Lab 8: Integer Arithmetic and Bit Manipulation

Contents

8.1 Bitwise Logical Instructions

The IA-32 instruction set contains the AND, OR, XOR, NOT, and TEST instructions that implement bitwise logical operations. The source and destination operands can be bytes, words, or double words, and they must be of the same size. These instructions are listed in the table shown below:

Instruction	Description
AND *destination*, *source*	Bitwise AND: Result bit is 1 if both bits are 1. Modifies ZF, SF, and PF flags according to the result value. Always clear the CF and OF flags
OR *destination*, *source*	Bitwise OR: Result bit is 1 if at least one bit is 1. Modifies ZF, SF, and PF flags according to the result value. Always clear the CF and OF flags.
XOR *destination, source*	Bitwise XOR: Result bit is 1 if one bit is 1 and the other bit is 0. Modifies ZF, SF, and PF flags according to the result value. Always clear the CF and OF flags.
NOT *destination*	Bitwise NOT: Toggles all bits in an operand (1's complement). No flags are affected by the NOT instruction.
TEST *destination, source*	Bitwise TEST: does an AND, but does not write *destination*. Modifies ZF, SF, and PF flags in accordance to the AND instruction. Always clear the CF and OF flags.

8.1.1 The CPU Flags

Recall from Lab 4 (Basic Instructions) the zero flag (ZF), the sign flag (SF) the carry flag (CF), the overflow flag (OF), and the parity flag (PF):

- The Zero flag is set when the result of an operation is zero.
- The Sign flag is set when the high bit of the destination operand is 1 (or negative).
- The Carry flag is set when the unsigned result is out of range.
- The Overflow flag is set when the signed result is out of range.
- The Parity flag is set when an even number of 1 bits exist in the low byte of the result.

8.1.2 Converting the Letter Case

Compare the ASCII codes of capital **'A'** and lowercase **'a'**. Only bit 5 is different.

```
0 1 0 0 0 0 0 1 = 41h = 'A'
0 1 1 0 0 0 0 1 = 61h = 'a'
```

The AND instruction provides a simple way to change a letter to uppercase:

```
AND AL, 11011111b ; clear bit 5 of AL
```

The OR instruction provides a simple way to change a letter to lowercase:

```
OR  AL, 00100000b ; set bit 5 of AL
```

The XOR instruction toggles the letter case (from uppercase to lowercase and vice versa):

```
XOR AL, 00100000b ; toggle bit 5 of AL
```

The AND instruction is used to clear selected bits of a destination operand, the OR is used to set selected bits, and the XOR instruction is used to complement selected bits.

8.1.3 Cutting and Pasting Bits

The AND and OR instructions can be used together to “cut and paste” selected bits from two or more operands. The following code creates a new byte in the AL register by combining even bits from AL with odd bits from the BL register:

```
AND AL, 55h  ; Clear odd bits of AL (55h = 01010101b)
AND BL, 0AAh ; Clear even bits of BL (0AAh = 10101010b)
OR  AL, BL   ; Paste them together
```

8.1.4 Practice on Bitwise Logical Instructions

Show the value of EAX and flags where indicated:

```
mov  eax, 8A4B401Ch
and  eax, 7C3F89D6h     ; EAX =

mov  eax, 8A4B401Ch
or   eax, 7C3F89D6h     ; EAX =

mov  eax, 8A4B401Ch
xor  eax, 7C3F89D6h     ; EAX =

mov  eax, 8A4B401Ch
not  eax                ; EAX =

mov  eax, 8A4B401Ch
test eax, 0FEh          ; SF =       ZF =        PF =

mov  eax, 8A4B401Ch
bt   eax, 10            ; CF =       EAX =
```

To verify your answers write the above instructions in a program and trace its execution.

8.1.5 String Encryption

The XOR instruction provides an easy way to perform data encryption. A string entered by the user is transformed into an unintelligible string (called *cipher text*) using a *key*. The cipher text can be stored or transmitted to a remote location without unauthorized persons being able to read it. The intended viewer uses a program to decrypt the cipher text and produce the original plain text.

The following *encrypt* procedure uses a technique called *symmetric encryption*, which means that the *same key* is used for both encryption and decryption. The *encrypt* procedure uses the XOR instruction to perform the encryption and decryption of characters in a string. The following example demonstrates the encryption and decryption of character ‘T’.

```
              Encryption                                    Decryption

XOR  01010100 = Original character ‘T’          XOR  01000110 = Encrypted character ‘F’
     00010010 = Encryption Key                       00010010 = Encryption Key
   ------------------------------------            ------------------------------------
     01000110 = Encrypted character ‘F’              01010100 = Original character ‘T’
```

```
encrypt PROC
    push ecx                ; save registers
    push edx
L1:
    xor  [edx], AL          ; encrypt char pointed by edx
    inc  edx                ; point to next character
    loop L1

    pop  edx                ; restore register values
    pop  ecx
    ret                     ; return
encrypt ENDP
```

8.1.6 Assemble, Link, and Trace the Execution of Program *encrypt.asm*

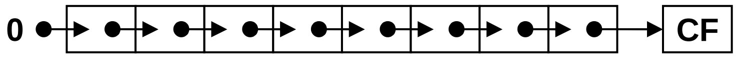
Show the encryption of string "**`Top Secret Message!`**" if the encryption key = **`00010010b`**:

Show the encryption of string "**`Attack at dawn.`**" if the encryption key = **`00010010b`**:

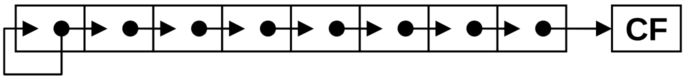
Show the encryption of string "**`Attack at dawn.`**" if the encryption key = **`00011010b`**:

8.2 Shift and Rotate Instructions

Shifting is to move bits left and right inside an operand. There are two basic ways to shift the bits. The first, called a *logical shift*, fills the newly created bit position with zero. In the following diagram, a byte is logically shifted one position to the right. The highest bit is assigned 0 and the bit which is shifted out is stored in the carry flag.

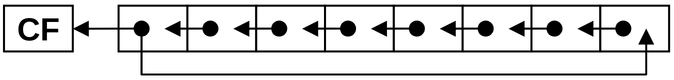


The other type of shift is called an *arithmetic shift*. The newly created bit position is filled with a copy of the original number's sign bit. It works only with *shift arithmetic right*.

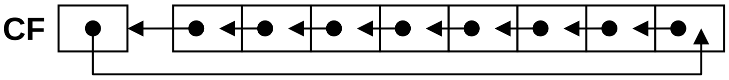


A drawback of the shift instructions is that the bits that are shifted out are lost, except that the last bit shifted out is stored in the carry flag. There may be situations where we want to keep these bits. The rotate instructions can be used instead. They are divided into two types: the normal rotate, and the rotate through the carry flag.

An example of a rotate instruction is ROL (rotate left). This instruction shifts each bit to the left according to a specific count. The highest bit rotates left to become the lowest bit. The last highest bit that was rotated left is stored in the carry flag.



The rotate through carry flag includes the carry flag in the rotation process. For example, the RCL (rotate carry left) instruction rotates the carry flag to become the lowest bit. You can think of the carry flag as an extra bit attached to the number, as if it were the highest bit.



The shift and rotate instructions are listed in the following table. The *destination* can be an 8, 16, or 32-bit operand stored either in a register or in memory. The *count* operand specifies the number of bit positions to be shifted. The *count* can be given as an 8-bit constant value or in the CL register. These instructions modify the *Zero*, *Sign*, and *Parity* flags according to the result value. They modify the *Carry* flag according to the bit that was last shifted out. They set the *Overflow* flag for a single-bit shift or rotate if the sign bit has changed. The *Overflow* flag is undefined if the *count* is greater than 1.

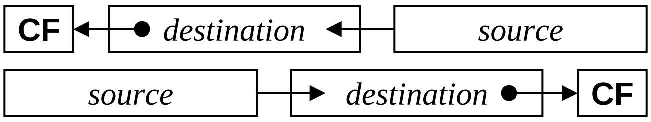
Instruction	**Description**
SHL *destination*, *count*	Shift left (logical). Zero is inserted from the right.
SHR *destination*, *count*	Shift right (logical). Zero is inserted from the left.
SAL *destination*, *count*	Shift arithmetic left. Same as SHL.
SAR *destination*, *count*	Shift arithmetic right. Sign bit is inserted from the left.
ROL *destination*, *count*	Rotate left.
ROR *destination*, *count*	Rotate right.
RCL *destination*, *count*	Rotate carry left.
RCR *destination*, *count*	Rotate carry right.

8.2.1 SHLD and SHRD Instructions

The SHLD and SHRD are double precision shift instructions that operate on three operands. The *destination* and *source* operands can be 16 or 32 bits, and they must be of the same size. The *count* can be an 8-bit constant or the CL register.

SHLD *destination*, *source*, *count*

SHRD *destination*, *source*, *count*



The SHLD/SHRD instructions shift a *destination* operand a given number of bits specified by *count*. The *source* operand supplies the bits that have to be shifted into the *destination* operand. However, the *source* operand is not modified. The last bit shifted out is stored in the *Carry* flag. The *destination* can be a register or memory, but the *source* should be a register.

8.2.2 Practice on Shift and Rotate Instructions

Show the value of CF and EAX after each shift/rotate instruction has executed:

```
mov  eax, 8A4B401Ch
shl  eax, 1          ; CF =     EAX =

mov  eax, 8A4B401Ch
shr  eax, 1          ; CF =     EAX =

mov  eax, 8A4B401Ch
sar  eax, 1          ; CF =     EAX =

mov  eax, 8A4B401Ch
rol  eax, 4          ; CF =     EAX =

mov  eax, 8A4B401Ch
ror  eax, 8          ; CF =     EAX =

stc  ; set carry flag
mov  eax, 8A4B401Ch
rcl  eax, 1          ; CF =     EAX =
stc  ; set carry flag
mov  eax, 8A4B401Ch
rcr  eax, 4            ; CF =      EAX =

mov  eax, 8A4B401Ch
mov  ebx, 7C3F89D6h
shld eax, ebx, 8       ; CF =      EAX =

mov  eax, 8A4B401Ch
mov  ebx, 7C3F89D6h
shrd eax, ebx, 12      ; CF =      EAX =
```

To verify your answers write the above instructions in a program and trace its execution.

8.2.3 Binary Multiplication and Division

Shift operations are very effective in multiplying or dividing binary numbers by a power of 2. In the binary number system, if we want to multiply a number by 2, we simply append a 0 to the right, which is analogous to shifting left by 1 bit. In general, shifting an integer *n* bits to the left multiplies it by 2^n.

Similarly, division by 2 is analogous to shifting right by 1 bit. In general, shifting an integer *n* bits to the right divides it by 2^n. This division process corresponds to integer division, which discards any fractional part of the result. For signed integers, we use *shift arithmetic right* instead of *shift right* to preserve the sign of the integer. The following are examples:

SHL	**AL = 00001011 = +11**	**AL = 11110101 = –11**

SHL AL, 1
SHL AL, 2
SHL AL, 3
	AL = 00010110 = +22
	AL = 00101100 = +44
	AL = 01011000 = +88
		AL = 11101010 = –22
		AL = 11010100 = –44
		AL = 10101000 = –88

SAR	**AL = 01010000 = +80**	**AL = 10110000 = –80**

SAR AL, 1
SAR AL, 2
SAR AL, 3
	AL = 00101000 = +40
	AL = 00010100 = +20
	AL = 00001010 = +10
		AL = 11011000 = –40
		AL = 11101100 = –20
		AL = 11110110 = –10

8.2.4 Practice on Binary Multiplication and Division

Write instructions that calculate EAX × 24, using binary multiplication. Hint: $24 = 2^4 + 2^3$.

Write instructions that calculate EAX × 15, using binary multiplication. Hint: $15 = 2^4 – 2^0$.

Write an instruction that calculates EAX / 16, using signed binary division.

8.2.5 Converting a Number to ASCII Hexadecimal Format

The following procedure converts a 32-bit number stored in the EAX register into ASCII hexadecimal format. It stores the hexadecimal characters in a string passed by reference. The address of the string is passed as a parameter in the EDX register.

A loop is used to traverse all the bits of the EAX register. At the beginning of the loop iteration, the upper 4 bits of EAX are rotated left to become the lowest 4 bits. The ROL instruction is used for this purpose. Then, the AND instruction keeps only the lower 4 bits in EBX by clearing all the remaining bits. These 4 bits are used to index *hexarray*, which converts them into a hexadecimal character. After repeating the loop 8 iterations, all the bits of EAX are traversed and converted. Because the ROL instruction is used in loop L1, the value of the EAX register is brought back to its initial value at the end of the loop.

```
Convert2Hex PROC
    push ebx               ; save registers
    push ecx
    push edx
    mov  ecx, 8            ; 8 iterations
L1:
    rol  eax, 4            ; rotate upper 4 bits of eax
    mov  ebx, eax
    and  ebx, 15           ; keep lower 4 bits in ebx
    mov  bl,  hexarray[ebx] ; convert 4 bits to Hex character
    mov  [edx], bl         ; store Hex char in string
    add  edx, 1            ; point to next char in string
    loop L1

    mov  BYTE PTR [edx], 0  ; Terminate string with a NULL char
    pop  edx               ; restore register values
    pop  ecx
    pop  ebx
    ret                    ; return
    hexarray BYTE "0123456789ABCDEF"
Convert2Hex ENDP
```

8.2.6 Lab Work: Assemble, Link, and Trace Program *convert.asm*

What is the return string of *Convert2Hex* when EAX = 123456789? ....

What is the return string of *Convert2Hex* when EAX = 987654321? ....

8.2.7 Lab Work: Complete the *Convert2Bin* Procedure

Complete the writing of the *Convert2Bin* procedure that converts a number in EAX to ASCII binary format. Test your procedure by calling it from the *main* procedure.

8.3 Integer Multiplication and Division

The Intel instruction set lets you multiply and divide 8-bit, 16-bit, and 32-bit integers using the MUL, IMUL, DIV, and IDIV instructions.

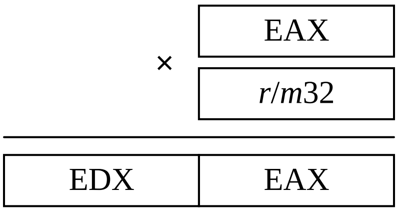
8.3.1 MUL and IMUL Instructions

The MUL (unsigned multiply) instruction multiplies an 8-bit, 16-bit, or 32-bit operand by AL, AX, or EAX. This instruction takes only one operand, which is the multiplier. The multiplicand defaults to the AL, AX, or EAX register. It has the following format:

```
MUL multiplier  ; Multiplicand is AL, AX, or EAX depending on size of multiplier
```

The product is twice the size of the multiplicand and multiplier and is stored in the AX, DX:AX, or EDX:EAX registers respectively. The following table shows the details:

Multiplicand	Multiplier	Product
AL	*r*/*m*8	AX
AX	*r*/*m*16	DX:AX
EAX	*r*/*m*32	EDX:EAX



The *r/m32* notation means that the multiplier should be a 32-bit register or memory operand. MUL sets the Carry and Overflow flags if the upper half of the product is not equal to zero.

The IMUL (integer multiply) instruction performs signed integer multiplication. It has the same syntax and uses the same operands as the MUL instruction. What is different is that it preserves the sign of the product. IMUL sets the Carry and Overflow flags if the upper half of the product is not a sign extension of the lower half.

The IMUL instruction provides two more general-purpose formats:

```
IMUL destination, source
IMUL destination, source, constant
```

In the two- and three-operand formats, the *source* and *destination* must be both either 16-bit or 32-bit operands. In the two-operand format, the result of *destination* × *source* is stored in *destination*. In the three-operand format, the result of *source* × *constant* is stored in *destination*. The result is of the same length as the operands. While *source* can be either in a register or memory, the *destination* must be a register.

8.3.2 Lab Work: Practice on MUL and IMUL Instructions

Guess and show the values of the specified registers and flags:

```
mov  al, -4         ; AL = 0FCh = 252
mov  bl, 4
mul  bl             ; CF =     AX =

mov  al, -4
mov  bl, 4
imul bl             ; OF =     AX =

mov  ax, 2000h
mov  bx, 100h
mul  bx             ; CF =     DX =                 AX =

mov  eax, 12345h
mov  ebx, 1000h
mul  ebx            ; CF =     EDX =                EAX =

mov  ecx, -16
mov  edx, -20
imul ecx, edx       ; OF =     ECX =

mov  ecx, 12345h
imul ebx, ecx, 200h ; OF =     EBX =
```

To verify your answers, assemble, link, and trace the execution of program *mul.asm*.

8.3.3 DIV and IDIV Instructions

The DIV (unsigned divide) instruction performs 8-bit, 16-bit, and 32-bit division on unsigned operands. A single register or memory operand is supplied which is assumed to be the divisor. The dividend is implicit and stored in the AX, DX:AX, or EDX:EAX register and depends on the size of the divisor. The instruction format is given below:

```
DIV divisor ; Dividend is either AX, DX:AX, or EDX:EAX
```

The integer division results in a *quotient* and a *remainder*. The quotient is stored in the AL, AX, or EAX register and the remainder is stored in the AH, DX, or EDX register. The quotient and remainder are determined according to the size of the divisor as shown below:

Dividend	Divisor	Quotient	Remainder
AX	*r*/*m*8	AL	AH
DX:AX	*r*/*m*16	AX	DX
EDX:EAX	*r*/*m*32	EAX	EDX

The following diagram shows the operation of DIV when a 32-bit divisor is used:

$$\frac{\boxed{\text{EDX}}\boxed{\text{EAX}}}{\boxed{r/m32}} = \boxed{\text{EAX (quotient)}}\ \boxed{\text{EDX (remainder)}}$$

The IDIV (integer divide) instruction performs signed integer division, using the same format and operands as the DIV instruction. For both DIV and IDIV, all the arithmetic flags are undefined after the operation.

8.3.4 CBW, CWD, and CDQ Instructions

Before doing signed integer division, the sign of a register must be extended into another register. The CBW (Convert Byte to Word) instruction extends the sign bit of AL into the AH register. The CWD (Convert Word to Double-word) instruction extends the sign bit of AX into the DX register. The CDQ (Convert Double-word to Quad-word) instruction extends the sign bit of EAX into the EDX register.

8.3.5 Lab Work: Practice on Integer Division

Guess and show the values of the specified registers and flags:

```
mov  ax, 0A85h
mov  bl, 10h
div  bl              ; AL =                        AH =

mov  ax, -211
cwd
mov  bx, 2
idiv bx              ; AX =                        DX =

mov  edx, 90h
mov  eax, 12345678h
mov  ecx, 1000h
div  ecx             ; EAX =                       EDX =

mov  eax, -500003
cdq
mov  ebx, 5
idiv ebx             ; EAX =                       EDX =
```

To verify your answers, assemble, link, and trace the execution of program *div.asm*.

8.3.6 Divide Overflow

If a division operand produces a quotient that is too large to fit in the destination register, a *divide overflow* condition results. This causes a CPU interrupt, and the current program halts. For example, the following instructions generate a divide overflow because the quotient (500h) cannot fit in the AL register.

```
mov  ax, 1000h
mov  bl, 2
div  bl             ; Divide overflow: AL cannot hold 500h
```

8.3.7 Converting an Unsigned Integer to ASCII Decimal Format

To convert an unsigned integer to ASCII decimal format, we should divide it by 10 and find the remainder. The remainder will be a number between 0 and 9. It represents the least significant digit. We convert the digit to ASCII format by adding the character '0'. We repeat the DIV instruction until the quotient becomes zero. We collect the digits one by one and store them in a string. Since the least significant digit is computed first and the most significant digit is computed last, we push them on the stack to avoid storing them in the string in reverse order. A second loop is used to pop the digit characters off the stack and store them in the string, as shown in the following *Convert2Dec* procedure:

```
Convert2Dec PROC
    pushad                  ; save all general-purpose registers
    mov  esi, edx           ; ESI = string address
    mov  ecx, 0             ; counts decimal digits
    mov  ebx, 10            ; divisor = 10
L1:
    mov  edx, 0             ; dividend = EDX:EAX
    div  ebx                ; EDX = remainder digit = 0 to 9 (stored in DL)
    add  dl, '0'            ; convert DL to ASCII digit
    push dx                 ; save digit on the stack
    inc  ecx                ; count digit
    cmp  eax, 0
    jnz  L1                 ; loop back if EAX != 0
L2:
    pop  dx                 ; last digit pushed is the most significant
    mov  [esi], dl          ; save ASCII digit in string
    inc  esi
    loop L2

    mov  BYTE PTR [esi], 0  ; Terminate string with a NULL char
    popad                   ; restore all general-purpose registers
    ret                     ; return
Convert2Dec ENDP
```

8.3.8 Lab Work: Complete the *Convert2Int* Procedure

The *Convert2Dec* procedure is written in the *convert.asm* program. Add instructions to the main procedure to call and test *Convert2Dec*. Also, complete the writing of the *Convert2Int* procedure that converts a signed integer in EAX to ASCII format prefixed with sign. Also, test the *Convert2Int* procedure by calling it from the main procedure. To simplify your task, let *Convert2Int* call *Convert2Dec* after checking the sign of EAX. If the number is negative, use the NEG instruction to convert it to positive before calling *Convert2Dec*.

8.4 Multiword Arithmetic

The arithmetic instructions like **add**, **sub**, and **mul** operate on 8-, 16-, and 32-bit operands. What if a program requires number larger than 32 bits? Such program requires arithmetic to be done on multiword operands.

8.4.1 Extended Addition and Subtraction

The ADC (add with carry) instruction adds both a *source* operand and the content of the *carry* flag to a *destination* operand. The SBB (subtract with borrow) instruction subtracts both a *source* operand and the value of the *carry* flag from a *destination* operand. All the arithmetic flags are affected by both instructions.

Instruction	Description
ADC *destination, source*	*destination = destination + source + carry*
SBB *destination, source*	*destination = destination – source – carry*

The procedure *add64* performs addition of two 64-bit numbers in EBX:EAX and EDX:ECX. The result is returned in EBX:EAX. Carry/Overflow conditions are indicated by CF and OF.

```
add64 PROC
    add eax, ecx
    adc ebx, edx
    ret
add64 ENDP
```

The 64-bit subtraction is also simple and similar to the 64-bit addition.

8.4.2 Lab Work: Complete the *extadd* Procedure in *extadd.asm*

The *extadd* procedure in *extadd.asm* generalizes extended addition by operating on arrays of double-words. The procedure receives the address of the source array in ESI, the address of the destination array in EDI, and their length in ECX. Addition should proceed from the least significant double-word, stored as the first array element, to the most significant one, stored as the last array element. The result should be stored in the destination array. Complete the writing of the *extadd* procedure and test its execution by calling it from the *main* procedure.

Review Questions

1. Which instruction sets the upper 8 bits of **eax** without modifying the remaining bits?
2. Which instruction clears the lower 16 bits of **eax** without modifying the remaining bits?
3. Which instruction reverses the lower 10 bits of **eax** without modifying the remaining bits?
4. Which instruction sets the Zero flag if **eax** is even and clears it if **eax** is odd?
5. Using the AND and OR instructions, cut the upper 4 bits of AL and the lower 4 bits of BL and paste them into the BL register.
6. Suppose that the Intel instruction set did not support the NOT instruction. How do you implement NOT using the XOR instruction?
7. How is the IMUL instruction different from MUL in the way it generates a product?
8. When does the IMUL instruction set the Carry and Overflow flags?
9. When BX is the divisor in a DIV instruction, which register holds the quotient?
10. Write the instructions that shift three memory words to the left by 1 bit position:
    ```
    wordarray WORD 810Dh, 0C064h, 93ABh
    ```

Programming Exercises

1. Write a procedure that multiplies any two 16-bit unsigned integers using shifting and addition. The parameters should be passed on the stack. The result should be 32 bits returned in the EAX register. Test your procedure by calling it from the main procedure.
2. Write a procedure that shifts an array of double-word integers using the SHRD instruction. The procedure should receive the address of the array, the length of the array, and the shift amount as parameters. Test your procedure by calling it from *main*.
3. Suppose that we use a 16-bit word to represent a date in binary as follows:

   Bits 0 – 4 are used to represent the day: 1 – 31,

   Bits 5 – 8 are used to represent the month: 1 – 12,

   Bits 9 – 15 are used to represent the year relative to 1980.

   For example, the date September 17, 2006 is represented as:

   2006 is represented as 26 in the year field, relative to 1980.

0011010	**1001**	**10001**
year	*month*	*day*

   Write a procedure that receives a data in binary in the AX register, converts, and returns the date as a string. The string address should be passed as a second parameter.
4. Write a procedure to convert temperature from *Celsius* to *Fahrenheit*. The formula is:

   $$F = \frac{9}{5} \times C + 32$$

5. Write a program to read the length *L*, width *W*, and height *H* of a box from input and displays the *volume* and *surface area* of the box:

   Volume = $L \times W \times H$, *Surface Area* = $2 \times (L \times H + L \times W + W \times H)$
6. Write a procedure to perform ASCII decimal to binary number conversion. The procedure should receive as parameter the address of a string containing the number in ASCII decimal format. It should convert the string to a binary number and return it in EAX.
7. *Challenge:* Write a procedure to perform 64-bit unsigned multiplication. The first 64-bit number is received in the EBX:EAX registers and the second number in the EDX:ECX registers. The 128-bit result should be returned in the EDX:ECX:EBX:EAX registers. Use the MUL instruction for 32-bit unsigned multiplication and accumulate the sum using the ADD and ADC instructions.