Questão 1:

Practice Problem 3.1

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Fill in the following table showing the values for the indicated operands:

Operand	Value
%eax	
0x104	
$0x108	
(%eax)	
4(%eax)	
9(%eax,%edx)	
260(%ecx,%edx)	
0xFC(,%ecx,4)	
(%eax,%edx,4)	

Solução da questão 1

Solution to Problem 3.1 (page 170)

This exercise gives you practice with the different operand forms.

Operand	Value	Comment
%eax	0x100	Register
0x104	0xAB	Absolute address
$0x108	0x108	Immediate
(%eax)	0xFF	Address 0x100
4(%eax)	0xAB	Address 0x104
9(%eax,%edx)	0x11	Address 0x10C
260(%ecx,%edx)	0x13	Address 0x108
0xFC(,%ecx,4)	0xFF	Address 0x100
(%eax,%edx,4)	0x11	Address 0x10C

Questão 2

Practice Problem 3.2

For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands. (For example, `mov` can be rewritten as `movb`, `movw`, or `movl`.)

```
1      mov   %eax, (%esp) movl
2      mov   (%eax), %dx  movw
3      mov   $0xFF, %bl   movb
```

Solução da questão 2

Solution to Problem 3.2 (page 174)

As we have seen, the assembly code generated by GCC includes suffixes on the instructions, while the disassembler does not. Being able to switch between these two forms is an important skill to learn. One important feature is that memory references in IA32 are always given with double-word registers, such as %eax, even if the operand is a byte or single word.

Here is the code written with suffixes:

```
1      movl   %eax, (%esp)
2      movw   (%eax), %dx
3      movb   $0xFF, %bl
```

Questão 3

Practice Problem 3.3

Each of the following lines of code generates an error message when we invoke the assembler. Explain what is wrong with each line.

```
1      movb $0xF, (%bl)          o ponteiro tem que ser 32 bits
2      movl %ax, (%esp)          %ax tem apenas 16 bit e a inst. é movl
3      movw (%eax),4(%esp)       movimentaçao memoria-memoria nao pod
4      movb %ah,%sh              destino nao existe
5      movl %eax,$0x123          o destino nao pode ser um valor imediato
```

Solução da questão 3

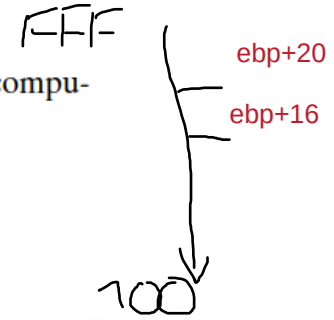
```
1 movb $0xF, (%bl)      Não é possível usar %bl como registrador de endereço
2 movl %ax, (%esp)      Incompatibilidade entre o sufixo da instrução e o registrador
3 movw (%eax),4(%esp)   Não é permitido que tanto a origem quanto o destino sejam refe
4 movb %ah,%sh          Não existe registrador chamado %sh
5 movl %eax,$0x123      Não é possível ter um imediato como destino
```

Questão 4

Practice Problem 3.12

Consider the following C function prototype, where `num_t` is a data type declared using `typedef`:

```
void store_prod(num_t *dest, unsigned x, num_t y) {
    *dest = x*y;
}
```



GCC generates the following assembly code implementing the body of the computation:

FFFF FFFF
2

IA32 é little endian = bit menos significativo no menor endereço de memoria

```
    dest at %ebp+8, x at %ebp+12, y at %ebp+16
1     movl    12(%ebp), %eax    %eax = x
2     movl    20(%ebp), %ecx    %ecx = parte alta de y
3     imull   %eax, %ecx        %ecx = (parte alta de y ) * x
4     mull    16(%ebp)          (parte baixa do y) * x --> %edx = parte alta do resultado e %eax guarda a parte baixa
5     leal    (%ecx,%edx), %edx %edx = parte alta do resultado multiplicação de x * y
6     movl    8(%ebp), %ecx     %ecx = dest
7     movl    %eax, (%ecx)      *(parte baixa de dest) = %eax
8     movl    %edx, 4(%ecx)
```

dest = 0x100 \\\ dest + 4 = *(0X104)

Observe that this code requires two memory reads to fetch argument y (lines 2 and 4), two multiplies (lines 3 and 4), and two memory writes to store the result (lines 7 and 8).

A. What data type is `num_t`?

B. Describe the algorithm used to compute the product and argue that it is correct.

Não tem a letra C!!!!

Solução da questão 4

Solution to Problem 3.12 (page 184)

A. We can see that the program is performing multiprecision operations on 64-bit data. We can also see that the 64-bit multiply operation (line 4) uses unsigned arithmetic, and so we conclude that `num_t` is `unsigned long long`.

B. Let x denote the value of variable x, and let y denote the value of y, which we can write as $y = y_h \cdot 2^{32} + y_l$, where y_h and y_l are the values represented by the high- and low-order 32 bits, respectively. We can therefore compute $x \cdot y = x \cdot y_h \cdot 2^{32} + x \cdot y_l$. The full representation of the product would be 96 bits long, but we require only the low-order 64 bits. We can therefore let s be the low-order 32 bits of $x \cdot y_h$ and t be the full 64-bit product $x \cdot y_l$, which

Questão 5

Practice Problem 3.15

In the following excerpts from a disassembled binary, some of the information has been replaced by Xs. Answer the following questions about these instructions.

A. What is the target of the `je` instruction below? (You don't need to know anything about the `call` instruction here.)

```
804828f:    74 05              je    XXXXXXX
8048291:    e8 1e 00 00 00     call  80482b4
```

B. What is the target of the `jb` instruction below?

```
8048357:    72 e7              jb    XXXXXXX
8048359:    c6 05 10 a0 04 08 01   movb  $0x1,0x804a010
```

1110 0111

0001 1000

0001 1001 = -25 = -0x19

Solução da questão 5

Solution to Problem 3.15 (page 192)

This exercise requires you to examine disassembled code in detail and reason about the encodings for jump targets. It also gives you practice in hexadecimal arithmetic.

A. The `je` instruction has as target 0x8048291 + 0x05. As the original disassembled code shows, this is 0x8048296:

```
804828f:    74 05              je    8048296
8048291:    e8 1e 00 00 00     call  80482b4
```

B. The `jb` instruction has as target 0x8048359 − 25 (since 0xe7 is the 1-byte, two's-complement representation of −25). As the original disassembled code shows, this is 0x8048340:

```
8048357:    72 e7              jb    8048340
8048359:    c6 05 10 a0 04 08 01   movb  $0x1,0x804a010
```

0x59 - 0x19 = 0x40

Questão 6 *(acho que não cai switch)*

Practice Problem 3.29

For a C function `switcher` with the general structure

```
1   int switcher(int a, int b, int c)
2   {
3       int answer;
4       switch(a) {
5       case _________:       /* Case A */
6           c = _________;
7           /* Fall through */
8       case _________:       /* Case B */
9           answer = _________;
10          break;
11      case _________:       /* Case C */
12      case _________:       /* Case D */
13          answer = _________;
14          break;
15      case _________:       /* Case E */
16          answer = _________;
17          break;
18      default:
19          answer = _________;
20      }
21      return answer;
22  }
```

GCC generates the assembly code and jump table shown in Figure 3.20.

Fill in the missing parts of the C code. Except for the ordering of case labels `C` and `D`, there is only one way to fit the different cases into the template.

```
   a at %ebp+8, b at %ebp+12, c at %ebp+16
1     movl    8(%ebp), %eax
2     cmpl    $7, %eax
3     ja      .L2
4     jmp     *.L7(,%eax,4)
5   .L2:
6     movl    12(%ebp), %eax
7     jmp     .L8
8   .L5:
9     movl    $4, %eax
10    jmp     .L8
11  .L6:
12    movl    12(%ebp), %eax
13    xorl    $15, %eax
14    movl    %eax, 16(%ebp)
15  .L3:
16    movl    16(%ebp), %eax
17    addl    $112, %eax
18    jmp     .L8
19  .L4:
20    movl    16(%ebp), %eax
21    addl    12(%ebp), %eax
22    sall    $2, %eax
23  .L8:
```

```
1   .L7:
2     .long   .L3
3     .long   .L2
4     .long   .L4
5     .long   .L2
6     .long   .L5
7     .long   .L6
8     .long   .L2
9     .long   .L4
```

Solução da questão 6

Solution to Problem 3.29 (page 218)

The key to reverse engineering compiled `switch` statements is to combine the information from the assembly code and the jump table to sort out the different cases. We can see from the `ja` instruction (line 3) that the code for the default case

has label `.L2`. We can see that the only other repeated label in the jump table is `.L4`, and so this must be the code for the cases C and D. We can see that the code falls through at line 14, and so label `.L6` must match case A and label `.L3` must match case B. That leaves only label `.L2` to match case E.

The original C code is as follows. Observe how the compiler optimized the case where a equals 4 by setting the return value to be 4, rather than a.

```
1  int switcher(int a, int b, int c)
2  {
3      int answer;
4      switch(a) {
6      case 5:
7          c = b ^ 15;
8          /* Fall through */
9      case 0:
10         answer = c + 112;
11         break;
12     case 2:
13     case 7:
14         answer = (c + b) << 2;
15         break;
16     case 4:
17         answer = a;  /* equivalently, answer = 4 */
18         break;
19     default:
20         answer = b;
21     }
22     return answer;
23 }
```