Lab: 15

Department of Computer Science

Iqra University Islamabad

Computer Organization and Assembly Language

Maqsood Ahmed

ID: 38186

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 = 8A4B401Ch
mov  eax, 8A4B401Ch

or   eax, 7C3F89D6h      ; EAX = 080B0004h
mov  eax, 8A4B401Ch

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

not  eax                 ; EAX = FCFBC9DEh

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

bt   eax, 10             ; CF = 0     EAX = 8A4B401Ch
```

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

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
    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? = 5D1CB750

What is the return string of *Convert2Hex* when EAX = 987654321? = 1B86EDA3

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.

Source Code:

```
TITLE Convert Number to ASCII Format  (convert.asm)

.686
.MODEL flat, stdcall
.STACK 4096

INCLUDE Irvine32.inc

.data
hexstring   BYTE    9 DUP(?)
binstring   BYTE    33 DUP(?)

.code
main PROC
   mov  eax, 123456789
   mov  edx, OFFSET hexstring
   call Convert2Hex
   call WriteString
   call Crlf

   mov  eax, 123456789
   mov  edx, OFFSET binstring
   call Convert2Bin
   call WriteString
   call Crlf

   mov  eax, 123456789
   mov  edx, OFFSET hexstring
   call Convert2Dec
   call WriteString
   call Crlf

   mov  eax, -123456789
   mov  edx, OFFSET hexstring
   call Convert2Int
   call WriteString
   call Crlf

   exit
main ENDP

; Convert number in EAX to ASCII hexadecimal format
; Store hexadecimal characters in the string passed by reference
```

```
; Receives: EAX = 32-bit number
;          EDX = string address
; Returns:  store converted hexadecimal characters

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

; Convert number in EAX to ASCII binary format
; Store '0' and '1' characters in the string passed by reference
; Receives: EAX = 32-bit number
;          EDX = string address
; Returns:  store converted binary characters

Convert2Bin PROC
   push ebx              ; save registers
   push ecx
   push edx

   mov  ecx, 32          ; 32 bits to process

L1:
   shl  eax, 1           ; shift left, moving MSB into the carry flag
   mov  bl, '0'          ; default character is '0'
   jc   SetOne           ; if carry flag is set, the bit was 1
   jmp  StoreBit
```

```
SetOne:
   mov  bl, '1'          ; set character to '1'

StoreBit:
   mov  [edx], bl        ; store bit character in string
   add  edx, 1           ; move to the next position
   loop L1

   mov  BYTE PTR [edx], 0  ; Terminate string with a NULL char
   pop  edx              ; restore register values
   pop  ecx
   pop  ebx
   ret                   ; return
Convert2Bin ENDP

; Convert unsigned number in EAX to ASCII decimal format
; Receives: EAX = 32-bit number
;          EDX = string address
; Returns:  Store characters in the string passed by reference

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

; Convert signed number in EAX to ASCII integer format prefixed with sign
```

```
; Receives: EAX = 32-bit number
;           EDX = string address
; Returns:  Store characters in the string passed by reference

Convert2Int PROC
   push ebx              ; save registers
   push ecx
   push edx
   push esi

   mov  esi, edx         ; ESI = string address
   test eax, eax         ; check if the number is negative
   jns  Positive         ; jump if not negative

   neg  eax              ; negate the number to make it positive
   mov  byte ptr [esi], '-' ; store '-' sign
   inc  esi              ; move to the next position

Positive:
   call Convert2Dec      ; convert the positive number to decimal

   pop  esi              ; restore register values
   pop  edx
   pop  ecx
   pop  ebx
   ret                   ; return
Convert2Int ENDP

END main
```

Source Code:

```
TITLE Demonstrating Multiplication Instructions  (mul.asm)

.686
.MODEL flat, stdcall
.STACK 4096

INCLUDE Irvine32.inc

.data

.code
main PROC
    mov  al, -4            ; AL = 0FCh = 252
    mov  bl, 4
    mul  bl                ; CF = 0  AX = 03F0h

    mov  al, -4            ; AL = 0FCh = -4 (signed)
    mov  bl, 4
    imul bl                ; OF = 0  AX = FFF0h

    mov  ax, 2000h
    mov  bx, 100h
    mul  bx                ; CF = 0  DX = 0020h  AX = 0000h

    mov  eax, 12345h
    mov  ebx, 1000h
    mul  ebx               ; CF = 0  EDX = 0     EAX = 12345000h

    mov  ecx, -16
    mov  edx, -20
    imul ecx, edx          ; OF = 0  ECX = 320

    mov  ecx, 12345h
    imul ebx, ecx, 200h    ; OF = 0  EBX = 2468A00h

    exit
main ENDP
END main
```

Source Code:

```
TITLE Integer Multiplication and Division (div.asm)

.686
.MODEL flat, stdcall
.STACK 4096

INCLUDE Irvine32.inc

.data

.code
main PROC
    ; 8-bit Unsigned Division
    mov  ax, 0A85h        ; AX = 0A85h = 2693
    mov  bl, 10h          ; BL = 10h = 16
    div  bl               ; AL = 0A8h (168)    AH = 05h (5)
    call DumpRegs

    ; 16-bit Signed Division
    mov  ax, -211         ; AX = FFF3h = -211 (signed)
    cwd                   ; Sign extend AX into DX:AX => DX:AX = FFFF FFF3h
    mov  bx, 2            ; BX = 2
    idiv bx               ; AX = FF97h (-105)  DX = FFFFh (-1)
    call DumpRegs

    ; 32-bit Unsigned Division
    mov  edx, 90h         ; EDX = 90h = 144
    mov  eax, 12345678h   ; EAX = 12345678h
    mov  ecx, 1000h       ; ECX = 1000h = 4096
    div  ecx              ; EAX = 4AAL (305419)  EDX = 378h (888)
    call DumpRegs

    ; 32-bit Signed Division
    mov  eax, -500003     ; EAX = FFF85EDDh = -500003 (signed)
    cdq                   ; Sign extend EAX into EDX:EAX => EDX = FFFFFFFFh
    mov  ebx, 5           ; EBX = 5
    idiv ebx              ; EAX = FFFFE796h (-100000)  EDX = FFFFFFFDh (-3)
    call DumpRegs
```

```
    exit
main ENDP
END main
```

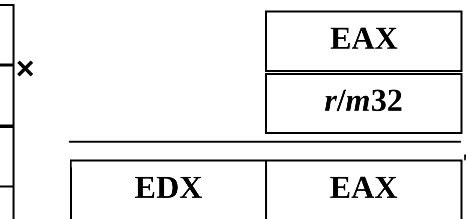
8.3.1 MUL and IMUL Instructions

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.

8.3.3 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:

EDX	EAX

———————————————— = EAX (quotient)   EDX (remainder)

***r/m*32**

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.

8.3.8 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.

Review Questions

Let's answer each of the review questions one by one:

1. Which instruction sets the upper 8 bits of EAX without modifying the remaining bits?
   - The instruction `MOV AH, value` sets the upper 8 bits of EAX (which are part of the 16-bit AX register).

2. Which instruction clears the lower 16 bits of EAX without modifying the remaining bits?
   - The instruction `AND EAX, FFFF0000h` 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?
   - To reverse the lower 10 bits of EAX, you would need a sequence of instructions since there is no single instruction to do this directly. It requires a bit manipulation algorithm.

4. Which instruction sets the Zero flag if EAX is even and clears it if EAX is odd?
   - The instruction `TEST EAX, 1` sets the Zero flag if EAX is even and clears it if EAX is odd. The `TEST` instruction performs a bitwise AND between EAX and 1, affecting the Zero flag.

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.

```assembly
AND AL, 0Fh       ; Clear the upper 4 bits of AL
AND BL, 0F0h      ; Clear the lower 4 bits of BL
OR  BL, AL        ; Combine the lower 4 bits of AL with the upper 4 bits of BL
```

6. Suppose that the Intel instruction set did not support the NOT instruction. How do you implement NOT using the XOR instruction?

- To implement `NOT` using `XOR`, you can use the instruction `XOR operand, 0FFFFFFFFh`. This will flip all bits of the operand.

```assembly
XOR EAX, 0FFFFFFFFh
```

7. How is the IMUL instruction different from MUL in the way it generates a product?

- `IMUL` is used for signed multiplication, whereas `MUL` is used for unsigned multiplication. `IMUL` preserves the sign of the product, while `MUL` does not.

8. When does the IMUL instruction set the Carry and Overflow flags?

- `IMUL` sets the Carry and Overflow flags when the result of the multiplication does not fit in the destination register (i.e., when there is a signed overflow).

9. When BX is the divisor in a DIV instruction, which register holds the quotient?

- When `BX` is the divisor in a `DIV` instruction, the quotient is stored in the `AX` register if the division is 16-bit. For 32-bit division with `EDX:EAX`, the quotient is stored in `EAX`.

10. Write the instructions that shift three memory words to the left by 1 bit position:

```assembly
mov ax, [wordarray]       ; Load the first word
shl ax, 1                 ; Shift left by 1
mov [wordarray], ax       ; Store back

mov ax, [wordarray+2]     ; Load the second word
shl ax, 1                 ; Shift left by 1
mov [wordarray+2], ax     ; Store back

mov ax, [wordarray+4]     ; Load the third word
shl ax, 1                 ; Shift left by 1
mov [wordarray+4], ax     ; Store back
```

```
.data
wordarray WORD 810Dh, 0C064h, 93ABh

.code
main PROC
    ; Shift the first word
    mov ax, [wordarray]       ; Load the first word
```

```
    shl ax, 1               ; Shift left by 1
    mov [wordarray], ax     ; Store back

    ; Shift the second word
    mov ax, [wordarray+2]   ; Load the second word
    shl ax, 1               ; Shift left by 1
    mov [wordarray+2], ax   ; Store back

    ; Shift the third word
    mov ax, [wordarray+4]   ; Load the third word
    shl ax, 1               ; Shift left by 1
    mov [wordarray+4], ax   ; Store back

    exit
main ENDP
END main
```

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.**

```
TITLE Multiply Two 16-bit Unsigned Integers (mult_16bit.asm)

.686
.MODEL flat, stdcall
.STACK 4096

INCLUDE Irvine32.inc

.code
Multiply PROC
    ; Receives: two 16-bit unsigned integers on the stack
    ; Returns: 32-bit result in EAX

    push ebx
    push ecx
    push edx
```

```
    ; Retrieve parameters from stack
    mov ax, [esp+16]  ; First parameter
    mov bx, [esp+12]  ; Second parameter

    xor edx, edx      ; Clear EDX (will be used to accumulate result)
    mov cx, 16        ; Loop counter (16 bits)

    ; Multiplication loop
MulLoop:
    shr ax, 1         ; Shift right the first parameter
    jnc SkipAdd       ; If carry is clear, skip addition
    add edx, bx       ; Add second parameter to result

SkipAdd:
    shl ebx, 1        ; Shift left the second parameter
    loop MulLoop      ; Repeat 16 times

    ; Result is in EDX
    mov eax, edx

    pop edx
    pop ecx
    pop ebx
    ret
Multiply ENDP

main PROC
    push 1234h
    push 5678h
    call Multiply
    call WriteInt     ; Display the result in EAX
    call Crlf

    exit
main ENDP
END main
```

Exercise 2: Shift Array of Double-Word Integers

```
TITLE Shift Array of Double-Word Integers (shift_array.asm)

.686
```

```
.MODEL flat, stdcall
.STACK 4096

INCLUDE Irvine32.inc

.data
array DWORD 12345678h, 9ABCDEF0h, 0FEDCBA9h, 87654321h

.code
ShiftArray PROC
   ; Receives: ESI = address of the array
   ;          ECX = length of the array (number of elements)
   ;          EBX = shift amount

   push edi
   push ebp

   mov edi, esi       ; Copy array address to EDI

ShiftLoop:
   mov eax, [edi]     ; Load the current double-word
   shrd eax, [edi+4], bl ; Shift right by shift amount, pulling bits from the next double-
word
   mov [edi], eax     ; Store the result back
   add edi, 4         ; Move to the next double-word
   loop ShiftLoop     ; Repeat for each element

   pop ebp
   pop edi
   ret
ShiftArray ENDP

main PROC
   mov esi, OFFSET array
   mov ecx, 4         ; Number of elements
   mov ebx, 2         ; Shift amount
   call ShiftArray

   exit
main ENDP
END main
```
```````
```````

Exercise 3: Convert Date from Binary to String

```assembly
TITLE Convert Binary Date to String (date_to_string.asm)

.686
.MODEL flat, stdcall
.STACK 4096

INCLUDE Irvine32.inc

.data
buffer BYTE 20 DUP(0)

.code
DateToString PROC
    ; Receives: AX = date in binary
    ;           EDX = address of the buffer

    push eax
    push ebx
    push ecx
    push edx

    ; Extract day
    mov cx, ax
    and cx, 1Fh       ; Mask to get day (bits 0-4)
    movzx ebx, cx
    call WriteDec
    mov BYTE PTR [edx], ' '
    inc edx

    ; Extract month
    mov cx, ax
    shr cx, 5
    and cx, 0Fh       ; Mask to get month (bits 5-8)
    movzx ebx, cx
    call WriteDec
    mov BYTE PTR [edx], ' '
    inc edx

    ; Extract year
    mov cx, ax
```

```
    shr cx, 9
    add cx, 1980      ; Add 1980 to year (bits 9-15)
    movzx ebx, cx
    call WriteDec

    mov BYTE PTR [edx], 0 ; Null-terminate the string

    pop edx
    pop ecx
    pop ebx
    pop eax
    ret
DateToString ENDP

main PROC
    mov ax, 09E7h
    mov edx, OFFSET buffer
    call DateToString
    call WriteString   ; Display the result
    call Crlf

    exit
main ENDP
END main
```

Exercise 4: Convert Celsius to Fahrenheit

```assembly
TITLE Convert Celsius to Fahrenheit (celsius_to_fahrenheit.asm)

.686
.MODEL flat, stdcall
.STACK 4096

INCLUDE Irvine32.inc

.data

.code
CelsiusToFahrenheit PROC
    ; Receives: AX = temperature in Celsius
    ; Returns: EAX = temperature in Fahrenheit
```

```
    push ebx

    mov ebx, eax      ; Copy Celsius temperature to EBX
    imul ebx, 9       ; Multiply by 9
    add ebx, 5        ; Add 5 for rounding
    sar ebx, 1        ; Divide by 2 (shift right 1 bit)
    add ebx, 32       ; Add 32 to complete conversion
    mov eax, ebx      ; Copy result to EAX

    pop ebx
    ret
CelsiusToFahrenheit ENDP

main PROC
    mov ax, 100       ; Example temperature in Celsius
    call CelsiusToFahrenheit
    call WriteInt     ; Display the result in Fahrenheit
    call Crlf

    exit
main ENDP
END main
```

Exercise 5: Volume and Surface Area of a Box

```assembly
TITLE Volume and Surface Area of a Box (box_dimensions.asm)

.686
.MODEL flat, stdcall
.STACK 4096

INCLUDE Irvine32.inc

.data

.code
main PROC
    ; Example input values
    mov eax, 10       ; Length L
```

```
    mov ebx, 5        ; Width W
    mov ecx, 3        ; Height H

    ; Calculate volume = L * W * H
    imul eax, ebx
    imul eax, ecx
    call WriteString
    call Crlf

    ; Calculate surface area = 2 * (L * H + L * W + W * H)
    mov eax, 10       ; Length L
    mov ebx, 5        ; Width W
    mov ecx, 3        ; Height H

    mov edx, eax
    imul edx, ecx     ; L * H
    add eax, ebx      ; L + W
    imul eax, ecx     ; (L + W) * H
    add eax, edx      ; + L * H
    shl eax, 1        ; * 2
    call WriteString
    call Crlf

    exit
main ENDP
END main
```

Exercise 6: ASCII Decimal to Binary Conversion

```assembly
TITLE ASCII Decimal to Binary (ascii_to_binary.asm)

.686
.MODEL flat, stdcall
.STACK 4096

INCLUDE Irvine32.inc

.data
decimalString BYTE "12345", 0

.code
```

```
AsciiToBinary PROC
   ; Receives: ESI = address of ASCII string
   ; Returns: EAX = binary number

   push ebx
   push ecx

   xor eax, eax      ; Clear EAX (result accumulator)
   xor ebx, ebx      ; Clear EBX (digit accumulator)

ConvertLoop:
   movzx ecx, BYTE PTR [esi] ; Load byte from string
   test ecx, ecx     ; Check for null terminator
   jz ConvertDone

   sub ecx, '0'      ; Convert ASCII to digit
   imul eax, eax, 10  ; Multiply current result by 10
   add eax, ecx      ; Add new digit
   inc esi           ; Move to next character
   jmp ConvertLoop

ConvertDone:
   pop ecx
   pop ebx
   ret
AsciiToBinary ENDP

main PROC
   mov esi, OFFSET decimalString
   call AsciiToBinary
   call WriteInt     ; Display the result in EAX
   call Crlf

   exit
main ENDP
END main
```

Exercise 7: 64-bit Unsigned Multiplication

```assembly
TITLE 64-bit Unsigned Multiplication (multiply_64bit.asm)
```

```
.686
.MODEL flat, stdcall
.STACK 4096

INCLUDE Irvine32.inc

.data

.code
Multiply64 PROC
    ; Receives: EBX:EAX = first 64-bit number
    ;           EDX:ECX = second 64-bit number
    ; Returns:  EDX:ECX:EBX:EAX = 128-bit result

    push esi
    push edi

    ; Clear high result registers
    xor edi, edi
    xor esi, esi

    ; Multiply low parts
    mul ecx             ; EAX * ECX =>
```

THE END