# EE2016: Microprocessor Lab

# Experiment 7: Computations using Atmel Atmega8 AVR through Assembly Program Emulation

Ajeet E S, EE22B086
Amogh Agrawal, EE22B087

27 September, 2023

## Contents

## List of Listings

## List of Figures

## 1 Aim

To implement arithmetic and logical manipulation programs using Atmel Atmega8 microcontroller in assembly program emulation.

## 2 Google Drive Link

The link to the codes and screenshots for the Experiment are uploaded here:
Code and Screenshots

## 3 Problem 1: Common 8-bit Mathematical Operations

### 3.1 Problem Statement

Given two 8-bit binary words (byte), compute the sum and product, store it in two separate registers.

### 3.2 Approach

We use the ADD directive to add two numbers. The Syntax of this command is:

```
ADD Rd,Rr
```

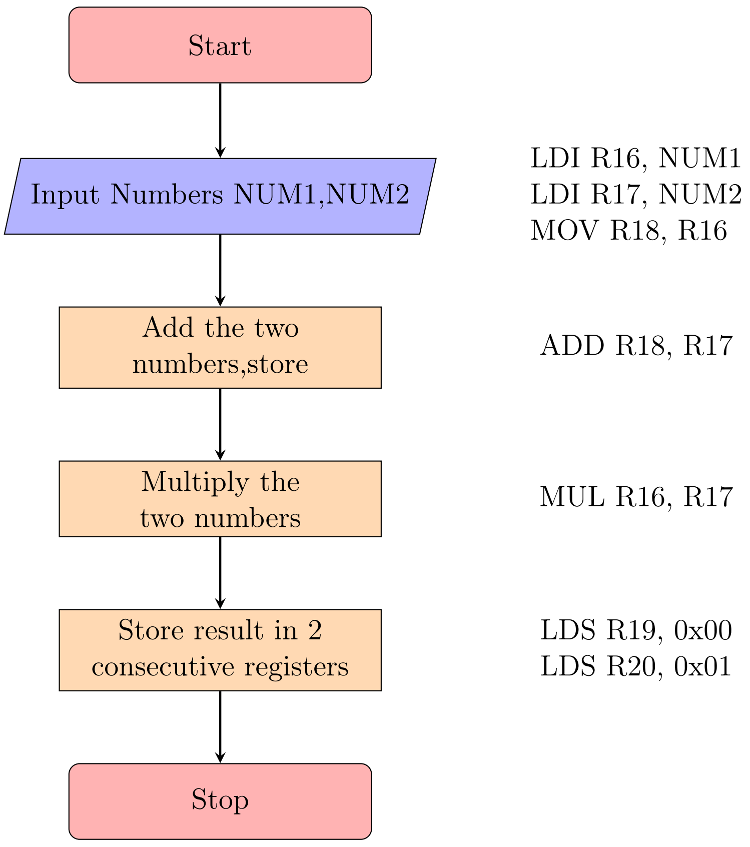
This adds Rd and Rr, and stores the result in Rd.

We use the MUL directive to add two numbers. The Syntax of this command is:

```
MUL Rd,Rr
```

It multiplies both the numbers and stores the Lower byte in R0 and the Higher Byte in R1. These values are then extracted from there and stored in two consecutive registers

## 3.3 Flowchart



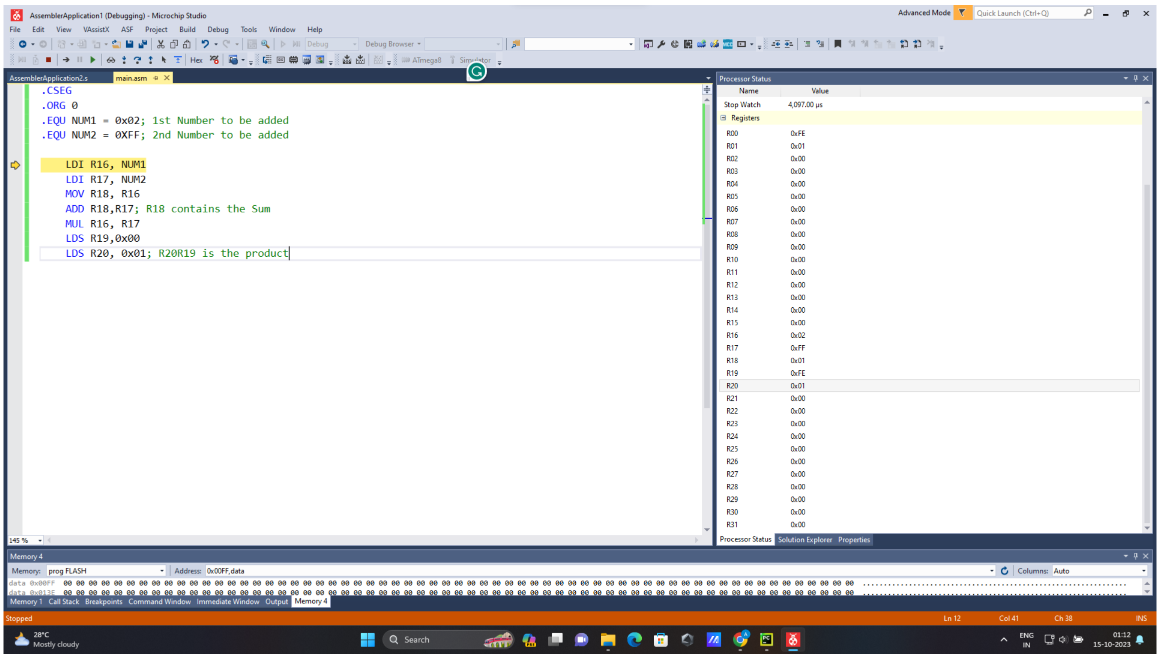
## 3.4 Code

The code used to perform addition and multiplication is given below:

```
.CSEG
.ORG 0
.EQU NUM1 = 0x02; 1st Number to be added
.EQU NUM2 = 0XFF; 2nd Number to be added

        LDI R16, NUM1
        LDI R17, NUM2
        MOV R18, R16
        ADD R18,R17; R18 contains the Sum
        MUL R16, R17
        LDS R19,0x00
        LDS R20, 0x01; R20R19 is the product
```

Listing 1: Code to Add and Multiply two Numbers

## 3.5 Screenshots



```
.CSEG
.ORG 0
.EQU NUM1 = 0x02; 1st Number to be added
.EQU NUM2 = 0XFF; 2nd Number to be added

    LDI R16, NUM1
    LDI R17, NUM2
    MOV R18, R16
    ADD R18,R17; R18 contains the Sum
    MUL R16, R17
    LDS R19,0x00
    LDS R20, 0x01; R20R19 is the product
```

| Name | Value |
|---|---|
| Stop Watch | 4,097.00 µs |
| Registers | |
| R00 | 0xFE |
| R01 | 0x01 |
| R02 | 0x00 |
| R03 | 0x00 |
| R04 | 0x00 |
| R05 | 0x00 |
| R06 | 0x00 |
| R07 | 0x00 |
| R08 | 0x00 |
| R09 | 0x00 |
| R10 | 0x00 |
| R11 | 0x00 |
| R12 | 0x00 |
| R13 | 0x00 |
| R14 | 0x00 |
| R15 | 0x00 |
| R16 | 0x02 |
| R17 | 0xFF |
| R18 | 0x01 |
| R19 | 0xFE |
| R20 | 0x01 |
| R21 | 0x00 |
| R22 | 0x00 |
| R23 | 0x00 |
| R24 | 0x00 |
| R25 | 0x00 |
| R26 | 0x00 |
| R27 | 0x00 |
| R28 | 0x00 |
| R29 | 0x00 |
| R30 | 0x00 |
| R31 | 0x00 |

AssemblerApplication1 (Debugging) - Microchip Studio
File Edit View VAssistX ASF Project Build Debug Tools Window Help
AssemblerApplication2.s
main.asm
Processor Status
Memory: prog FLASH
Address: 0x00FF,data
Stopped

Figure 1: Screenshot for Problem 1

## 3.6 Inferences

We infer there are in-built instructions for addition and multiplication in AVR.

```
ADD R17,R18
MUL R16, R17
```

The MUL statement will store the multiplication contents in the R0 and R1 Registers.

### 3.6.1 Register Transfers

1. Load: Two LDI to initialise R16 and R17, Two LDI to store data to R19 and R20
2. Store: No Store has been done
3. Move: MOV to transfer data from R16 to R18

### 3.6.2 Clock Cycles per Instruction

1. LDI: 1 Clock Cycle
2. MOV: 1 Clock Cycle
3. ADD: 1 Clock Cycle
4. MUL: 2 Clock Cycles
5. LDS: 2 Clock Cycles

### 3.6.3 Latency

The latency of this AVR Assembly Code is $10\mu s$

#### 3.6.4 Throughput

Throughput of a code is defined as:

$$Throughput \equiv \frac{\text{Number of Computations(tasks) Done}}{\text{Time Taken}}$$

Here, we define the task to be the number of (Addition+ Multiplication) per unit time
$Throughput = \frac{1}{10\mu s} = 10^5$ Tasks /s

#### 3.6.5 Limitations

1. This code can only add and multiply 8-bit numbers

## 4 Problem 2: Implementing Division on AVR

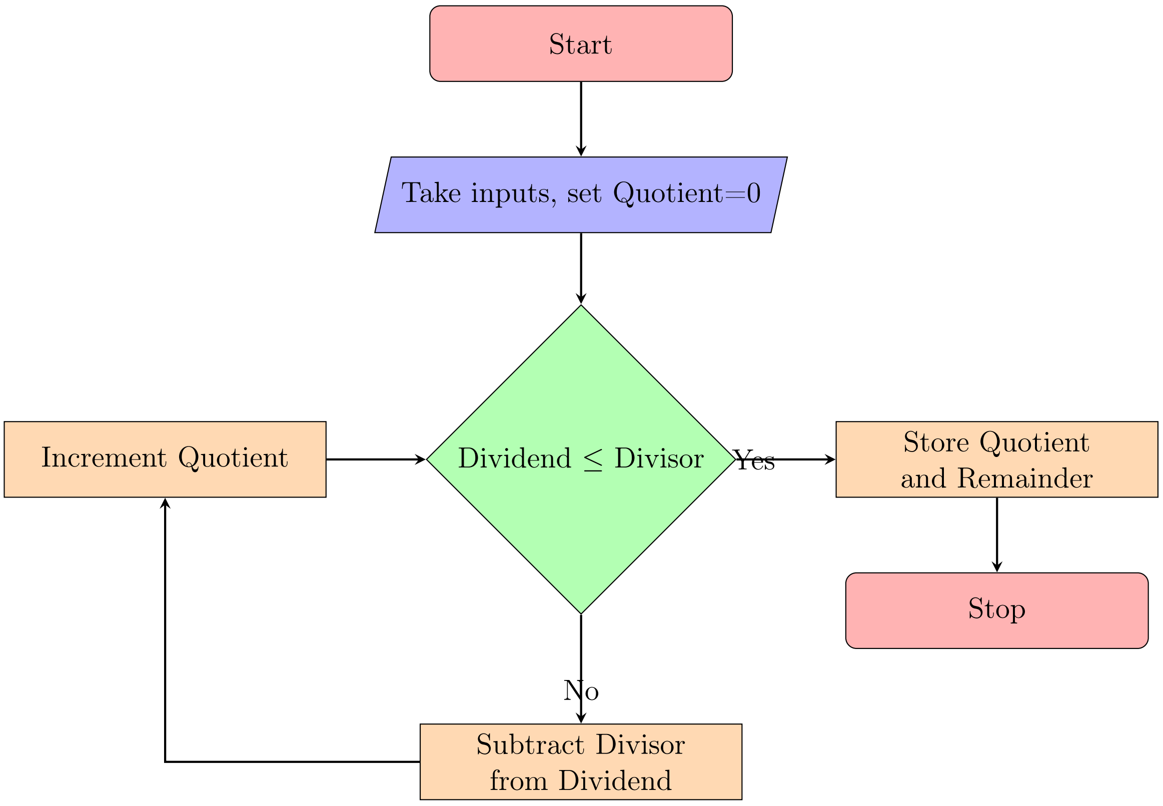
### 4.1 Problem Statement

AVR has no division operation. Implement a code that takes the dividend from memory location A at 0x0E and the divisor from memory location B at 0x10. Note that these locations are not consecutive. You need to set the locations yourself. The TA would enter random numbers at these memory locations. Store the quotient in location 0xF0 and remainder in location 0xFF. For the sake of simplicity, assume that you are implementing unsigned division. (If you identify more than one algorithm for division, you can compute the difference in computational latency between the two).

### 4.2 Approach

The code takes values from the memory locations mentioned (Here, we have added code to write data onto those memory locations for demonstration purposes). Then, we perform repetitive subtraction of the divisor from the dividend in place until divisor>dividend while incrementing the quotient(stored in another register). At this point, the register containing the dividend will contain the remainder, and the quotient register will contain the quotient. These are then moved into the specified memory locations.

### 4.3 Flowchart


Start
Take inputs, set Quotient=0
Dividend ≤ Divisor
Yes
Store Quotient and Remainder
Stop
No
Subtract Divisor from Dividend
Increment Quotient

### 4.4 Code

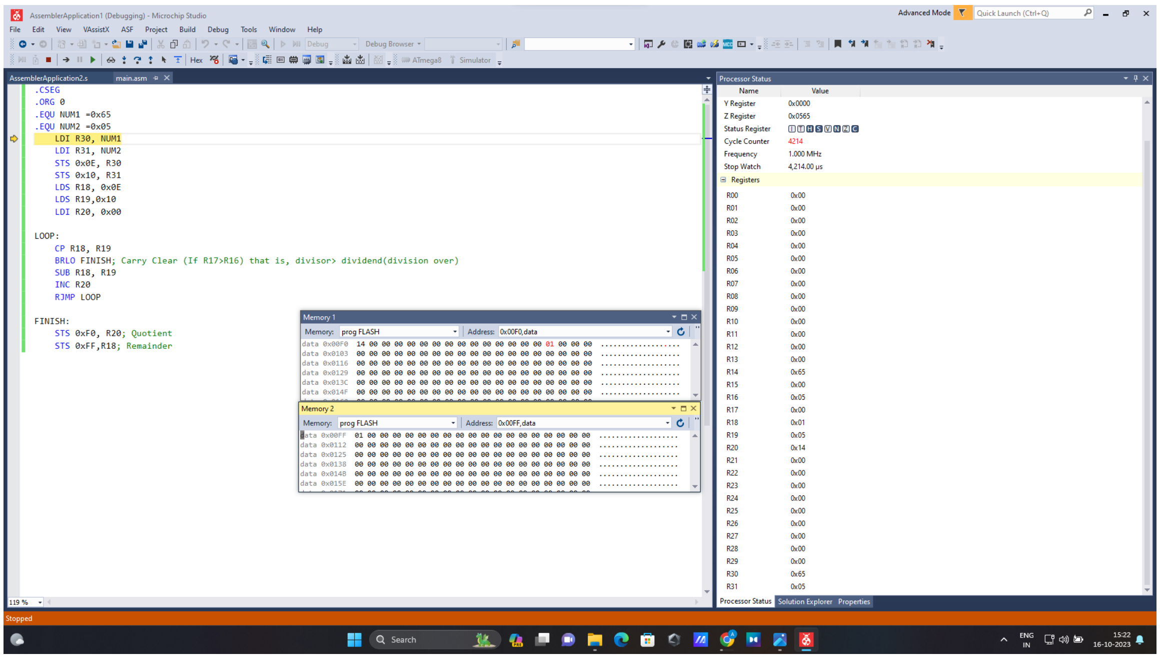
The code used to perform division is given below:

```
.CSEG
.ORG 0
.EQU NUM1 =0x65
.EQU NUM2 =0x05
        LDI R30, NUM1
        LDI R31, NUM2
        STS 0x0E, R30
        STS 0x10, R31
        LDS R18, 0x0E
        LDS R19,0x10
        LDI R20, 0x00

LOOP:
        CP R18, R19
        BRLO FINISH; Carry Clear (If R17>R16) that is, divisor> dividend
         ↪  (division over)
        SUB R18, R19
        INC R20
        RJMP LOOP

FINISH:
```

```
STS 0xF0, R20; Quotient
STS 0xFF,R18; Remainder
```

Listing 2: Code to Perform Division

## 4.5 Screenshots



AssemblerApplication1 (Debugging) - Microchip Studio

```
.CSEG
.ORG 0
.EQU NUM1 =0x65
.EQU NUM2 =0x05
    LDI R30, NUM1
    LDI R31, NUM2
    STS 0x0E, R30
    STS 0x10, R31
    LDS R18, 0x0E
    LDS R19,0x10
    LDI R20, 0x00

LOOP:
    CP R18, R19
    BRLO FINISH; Carry Clear (If R17>R16) that is, divisor> dividend(division over)
    SUB R18, R19
    INC R20
    RJMP LOOP

FINISH:
    STS 0xF0, R20; Quotient
    STS 0xFF,R18; Remainder
```

Memory 1 — Memory: prog FLASH — Address: 0x00F0,data

```
data 0x00F0  14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00  ...................
data 0x0103  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x0116  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x0129  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x013C  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x014F  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
```

Memory 2 — Memory: prog FLASH — Address: 0x00FF,data

```
data 0x00FF  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x0112  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x0125  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x0138  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x014B  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x015E  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
```

Processor Status

| Name | Value |
|---|---|
| Y Register | 0x0000 |
| Z Register | 0x0565 |
| Status Register | I T H S V N Z C |
| Cycle Counter | 4214 |
| Frequency | 1.000 MHz |
| Stop Watch | 4,214.00 µs |
| Registers | |
| R00 | 0x00 |
| R01 | 0x00 |
| R02 | 0x00 |
| R03 | 0x00 |
| R04 | 0x00 |
| R05 | 0x00 |
| R06 | 0x00 |
| R07 | 0x00 |
| R08 | 0x00 |
| R09 | 0x00 |
| R10 | 0x00 |
| R11 | 0x00 |
| R12 | 0x00 |
| R13 | 0x00 |
| R14 | 0x65 |
| R15 | 0x00 |
| R16 | 0x05 |
| R17 | 0x00 |
| R18 | 0x01 |
| R19 | 0x05 |
| R20 | 0x14 |
| R21 | 0x00 |
| R22 | 0x00 |
| R23 | 0x00 |
| R24 | 0x00 |
| R25 | 0x00 |
| R26 | 0x00 |
| R27 | 0x00 |
| R28 | 0x00 |
| R29 | 0x00 |
| R30 | 0x65 |
| R31 | 0x05 |

Figure 2: Screenshot for Problem 2

## 4.6 Inferences

This assembly code is designed to perform integer division of two numbers, with NUM1 as the dividend and NUM2 as the divisor. The code calculates the quotient and remainder of the division and stores the results in memory locations 0xF0 (for quotient) and 0xFF (for remainder).
Since no in-built function exists, we use the continuous subtraction method to calculate the quotient and remainder.

### 4.6.1 Register Transfers

1. Load: LDI, LDS to load data
2. Store: STS to store
3. Move: SUB- Register to Register Moves

### 4.6.2 Clock Cycles per Instruction

1. LDI: 1 Clock Cycle
2. MOV: 1 Clock Cycle
3. ADD: 1 Clock Cycle

4. MUL: 2 Clock Cycles
5. LDS: 2 Clock Cycles
6. STS: 2 Clock Cycles
7. CP: 1 Clock Cycle
8. INC: 1 Clock Cycle
9. RJMP: 2 Clock Cycles
10. Branch Statements: 1 or 2 Clock Cycles depending on result

#### 4.6.3 Latency

The latency of this AVR Assembly Code is $11 + 6\lfloor \frac{Divident}{Divisor} \rfloor + 7 \mu s$

#### 4.6.4 Throughput

Throughput of a code is defined as:

$$Throughput \equiv \frac{\text{Number of Computations(tasks) Done}}{\text{Time Taken}}$$

Here, we define the task to be the number of Divisions per unit time
$Throughput = \frac{10^6}{18+6\lfloor \frac{Divident}{Divisor} \rfloor}$ Tasks /s

#### 4.6.5 Limitations

1. This code can only divide an unsigned 8-bit number by another unsigned 8-bit number

## 5 Problem 3: Parity Detection

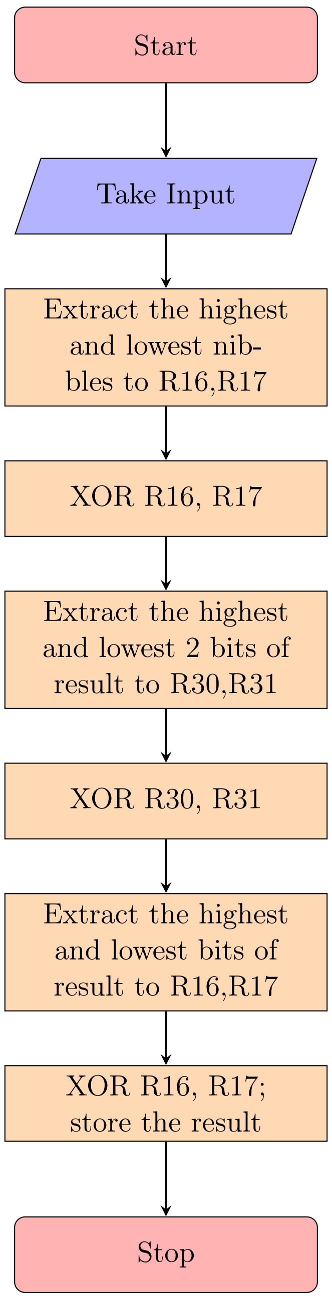
### 5.1 Problem Statement

Given an 8-bit number stored in location 0xFF, generate the odd parity of this number and store it at memory location 0xF0.

### 5.2 Approach

For this problem, if we have an 8-bit number, we divide it into two 4-bit numbers, take xor and then divide the next 4 bits into two two-bit numbers and continue this process until we get a single bit. This single bit is the odd parity.

### 5.3 Flowchart


Start
Take Input
Extract the highest and lowest nibbles to R16,R17
XOR R16, R17
Extract the highest and lowest 2 bits of result to R30,R31
XOR R30, R31
Extract the highest and lowest bits of result to R16,R17
XOR R16, R17; store the result
Stop

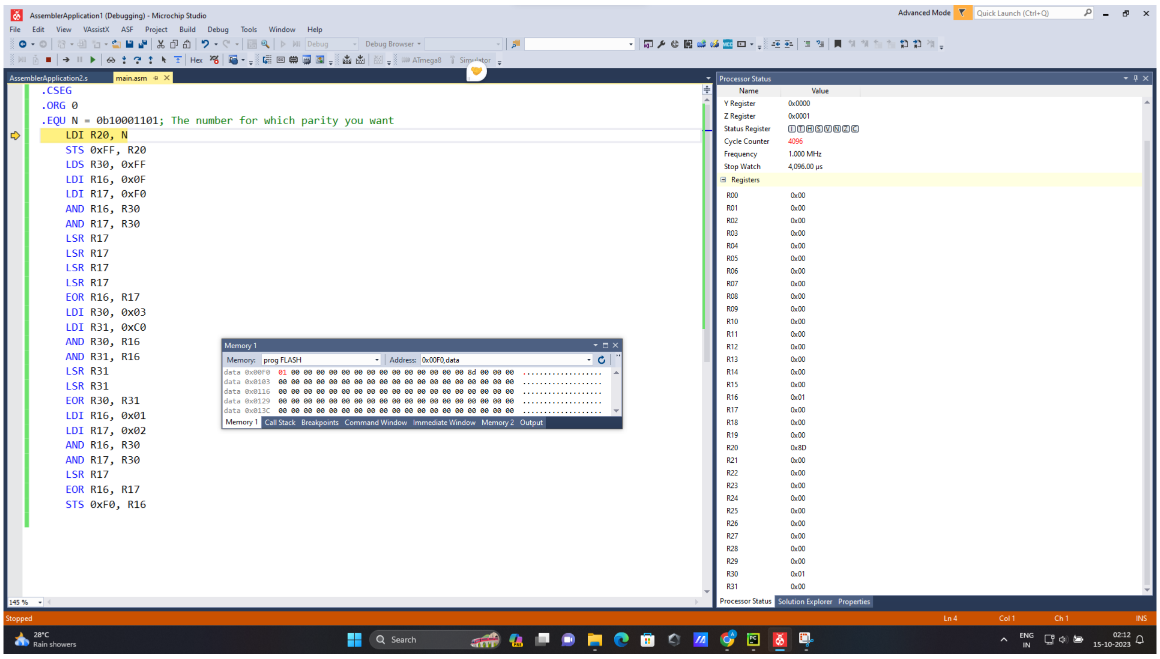
### 5.4 Code

The code used to find the odd parity of a given 8-bit number is given below:

```
.CSEG
.ORG 0
.EQU N = 0b10001101; The number for which parity you want
        LDI R20, N
        STS 0xFF, R20
        LDS R30, 0xFF
        LDI R16, 0x0F
        LDI R17, 0xF0
        AND R16, R30
        AND R17, R30
        LSR R17
        LSR R17
        LSR R17
        LSR R17
        EOR R16, R17
        LDI R30, 0x03
        LDI R31, 0xC0
        AND R30, R16
        AND R31, R16
        LSR R31
        LSR R31
        EOR R30, R31
        LDI R16, 0x01
        LDI R17, 0x02
        AND R16, R30
        AND R17, R30
        LSR R17
        EOR R16, R17
        STS 0xF0, R16
```

Listing 3: Code to Generate Odd Parity

## 5.5 Screenshots



```
.CSEG
.ORG 0
.EQU N = 0b10001101; The number for which parity you want
    LDI R20, N
    STS 0xFF, R20
    LDS R30, 0xFF
    LDI R16, 0x0F
    LDI R17, 0xF0
    AND R16, R30
    AND R17, R30
    LSR R17
    LSR R17
    LSR R17
    LSR R17
    EOR R16, R17
    LDI R30, 0x03
    LDI R31, 0xC0
    AND R30, R16
    AND R31, R16
    LSR R31
    LSR R31
    EOR R30, R31
    LDI R16, 0x01
    LDI R17, 0x02
    AND R16, R30
    AND R17, R30
    LSR R17
    EOR R16, R17
    STS 0xF0, R16
```

Memory 1

Memory: prog FLASH    Address: 0x00F0,data

```
data 0x00F0  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 8d 00 00 00  ...................
data 0x0103  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x0116  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x0129  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x013C  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
```

Memory 1 | Call Stack | Breakpoints | Command Window | Immediate Window | Memory 2 | Output

Processor Status

| Name | Value |
|---|---|
| Y Register | 0x0000 |
| Z Register | 0x0001 |
| Status Register | I T H S V N Z C |
| Cycle Counter | 4096 |
| Frequency | 1.000 MHz |
| Stop Watch | 4,096.00 µs |
| Registers | |
| R00 | 0x00 |
| R01 | 0x00 |
| R02 | 0x00 |
| R03 | 0x00 |
| R04 | 0x00 |
| R05 | 0x00 |
| R06 | 0x00 |
| R07 | 0x00 |
| R08 | 0x00 |
| R09 | 0x00 |
| R10 | 0x00 |
| R11 | 0x00 |
| R12 | 0x00 |
| R13 | 0x00 |
| R14 | 0x00 |
| R15 | 0x00 |
| R16 | 0x01 |
| R17 | 0x00 |
| R18 | 0x00 |
| R19 | 0x00 |
| R20 | 0x8D |
| R21 | 0x00 |
| R22 | 0x00 |
| R23 | 0x00 |
| R24 | 0x00 |
| R25 | 0x00 |
| R26 | 0x00 |
| R27 | 0x00 |
| R28 | 0x00 |
| R29 | 0x00 |
| R30 | 0x01 |
| R31 | 0x00 |

Processor Status | Solution Explorer | Properties

Figure 3: Screenshot for Problem 3

## 5.6 Inferences

This assembly code calculates the odd parity of the binary number N to be input by the TA and stores the result in memory location 0xF0. We infer from this that parity generation is essential since the data we get may be incorrect.

#### 5.6.1 Register Transfers

1. Load: LDI, LDS to load data
2. Store: STS to store
3. Move: SUB- Register to Register Moves

#### 5.6.2 Clock Cycles per Instruction

1. LDI: 1 Clock Cycle
2. LDS: 2 Clock Cycles
3. STS: 2 Clock Cycles
4. LSR: 1 Clock Cycle
5. EOR: 1 Clock Cycle
6. AND: 1 Clock Cycle

#### 5.6.3 Latency

The latency of this AVR Assembly Code is $29\mu s$

#### 5.6.4 Throughput

Throughput of a code is defined as:

$$Throughput \equiv \frac{\text{Number of Computations(tasks) Done}}{\text{Time Taken}}$$

Here, we define the task to be finding the Parity of the given number
$Throughput = \frac{1}{29\mu s} = 3.44 \times 10^4$ Tasks /s

#### 5.6.5 Limitations

1. This code can only find the parity of an 8- bit number

## 6 Problem 4: Largest and Smallest of a number set given

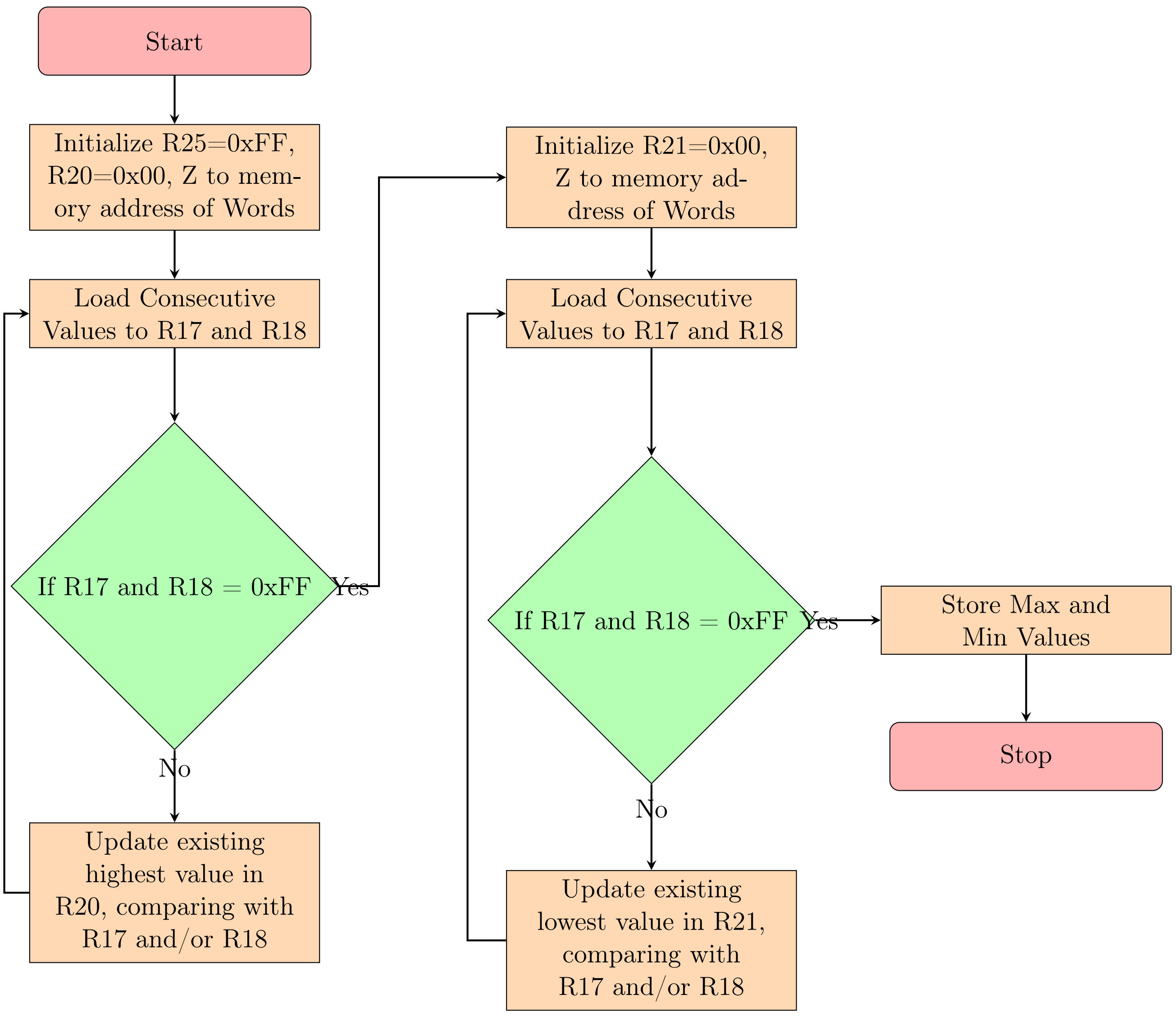
### 6.1 Problem Statement

Given a finite set of binary words, identify the largest and smallest number in the given set and store it at 0xF0 and 0xFF, respectively. Note that the TA will enter the finite set. You are supposed to develop a code that works for an arbitrary number of numbers.

### 6.2 Approach

For this problem, we first ask the TA's to store the set of values at the end of the program with the db tag. Then, the Values are taken one by one through the Z register, and this process is terminated if we get two consecutive 0xFF. While loop in through the Z register we keep on updating the max and the min value.

## 6.3 Flowchart

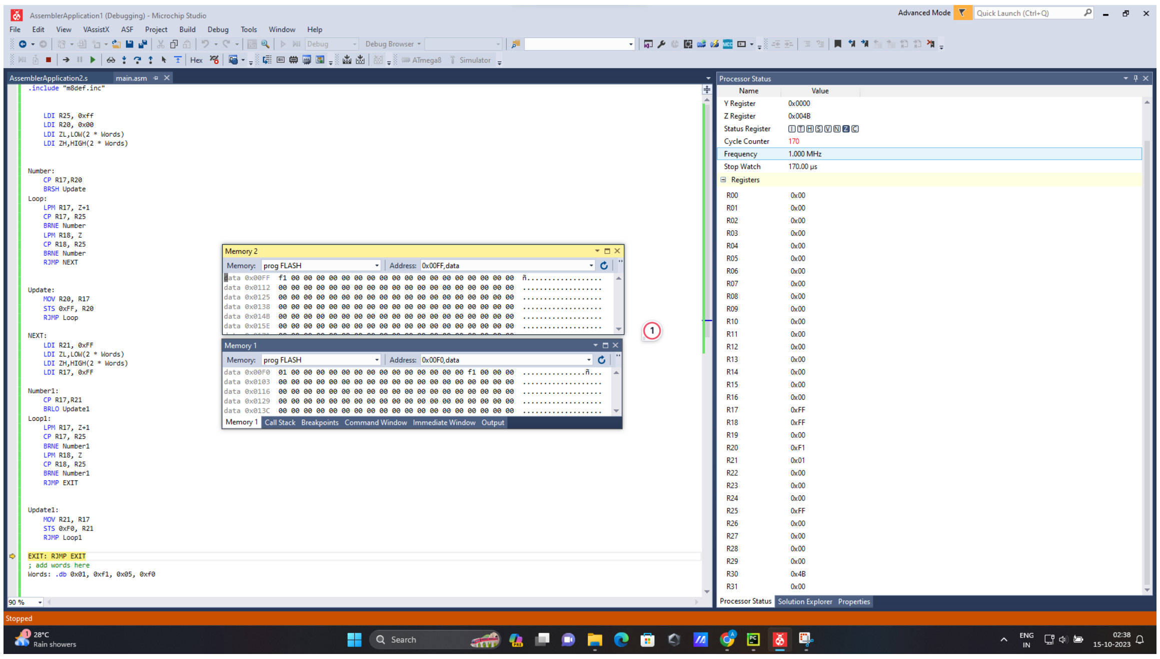

Start
Initialize R25=0xFF, R20=0x00, Z to memory address of Words
Load Consecutive Values to R17 and R18
If R17 and R18 = 0xFF
Yes
No
Update existing highest value in R20, comparing with R17 and/or R18
Initialize R21=0x00, Z to memory address of Words
Load Consecutive Values to R17 and R18
If R17 and R18 = 0xFF
Yes
No
Update existing lowest value in R21, comparing with R17 and/or R18
Store Max and Min Values
Stop

## 6.4 Code

```
.include "m8def.inc"


        LDI R25, 0xff
        LDI R20, 0x00
        LDI ZL,LOW(2 * Words)
        LDI ZH,HIGH(2 * Words)


Number:
        CP R17,R20
        BRSH Update
Loop:
        LPM R17, Z+1
        CP R17, R25
        BRNE Number
        LPM R18, Z
        CP R18, R25
        BRNE Number
        RJMP NEXT


Update:
        MOV R20, R17
        STS 0xFF, R20
        RJMP Loop

NEXT:
        LDI R21, 0xFF
        LDI ZL,LOW(2 * Words)
        LDI ZH,HIGH(2 * Words)
        LDI R17, 0xFF

Number1:
        CP R17,R21
        BRLO Update1
Loop1:
        LPM R17, Z+1
        CP R17, R25
        BRNE Number1
        LPM R18, Z
        CP R18, R25
        BRNE Number1
        RJMP EXIT


Update1:
        MOV R21, R17
        STS 0xF0, R21
        RJMP Loop1

EXIT: RJMP EXIT
; add words here
Words: .db 0x01, 0xf1, 0x05, 0xf0
```

Listing 4: Code to Find the Largest and Smallest Number from a Given Set of Numbers

## 6.5 Screenshots


AssemblerApplication1 (Debugging) - Microchip Studio
File Edit View VAssistX ASF Project Build Debug Tools Window Help
Advanced Mode
Quick Launch (Ctrl+Q)
AssemblerApplication2.s
main.asm
Stopped
28°C Rain showers
ENG IN
02:38 15-10-2023

```
.include "m8def.inc"


    LDI R25, 0xff
    LDI R20, 0x00
    LDI ZL,LOW(2 * Words)
    LDI ZH,HIGH(2 * Words)


Number:
    CP R17,R20
    BRSH Update
Loop:
    LPM R17, Z+1
    CP R17, R25
    BRNE Number
    LPM R18, Z
    CP R18, R25
    BRNE Number
    RJMP NEXT


Update:
    MOV R20, R17
    STS 0xFF, R20
    RJMP Loop

NEXT:
    LDI R21, 0xFF
    LDI ZL,LOW(2 * Words)
    LDI ZH,HIGH(2 * Words)
    LDI R17, 0xFF

Number1:
    CP R17,R21
    BRLO Update1
Loop1:
    LPM R17, Z+1
    CP R17, R25
    BRNE Number1
    LPM R18, Z
    CP R18, R25
    BRNE Number1
    RJMP EXIT


Update1:
    MOV R21, R17
    STS 0xF0, R21
    RJMP Loop1

EXIT: RJMP EXIT
; add words here
Words: .db 0x01, 0xf1, 0x05, 0xf0
```

Memory 2 — Memory: prog FLASH — Address: 0x00FF,data

```
data 0x00FF  f1 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ñ..................
data 0x0112  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x0125  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x0138  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x014B  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x015E  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
```

Memory 1 — Memory: prog FLASH — Address: 0x00F0,data

```
data 0x00F0  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 f1 00 00 00  ...............ñ...
data 0x0103  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x0116  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x0129  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
data 0x013C  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
```

Memory 1 | Call Stack | Breakpoints | Command Window | Immediate Window | Output

Processor Status

| Name | Value |
|---|---|
| Y Register | 0x0000 |
| Z Register | 0x004B |
| Status Register | I T H S V N Z C |
| Cycle Counter | 170 |
| Frequency | 1.000 MHz |
| Stop Watch | 170.00 µs |
| Registers | |
| R00 | 0x00 |
| R01 | 0x00 |
| R02 | 0x00 |
| R03 | 0x00 |
| R04 | 0x00 |
| R05 | 0x00 |
| R06 | 0x00 |
| R07 | 0x00 |
| R08 | 0x00 |
| R09 | 0x00 |
| R10 | 0x00 |
| R11 | 0x00 |
| R12 | 0x00 |
| R13 | 0x00 |
| R14 | 0x00 |
| R15 | 0x00 |
| R16 | 0x00 |
| R17 | 0xFF |
| R18 | 0xFF |
| R19 | 0x00 |
| R20 | 0xF1 |
| R21 | 0x01 |
| R22 | 0x00 |
| R23 | 0x00 |
| R24 | 0x00 |
| R25 | 0xFF |
| R26 | 0x00 |
| R27 | 0x00 |
| R28 | 0x00 |
| R29 | 0x00 |
| R30 | 0x4B |
| R31 | 0x00 |

Processor Status | Solution Explorer | Properties

Figure 4: Screenshot for Problem 4

## 6.6 Inferences

### 6.6.1 Register Transfers

1. Load: LDI, LDS to load data
2. Store: STS to store
3. Move: SUB- Register to Register Moves

### 6.6.2 Clock Cycles per Instruction

1. LDI: 1 Clock Cycle
2. CP: 1 Clock Cycle
3. Branch Statements: 1 or 2 Clock Cycles depending on result
4. LPM: 3 Clock Cycles
5. RJMP: 2 Clock Cycles
6. MOV: 1 Clock Cycle
7. LDS: 2 Clock Cycles
8. STS: 2 Clock Cycles

### 6.6.3 Limitations

1. This code cannot find maximum or minimum of the set of numbers if there are two or more repeating 0xFF. The code will not check further if that occurs.

2. The numbers being compared are assumed to be 8-bit numbers.

## 7 Problem 5: Fibonacci Sequence

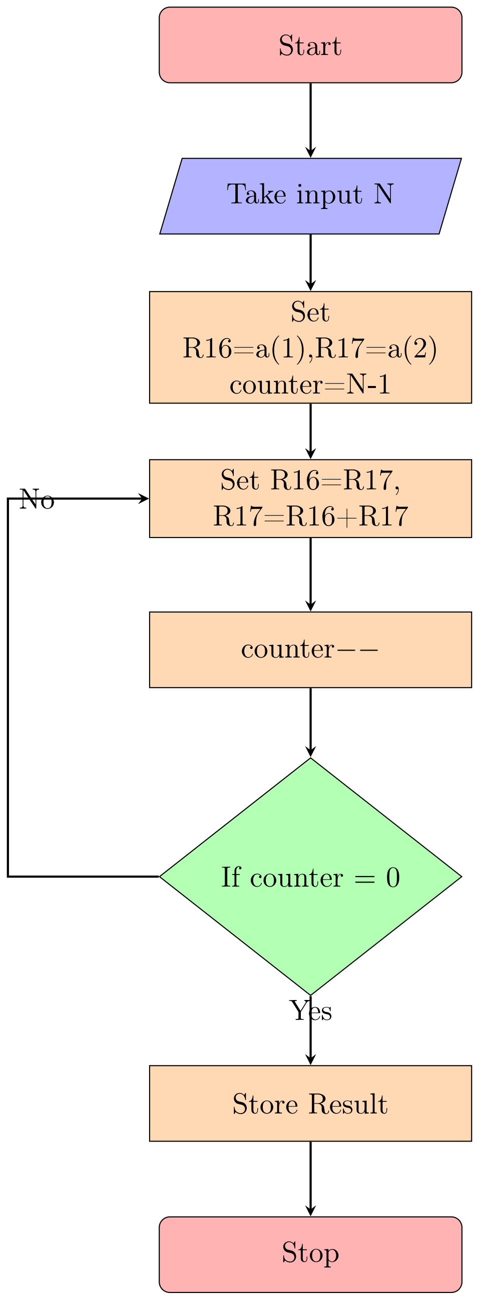
### 7.1 Problem Statement

The Fibonacci sequence is a sequence of numbers that follows the recurrence $a(n+1) = a(n) + a(n-1)\ \forall\ n > 2$. The initial two terms of the sequence are $a(1) = 0$ and $a(2) = 1$. Given a number $N$ , calculate the $N^{th}$ term of the Fibonacci sequence $a(N)$ and store it in register $R0$

### 7.2 Approach

For this program, we ask the TA first to input the Nth value of the Fibonacci series they want. Using this, we then calculate the values of all the Fibonacci series till the Nth value and store the Nth value in a memory location. This is done progressively by shifting the values of two temporary registers to the (N-1)Th and (N-2)Th values.

### 7.3 Flowchart

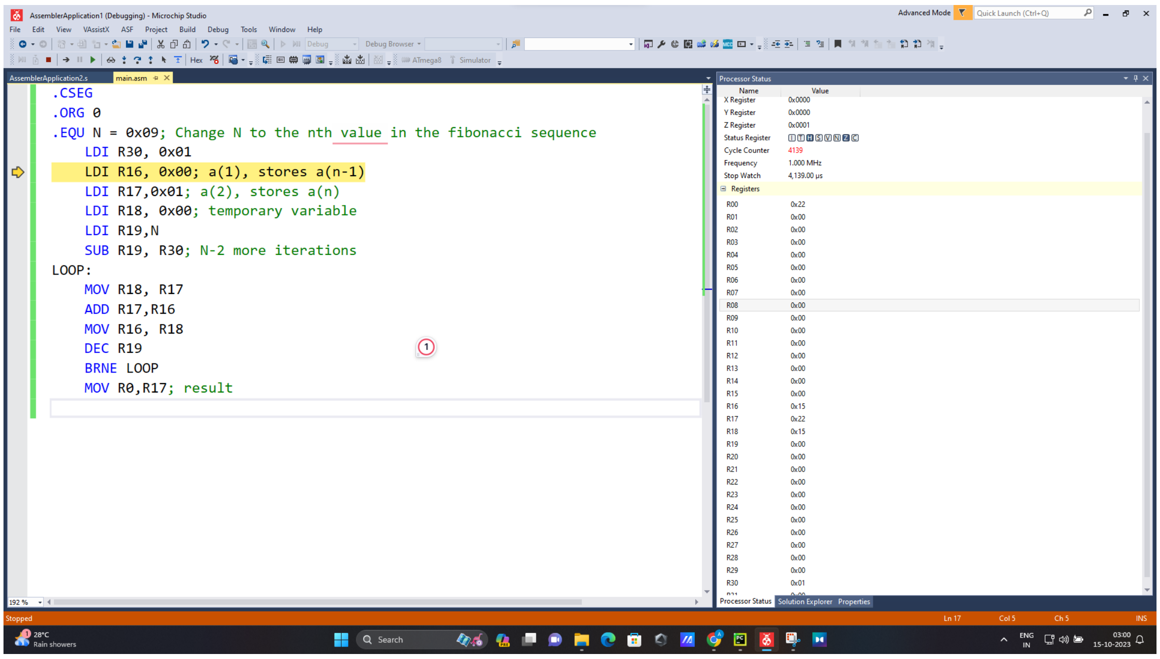


### 7.4 Code

```
.CSEG
.ORG 0
.EQU N = 0x09; The value of N for which a(N) is to be found
        LDI R30, 0x01
        LDI R16, 0x00; a(1), stores a(n-1)
        LDI R17,0x01; a(2), stores a(n)
        LDI R18, 0x00; temporary variable
        LDI R19,N
        SUB R19, R30; N-2 more iterations
LOOP:
        MOV R18, R17
        ADD R17,R16
        MOV R16, R18
        DEC R19
        BRNE LOOP
        MOV R0,R17; Result stored in R0
```

Listing 5: Code to find $N^{th}$ Term of Fibonacci Series

## 7.5 Screenshots



```
.CSEG
.ORG 0
.EQU N = 0x09; Change N to the nth value in the fibonacci sequence
    LDI R30, 0x01
    LDI R16, 0x00; a(1), stores a(n-1)
    LDI R17,0x01; a(2), stores a(n)
    LDI R18, 0x00; temporary variable
    LDI R19,N
    SUB R19, R30; N-2 more iterations
LOOP:
    MOV R18, R17
    ADD R17,R16
    MOV R16, R18
    DEC R19
    BRNE LOOP
    MOV R0,R17; result
```

Processor Status

| Name | Value |
|---|---|
| X Register | 0x0000 |
| Y Register | 0x0000 |
| Z Register | 0x0001 |
| Status Register | I T H S V N Z C |
| Cycle Counter | 4139 |
| Frequency | 1.000 MHz |
| Stop Watch | 4,139.00 µs |
| Registers | |
| R00 | 0x22 |
| R01 | 0x00 |
| R02 | 0x00 |
| R03 | 0x00 |
| R04 | 0x00 |
| R05 | 0x00 |
| R06 | 0x00 |
| R07 | 0x00 |
| R08 | 0x00 |
| R09 | 0x00 |
| R10 | 0x00 |
| R11 | 0x00 |
| R12 | 0x00 |
| R13 | 0x00 |
| R14 | 0x00 |
| R15 | 0x00 |
| R16 | 0x15 |
| R17 | 0x22 |
| R18 | 0x15 |
| R19 | 0x00 |
| R20 | 0x00 |
| R21 | 0x00 |
| R22 | 0x00 |
| R23 | 0x00 |
| R24 | 0x00 |
| R25 | 0x00 |
| R26 | 0x00 |
| R27 | 0x00 |
| R28 | 0x00 |
| R29 | 0x00 |
| R30 | 0x01 |

AssemblerApplication1 (Debugging) - Microchip Studio
File Edit View VAssistX ASF Project Build Debug Tools Window Help
Advanced Mode
Quick Launch (Ctrl+Q)
AssemblerApplication2.s
main.asm
Processor Status Solution Explorer Properties
Stopped
Ln 17 Col 5 Ch 5 INS
192 %
28°C Rain showers
Search
ENG IN
03:00 15-10-2023

Figure 5: Screenshot for Problem 5

## 7.6 Inferences

This assembly code calculates the nth value in the Fibonacci sequence, where the value of n is specified as N. It uses a loop and a set of registers (R16, R17, R18, R19, and R30) to calculate.

### 7.6.1 Register Transfers

1. Load: LDI, LDS to load data
2. Store: STS to store
3. Move: SUB- Register to Register Moves

### 7.6.2 Clock Cycles per Instruction

1. LDI: 1 Clock Cycle
2. SUB: 1 Clock Cycle
3. MOV: 1 Clock Cycle

4. MOV: 1 Clock Cycle
5. ADD: 1 Clock Cycle
6. DEC: 1 Clock Cycle
7. Branch Statements: 1 or 2 Clock Cycles depending on result

### 7.6.3 Latency

The latency of this AVR Assembly Code is $6N\mu s$, where $N$ is the number N for which a(N) has to be found

### 7.6.4 Throughput

Throughput of a code is defined as:

$$Throughput \equiv \frac{\text{Number of Computations(tasks) Done}}{\text{Time Taken}}$$

Here, we define the task to be finding the $N^{th}$ number in the Fibonacci Series
$Throughput = \frac{10^6}{6N}$ Tasks /s

### 7.6.5 Limitations

1. This code, like the rest, is limited by the size. It can only find $N^{th}$ term as long as it can be stored within 8 bits