Series Parallel DC Circuits

Experiment #8

By:

Pedro Cabrera

Submitted to:

Dr. V. Rajaravivarma

CET-2123C Fundamentals of Microprocessors

Department of Electronics Engineering Technology

Valencia College – West Campus

3/8/2024

Introduction:

In this lab, we explore the calculation of total resistance in series and parallel DC circuits using the x86 microprocessor. Despite its limited mathematical capabilities, the exercise demonstrates effective programming strategies to address this classic electrical engineering problem. Through a series of approaches—linear programming, subroutines, and stack manipulation—we aim to illustrate the computational techniques for determining total resistance. This hands-on experience not only enhances understanding of resistive circuits but also hones practical programming skills necessary for electrical engineering applications. The lab thereby serves as a bridge between theoretical concepts and their real-world applications, equipping students with the knowledge and skills to tackle similar challenges.

Parts List:

- Windows based computer
- Visual Studio Code 2022

Discussion:

Program 1- RESISTIVE SERIES AND PARALLEL CIRCUITS AS A LINEAR PROGRAM

1. **Initialization**: The program begins by clearing the contents of the registers **EAX**, **EBX**, **ECX**, and **EDX** to ensure they do not contain any previous data that could affect the calculations.
2. **Series Resistance Calculation**: The code then moves the value of R2 into the **AX** register and adds the value of R3 to it. This sum represents the total series resistance of R2 and R3 because, in a series circuit, resistances simply add together.
3. **Parallel Resistance Calculation**:
    - The value of R1 is added to **BX**, which now holds the total resistance of the series circuit (R2 + R3). This sum is the denominator of the formula for calculating the total resistance of parallel resistors.
    - The code then multiplies R1 by the sum of R2 and R3, storing the result in **AX**. This multiplication gives us the numerator of the parallel resistance formula.

- Finally, the **DIV** instruction divides the numerator in **AX** by the denominator in **BX**, giving us the total parallel resistance, which is stored in **AX**.

4. **Result**: The final resistance value is now in the **AX** register, ready to be used or displayed as needed. The **RET** instruction at the end signals the end of the procedure, allowing the program to exit or continue with further instructions if they were present.

```
.MODEL FLAT
.DATA
R1 DW 100
R2 DW 400
R3 DW 600
.CODE

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

    ; First calculate the series resistance of R2+R3
    mov ax, R2 ; load R2 into AX
    add ax, R3 ; add R3 to AX, now AX contains the total series resistance
    mov bx, ax ; move the sum into BX

    ; Now we will calculate the denominator of the parallel resistance
    add bx, R1 ; add R1 to BX, BX is now the denominator
               ; for the parallel resistance

    ; Now we'll calculate the numerator of the parallel resistance
    mul R1 ; multiply the sum of R2+R3 by R1, now ax contains
           ; the numerator for the parallel resistance

    ; Finally, divide the numerator (AX) by the denominator (BX)
    div bx

    ; AX contains the result

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

END
```

Figure 1

```
EAX = 0000005A
  EBX = 0000044C
  ECX = 00000000
  EDX = 000003E8
  ESI = 00391005
  EDI = 00391005
  EIP = 00391039
  ESP = 0024FAA4
  EBP = 0024FAB0
  EFL = 00000A87

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

Figure 2

Program 2 - RESISTIVE SERIES AND PARALLEL CIRCUITS USING SUBROUTINES

1. **Initialization**: The code begins by zeroing out the general-purpose registers **EAX**, **EBX**, **ECX**, and **EDX** to ensure they start with a clean state.

2. **Series Resistance Calculation**: The program then moves the value of R2 into **AX** and the value of R3 into **BX**. The **series** procedure is called, which adds the contents of **BX** to **AX**, calculating the total series resistance of R2 and R3 and storing the result back into **AX**.

3. **Parallel Resistance Calculation**: After the series calculation, R1's value is moved into **BX** before calling the **parallel** procedure. This procedure performs the parallel resistance calculation by first moving the contents of **BX** into **CX**, then adding **AX** to **CX** to get the denominator of the calculation, and multiplying **BX** by **AX** to get the numerator. The numerator is then divided by the denominator using **CX**, with the result of the division—the total parallel resistance—stored in **AX**.

4. **Result**: Upon completion of the **parallel** procedure, **AX** contains the final calculated total resistance. The program concludes with the **ret** instruction, which indicates the end of the procedure.

```
.MODEL FLAT
.DATA
R1 DW 100
R2 DW 400
R3 DW 600
.CODE

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

    ; First calculate the series resistance of R2+R3
    mov ax, R2     ; load R2 into AX
    mov bx, R3     ; load R3 into BX
    call series    ; the series result is stored in AX on return

    mov bx, R1     ; load R1 into BX
    call parallel ; the parallel result is stored in AX on return

    ; AX contains the result

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

series proc
    add ax, bx     ; AX is the series resistance
    ret
series endp

parallel proc
    mov cx, bx
    add cx, ax     ; DX is the denominator
    mul bx         ; BX is the numerator
    div cx         ; AX is the parallel resistance
    ret
parallel endp
END
```

Figure 3

```
EAX = 0000005A
  EBX = 00000064
  ECX = 0000044C
  EDX = 000003E8
  ESI = 004E100A
  EDI = 004E100A
  EIP = 004E1056
  ESP = 012FFAC0
  EBP = 012FFACC
  EFL = 00000A87

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

Figure 4

Program 3 - RESISTIVE SERIES AND PARALLEL CIRCUITS USING SUBROUTINES AND THE STACK

1. **Initialization**: It starts by zeroing the **EAX**, **EBX**, **ECX**, and **EDX** registers to avoid unexpected results due to leftover data.
2. **Loading Resistors**: The program uses the **ESI** register to point to the start of the resistors list. It then enters a loop, where it uses the **LODSW** instruction to load the word at the current **ESI** pointer into **AX** and increment **ESI** by the size of a word (2 bytes).
3. **Stack Operations**: Each value loaded into **AX** is immediately pushed onto the stack. When a zero value is encountered, signifying the end of the list, the program proceeds to calculate the total parallel resistance.
4. **Parallel Resistance Calculation**: The **parallel** procedure is called with two values on the stack (the result of the previous parallel calculations and the next resistor value). It pops these two values and the return address off the stack, calculates the parallel resistance, and then pushes the result back onto the stack.
5. **Final Result**: After all resistor values have been processed, the final result, which is the total resistance of all resistors in parallel, is popped off the stack into **AX**.
6. **Completion**: The program then exits with the **RET** instruction, leaving the final resistance value in **AX**.

```
.MODEL FLAT
.DATA
; Note, the last resistor value must be zero
resistors DW 100,400,600,0
.CODE

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

    lea esi, resistors ; load the effective address of the first resistor
    lodsw              ; load value at DS:ESI into AX, increment ESI
    push ax            ; Push AX onto the stack
next:
    lodsw              ; load value at DS:ESI into AX, increment ESI
    cmp ax, 0
    jz done            ; If AX is zero, we're at the end of the list
    push ax            ; If AX is not zero, push it onto stack
    ; At this point we have two values on the stack to calculate their parallel resistance
    call parallel      ; Calculate the parallel resistance
                       ; The result is stored on the stack

    jmp next
done:
    pop ax             ; Since we're done, we'll pop the result off the stack
                       ; and place it into AX

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

parallel proc
    pop ebp            ; pop the return address off the stack into EBP
    pop ax             ; pop the first value into AX
    pop bx             ; pop the second value into BX
    ; First calculate the denominator (AX+BX) and store in CX
    mov cx, ax
    add cx, bx
    ; Second calculate the numerator (AX*BX)
    mul bx
    div cx             ; Divide numerator by denominator
    push ax            ; Push result (AX) onto stack
    push ebp           ; Put the return address back onto the stack
    ret
parallel endp

END
```

Figure 5

```
EAX = 00000046 EBX = 00000050
  ECX = 000002A8 EDX = 00000190
  ESI = 00854008 EDI = 0085100A
  EIP = 00851045 ESP = 00D9FF0C
  EBP = 00851041 EFL = 00000246

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

Figure 6

<u>Validation of Data:</u>

Q1 Calculate the RTOTAL using the above equation for the three values listed in the program. Does your value match the value displayed in the AX register at the end of the program's execution?

$$R_T = \frac{(600 + 400) * 100}{(600 + 400) + 100} = 90.91\Omega$$

This value matches the value of AX which is $5A_{HEX}$ or 90_{DEC}.

Q2 Change R2 from 400 to 500 ohms and re-run your program, then calculate R_{TOTAL} by hand. Why does the computer answer differ from your hand calculation?

```
.MODEL FLAT
.DATA
R1 DW 100
R2 DW 500
R3 DW 600
.CODE

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

    ; First calculate the series resistance of R2+R3
    mov ax, R2 ; load R2 into AX
    add ax, R3 ; add R3 to AX, now AX contains the total series resistance
    mov bx, ax ; move the sum into BX

    ; Now we will calculate the denominator of the parallel resistance
    add bx, R1 ; add R1 to BX, BX is now the denominator
               ; for the parallel resistance

    ; Now we'll calculate the numerator of the parallel resistance
    mul R1 ; multiply the sum of R2+R3 by R1, now ax contains
           ; the numerator for the parallel resistance

    ; Finally, divide the numerator (AX) by the denominator (BX)
    div bx

    ; AX contains the result

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

END
```

Figure 7

```
EAX = 0000005B
  EBX = 000004B0
  ECX = 00000000
  EDX = 00000320
  ESI = 00881005
  EDI = 00881005
  EIP = 00881039
  ESP = 010FFB78
  EBP = 010FFB84
  EFL = 00000A83

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

Figure 8

The program outputs $5B_{HEX}$ which is 91_{DEC} which is correct. The calculated value is 91.66 Ω.

Q3 Explain in your own words the benefits of using subroutines.

Subroutines are key to writing clear and maintainable code, allowing programmers to create reusable components that simplify complex operations into single tasks. This modularity not only enhances readability but also minimizes errors by isolating functionality for individual testing and debug. Furthermore, subroutines facilitate easier maintenance and updates without affecting the main body of the program, and support collaborative work by enabling different programmers to manage distinct sections of code. They also contribute to efficient memory usage by avoiding repetitive code and help structure the program logically to align with the problem it solves.

In the specific context of adding a resistor to a program, issues might arise due to several factors like register overflow, stack mismanagement, or syntax errors. These could prevent the program from running correctly, with the precise cause identifiable through error messages during compilation or execution.

Q4 Modify the program by adding an additional resistor (R4) with a value of 100k ohms (100000) and add the two additional lines to calculate this new resistor in parallel with the previous result after "call parallel". (If you need help with the lines to add, they're listed after Q5, but try to write them yourself first!) Why did your program fail to run?

```
.MODEL FLAT
.DATA
R1 DW 100
R2 DW 400
R3 DW 600
R4 DD 100000  ; Declare R4 with a value of 100k ohms
.CODE

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

    ; First calculate the series resistance of R2+R3
    mov ax, R2    ; load R2 into AX
    mov bx, R3    ; load R3 into BX
    call series   ; the series result is stored in AX on return

    mov bx, R1    ; load R1 into BX
    call parallel ; the parallel result is stored in AX on return
    mov ebx, R4      ; Load R4 into BX
    call parallel   ; Calculate the new parallel result with R4
    ; AX contains the result

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

series proc
    add ax, bx    ; AX is the series resistance
    ret
series endp

parallel proc
    mov cx, bx
    add cx, ax    ; DX is the denominator
    mul bx        ; BX is the numerator
    div cx        ; AX is the parallel resistance
    ret
parallel endp

END
```

Figure 9

```
EAX = 00000059
  EBX = 000186A0
  ECX = 000086FA
  EDX = 00006756
  ESI = 009B100A
  EDI = 009B100A
  EIP = 009B1061
  ESP = 012FFAA8
  EBP = 012FFAB4
  EFL = 00000A13

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

Figure 10

The original issue with declaring R4 as a word (**DW**) is due to the 16-bit size limit, which cannot accommodate a value larger than 65,535. Correctly, R4 should be declared as a double word (**DD**) to handle the 100,000-ohm value. To accommodate this larger value, which is a 32-bit number, it's necessary to use the 32-bit register **EBX** instead of the 16-bit **BX**. This ensures the program can handle and calculate the resistance values accurately.

When R4 is declared as a double word to accommodate its 100,000-ohm value, it must be loaded into a 32-bit register like **EBX**, not **BX**, which is only 16 bits. Due to the **div** instruction using integer division, the final result is rounded down, with the remainder discarded. This is why the program's output is one ohm less than the actual calculated value, which would include the remainder if it were considered.

Q5 Change the value of R4 to 10000 and run your program. What is the value of AX this time? Why was the change so small?

The change in AX is relatively small because when resistors are combined in parallel, the total resistance is always less than the smallest individual resistance. Adding another resistor in parallel will decrease the total resistance, but the change becomes less significant if the added resistor's value is large compared to the other resistors in the parallel network. If R4 is changed from 100,000 ohms to 10,000 ohms, it is still quite large compared to the other resistors (R1, R2, R3), hence the small change in the total resistance observed in AX.

Q6: Explain the operation of the PUSH and POP instructions in your own words.

The PUSH instruction is used to place data onto the stack. It decrements the stack pointer and then stores the specified value at the new top location of the stack. Conversely, the POP instruction removes data from the top of the stack; it retrieves the value from the current top of the stack and then increments the stack pointer to its next position.

Q7: Can PUSH and POP be used with any register? Can you PUSH a constant onto the stack? Can you POP to a memory location?

PUSH and POP can be used with general-purpose registers. Yes, you can PUSH a constant onto the stack. However, POP is typically used to retrieve a value from the stack into a register, but it can also be used to POP a value directly to a memory location if the instruction set and processor architecture allow it.

Q8: Realizing that all three resistors are in parallel in this program, calculate the R_TOTAL by hand and compare it to your program's value in AX. Are they the same?

$$R_T = \left(\frac{1}{100} + \frac{1}{400} + \frac{1}{600}\right)^{-1} = 70.58\,\Omega$$

The AX register gave 46_{HEX} Which is 70_{Dec}.

Q9: Add two more values to the resistor list and re-run your program. Recalculate the final result, does your program agree?

```
.MODEL FLAT
.DATA
; Note, the last resistor value must be zero
resistors DW 100,400,600,200,300,0
.CODE
```

Figure 11

```
EAX = 0000002B EBX = 00000033
  ECX = 0000015F EDX = 000000CF
  ESI = 00BF400C EDI = 00BF100A
  EIP = 00BF1045 ESP = 004FFD4C
  EBP = 00BF1041 EFL = 00000246

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

Figure 12

Result was off by 1Ω. The calculated answer was 44.44Ω.

Q10: Run your program again, this time looking at the ESI register each time the LODSW instruction executes, why does the value change? What is the increment at which it is changing? Why does it change at that increment (how many bits is a WORD)?

When the LODSW instruction executes, it loads a word (16 bits) from the address pointed to by ESI into AX and then increments ESI by the size of the word loaded. The value changes because LODSW automatically advances ESI to point to the next word in memory. The increment is 2 bytes because a word is 16 bits, which is equivalent to 2 bytes.

Q11: Fill in the missing elements in the procedure using POP and PUSH to calculate series resistance.

```
series proc
  pop ebp          ; pop the return address off the stack into EBP
  pop ax          ; pop the first value into AX
  pop bx          ; pop the second value into BX
  add ax, bx        ; add BX to AX
  push ax          ; Push result (AX) onto stack
  push ebp          ; Put the return address back onto the stack
  ret
series endp
```

<u>Conclusion:</u>

In this lab on Series and Parallel DC Circuits, I've gained practical insights into the principles of electrical circuits and their simulation through assembly language programming. My experience began with theoretical electrical concepts and extended to their practical implementation in code.

I've learned to manipulate data within registers, use stack operations for dynamic data handling, and harness the power of subroutines to streamline complex tasks, which are crucial skills for low-level programming and microprocessor applications. Translating electrical calculations into assembly language illustrated the necessity for accuracy in coding and brought to light the nuances of integer arithmetic in computational tasks.

The lab exercises sharpened my understanding of stack management and subroutines efficiency and brought to my attention the importance of choosing the correct data types and understanding the limitations of the processor's registers. Comparing the outcomes of my program with hand calculations provided a clear picture of the challenges in numerical computations using assembly language.

To conclude, this lab has solidified my foundation in electrical engineering concepts and their application in the field of computer science. This experience has equipped me with the interdisciplinary skills essential for bridging theoretical knowledge with practical technical execution, a valuable asset for my future projects and professional development.