C com Assembly

C com Assembly

- Existem dois caminhos para se escrever código misto C/Assembly:

  - Método inline

  - Modulos assembly separados
- No método inline, o programa em C contém instruções em assembly.
- Muitos compiladores tais como o gcc permite embarcar instruções assembly através do préfixo **asm.**
- Este método é preferível quando se tem pouca instrução assembly para embarcar.

C com Assembly

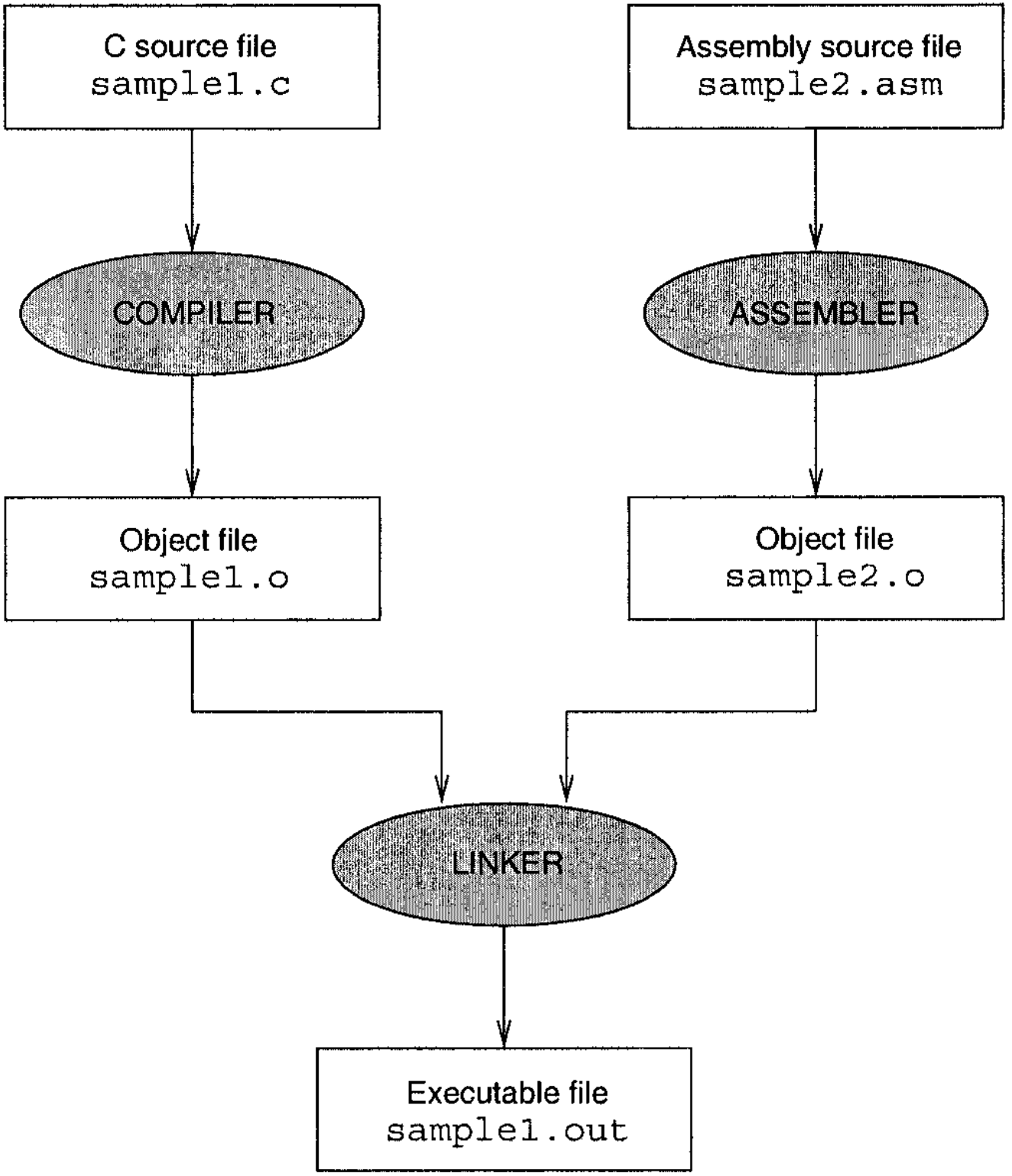
- O método dos módulos separados é utilizado para uma maior quantidade de instruções.
- Utilizamos um compilador C para compilar módulos em C e um montador para montar códigos em Assembly.
- Suponha que o modo misto possui os arquivos sample1.c e sample2.asm.
- Para produzir o arquivo executável:

```
nasm -f elf sample2.asm

gcc -o sample1.out sample1.c sample2.o
```

C com Assembly

- `nasm -f elf sample2.asm` Cria o arquivo sample2.o
- `gcc -o sample1.out sample1.c sample2.o` cria o arquivo sample1.o. O linker é invocado para linkar sample1.o com sample2.o para produzir o executável sample1.out


C source file
sample1.c
Assembly source file
sample2.asm
COMPILER
ASSEMBLER
Object file
sample1.o
Object file
sample2.o
LINKER
Executable file
sample1.out

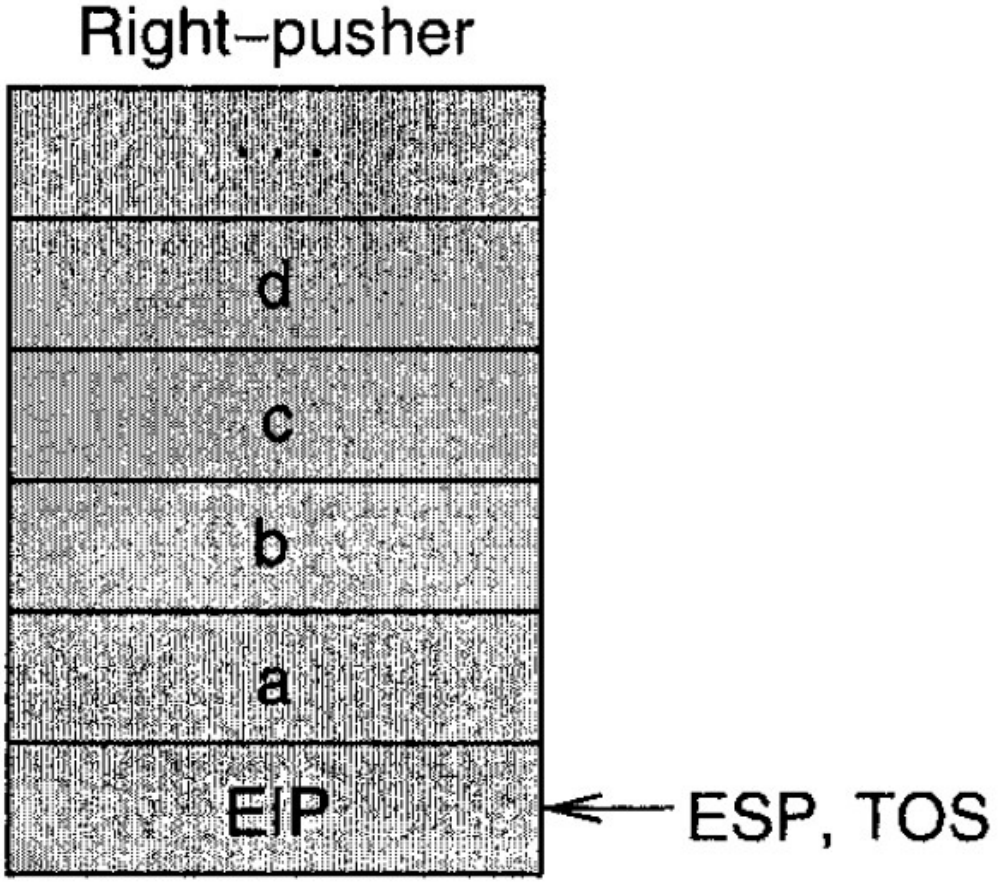
Chamando procedures assembly do código C

- Como os duas procedures (main e assembly) podem trocar parametros e resultados ?
- O canal de comunicação entre o C e o assembly é a pilha.
- É preciso saber como a função em C passa parâmetros para a pilha e onde ela espera o resultado retornado pela procedure assembly.

Chamando procedures assembly do código C

- Passagem de parâmetro: argumentos são empurrados para a pilha após a execução de uma função:

```
sum(a,b,c,d)
```



Código c que chama uma função externa

```
int main(void)
{
    int      x=25, y=70;
    int      value;
    extern   int test(int, int, int);

    value = test (x, y, 5);
    . . .
}
```

Código em assembly representando a chamada a função externa em C

```
push    5
push    70
push    25
call    test
add     ESP,12
mov     [EBP-12],EAX
```

Chamando procedures assembly do código C

- Valores de retorno – se utiliza o registrador eax para armazenar o valor de retorno da função. Quando se quer armazenar valores de 64 bits este é armazenado em EDX:EAX.
- Extern – funções que não são definidas no mesmo módulo são declaradas como **Extern.**
- Global - As procedures que são acessadas por outro módulo são declaradas como **global**.

Chamando assembly do código C

```
 6:  segment .text
 7:
 8:  global  test1
 9:
10:  test1:
11:         enter   0,0
12:         mov     EAX,[EBP+8]          ; get argument1 (x)
13:         add     EAX,[EBP+12]         ; add argument 2 (y)
14:         sub     EAX,[EBP+16]         ; subtract argument3 (5)
15:         leave
16:         ret
```

Chamando assembly do código C

```
 6:  #include          <stdio.h>
 7:
 8:  int main(void)
 9:  {
10:          int     x = 25, y = 70;
11:          int     value;
12:          extern int test1 (int, int, int);
13:
14:          value = test1(x, y, 5);
15:          printf("Result = %d\n", value);
16:
17:          return 0;
18:  }
```

Exemplo min-max

```
 7:  #include <stdio.h>
 8:  int main(void)
 9:  {
10:        int     value1, value2, value3;
11:        int     min, max;
12:        extern  void min_max (int, int, int, int*, int*);
13:
14:        printf("Enter number 1 = ");
15:        scanf("%d", &value1);
16:        printf("Enter number 2 = ");
17:        scanf("%d", &value2);
18:        printf("Enter number 3 = ");
19:        scanf("%d", &value3);
20:
21:        min_max(value1, value2, value3, &min, &max);
22:        printf("Minimum = %d, Maximum = %d\n", min, max);
23:        return 0;
24:  }
```

```
 7:  global  min_max
 8:
 9:  min_max:
10:         enter   0,0
11:         ; EAX keeps minimum number and EDX maximum
12:         mov     EAX,[EBP+8]      ; get value 1
13:         mov     EDX,[EBP+12]     ; get value 2
14:         cmp     EAX,EDX          ; value 1 < value 2?
15:         jl      skip1            ; if so, do nothing
16:         xchg    EAX,EDX          ; else, exchange
17:  skip1:
18:         mov     ECX,[EBP+16]     ; get value 3
19:         cmp     ECX,EAX          ; value 3 < min in EA
20:         jl      new_min
21:         cmp     ECX,EDX          ; value 3 < max in ED
22:         jl      store_result
23:         mov     EDX,ECX
24:         jmp     store_result
25:  new_min:
26:         mov     EAX,ECX
27:  store_result:
28:         mov     EBX,[EBP+20]     ; EBX = &min
29:         mov     [EBX],EAX
30:         mov     EBX,[EBP+24]     ; EBX = &max
31:         mov     [EBX],EDX
32:         leave
33:         ret
```

Exemplo soma Array

```
 6:  #include        <stdio.h>
 7:
 8:  #define  SIZE  10
 9:
10:  int main(void)
11:  {
12:      int    value[SIZE], sum, i;
13:      extern int array_sum(int*, int);
14:
15:      printf("Input %d array values:\n", SIZE);
16:      for (i = 0; i < SIZE; i++)
17:          scanf("%d",&value[i]);
18:
19:      sum = array_sum(value,SIZE);
20:      printf("Array sum = %d\n", sum);
21:
22:      return 0;
23:  }
```

Exemplo soma Array

```
 5:  segment .text
 6:
 7:  global  array_sum
 8:
 9:  array_sum:
10:         enter   0,0
11:         mov     EDX,[EBP+8]     ; copy array pointer to EDX
12:         mov     ECX,[EBP+12]    ; copy array size to ECX
13:         sub     EBX,EBX         ; array index = 0
14:         sub     EAX,EAX         ; sum = 0 (EAX keeps the sum)
15:  add_loop:
16:         add     EAX,[EDX+EBX*4]
17:         inc     EBX             ; increment array index
18:         cmp     EBX,ECX
19:         jl      add_loop
20:         leave
21:         ret
```

Chamando funções C do código assembly

- A pilha também é utilizada.

```
 6:  #include        <stdio.h>
 7:
 8:  #define  SIZE  10
 9:
10:  int main(void)
11:  {
12:      int    value[SIZE];
13:      extern int array_sum(int*, int);
14:
15:      printf("sum = %d\n",array_sum(value,SIZE));
16:
17:      return 0;
18:  }
```

Chamando funções C do código assembly

```
1:  ;-----------------------------------------------------
2:  ; This procedure receives an array pointer and its size
3:  ; via the stack. It first reads the array input from the
4:  ; user and then computes the array sum.
5:  ; The sum is returned to the C program.
6:  ;-----------------------------------------------------
 7:  segment .data
 8:  scan_format    db   "%d",0
 9:  printf_format  db   "Input %d array values:",10,13,0
10:
11:  segment .text
12:
13:  global  array_sum
14:  extern  printf,scanf
```

```
15:
16:  array_sum:
17:         enter   0,0
18:         mov     ECX,[EBP+12]    ; copy array size to ECX
19:         push    ECX             ; push array size
20:         push    dword printf_format
21:         call    printf
22:         add     ESP,8           ; clear the stack
23:
24:         mov     EDX,[EBP+8]     ; copy array pointer to EDX
25:         mov     ECX,[EBP+12]    ; copy array size to ECX
26:  read_loop:
27:         push    ECX             ; save loop count
28:         push    EDX             ; push array pointer
29:         push    dword scan_format
30:         call    scanf
31:         add     ESP,4           ; clear stack of one argument
32:         pop     EDX             ; restore array pointer in EDX
33:         pop     ECX             ; restore loop count
34:         add     EDX,4           ; update array pointer
35:         dec     ECX
36:         jnz     read_loop
37:
38:         mov     EDX,[EBP+8]     ; copy array pointer to EDX
```

Chamando funções C do código assembly

```
39:         mov     ECX,[EBP+12]    ; copy array size to ECX
40:         sub     EAX,EAX         ; EAX = 0 (EAX keeps the sum)
41:  add_loop:
42:         add     EAX,[EDX]
43:         add     EDX,4           ; update array pointer
44:         dec     ECX
45:         jnz     add_loop
46:         leave
47:         ret
```

Assembly Inline

- É possível embarcar código assembly no código C através da construção asm.
- A sintaxe gcc para declarações assembly segue a sintaxe AT&T, diferente da sintaxe INTEL.
- A sintaxe AT&T possui algumas diferenças:

  - nome de registrador: inicia com %. Por exemplo, %eax.

  - fonte e destino: mov eax, ebx é escrito como movl %ebx, %eax

Assembly Inline

- Tamanho do operando: byte (b), word (w) e long word (l):

```
movb    %bl,%al        ; moves contents of bl to al
movw    %bx,%ax        ; moves contents of bx to ax
movl    %ebx,%eax      ; moves contents of ebx to eax
```

- Constantes: são especificados pelo prefixo $:

```
movb    $255,%al
movl    $0xFFFFFFFF,%eax
```

Assembly inline

- Carrega o endereço da variável global total em C no registrador eax:

```
movl    $total,%eax
```

- Endereçamento: para especificar endereçamento indireto `mov eax, [ebx]`

  é escrito como `movl (%ebx),%eax`

Assembly inline

- Declarações inline:

```
asm("incl %eax");

asm("pushl    %eax");
asm("incl     %eax");
asm("popl     %eax");
```

- Pode ser agrupada:

```
asm("pushl    %eax; incl     %eax; popl     %eax");
```

ou

```
asm("pushl    %eax;"
    "incl     %eax;"
    "popl     %eax");
```

Comparando as sintaxes

Intel Code	AT&T Code
mov eax,1	movl $1,%eax
mov ebx,0ffh	movl $0xff,%ebx
int 80h	int $0x80
mov ebx, eax	movl %eax, %ebx
mov eax,[ecx]	movl (%ecx),%eax
mov eax,[ebx+3]	movl 3(%ebx),%eax
mov eax,[ebx+20h]	movl 0x20(%ebx),%eax
add eax,[ebx+ecx*2h]	addl (%ebx,%ecx,0x2),%eax
lea eax,[ebx+ecx]	leal (%ebx,%ecx),%eax
sub eax,[ebx+ecx*4h-20h]	subl -0x20(%ebx,%ecx,0x4),%eax

Declarações inline estendidas

- Formato:

```
asm(assembly code
    :outputs
    :inputs
    :clobber list);
```

- Assembly code – código assembly
- Outputs – especifica o operando de saída para o código assembly:

```
"=op-constraint" (C-expression)
```

= especifica que isso é uma restrição de saída

`"=r" (sum)` especifica que a variável sum do **C** deve ser mapeada para um registrador com indicado por **r**

Declarações inline estendidas

- Outras escolhas podem ser permitidas: r, m (memory), i (immediate), g (general)
- Inputs – não utiliza =

```
asm("movl %1,%0"
      :"=r"(sum)      /* output */
      :"r"(number1)   /* input  */
     );
```

- No código assembly sum é identificado por 0 e number2 por 1

Declarações inline estendidas

- Se quer realizar a operação `sum = sum + number1`

```
asm("addl %1,%0"
      :"=r"(sum)                    /* output */
      :"r"(number1), "0"(sum)       /* inputs */
    );
```

- Clobber list – o gcc será informado sobre os registradores que foram modificados.

```
asm("movl %0,%%eax"
      : /* no output */
      :"r"(number1)   /* inputs */
      :"%eax"         /* clobber list */
    );
```

Especificando registradores particulares

Letter	Register set
a	EAX register
b	EBX register
c	ECX register
d	EDX register
S	ESI register
D	EDI register
r	Any of the eight general registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP)
q	Any of the four data registers (EAX, EBX, ECX, EDX)
A	A 64-bit value in EAX and EDX
f	Floating-point registers
t	Top floating-point register
u	Second top floating-point register

Exemplo inline

```
 6:  #include        <stdio.h>
 7:
 8:  int main(void)
 9:  {
10:          int    x = 25, y = 70;
11:          int    value;
12:          extern int test1 (int, int, int);
13:
14:          value = test1(x, y, 5);
15:          printf("Result = %d\n", value);
16:
17:          return 0;
18:  }
19:
20:  int test1(int x, int y, int z)
21:  {
22:         asm("movl  %0,%%eax;"
23:             "addl  %1,%%eax;"
24:             "subl  %2,%%eax;"
25:              :/* no outputs */          /* outputs */
26:              : "r"(x), "r"(y), "r"(z)  /* inputs */
27:              :"cc","%eax");            /* clobber list */
28:  }
```

Exemplo2 inline

```
sum += value[i];
```

```
 6:  #include        <stdio.h>
 7:
 8:  #define  SIZE  10
 9:
10:  int main(void)
11:  {
12:      int    value[SIZE], sum, i;
13:      int    array_sum(int*, int);
14:
15:      printf("Input %d array values:\n", SIZE);
16:      for (i = 0; i < SIZE; i++)
17:          scanf("%d",&value[i]);
18:
19:      sum = array_sum(value,SIZE);
20:      printf("Array sum = %d\n", sum);
21:
22:      return 0;
23:  }
24:
25:  int array_sum(int* value, int size)
26:  {
27:      int  i, sum=0;
28:      for (i = 0; i < size; i++)
29:           asm("addl (%1,%2,4),%0"
30:                :"=r"(sum)                        /* output */
31:                :"r"(value),"r"(i),"0"(sum)    /* inputs */
32:                :"cc");                    /* clobber list */
33:      return(sum);
34:  }
```