Ordem de Bytes: Big Endian e Little Endian

Dependendo da máquina que se usa, tem de se considerar a ordem pela qual os números constituídos por múltiplos bytes estão armazenados. As ordem são duas: big endian e little endian.

Big Endian:

Significa que o byte de ordem mais elevada do número está armazenado no endereço de memória mais baixo e de ordem mais baixa, no endereço mais elevado.

Os processadores Motorola (dos Mac's) usam ordem de byte Big Endian.

Little Endian:

Significa que o byte de ordem mais baixa do número está armazenado no endereço de memória mais baixo e de ordem mais elevada, no endereço mais elevado.

Os processadores Intel (dos PC's) usam ordem de byte Little Endian.

Casos Particulares da Representação em Binário de Números em Vírgula Flutuante Pela Norma IEEE754

Representação de Valores Desnormalizados:

A norma contempla este tipo de situações reservando para os valores desnormalizados os valores com:

expoente = 0000 0000b
mantissa != 0

Representação do Zero:

Caso particular dos valores desnormalizados, com:

expoente = 0000 0000b
mantissa = 0

Representação de +/- Infinito:

A norma contempla este tipo de situações reservando para estes valores a outra extremidade de representação do expoente:

expoente = 1111 1111b
mantissa = 0

Representação de números não reais:

Quando os números são não reais, a norma prevê uma forma de o indicar para posterior tratamento por rotinas de excepção (NaN):

expoente = 1111 1111b
mantissa != 0

Vantagens de Programar em Linguagens de Alto Nível

- **Tem-se disponível um corrector de código que nos ajuda a detectar erros e verifica se os dados estão referenciados e manipulados de forma consistente;**

- **O código gerado pelos compiladores de hoje em dia é tão ou mais eficiente do que um programador de linguagem Assembly implementaria manualmente;**

- **Os programas podem ser compilados e executados em diferentes máquinas, enquanto o Assembly é específico de cada máquina.**

Representação da Informação do Programa Fonte

Considerando o programa sugerido por Kernighan e Ritchie para introdução à linguagem C: “hello world.c”:

```
1 #include <stdio.h>
2
3 int main()
4 {
5 printf("hello, world\n");
6 }
```

A sua representação em ASCII será:

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	<sp>	w	o	r	l	d	\	n	“	)	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

Níveis de abstracção num computador

Nível da linguagem máquina (em binário): instruções e variáveis totalmente codificadas em binário, sendo a codificação das instruções sempre associada a um dado processador e tendo como objectivo a execução eficiente e rápida dos comandos; a sua utilização é pouco adequada para seres humanos;

• **Nível da linguagem *assembly*** (tradução literal do inglês: "de montagem"): equivalente ao nível anterior, mas em vez da notação puramente binária, a linguagem usa mnemónicas para especificar as operações pretendidas, bem como os valores ou localizações dos operandos; embora este nível seja melhor manuseado por seres humanos, ele ainda é inteiramente dependente do conjunto de instruções dum dado processador, isto é, não é portátil entre processadores de famílias diferentes, e as estruturas que manipula, quer de controlo, quer de dados, são de muito baixo nível;

• **Nível das linguagens HLL** (*High Level Languages*, como o C, Pascal, FORTRAN, ...): linguagens mais poderosas e mais próximas dos seres humanos, que permitem a construção de programas para execução eficiente em qualquer processador.

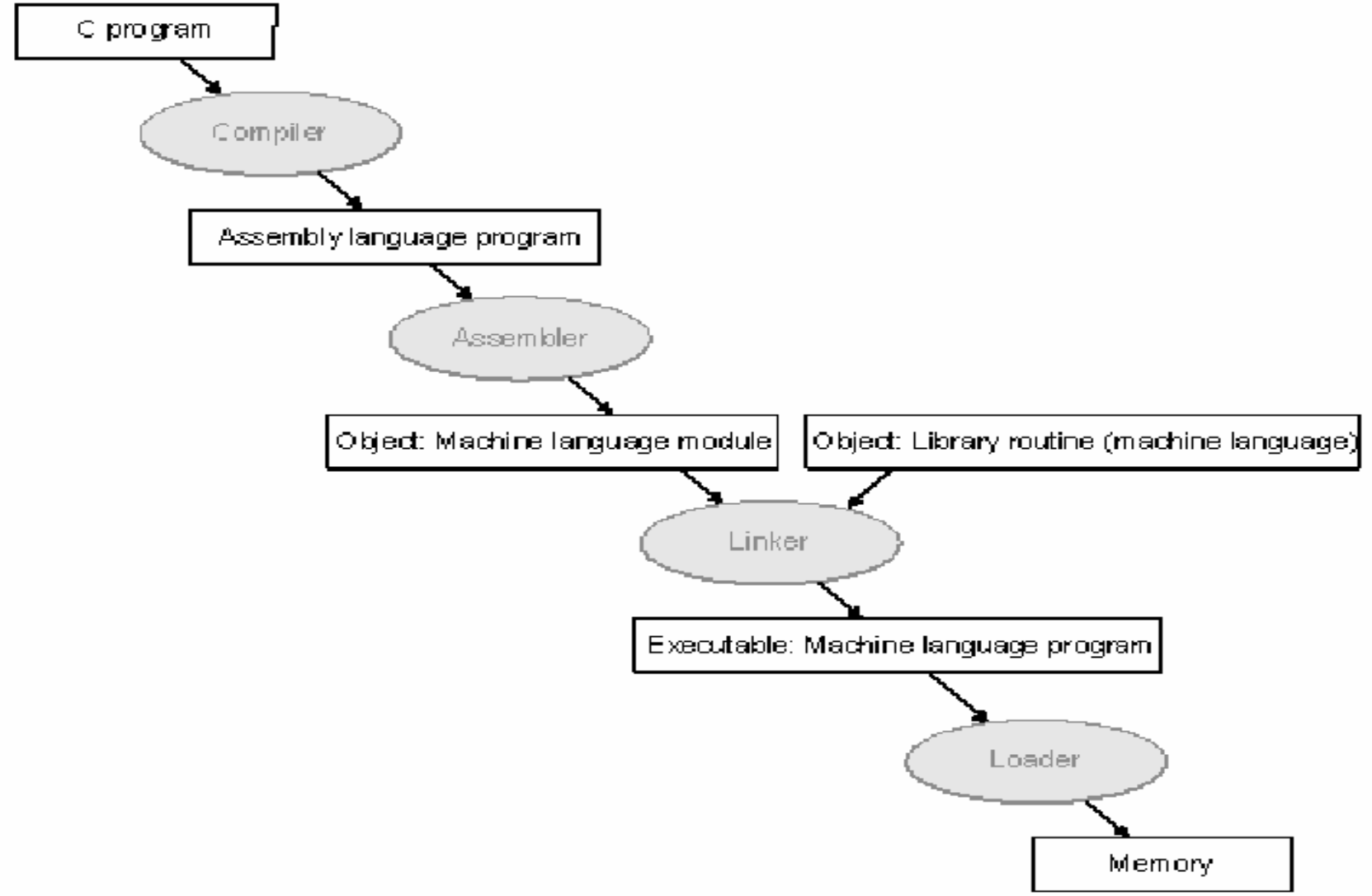
Níveis de abstracção num computador

Dado que o processador apenas "entende" os comandos em linguagem máquina, é necessário converter os programas escritos em linguagens dos níveis de abstracção superiores para níveis mais baixos, até eventualmente se chegar à linguagem máquina. Estes tradutores ou conversores de níveis são normalmente designados por:

• **Compiladores**: programas que traduzem os programas escritos em HLL para o nível de abstracção inferior, i.e., para *assembly*; a maioria dos compiladores existentes incluem já os dois passos da tradução para linguagem máquina, isto é, traduzem de HLL directamente para linguagem máquina binária, sem necessitarem de um *assembler*;

• ***Assemblers*** : programas que lêem os ficheiros com os programas escritos em linguagem de montagem - *assembly language* – e os convertem para linguagem máquina em binário, i.e., "montam" as instruções em formato adequado ao processador (também designados por "montadores" na literatura brasileira).

Níveis de abstracção num computador

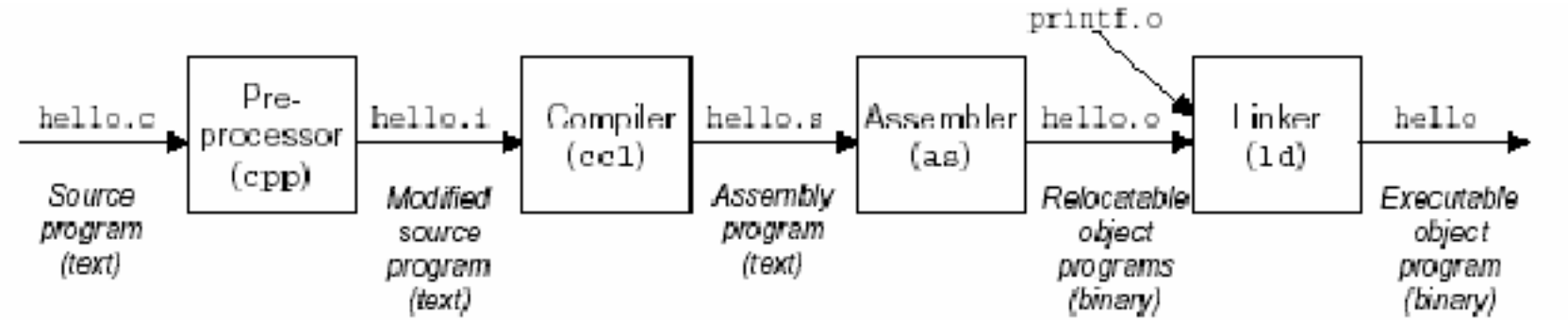


Sistema de Compilação

Um sistema de Compilação é composto pelos 4 programas:

- Pré-processador
- Compilador
- Assemblador
- Linker

Executando a linha de comandos: *gcc -o hello hello.c*



_____soma.c

```
int accum=0;

void soma (int p)
{
accum+=p;
}
```

_____soma.s

```
     .file     "soma.c"
     .version  "01.01"
gcc2_compiled.:
.globl accum
.data
     .align 4
     .type      accum,@object
     .size      accum,4
accum:
     .long     0
.text
     .align 4
.globl soma
     .type      soma,@function
soma:
     pushl     %ebp
     movl %esp, %ebp
     movl accum, %eax
     addl 8(%ebp), %eax
     movl %eax, accum
     popl %ebp
     ret
.Lfe1:
     .size      soma,.Lfe1-soma
     .ident    "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.3
2.96-113)"
```

_____soma.o

```
soma.o:     file format elf32-i386

Disassembly of section .text:

00000000 <soma>:
   0:     55                      push   %ebp
   1:     89 e5                   mov    %esp,%ebp
   3:     a1 00 00 00 00          mov    0x0,%eax
   8:     03 45 08                add    0x8(%ebp),%eax
   b:     a3 00 00 00 00          mov    %eax,0x0
  10:     5d                      pop    %ebp
  11:     c3                      ret
  12:     89 f6                   mov    %esi,%esi
```

_____prog.c

```
main()
{
int x;
soma(x);
}
```

_____prog.s

```
     .file     "prog.c"
     .version  "01.01"
gcc2_compiled.:
.text
     .align 4
.globl main
     .type      main,@function
main:
     pushl     %ebp
     movl %esp, %ebp
     subl $8, %esp
     subl $12, %esp
     pushl     -4(%ebp)
     call soma
     addl $16, %esp
     leave
     ret
.Lfe1:
     .size      main,.Lfe1-main
     .ident    "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.3
2.96-113)"
```

_____prog.o

```
prog.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:     55                      push   %ebp
   1:     89 e5                   mov    %esp,%ebp
   3:     83 ec 14                sub    $0x14,%esp
   6:     50                      push   %eax
   7:     e8 fc ff ff ff          call   8 <main+0x8>
   c:     c9                      leave
   d:     c3                      ret
   e:     89 f6                   mov    %esi,%esi
```

_____

_____prog2.c

```
main()
{
soma(10);
}
```

_____prog2.s

```
     .file     "prog2.c"
     .version  "01.01"
gcc2_compiled.:
.text
     .align 4
.globl main
     .type      main,@function
main:
     pushl     %ebp
     movl %esp, %ebp
     subl $8, %esp
     subl $12, %esp
     pushl     $10
     call soma
     addl $16, %esp
     leave
     ret
.Lfe1:
     .size      main,.Lfe1-main
     .ident    "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.3
2.96-113)"
```

_____prog2.o

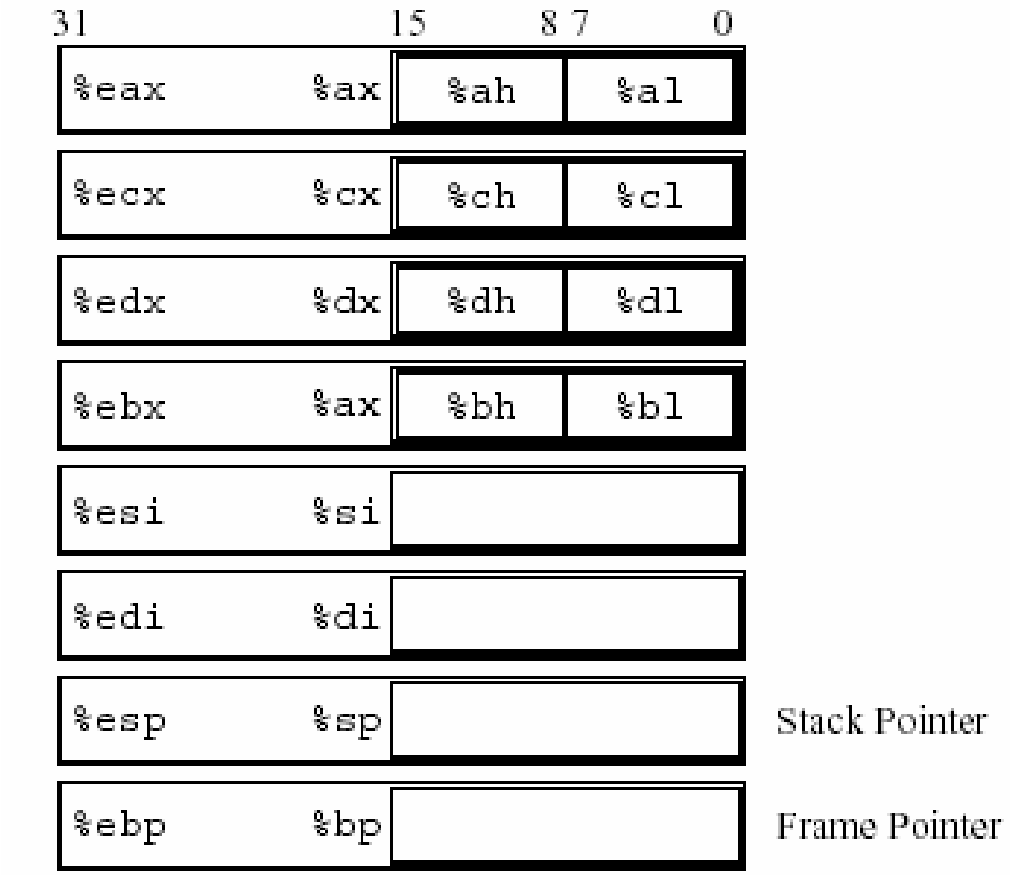
```
prog2.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:     55                      push   %ebp
   1:     89 e5                   mov    %esp,%ebp
   3:     83 ec 08                sub    $0x8,%esp
   6:     83 ec 0c                sub    $0xc,%esp
   9:     6a 0a                   push   $0xa
   b:     e8 fc ff ff ff          call   c <main+0xc>
  10:     83 c4 10                add    $0x10,%esp
  13:     c9                      leave
  14:     c3                      ret
  15:     8d 76 00                lea    0x0(%esi),%esi
```

C declaration	Intel Data Type	GAS suffix	Size (Bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double Word	`l`	4
`unsigned`	Double Word	`l`	4
`long int`	Double Word	`l`	4
`unsigned long`	Double Word	`l`	4
`char *`	Double Word	`l`	4
`float`	Single Precision	`s`	4
`double`	Double Precision	`l`	8
`long double`	Extended Precision	`t`	10/12

Representação de Tipos de Dados do C

Exemplo:

movl movw movb



Registos

Estes registos são usados para armazenar tanto inteiros como apontadores.

Os nomes dos registos do IA32 começam todos por %e.

Operandos:

Tipos de Operandos:

Imediato: para valores constantes (inteiros com 32 bits)

Exemplo: $-345, $0x2D

Registo: denota o conteúdo de um registo (32 bits ou um dos seus elementos, dependendo da operação ser double-word ou byte)

Exemplo: %eax, %al

Referência de memória: a forma complexa é Imm(E_b,E_i,s), todas as outras são simplificações desta:

Imm: offset de memória
E_b: registo base
E_i: registo índice
s: factor de escala (1, 2, 4 ou 8)

Exemplo: (%eax, %edx, 4)

Type	Form	Operand Value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	E_a	$Reg[E_a]$	Register
Memory	Imm	$Mem[Imm]$	Absolute
Memory	(E_a)	$Mem[Reg[E_a]]$	Indirect
Memory	$Imm\,(E_b)$	$Mem[Imm + Reg[E_b]]$	Base + Displacement
Memory	(E_b, E_i)	$Mem[Reg[E_b] + Reg[E_i]]$	Indexed
Memory	$Imm\,(E_b, E_i)$	$Mem[Imm + Reg[E_b] + Reg[E_i]]$	Indexed
Memory	$(, E_i, s)$	$Mem[Reg[E_i] \cdot s]$	Scaled Indexed
Memory	$Imm\,(, E_i, s)$	$Mem[Imm + Reg[E_i] \cdot s]$	Scaled Indexed
Memory	(E_b, E_i, s)	$Mem[Reg[E_b] + Reg[E_i] \cdot s]$	Scaled Indexed
Memory	$Imm\,(E_b, E_i, s)$	$Mem[Imm + Reg[E_b] + Reg[E_i] \cdot s]$	Scaled Indexed

Instruction		Effect	Description
`movl`	S, D	$D \leftarrow S$	Move Double Word
`movw`	S, D	$D \leftarrow S$	Move Word
`movb`	S, D	$D \leftarrow S$	Move Byte
`movsbl`	S, D	$D \leftarrow SignExtend(S)$	Move Sign-Extended Byte
`movzbl`	S, D	$D \leftarrow ZeroExtend(S)$	Move Zero-Extended Byte
`pushl`	S	$Reg[\%esp] \leftarrow Reg[\%esp] - 4$; $Mem[Reg[\%esp]] \leftarrow S$	Push
`popl`	D	$D \leftarrow Mem[Reg[\%esp]]$; $Reg[\%esp] \leftarrow Reg[\%esp] + 4$	Pop

Instruções para Transferência de Informação

```
movl $0x4050,%eax          Immediate--Register
movl %ebp,%esp             Register--Register
movl (%edi,%ecx),%eax      Memory--Register
movl $-17,(%esp)           Immediate--Memory
movl %eax,-12(%ebp)        Register--Memory
```

Não é possível o mov de memória para memória, o que tem de ser executado em 2 passos atrvés do uso de um registo intermédio.

Exemplo:

Assumir que %dh = 8D e %eax = 98765432

movb %dh, %al	%eax = 9876548D
movsbl %dh, %eax	%eax = FFFFFF8D
movzbl %dh, %eax	%eax = 0000008D

Instruction		Effect	Description
`leal`	*S*, *D*	*D* ← &*S*	Load Effective Address
`incl`	*D*	*D* ← *D* + 1	Increment
`decl`	*D*	*D* ← *D* - 1	Decrement
`negl`	*D*	*D* ← -*D*	Negate
`notl`	*D*	*D* ← ~*D*	Complement
`addl`	*S*, *D*	*D* ← *D* + *S*	Add
`subl`	*S*, *D*	*D* ← *D* - *S*	Subtract
`imull`	*S*, *D*	*D* ← *D* * *S*	Multiply
`xorl`	*S*, *D*	*D* ← *D* ^ *S*	Exclusive-Or
`orl`	*S*, *D*	*D* ← *D* | *S*	Or
`andl`	*S*, *D*	*D* ← *D* & *S*	And
`sall`	*k*, *D*	*D* ← *D* << *k*	Left Shift
`shll`	*k*, *D*	*D* ← *D* << *k*	Left Shift (same as `sall`)
`sarl`	*k*, *D*	*D* ← *D* >> *k*	Arithmetic Right Shift
`shrl`	*k*, *D*	*D* ← *D* >> *k*	Logical Right Shift

Operações aritméticas com inteiros

A instrução leal é uma variante da instrução movl. O seu primeiro operando apresenta-se como uma referência a uma posição de memória, mas em vez de ler o valor dessa localização, a instrução copia o seu endereço efectivo para o destino.

As operações de shift à direita são as únicas que distinguem entre operandos com e sem sinal.

code/asm/arith.c

```
1 int arith(int x,
2           int y,
3           int z)
4 {
5     int t1 = x+y;
6     int t2 = z*48;
7     int t3 = t1 & 0xFFFF;
8     int t4 = t2 * t3;
9
10    return t4;
11 }
```

code/asm/arith.c

```
1   movl 12(%ebp),%eax          Get y
2   movl 16(%ebp),%edx          Get z
3   addl 8(%ebp),%eax           Compute t1 = x+y
4   leal (%edx,%edx,2),%edx     Compute z*3
5   sall $4,%edx                Compute t2 = z*48
6   andl $65535,%eax            Compute t3 = t1&0xFFFF
7   imull %eax,%edx             Compute t4 = t2*t3
8   movl %edx,%eax              Set t4 as return val
```

Instruction		Based on	Description
`cmpb`	S_2, S_1	S_1 - S_2	Compare bytes
`testb`	S_2, S_1	S_1 & S_2	Test byte
`cmpw`	S_2, S_1	S_1 - S_2	Compare words
`testw`	S_2, S_1	S_1 & S_2	Test word
`cmpl`	S_2, S_1	S_1 - S_2	Compare double words
`testl`	S_2, S_1	S_1 & S_2	Test double word

Instruction		Synonym	Effect	Set Condition
`sete`	D	`setz`	$D \leftarrow$ `ZF`	Equal / Zero
`setne`	D	`setnz`	$D \leftarrow$ `~ZF`	Not Equal / Not Zero
`sets`	D		$D \leftarrow$ `SF`	Negative
`setns`	D		$D \leftarrow$ `~SF`	Nonnegative
`setg`	D	`setnle`	$D \leftarrow$ `~(SF ^ OF) & ~ZF`	Greater (Signed >)
`setge`	D	`setnl`	$D \leftarrow$ `~(SF ^ OF)`	Greater or Equal (Signed >=)
`setl`	D	`setnge`	$D \leftarrow$ `SF ^ OF`	Less (Signed <)
`setle`	D	`setng`	$D \leftarrow$ `(SF ^ OF) | ZF`	Less or Equal (Signed <=)
`seta`	D	`setnbe`	$D \leftarrow$ `~CF & ~ZF`	Above (Unsigned >)
`setae`	D	`setnb`	$D \leftarrow$ `~CF`	Above or Equal (Unsigned >=)
`setb`	D	`setnae`	$D \leftarrow$ `CF`	Below (Unsigned <)
`setbe`	D	`setna`	$D \leftarrow$ `CF & ~ZF`	Below or Equal (Unsigned <=)

Códigos de Condição

CF (CARRY FLAG): A OPERAÇÃO MAIS RECENTE GEROU UM BIT CARRY NO BIT MAIS SIGNIFICATIVO;
ZF(ZERO FLAG): A OPERAÇÃO MAIS RECENTE GEROU ZERO;
SF (SIGN FLAG): A OPERAÇÃO MAIS RECENTE GEROU UM VALOR NEGATIVO;
OF (OVERFLOW FLAG): A OPERAÇÃO MAIS RECENTE GEROU UM OVERFLOW EM COMPLEMENTO PARA 2 (PODE SER NEGATIVO OU POSITIVO).

Exemplo: supondo que se tem uma adição, t=a+b, usando variáveis do tipo inteiro em C.

Os códigos de condição seriam colocados de acordo com as seguintes expressões em C:

```
CF:   (unsigned t) < (unsigned a)                Unsigned overflow
ZF:   (t == 0)                                   Zero
SF:   (t < 0)                                    Negative
OF:   (a < 0 == b < 0) && (t < 0 != a < 0)       Signed overflow
```

Dado o seguinte código em Assembly IA32, complete o código em C a seguir apresentado, preenchendo as partes em falta (as comparações e as conversões de tipo).

```
1   movl 8(%ebp),%ecx          Get a
2   movl 12(%ebp),%esi         Get b
3   cmpl %esi,%ecx             Compare a:b
4   setl %al                   Compute t1
5   cmpl %ecx,%esi             Compare b:a
6   setb -1(%ebp)              Compute t2
7   cmpw %cx,16(%ebp)          Compare c:a
8   setge -2(%ebp)             Compute t3
9   movb %cl,%dl
10  cmpb 16(%ebp),%dl          Compare a:c
11  setne %bl                  Compute t4
12  cmpl %esi,16(%ebp)         Compare c:b
13  setg -3(%ebp)              Compute t5
14  testl %ecx,%ecx            Test a
15  setg %dl                   Compute t4
16  addb -1(%ebp),%al          Add t2 to t1
17  addb -2(%ebp),%al          Add t3 to t1
18  addb %bl,%al               Add t4 to t1
19  addb -3(%ebp),%al          Add t5 to t1
20  addb %dl,%al               Add t6 to t1
21  movsbl %al,%eax            Convert sum from char to int
```

```
1 char ctest(int a, int b, int c)
2 {
3   char t1 =         a __             b;
4   char t2 =         b __ (     )     a;
5   char t3 = (     ) c __ (     )     a;
6   char t4 = (     ) a __ (     )     c;
7   char t5 =         c __             b;
8   char t6 =         a __             0;
9   return t1 + t2 + t3 + t4 + t5 + t6;
10 }
```

Instruction		Synonym	Jump Condition	Description
`jmp`	*Label*		1	Direct Jump
`jmp`	**Operand*		1	Indirect Jump
`je`	*Label*	`jz`	`ZF`	Equal / Zero
`jne`	*Label*	`jnz`	`~ZF`	Not Equal / Not Zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (Signed >)
`jge`	*Label*	`jnl`	`~(SF ^ OF)`	Greater or Equal (Signed >=)
`jl`	*Label*	`jnge`	`SF ^ OF`	Less (Signed <)
`jle`	*Label*	`jng`	`(SF ^ OF) | ZF`	Less or Equal (Signed <=)
`ja`	*Label*	`jnbe`	`~CF & ~ZF`	Above (Unsigned >)
`jae`	*Label*	`jnb`	`~CF`	Above or Equal (Unsigned >=)
`jb`	*Label*	`jnae`	`CF`	Below (Unsigned <)
`jbe`	*Label*	`jna`	`CF & ~ZF`	Below or Equal (Unsigned <=)

```
 1    jle .L4                    If <, goto dest2
 2    .p2align 4,,7              Aligns next instruction to multiple of 8
 3  .L5:                       dest1:
 4    movl %edx,%eax
 5    sarl $1,%eax
 6    subl %eax,%edx
 7    testl %edx,%edx
 8    jg .L5                     If >, goto dest1
 9  .L4:                       dest2:
10    movl %edx,%eax
```

```
1     8:   7e 11                jle    1b <silly+0x1b>    Target = dest2
2     a:   8d b6 00 00 00 00    lea    0x0(%esi),%esi     Added nops
3    10:   89 d0                mov    %edx,%eax        dest1:
4    12:   c1 f8 01             sar    $0x1,%eax
5    15:   29 c2                sub    %eax,%edx
6    17:   85 d2                test   %edx,%edx
7    19:   7f f5                jg     10 <silly+0x10>    Target = dest1
8    1b:   89 d0                mov    %edx,%eax        dest2:
```