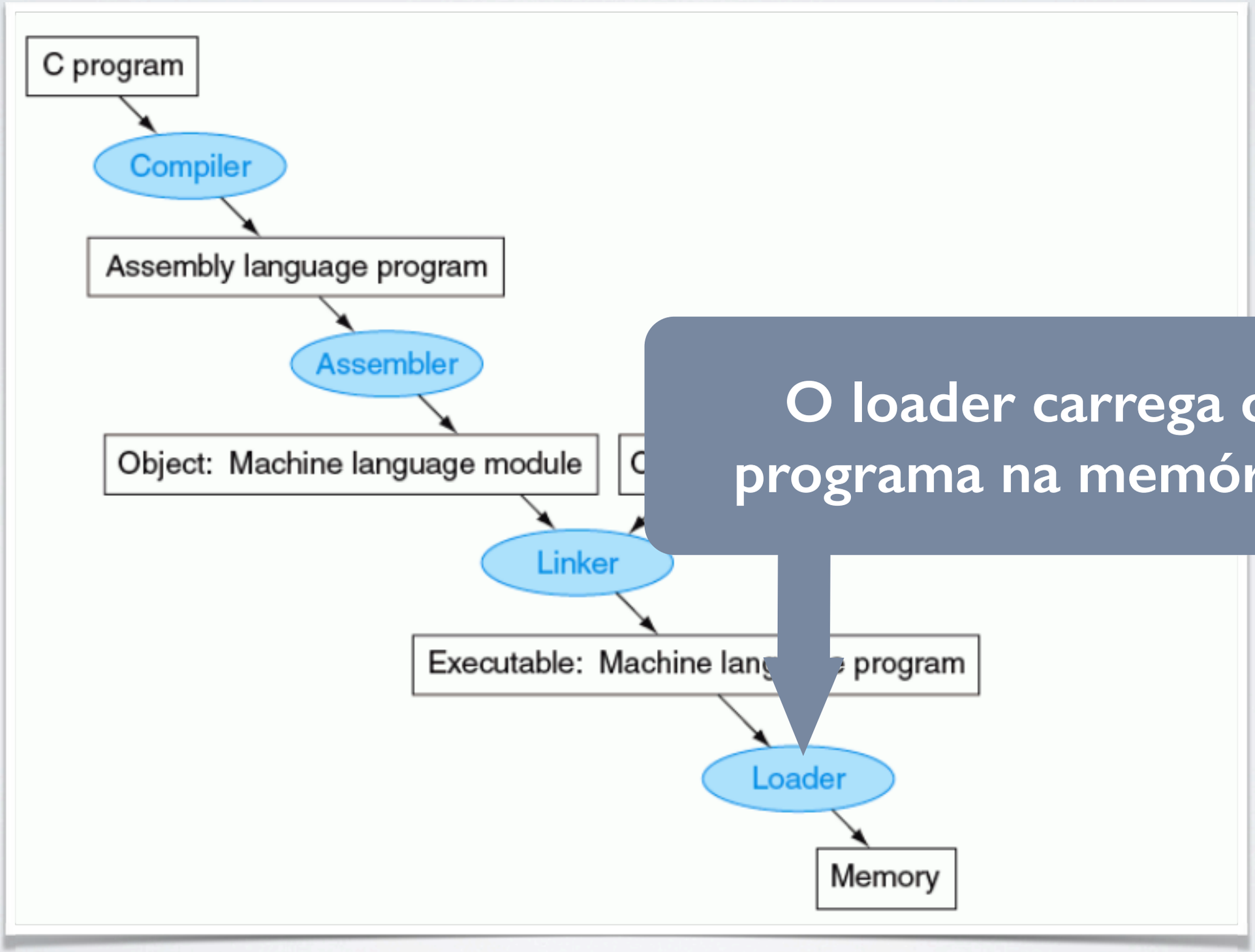
Além disso, é possível que algumas destas partes só existam em código de máquina. Por exemplo, uma biblioteca pré-compilada comprada de outro desenvolvedor.

TRAD



O Linker liga os vários códigos de máquina criados separadamente em uma imagem executável.

TRADUÇÃO


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Linker
Executable: Machine lang program
Loader
Memory
O loader carrega o programa na memória.

EXEMPLOS COMPLETOS

Vamos ver alguns exemplos de códigos para testar no *software* IAR Embedded WorkBench, da empresa IAR Systems.

EXEMPLOS COMPLETOS

Vamos ver alguns exemplos de códigos para testar no *software* IAR Embedded WorkBench, da empresa IAR Systems.

Os *softwares* Code Composer e GCC para MSP430 precisam algumas mudanças para rodar estes exemplos.

```
#include <msp430g2553.h>

RSEG CODE
main:
mov.w #WDTPW|WDTHOLD, &WDTCTL
mov.b #01000001b, P1OUT
mov.b #01000001b, P1DIR
jmp $

 RSEG RESET
 DW main
 END
```

```
#include <msp430g2553.h>

RSEG CODE

                                    VDTCTL

RSEG RESET
DW main
END
```

Define que o que vem a seguir irá na memória de programa.

```
#include <msp430g2553.h>

RSEG CODE
main:
mov.w #WDTPW|WDTHOLD, &WDTCTL
mov.b #01000001b, P1OUT
mov.b #01000001b, P1DIR
```

Desliga o Watchdog Timer
e liga os dois LEDs da Launchpad

```
#include <msp430g2553.h>

RSEG CODE
main:
mov.w #WDTPW|WDTHOLD, &WDTCTL
mov.b #01000001b, P1OUT
mov.b #01000001b, P1DIR
jmp $
END
```

Loop infinito: *jmp $* quer dizer "pule para esta mesma instrução"

```
#include <msp430g2553.h>

RSEG CODE
main:
                                  &WDTCTL
```

Esta seção indica que a subrotina *main* deve ser executada sempre que o sistema iniciar ou reiniciar.

```
RSEG RESET
DW main
END
```

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6

RSEG DATA16_N
   Contador DS 2

RSEG CODE
main: mov.w #WDTPW|
WDTHOLD, &WDTCTL
   clr.b #LEDS, P1OUT
   mov.b #LEDS, P1DIR
   mov.w #10, &Contador
Loop: tst &Contador
   jz Loop_end
   dec.w &Contador
   bis.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso1: dec R15
   jnz Atraso1
   bic.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso2: dec R15
   jnz Atraso2
   jmp Loop
Loop_end:
   bis.b #LEDS, &P1OUT
   jmp $
RSEG RESET
DW main
END
```

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6
RSEG CSTACK
main: mov.w #WDTPW|
WDTHOLD, &WDTCTL
    clr.b #LEDS, P1OUT
    mov.b #LEDS, P1DIR
    mov.w #10, &Contador
Loop: tst &Contador
    jz Loop_end
    dec.w &Contador
    bis.b #LEDS, &P1OUT
    mov.w #0xFFFF, R15
Atraso1: dec R15
    jnz Atraso1
    bic.b #LEDS, &P1OUT
    mov.w #0xFFFF, R15
Atraso2: dec R15
    jnz Atraso2
    jmp Loop
Loop_end:
    bis.b #LEDS, &P1OUT
    jmp $
RSEG RESET
DW main
END
```

Equivalente em C a
#define LEDS BIT0|BIT6

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6

RSEG DATA16_N
   Contador DS 2

RSEG CODE
main: mov.w #WDTPW|WDTHOLD, &WDTCTL
   mov.b #LEDS, P1DIR
   mov.w #10, &Contador
Loop: tst &Contador
   jz Loop_end
   dec.w &Contador
   bis.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso1: dec R15
   jnz Atraso1
   bic.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso2: dec R15
   jnz Atraso2
   jmp Loop
Loop_end:
   bis.b #LEDS, &P1OUT
   jmp $
RSEG RESET
DW main
END
```

Separe 2 bytes na memória de programa para a variável *Contador*

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6

RSEG DATA16_N
   Contador DS 2

RSEG CODE
main: mov.w #WDTPW|
WDTHOLD, &WDTCTL
   clr.b #LEDS, P1OUT
   mov.b #LEDS, P1DIR
   bis.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso1: dec R15
   jnz Atraso1
   bic.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso2: dec R15
   jnz Atraso2
   jmp Loop
Loop_end:
   bis.b #LEDS, &P1OUT
   jmp $
RSEG RESET
DW main
END
```

Desligue o Watchdog Timer e apague os LEDs da Launchpad, ligados aos pinos P1.0 e P1.6

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6

RSEG DATA16_N
   Contador DS 2

RSEG CODE
RESET: mov.w #0x0400, SP
   mov.w #WDTPW|WDTHOLD, &WDTCTL
   bis.b #LEDS, &P1DIR
   mov.w #10, &Contador
Loop: tst &Contador
   jz Loop_end
   dec.w &Contador
   bis.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso1: dec R15
   jnz Atraso1
   bic.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso2: dec R15
   jnz Atraso2
   jmp Loop
Loop_end:
   bis.b #LEDS, &P1OUT
   jmp $
RSEG RESET
DW main
END
```

Faça *Contador = 10*

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6

RSEG DATA16_N
   Contador DS 2

RSEG CODE
main: mov.w #WDTPW|
```

Se *Contador==0*, saia do loop

```
Loop: tst &Contador
   jz Loop_end
   dec.w &Contador
   bis.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso1: dec R15
   jnz Atraso1
   bic.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso2: dec R15
   jnz Atraso2
   jmp Loop
Loop_end:
   bis.b #LEDS, &P1OUT
   jmp $
RSEG RESET
DW main
END
```

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6

RSEG DATA16_N
   Contador DS 2

RSEG CODE
main: mov.w #WDTPW|
WDTHOLD, &WDTCTL
   dinb #LEDS,P1OUT
   dec.w &Contador
```

Se *Contador!=0*, decremente esta variável e prossiga

```
   bis.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso1: dec R15
   jnz Atraso1
   bic.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso2: dec R15
   jnz Atraso2
   jmp Loop
Loop_end:
   bis.b #LEDS, &P1OUT
   jmp $
RSEG RESET
DW main
END
```

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6

RSEG DATA16_N
   Contador DS 2

RSEG CODE
main: mov.w #WDTPW
WDTHOLD, &WDTCTL
   clr.b #LEDS, P1OUT
   mov.b #LEDS, P1DIR
   mov.w #10, &Contador
Loop: tst &Contador
   jz Loop_end
   dec.w &Contador
   bis.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso1: dec R15
   jnz Atraso1
   jmp Loop
Loop_end:
   bis.b #LEDS, &P1OUT
   jmp $
RSEG RESET
DW main
END
```

Acenda os LEDs e espere um tempo (enquanto R15 não for zerado).

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6

RSEG DATA16_N
   Contador DS 2

RSEG CODE
main: mov.w #WDTPW|
WDTHOLD, &WDTC
   clr.b #LEDS, P1OUT
   mov.b #LEDS, P1DIR
   mov.w #10, &Contac
Loop: tst &Contador
   jz Loop_end
   dec.w &Contador
   bis.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso1: dec R15
   jnz Atraso1
   bic.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso2: dec R15
   jnz Atraso2
RSEG RESET
DW main
END
```

Apague os LEDs e espere um tempo (enquanto R15 não for zerado).

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6

RSEG DATA16_N
   Contador DS 2

RSEG CODE
main: mov.w #WDTPW|
WDTHOLD, &WDTCTL
   clr.b #LEDS, P1OUT
   mov.b #LEDS, P1DIR
   mov.w #10, &Contado
Loop: tst &Contador
   jz Loop_end
   dec.w &Contador
   bis.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso1: dec R15
   jnz Atraso1
   bic.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso2: dec R15
   jnz Atraso2
   jmp Loop
   DW main
END
```

Agora que piscamos os LEDs uma vez, volte ao começo do loop

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6

RSEG DATA16_N
   Contador DS 2

RSEG CODE
main: mov.w #WDT
WDTHOLD, &WDT
   clr.b #LEDS, P1OUT
   mov.b #LEDS, P1DIR
   mov.w #10, &Contador
Loop: tst &Contador
   jz Loop_end
   dec.w &Contador
   bis.b #LEDS, &P1OUT
   mov.w #0xFFFF, R15
Atraso1: dec R15
   jnz Atraso1
   bis.b #LEDS, &P1OUT
Loop_end:
   bis.b #LEDS, &P1OUT
   jmp $
RSEG RESET
DW main
END
```

Ao final do loop, acenda os LEDs e entre em um loop infinito

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6
RSEG CSTACK
RSEG CODE
main: mov.w #WDTPW|
WDTHOLD, &WDTCTL
    mov.w #SFE(CSTACK), R1
    clr.b #LEDS, P1OUT
    mov.b #LEDS, P1DIR
    mov.w #10, R15
    call #Pisca
    bis.b #LEDS, P1OUT
    jmp $

Atraso: dec R15
    jnz Atraso
    ret
```

```
Pisca: tst R15
    jz Pisca_end
    dec R15
    push R15
    bis.b #LEDS, &P1OUT
    mov.w #0xFFFF, R15
    call #Atraso
    bis.b #LEDS, &P1OUT
    mov.w #0xFFFF, R15
    call #Atraso
    pop R15
    jmp Pisca
Pisca_end: ret

RSEG RESET
DW main
END
```

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6
RSEG CSTACK
RSEG RESET
    mov.w #SFE(CSTACK), R1
    clr.b #LEDS, P1OUT
    mov.b #LEDS, P1DIR
    mov.w #10, R15
    call #Pisca
    bis.b #LEDS, P1OUT
    jmp $

Atraso: dec R15
    jnz Atraso
    ret

Pisca: tst R15
    jz Pisca_end
    dec R15
    call #Atraso
    bis.b #LEDS, &P1OUT
    mov.w #0xFFFF, R15
    call #Atraso
    pop R15
    jmp Pisca
Pisca_end: ret

RSEG RESET
DW main
END
```

Definir a localização da pilha, e iniciar o *stack pointer* (R15)

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6
RSEG CSTACK
RSEG CODE

RESET: mov.w #0x0280, SP
mov.w #WDTPW|WDTHOLD, &WDTCTL
bis.b #LEDS, &P1DIR
mov.w #10, R15
call #Pisca
bis.b #LEDS, P1OUT
jmp $

Atraso: dec R15
jnz Atraso
ret

Pisca: tst R15
jz Pisca_end
dec R15
push R15
is.b #LEDS, &P1OUT
nov.w #0xFFFF, R15
all #Atraso
is.b #LEDS, &P1OUT
nov.w #0xFFFF, R15
call #Atraso
pop R15
jmp Pisca
Pisca_end: ret

RSEG RESET
DW main
END
```

Foi criada uma subrotina *Pisca*, que pisca os LEDs. O valor em R15 define a quantidade de piscadas

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6
RSEG CSTACK
RSEG CODE
main: mov.w #WDTPW|
WDTHOLD, &WDTCTL
    mov.w #SFE(CSTACK), R1
    clr.b #LEDS, P1OUT
    mov.b #LEDS, P1DIR
Pisca: tst R15
    jz Pisca_end
    dec R15
    push R15
    bis.b #LEDS, &P1OUT
    mov.w #0xFFFF, R15
    call #Atraso
    bis.b #LEDS, &P1OUT
    mov.w #0xFFFF, R15
    call #Atraso
    pop R15
    jmp Pisca
Pisca_end: ret

Atraso: dec R15
    jnz Atraso
    ret

RSEG RESET
DW main
END
```

Foi criada uma subrotina *Atraso*, que gera um atraso de acordo com o valor fornecido em R15

```
#include <msp430g2553.h>
LEDS EQU BIT0|BIT6
RSEG CSTACK
RSEG CODE
main: mov.w #WDTPW|
WDTHOLD, &WDTCTL
    mov.w #SFE(CSTACK), R1
    clr.b #LEDS, P1OUT
    mov.b #LEDS, P1DIR
    mov.w #10, R15
    call #Pisca
    bis.b #LEDS, P1OUT
    jmp $

Atraso: dec R15
    jnz Atraso
    ret
```

```
Pisca: tst R15
    jz Pisca_end
    dec R15
    push R15
    bis.b #LEDS, &P1OUT
    mov.w #0xFFFF, R15
    call #Atraso
    bis.b #LEDS, &P1OUT
    mov.w #0xFFFF, R15
    call #Atraso
    pop R15
    jmp Pisca
Pisca_end: ret
```

Subrotina *Pisca*, que chama a subrotina *Atraso* duas vezes

```
END
```