Counters

Experiment #6

By:

Pedro Cabrera

Submitted to:

Dr. V. Rajaravivarma

CET-2123C Fundamentals of Microprocessors

Department of Electronics Engineering Technology

Valencia College – West Campus

2/24/2024

INTRODUCTION:

The objective of this laboratory exercise is to deepen our understanding of the x86 microprocessor by writing a series of counter programs. The exercise will enhance our proficiency with the x86 instruction set, particularly in manipulating registers and implementing loop constructs to perform repetitive tasks. By creating counters that increment and decrement through a range of values, we aim to grasp the practical applications of assembly language in controlling program flow and managing data operations at a granular level. This lab will also refine our ability to debug and test assembly programs, ensuring they operate as intended across various scenarios.

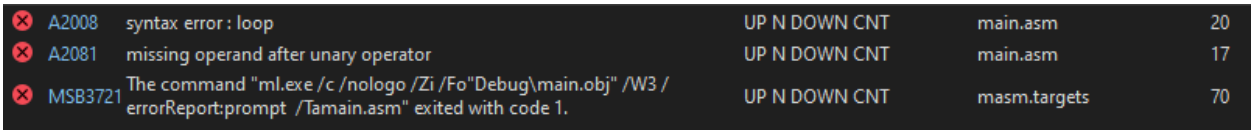
PARTS LIST:

- Windows based computer
- Visual Studio Code 2022

DISCUSSION:

PROGRAM 1 – 8-BIT UP AND DOWN COUNTER

```
1   .MODEL FLAT
2   .DATA
3   startnum DB 0
4   endnum DB 255
5
6   .CODE
7   main PROC
8       ;----- CLEAR CONTENTS OF REGISTERS -----
9       xor eax, eax
10      xor ebx, ebx
11      xor ecx, ecx
12      xor edx, edx
13      ;----- BEGIN MAIN PROCEDURE HERE -----
14
15      mov AL, startnum;
16  loop:
17      inc AL
18      cmp AL, endnum
19      jl loop
20
21      ;----- END MAIN PROCEDURE HERE -----
22      ret
23  main ENDP
24
25  END
```



These are some of the changes I had to perform in order for the code to properly run, and also to properly execute its main purpose which is to go from 0 to 255 using a loop.

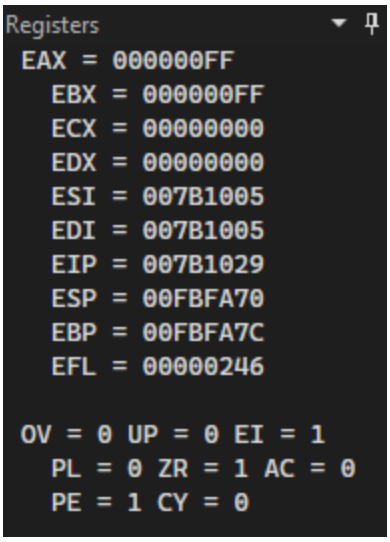
Here are the changes I made along with a brief explanation of the in between code segments:

1. **mov AL, startnum**
    - This line moves the value from the memory location labeled **startnum** into the **AL** register. It's setting up the initial value of **AL** for the loop.
2. **mov BL, endnum**
    - The value from the memory location labeled **endnum** is moved into the **BL** register. This sets the limit for the loop, as **BL** will hold the value up to which **AL** is incremented.
3. **my_loop:**
    - This is a label that marks the beginning of the loop. Labels are used as targets for jump instructions.
    - The label name was changed to **my_loop** to avoid conflict with any reserved words or previously used labels. It's a common practice to choose clear and unique labels that do not conflict with instruction mnemonics or reserved keywords.
4. **inc AL**
    - This instruction increments the value in **AL** by 1. It's the main operation of the loop, incrementing **AL** each time the loop runs.
5. **cmp AL, BL**
    - This instruction compares the values in **AL** and **BL**. It sets the flags in the FLAGS register based on the result of **AL - BL**.
6. **jb my_loop**
    - This instruction checks the Carry Flag (CF), which is set if the result of the last arithmetic operation (the **cmp** instruction) generated a borrow (in other words, if **AL** is less than **BL**). If the CF is set, indicating that **AL** is below **BL**,

the **jb** instruction will jump back to the **my_loop** label, causing the loop to repeat.

- The use of **jb** instead of **jl**. This is an important change because **jb** (jump if below) is for unsigned comparisons, and since **AL** and **BL** are being treated as unsigned bytes (0-255), **jb** is the appropriate instruction to use.

```
1   .MODEL FLAT
2   .DATA
3   startnum DB 0
4   endnum DB 255
5
6   .CODE
7   main PROC
8       ;----- CLEAR CONTENTS OF REGISTERS -----
9       xor eax, eax
10      xor ebx, ebx
11      xor ecx, ecx
12      xor edx, edx
13      ;----- BEGIN MAIN PROCEDURE HERE -----
14
15      mov AL, startnum
16      mov BL, endnum
17  my_loop:
18      inc AL
19      cmp AL, BL
20      jb my_loop ; Changed jl to jb since numbers are unsigned
21
22      ;----- END MAIN PROCEDURE HERE -----
23      ret
24  main ENDP
25
26  END
27
```



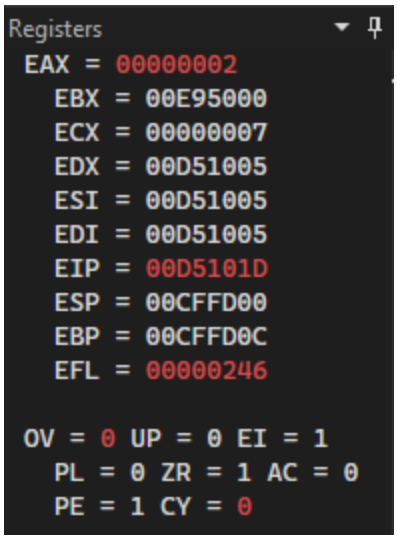
PROGRAM 2 – RING COUNTER

In this laboratory exercise, I constructed an assembly program to demonstrate the operation of an 8-bit ring counter using the **ROL** (rotate left) instruction. The objective was to familiarize myself with bitwise operations and loop constructs in the x86 assembly language within a 32-bit flat memory model environment. The program initializes by clearing the **EAX** register and then setting the **AL** register with a high-order bit as '1'. This initial state represents the starting point of the ring counter with a single '1' in an otherwise zero-filled byte.

The core of the program is a loop that executes eight times, equal to the number of bits in the **AL** register. Each iteration performs a left rotation on **AL**, effectively moving the '1' bit through each bit position in a cyclic fashion. The **ECX** register acts as a counter, which is decremented after each loop iteration by the **loop** instruction. To facilitate observation of the program's execution, a **NOP** instruction is strategically placed at the end of the loop, which acts as a convenient breakpoint during debugging. This allows for the inspection of the **AL** register's state at each stage without producing console output. The exercise successfully demonstrated how the ring counter cycles through all bit positions, providing a clear understanding of register manipulation and the functioning of conditional loops in assembly language.

```
.MODEL FLAT
.DATA
.CODE
main PROC
    ;----- CLEAR CONTENTS OF REGISTERS -----
    xor eax, eax  ; Clear the EAX register, which includes AL.
    xor ebx, ebx
    xor ecx, ecx  ; Clear the ECX register for loop counter.
    xor edx, edx

    mov al, 80h   ; Initialize AL with '10000000b' (the high bit set).
    mov ecx, 8    ; Set up a counter for 8 rotations, one for each bit in AL.

rotate_loop:
    rol al, 1     ; Rotate left the bits in AL.
    ; The above instruction will rotate the '1' bit in AL to the left each time.
    ; The 'rol' instruction rotates the bits through the carry flag.
    nop; This will allow me to see the results.
    loop rotate_loop  ; Decrement ECX and loop if ECX is not zero.

    ; The loop will run 8 times, and the state of AL can be observed in a debugger.

    ;----- END MAIN PROCEDURE HERE -----

main ENDP
END main
```

```
Registers
EAX = 00000001
  EBX = 00E95000
  ECX = 00000008
  EDX = 00D51005
  ESI = 00D51005
  EDI = 00D51005
  EIP = 00D5101D
  ESP = 00CFFD00
  EBP = 00CFFD0C
  EFL = 00000A47

OV = 1 UP = 0 EI = 1
  PL = 0 ZR = 1 AC = 0
  PE = 1 CY = 1
```


Registers
EAX = 00000002
EBX = 00E95000
ECX = 00000007
EDX = 00D51005
ESI = 00D51005
EDI = 00D51005
EIP = 00D5101D
ESP = 00CFFD00
EBP = 00CFFD0C
EFL = 00000246
OV = 0 UP = 0 EI = 1
PL = 0 ZR = 1 AC = 0
PE = 1 CY = 0

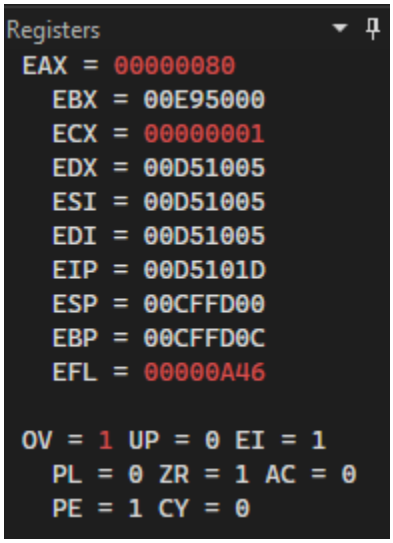

EAX = 00000004


EAX = 00000008


EAX = 00000010


EAX = 00000020


EAX = 00000040


Registers
EAX = 00000080
EBX = 00E95000
ECX = 00000001
EDX = 00D51005
ESI = 00D51005
EDI = 00D51005
EIP = 00D5101D
ESP = 00CFFD00
EBP = 00CFFD0C
EFL = 00000A46
OV = 1 UP = 0 EI = 1
PL = 0 ZR = 1 AC = 0
PE = 1 CY = 0

ANSWERS TO LAB QUESTIONS:

Q1 Why didn't this program run as entered? Why? What did you have to change to make it run? Note: This kind of mistake is very common when writing programs if you aren't careful with the words you choose for labels.

The initial program encountered an issue due to the use of the **jl** instruction within the loop. This instruction is designed for signed comparisons, which was not suitable for the task, as the values being compared were unsigned bytes. This could lead to erroneous behavior when the **AL** register exceeded 127, which is the maximum for a signed 8-bit integer, causing the loop to terminate prematurely.

In the revised program, the issue was addressed by replacing the **jl** instruction with **jb**, which stands for "jump if below" and is appropriate for unsigned comparisons. This ensures that the loop behaves as intended, incrementing **AL** until it is no longer below **BL**. The loop label was also changed from **loop** to **my_loop** to avoid confusion with the **loop** instruction in x86 assembly, which is a reserved keyword. Using clear and non-conflicting labels is a critical aspect of good programming practice and helps in preventing such issues.

Q2 Did you set a breakpoint and step through all 256 steps (from zero to 255)? After you get partway through the program with the breakpoint set on each loop, would it make sense to move the breakpoint to the "ret" instruction and just check the final result in EAX?

Setting a breakpoint at each iteration allows for a detailed examination of the loop's behavior, but it can be time-consuming. After confirming the loop works correctly for several steps, it's efficient to move the breakpoint to the **ret** instruction to quickly check the final result in **EAX**. This method assumes the loop continues to operate correctly after the initial checks.

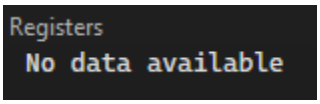
Q3 The EAX register is counting in hexadecimal, but the endnum is defined in decimal, to what value would you change it to make it hexadecimal? Try it to make sure your program still runs.

```
1   .MODEL FLAT
2   .DATA
3   startnum DB 0
4   endnum DB 0FFh
5
6   .CODE
7   main PROC
8       ;----- CLEAR CONTENTS OF REGISTERS -----
9       xor eax, eax
10      xor ebx, ebx
11      xor ecx, ecx
12      xor edx, edx
13      ;----- BEGIN MAIN PROCEDURE HERE -----
14
15      mov AL, startnum
16      mov BL, endnum
17  my_loop:
18      inc AL
19      cmp AL, BL
20      jb my_loop ; Changed jl to jb since numbers are unsigned
21
22      ;----- END MAIN PROCEDURE HERE -----
23      ret
24  main ENDP
25
26  END
27
```

```
EAX = 000000FF
  EBX = 000000FF
  ECX = 00000000
  EDX = 00000000
  ESI = 009C1005
  EDI = 009C1005
  EIP = 009C1029
  ESP = 00EFFC00
  EBP = 00EFFC0C
  EFL = 00000246

OV = 0 UP = 0 EI = 1
  PL = 0 ZR = 1 AC = 0
  PE = 1 CY = 0
```

Q4 Make your counter count by two, instead of one. Does your program operate as expected (i.e., did you have any problem when it got to the end with the jump instruction you used?)?

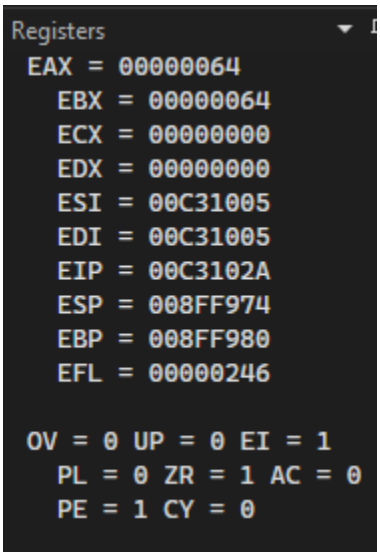
```
1   .MODEL FLAT
2   .DATA
3   startnum DB 0
4   endnum DB 0FFh
5
6   .CODE
7   main PROC
8       ;----- CLEAR CONTENTS OF REGISTERS -----
9       xor eax, eax
10      xor ebx, ebx
11      xor ecx, ecx
12      xor edx, edx
13      ;----- BEGIN MAIN PROCEDURE HERE -----
14
15      mov AL, startnum
16      mov BL, endnum
17  my_loop:
18      add al, 2
19      cmp AL, BL
20      jb my_loop ; Changed jl to jb since numbers are unsigned
21
22      ;----- END MAIN PROCEDURE HERE -----
23      ret
24  main ENDP
25
26  END
```



This could be due to the fact that when the count reaches 255, when one more is added it becomes 256, which causes the lower 8 bits of AL to all be 0. Therefore the loop would never end.

Q5 Change your counter to count from zero to 100. What did you have to change?

```
.MODEL FLAT
.DATA
startnum DB 0
endnum DB 100

.CODE
main PROC
    ;----- CLEAR CONTENTS OF REGISTERS -----
    xor eax, eax
    xor ebx, ebx
    xor ecx, ecx
    xor edx, edx
    ;----- BEGIN MAIN PROCEDURE HERE -----

    mov AL, startnum
    mov BL, endnum
my_loop:
    inc AL
    cmp AL, BL
    jb my_loop ; Changed jl to jb since numbers are unsigned

    ;----- END MAIN PROCEDURE HERE -----
    ret
main ENDP
END
```



All that needed to be changed was '**endnum**' to go from 255 to 100.

Q6 Modify your program to count down from the 255 to zero. Make sure your program makes sense to someone else reading it, or you reading it a year from now (i.e., make sure the variable [data] names reflect the correct numbers); is it important to redefine the startnum and endnum, or just simply change their place in the program?

```
1   .MODEL FLAT
2   .DATA
3   startnum DB 255 ; switched places
4   endnum DB 0
5
6   .CODE
7   main PROC
8       ;----- CLEAR CONTENTS OF REGISTERS -----
9       xor eax, eax
10      xor ebx, ebx
11      xor ecx, ecx
12      xor edx, edx
13      ;----- BEGIN MAIN PROCEDURE HERE -----
14
15      mov AL, startnum
16      mov BL, endnum
17  my_loop:
18      dec AL ; changed inc to dec
19      cmp AL, BL
20      ja my_loop ; Changed jl to ja bc now its jump if above.
21
22      ;----- END MAIN PROCEDURE HERE -----
23      ret  ≤ 1ms elapsed
24  main ENDP
25  END
```

```
Registers
 EAX = 00000000
   EBX = 00000000
   ECX = 00000000
   EDX = 00000000
   ESI = 00D51005
   EDI = 00D51005
   EIP = 00D51029
   ESP = 010FFA74
   EBP = 010FFA80
   EFL = 00000246

 OV = 0 UP = 0 EI = 1
   PL = 0 ZR = 1 AC = 0
   PE = 1 CY = 0
```

Another option to adjust the code would've been the **jnz** instruction, this action would've saved one line of code since all we cared about was if AL is zero. However, if we wanted AL to decrease until a different amount the code on the screenshot above is the best fit.

Q7 Modify your program to count down from 100 to zero. Was this as simple as it was to change the count up counter from 255 to 100?

```
.MODEL FLAT
.DATA
startnum DB 100 ; switched places
endnum DB 0

.CODE
main PROC
    ;----- CLEAR CONTENTS OF REGISTERS -----
    xor eax, eax
    xor ebx, ebx
    xor ecx, ecx
    xor edx, edx
    ;----- BEGIN MAIN PROCEDURE HERE -----

    mov AL, startnum
    mov BL, endnum
my_loop:
    dec AL ; changed inc to dec
    cmp AL, BL
    ja my_loop ; Changed jl to ja bc now its jump if above.

    ;----- END MAIN PROCEDURE HERE -----
    ret
main ENDP
END
```

```
EAX = 00000064
  EBX = 00000000
  ECX = 00000000
  EDX = 00000000
  ESI = 009C1005
  EDI = 009C1005
  EIP = 009C1023
  ESP = 008FF98C
  EBP = 008FF998
  EFL = 00000246

OV = 0 UP = 0 EI = 1
  PL = 0 ZR = 1 AC = 0
  PE = 1 CY = 0
```

```
EAX = 00000000
  EBX = 00000000
  ECX = 00000000
  EDX = 00000000
  ESI = 009C1005
  EDI = 009C1005
  EIP = 009C1029
  ESP = 008FF98C
  EBP = 008FF998
  EFL = 00000246

OV = 0 UP = 0 EI = 1
  PL = 0 ZR = 1 AC = 0
  PE = 1 CY = 0
```

Yes, this was as simple as it was to change 255 to 100.

Q8 Now that your counter is counting from 100 to zero, make it count from 100 to zero three times. Use another data segment value to specify the number of repetitions of the 100 to zero count.

For this program we just had to do a simple nested loop.

Here are some hints:

a) This program will consist of two separate loops, one inside the other.

b) You should use the ECX register for the outer loop, typically ECX is used as a counter by convention.

c) If you start ECX at zero, your outer loop increment to two, which should be the value in the data segment (for three total loops).

d) If you choose to count down, you might set the data segment value to three, and check for JNE to loop as you decrement ECX.

```
.MODEL FLAT
.DATA
startnum DB 100
endnum DB 0
counter DB 3 ;This will be the number of times the outter loop will repeat.
.CODE
main PROC
    ;----- CLEAR CONTENTS OF REGISTERS -----
    xor eax, eax
    xor ebx, ebx
    xor ecx, ecx
    xor edx, edx
    ;----- BEGIN MAIN PROCEDURE HERE -----

    mov CL, counter

outter_loop:
    mov AL, startnum
    mov BL, endnum

my_loop:
    dec AL ; changed inc to dec
    cmp AL, BL
    ja my_loop ; Changed jl to ja bc now its jump if above.

    dec CL
    jnz outter_loop ;this outter loop will run 3 times after my_loop exits
    ;----- END MAIN PROCEDURE HERE -----
    ret
main ENDP
END
```

For this program we simply introduced an outer lop that will run 3 times after **my_loop** has finished. I declared counter to be able to change that number easily at will.

I chose **JNZ** because it saved some lines of code by just decreasing **CX** and once it'd be zero it would exit.

Q9 Write a program to output the results shown above. Hint: You will use one of the shift instructions to help you with this.

```
.MODEL FLAT
.DATA
.CODE
main PROC
    ;----- CLEAR CONTENTS OF REGISTERS -----
    xor eax, eax  ; Clear the EAX register, which includes AL.
    xor ebx, ebx
    xor ecx, ecx  ; Clear the ECX register for loop counter.
    xor edx, edx

    mov al, 80h   ; Initialize AL with '10000000b' (the high bit set).
    mov ecx, 8    ; Set up a counter for 8 rotations, one for each bit in AL.

rotate_loop:
    rol al, 1     ; Rotate left the bits in AL.
    ; The above instruction will rotate the '1' bit in AL to the left each time.
    ; The 'rol' instruction rotates the bits through the carry flag.
    nop; This will allow me to see the results.
    loop rotate_loop  ; Decrement ECX and loop if ECX is not zero.

    ; The loop will run 8 times, and the state of AL can be observed in a debugger.

    ;----- END MAIN PROCEDURE HERE -----

main ENDP
END main
```

Q10 Modify your program to produce a ring counter with 16-bits instead of 8-bits.

```
.MODEL FLAT
.DATA
.CODE
main PROC
    ;----- CLEAR CONTENTS OF REGISTERS -----
    xor eax, eax  ; Clear the EAX register, which includes AL.
    xor ebx, ebx
    xor ecx, ecx  ; Clear the ECX register for loop counter.
    xor edx, edx

    mov ax, 8000h   ; Initialize AL with '1000000000000000b' (the high bit set).
    mov ecx, 16    ; Set up a counter for 8 rotations, one for each bit in AL.

rotate_loop:
    rol ax, 1     ; Rotate left the bits in AL.
    ; The above instruction will rotate the '1' bit in AL to the left each time.
    ; The 'rol' instruction rotates the bits through the carry flag.
    nop; This will allow me to see the results.
    loop rotate_loop  ; Decrement ECX and loop if ECX is not zero.

    ; The loop will run 16 times, and the state of AL can be observed in a debugger.

    ;----- END MAIN PROCEDURE HERE -----

main ENDP
END main
```

AX is used in place of **AL** to accommodate the 16-bit counter, and the loop counter **ECX** is set to 16 to match the number of bits in **AX**. The **rol** instruction with **ax, 1** will rotate **AX** left by one bit position each time the loop runs, just as it did with **AL**. The **nop** instruction at the end serves as a convenient place to set a breakpoint and observe the final state of **AX** after all rotations are completed in a debugger.

CONCLUSION:

In this lab, we successfully explored the functionality and implementation of counters using assembly language. By constructing and modifying various counter programs, we solidified our understanding of bitwise operations, specifically the **ROL** instruction for rotating bits in a register. We began with a simple 8-bit up-counter, then expanded our scope to include down-counters and ultimately developed a 16-bit ring counter. Each step of the lab built upon the last, reinforcing our skills in writing assembly code and employing debugging techniques to ensure the accuracy of our programs. By the end of the lab, we were able to write assembly programs that manipulated registers to perform precise counting sequences, both upwards and downwards, and to repeat these sequences a specified number of times. Through this exercise, we gained a deeper appreciation for the low-level operations that underpin higher-level programming constructs and furthered our proficiency in assembly language programming.