Workflow Manuals

Computer Architecture

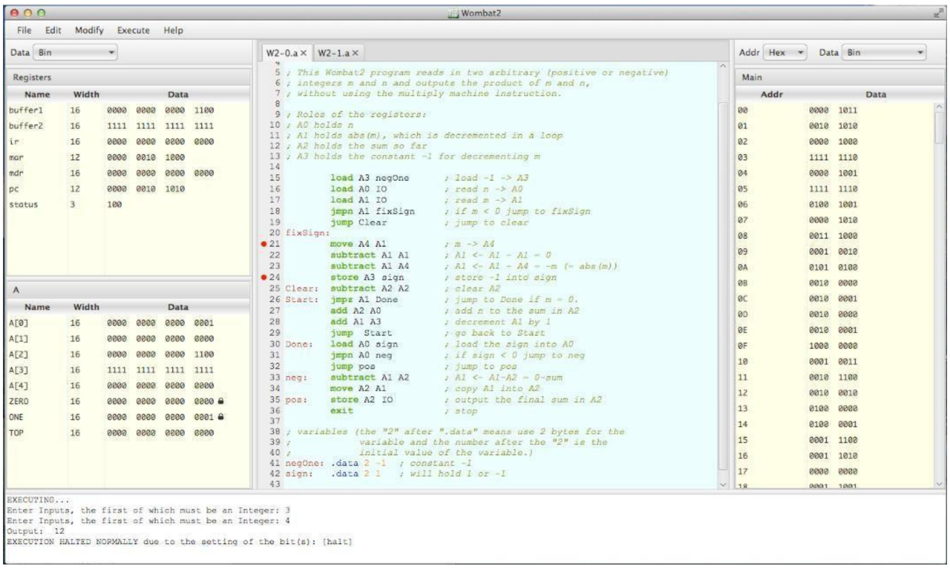
By
MUHAMMAD QASIM

Table of Contents

Tools & Technologies:

CPU Sim

CPU Sim is a tool used for simulating simple CPU architectures, helping students understand processor design, instruction execution, and debugging.



LAB 01

Introduction:

This lab introduces the fundamentals of CPU Sim, a simulation tool used to model basic CPU operations. Students will learn how to set up the software environment and explore key components of CPU architecture. By developing and testing simple assembly programs, they will gain hands-on experience in low-level programming. The lab also demonstrates the Fetch-Decode-Execute.

Registers used in CPU SIM

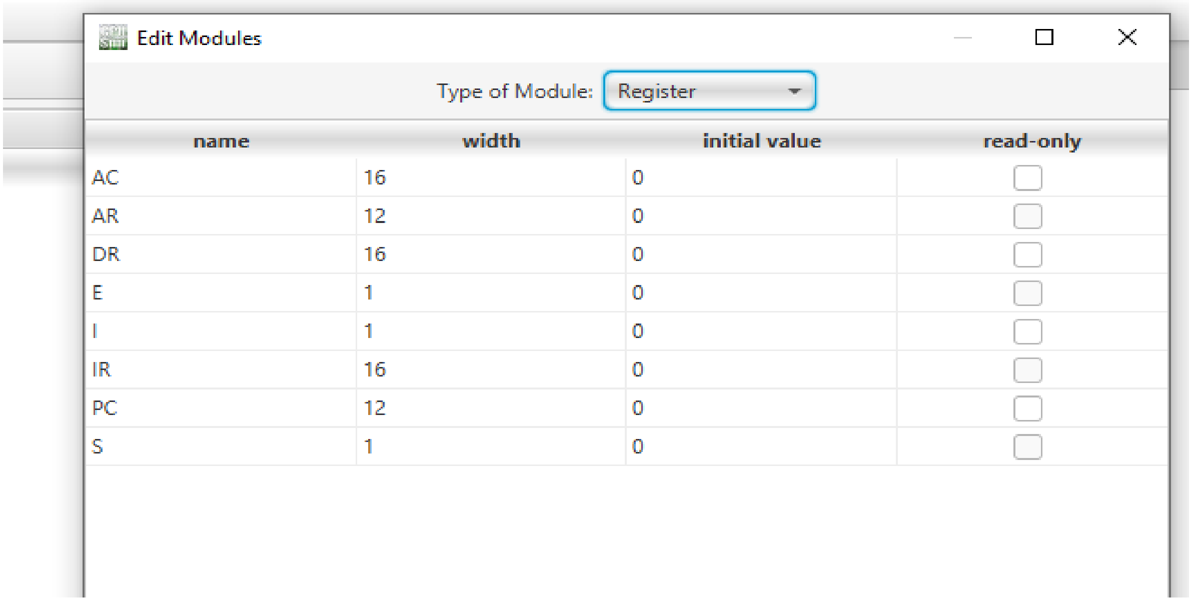
1. **Program Counter (PC)** – Stores the address of the next instruction (**12-bit**).
2. **Address Register (AR)** – Stores the memory address being accessed (**12-bit**).
3. **Instruction Register (IR)** – Holds the fetched instruction (**16-bit**).
4. **Accumulator Register (AC)** – Used for arithmetic operations (**16-bit**).
5. **Data Register (DR)** – Temporarily holds data being processed (**16-bit**).
6. **Temporary Register (TR)** – Optional register for intermediate calculations (**16-bit**).
7. **Condition Registers**
    - **E (Carry Bit)** – Used for carry operations.
    - **S (Status Register)** – Used for halt operations.
    - **I (Indexing Bit)** – Specifies direct or indirect addressing.

Registers

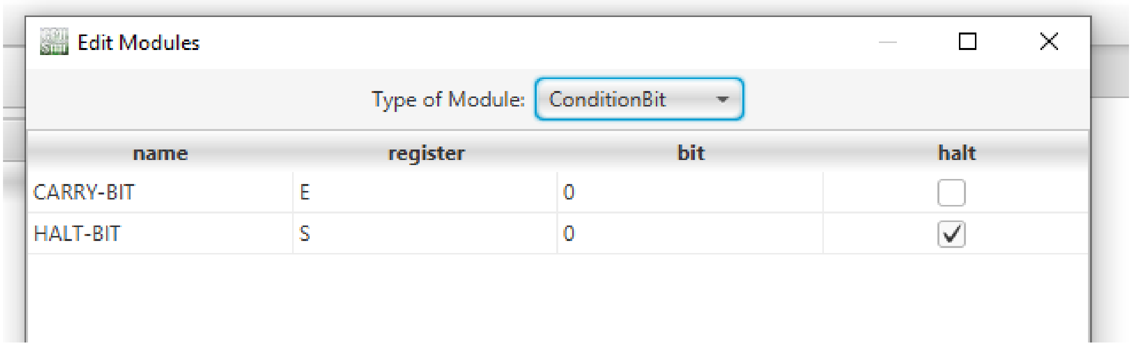
IR	DR	AC	AR	PC	I	E
0 15	0 15	0 15	0 11	0 11	1 bit	1 Bit

Setting up the machine for CPU Sim

- Setting up the registers



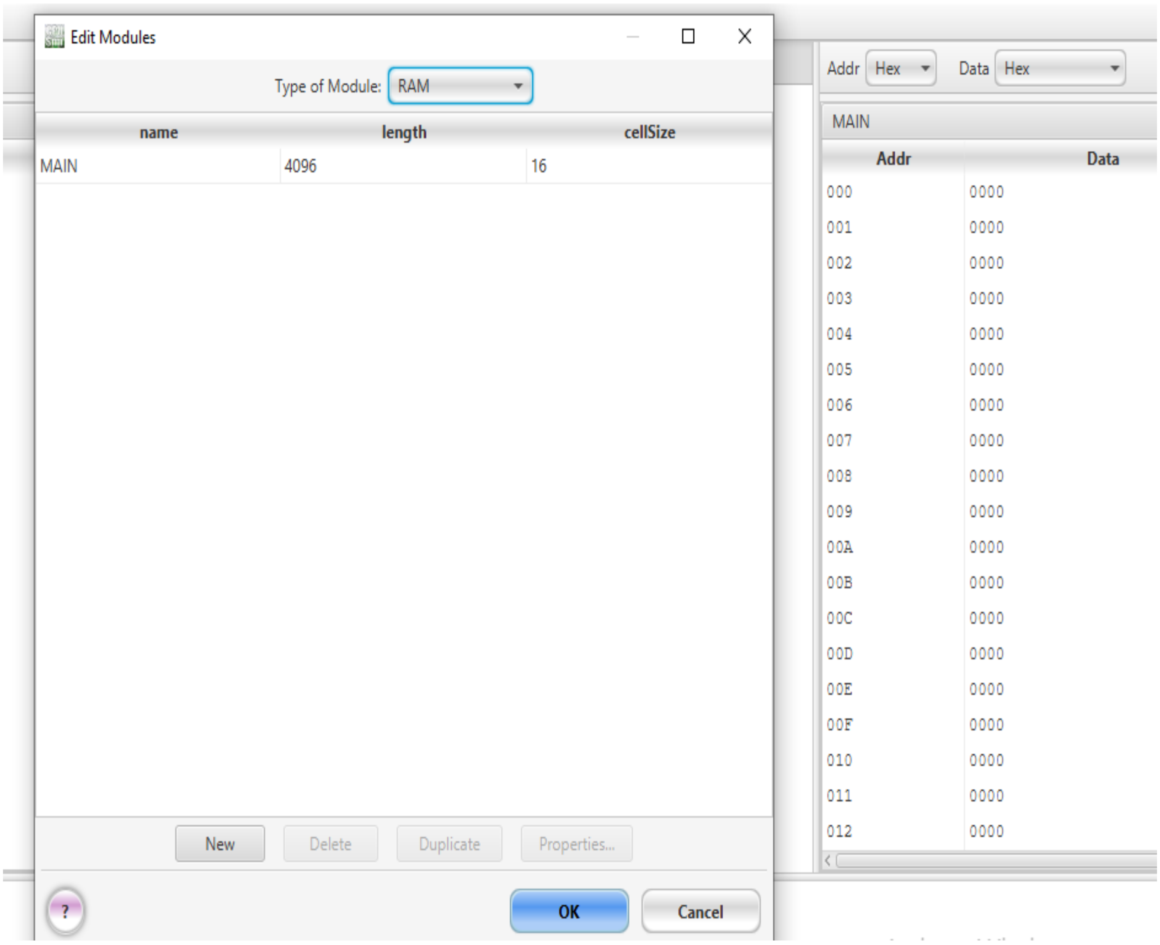
- Setting up condition registers



Setting up the memory

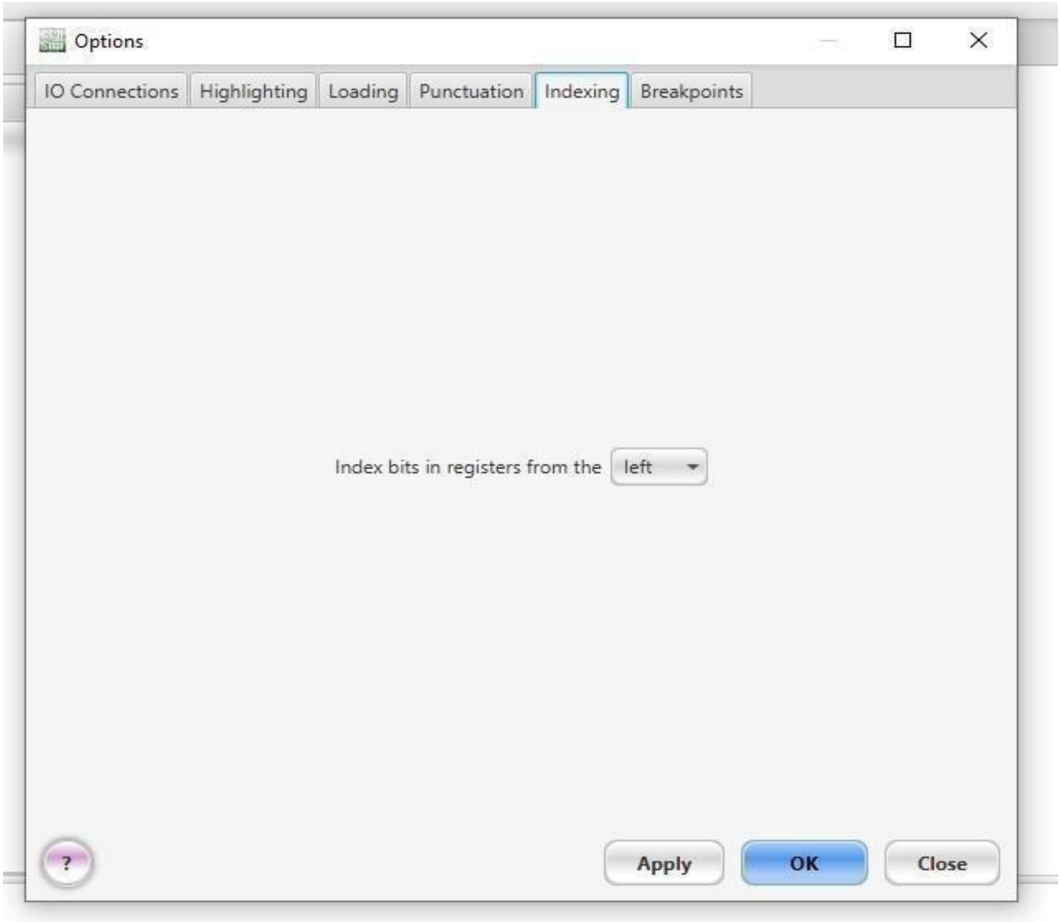
Open the **Hardware Module > Register > RAM**.

1. Create a new memory module:
    - **Size:** 4096 words
    - **Cell Size:** 16 bits
    - **Type:** RAM
2. Save the configuration with **.cpu** extension.



Setting up the indexing of instruction

1. Set the indexing from the left side.
2. Execute and then click on option and select indexing set it to left.

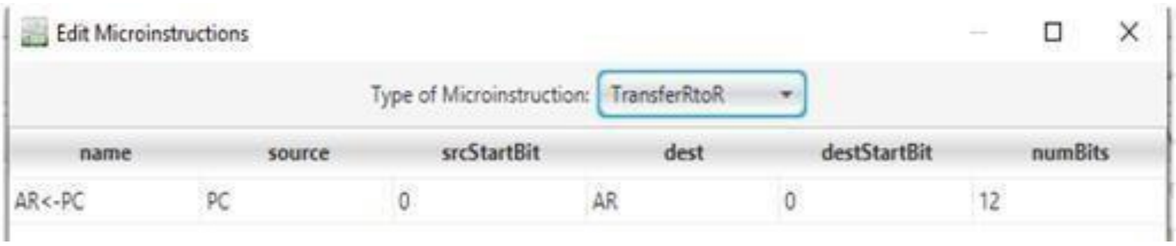


Fetch Cycle

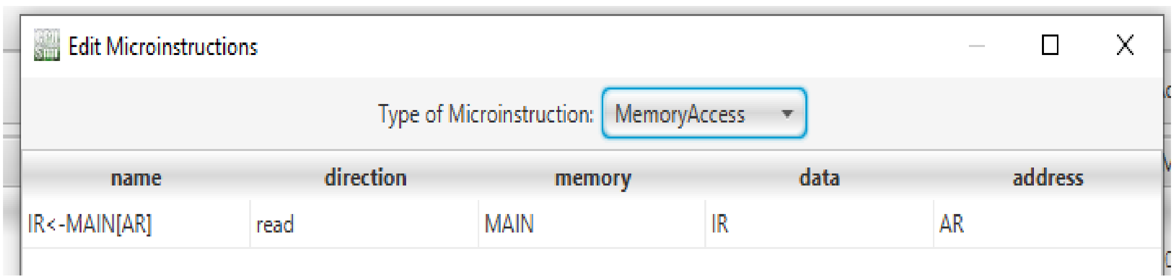
The Fetch Cycle involves the following steps:

1	AR <- PC
2	IR <- Main [AR]
3	PC - INCR
4	AR <- IR (4-15)
5	DECODE – IR

1. Transfer the address from **PC to AR**.



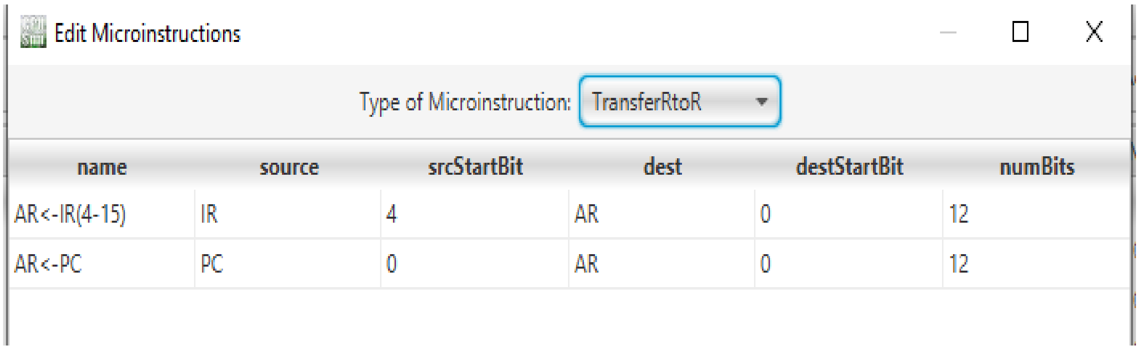
2. Read the instruction from memory into **IR**.



3. Increment **PC** to point to the next instruction.

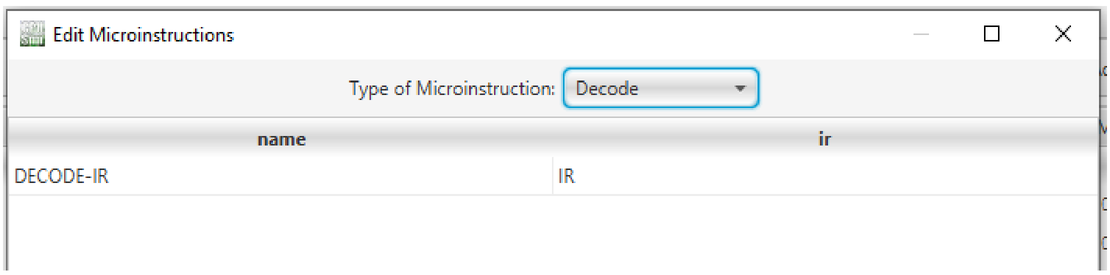
name	register	overflowBit	carryBit	delta
INCR-PC	PC	(none)	(none)	1

4. Extract the address part of the instruction and transfer it to **AR**.



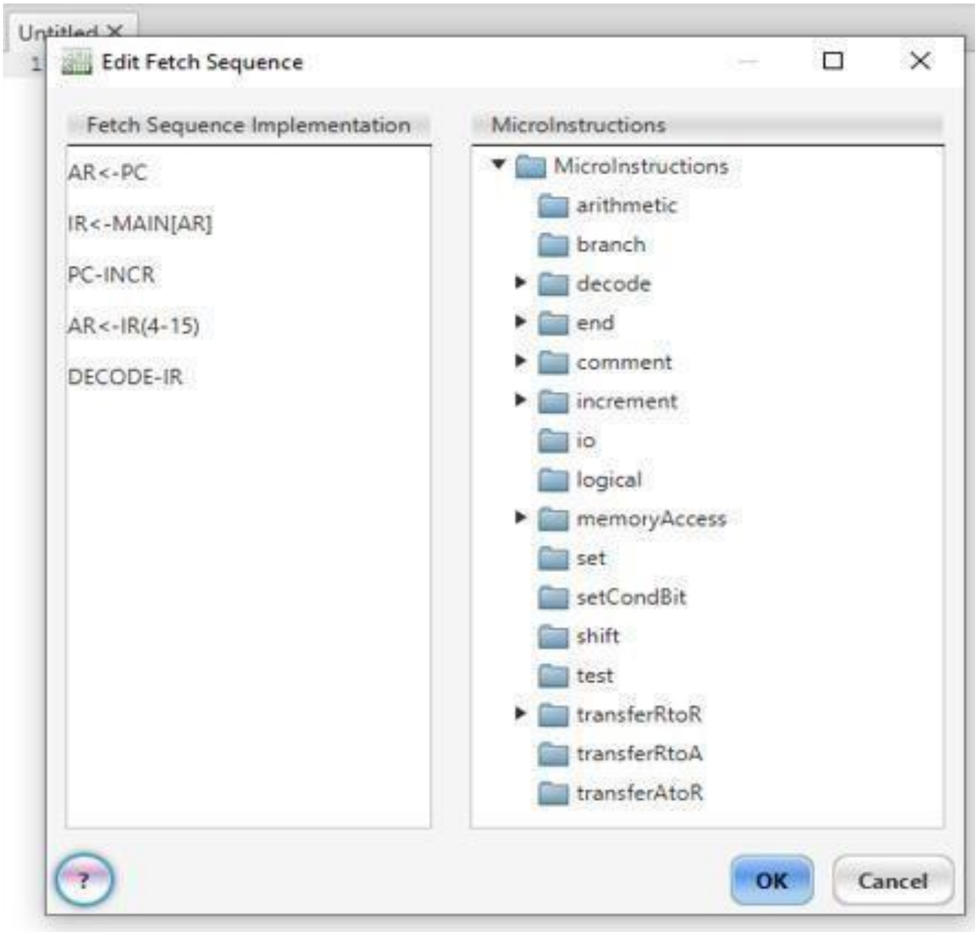
name	source	srcStartBit	dest	destStartBit	numBits
AR<-IR(4-15)	IR	4	AR	0	12
AR<-PC	PC	0	AR	0	12

5. Decode the instruction for execution.

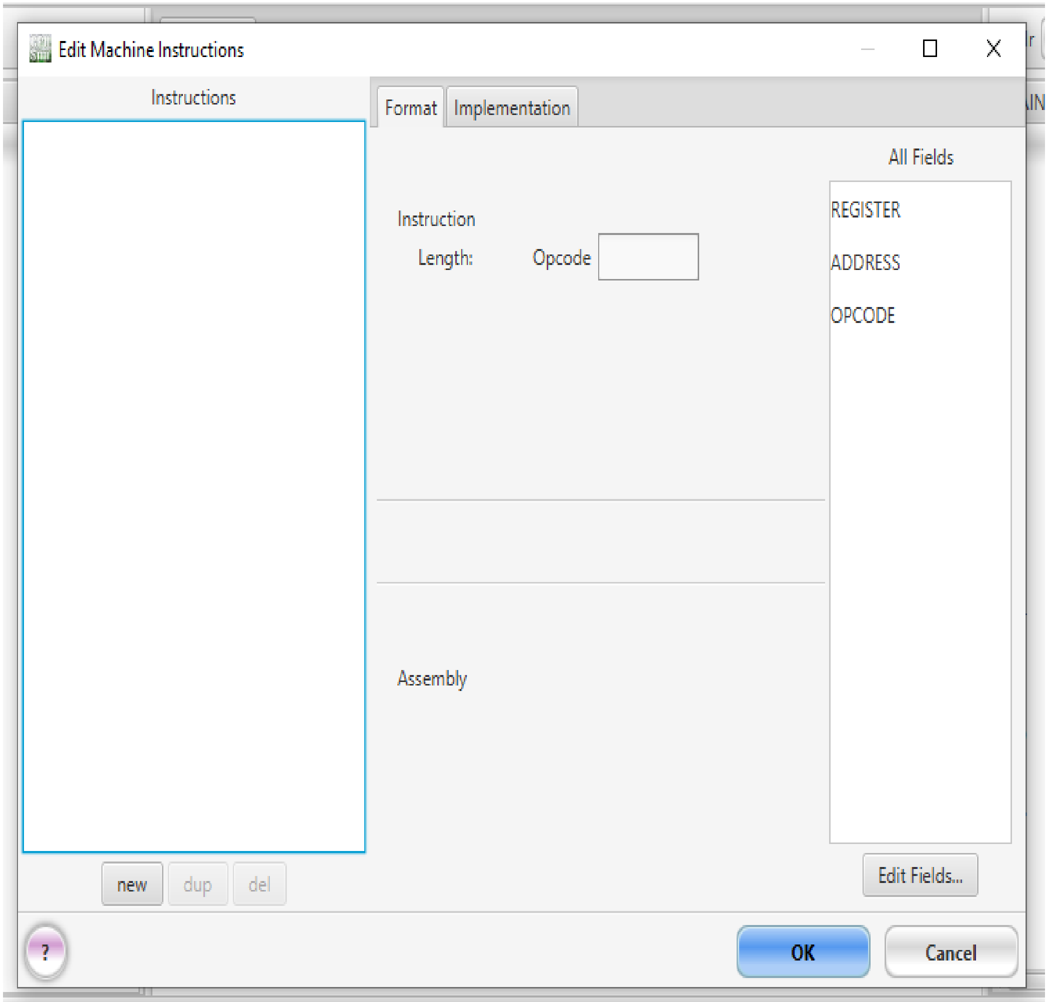


name	ir
DECODE-IR	IR

Fetch Execution



Edit machine Instructions



LAB 02

Introduction:

This lab focuses on understanding machine instructions and their critical role in the execution process. Students will explore instruction formats, including fields like opcodes and operands. The step-by-step execution of microinstructions will be demonstrated to show how control flows within the CPU. A practical task of adding two numbers using basic instructions will reinforce these core concepts

Program

```
START:
INP       ; Input a value and store it in the accumulator
STA NUM   ; Store the accumulator value in memory location "NUM"
INP       ; Input another value
ADD NUM   ; Add the value stored in "NUM" to the accumulator
OUT       ; Output the result
HLT       ; Halt execution
NUM: .data 1 0  ; Memory location labeled "NUM", initialized with 0
```

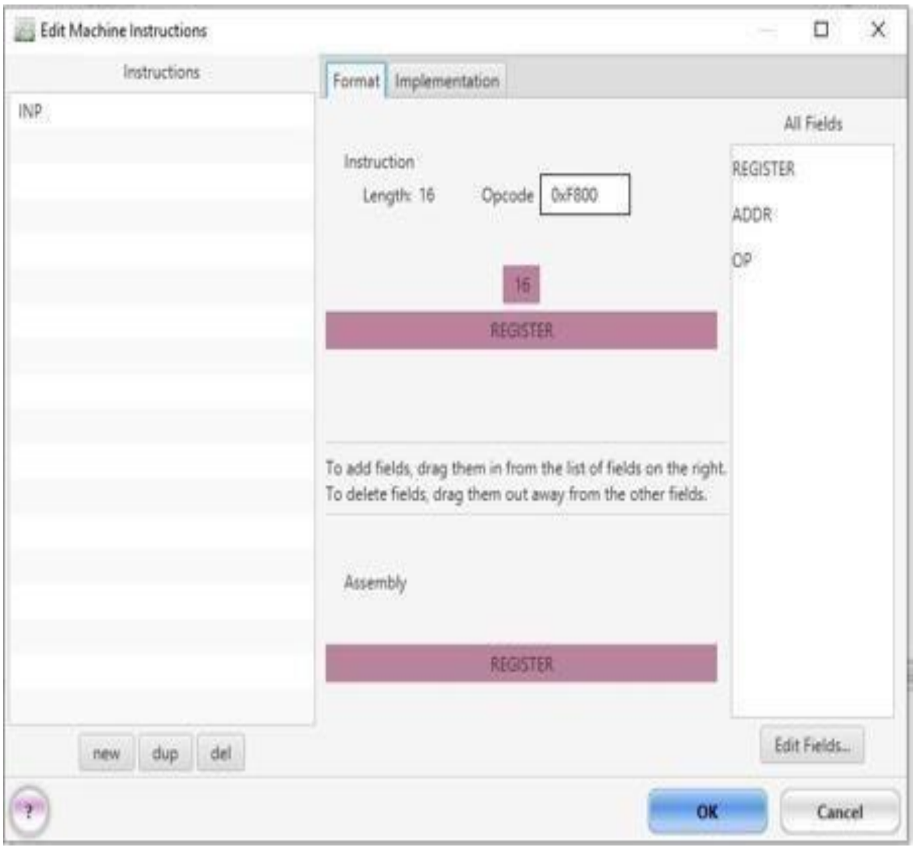
Edit machine instructions

1. First make a instructions for **INP**

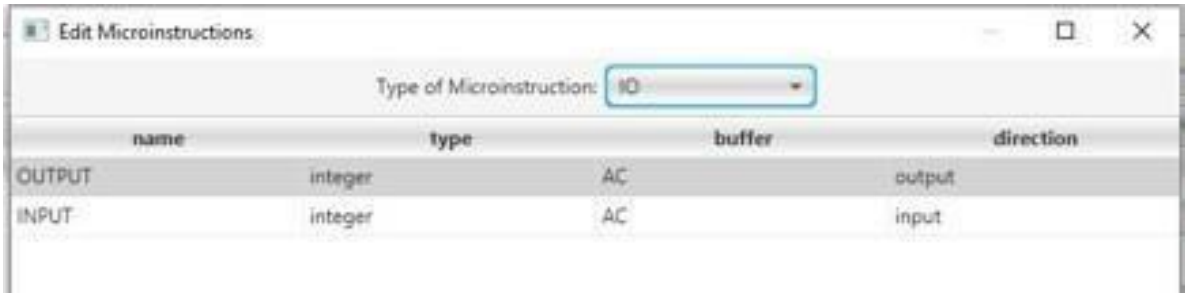
   First make a format of instruction by adding opcode according to given address.

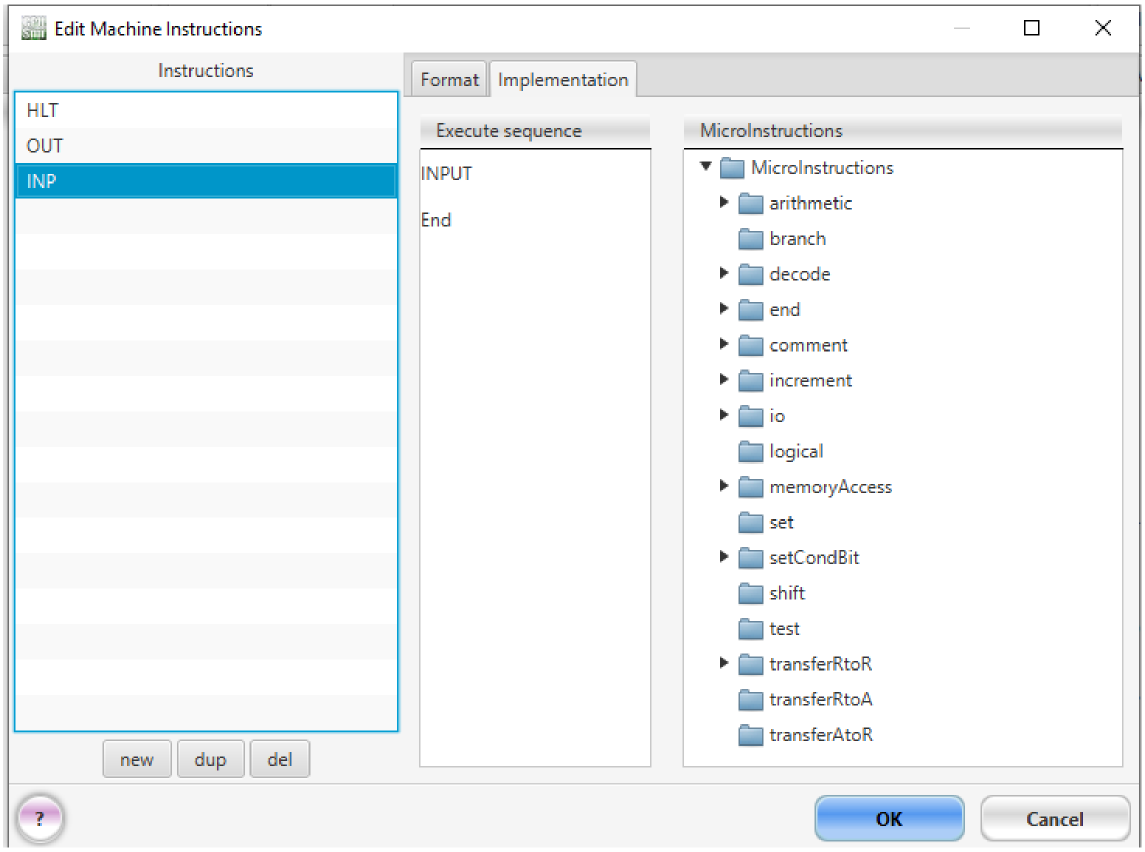
Input

OPCODE: F800



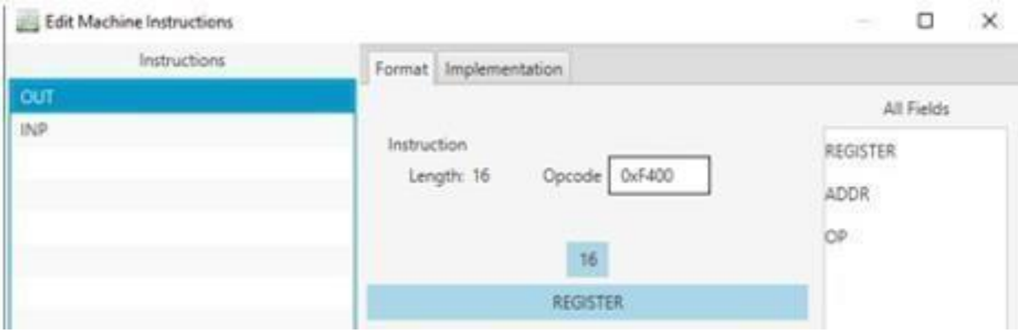
2. Second make a instructions for Microinstructions for input and set buffer.

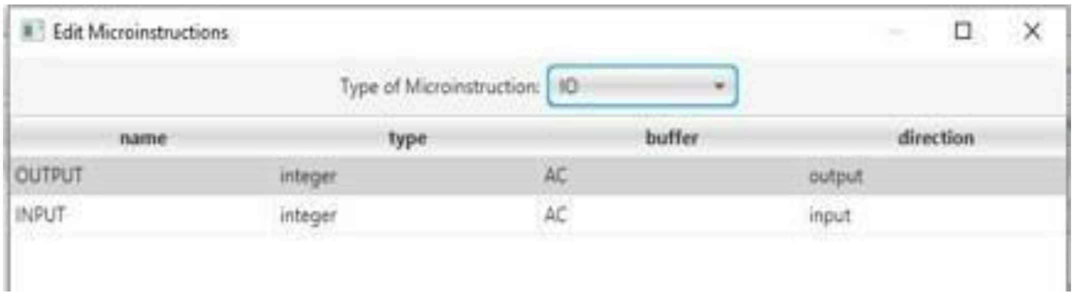


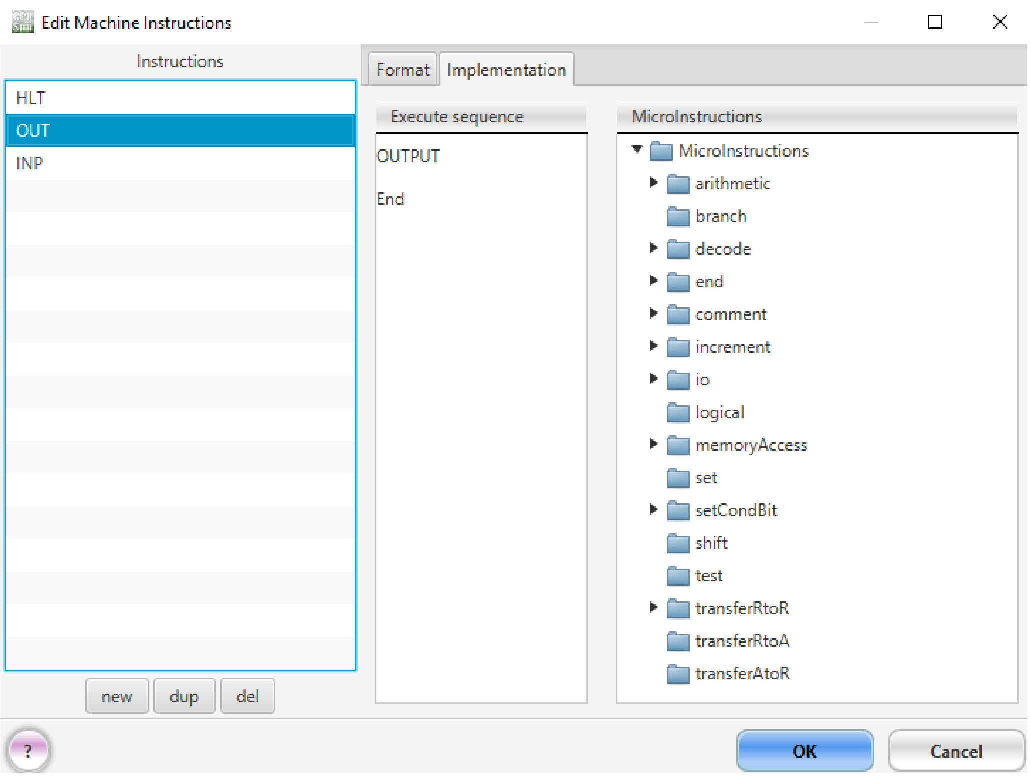


Output

OPCODE: F400

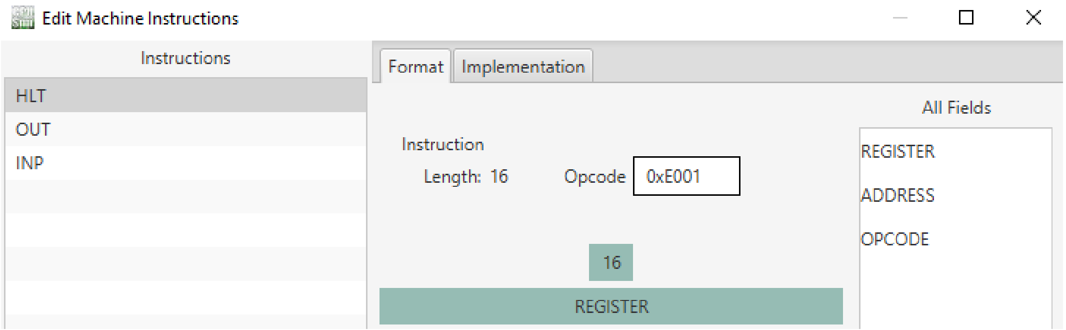




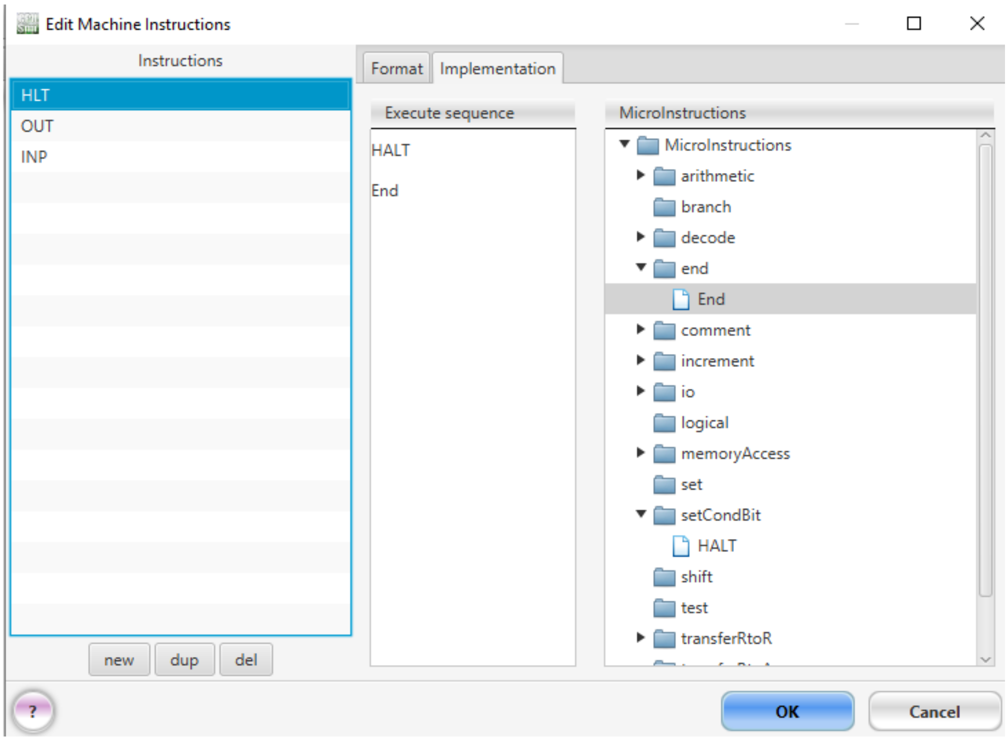


Halt

OPCODE: E001



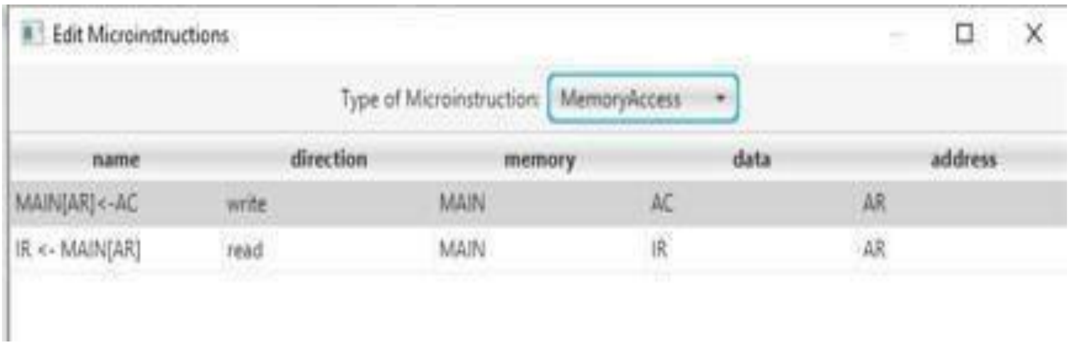
Set condition bit, for halt that is **S**

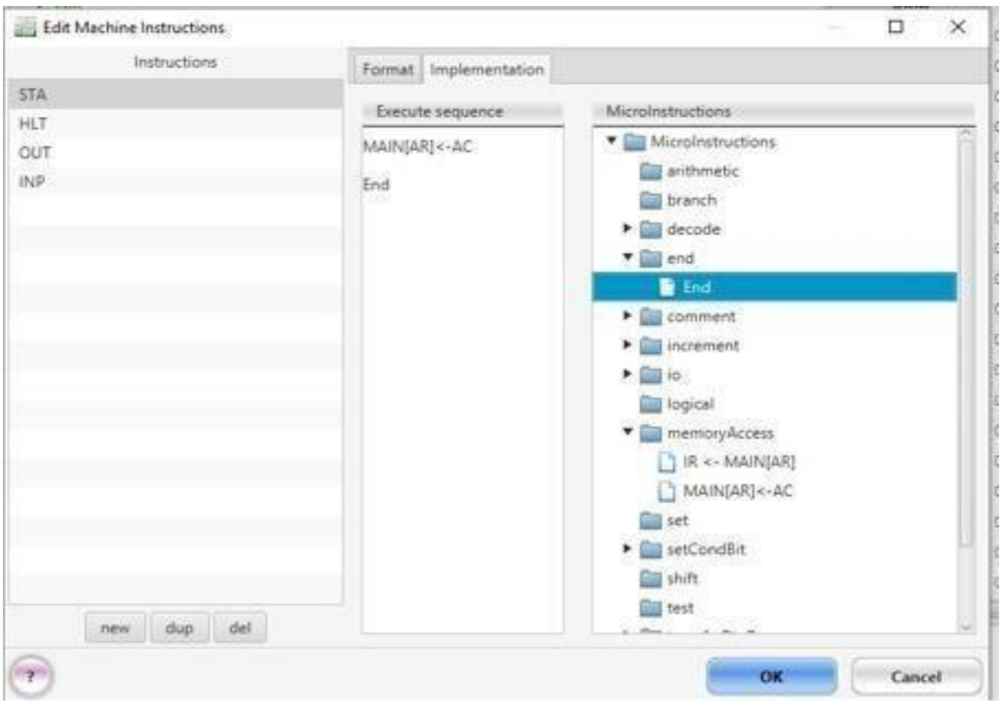


STA

OPCODE: 6

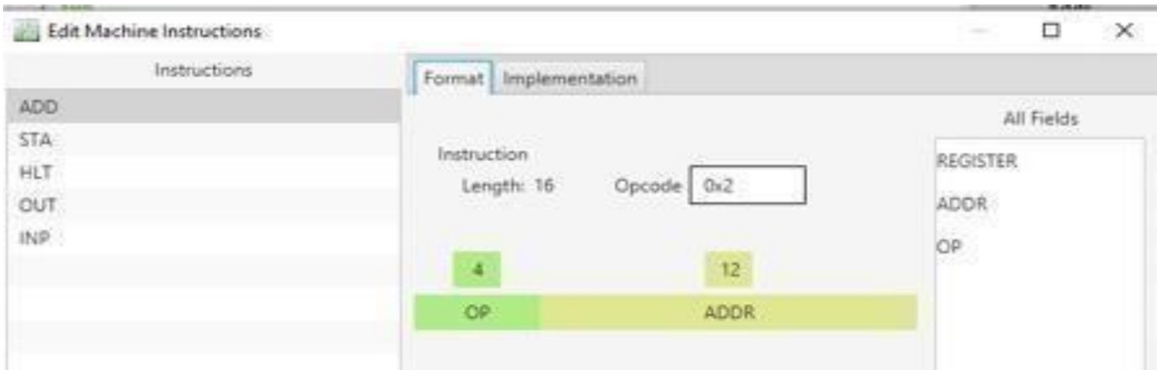

Edit Machine Instructions
Instructions
STA
HLT
OUT
INP
Format
Implementation
All Fields
Instruction
Length: 16
Opcode
0x6
REGISTER
ADDR
OP
4
12
OP
ADDR

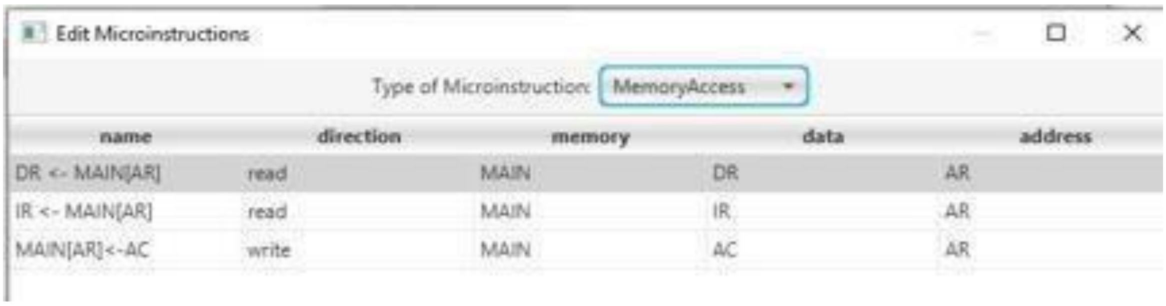

Edit Microinstructions
Type of Microinstruction:
MemoryAccess
name
direction
memory
data
address
MAIN[AR]<-AC
write
MAIN
AC
AR
IR <- MAIN[AR]
read
MAIN
IR
AR


Edit Machine Instructions
Instructions
STA
HLT
OUT
INP
Format
Implementation
Execute sequence
MAIN[AR]<-AC
End
MicroInstructions
MicroInstructions
arithmetic
branch
decode
end
End
comment
increment
io
logical
memoryAccess
IR <- MAIN[AR]
MAIN[AR]<-AC
set
setCondBit
shift
test
new
dup
del
?
OK
Cancel

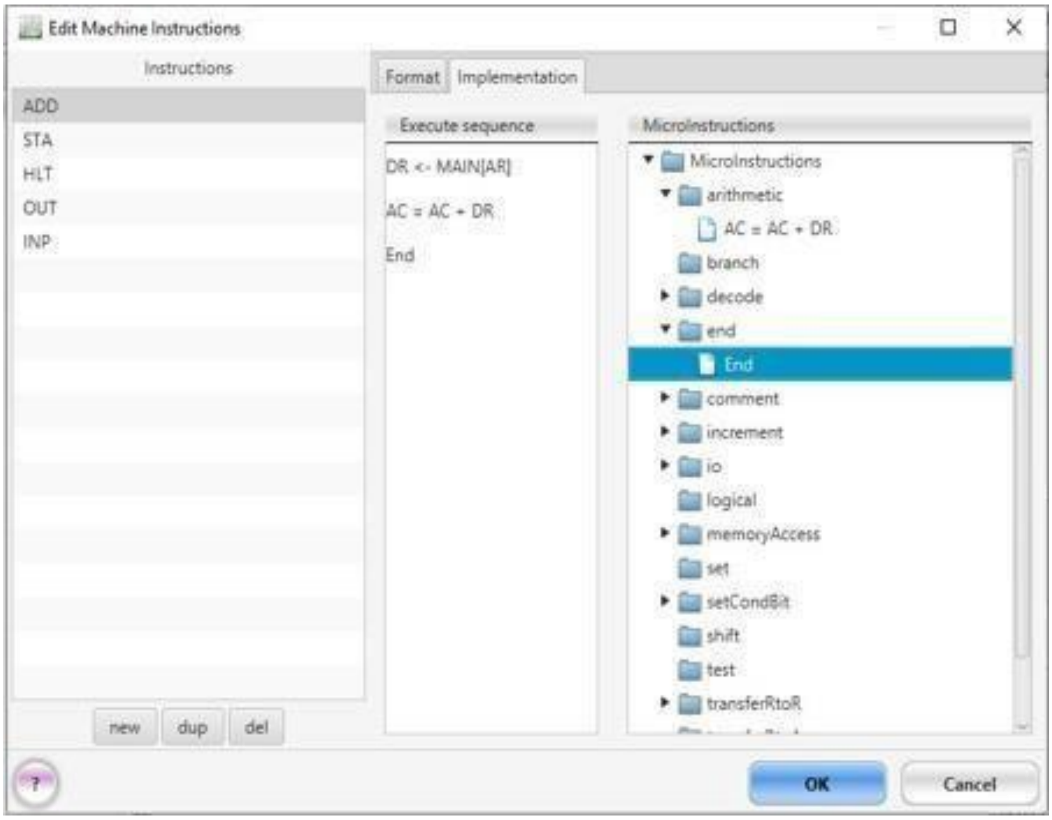
ADD

OPCODE: 2

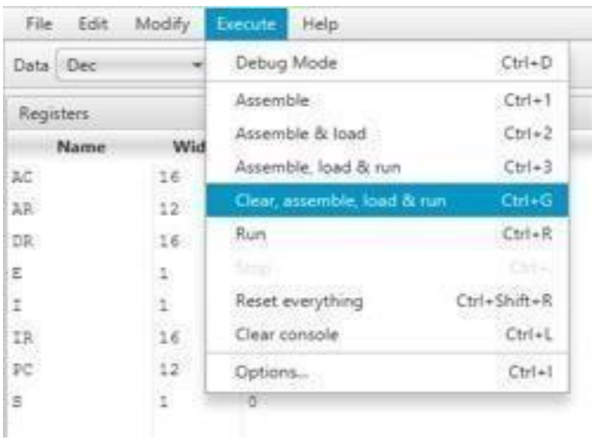

Edit Machine Instructions
Instructions
ADD
STA
HLT
OUT
INP
Format
Implementation
Instruction
Length: 16
Opcode
0x2
4
12
OP
ADDR
All Fields
REGISTER
ADDR
OP


Edit Microinstructions
Type of Microinstruction: MemoryAccess
name
direction
memory
data
address
DR <- MAIN[AR]
read
MAIN
DR
AR
IR <- MAIN[AR]
read
MAIN
IR
AR
MAIN[AR]<-AC
write
MAIN
AC
AR


Edit Microinstructions
Type of Microinstruction: Arithmetic
name
type
source1
source2
destination
overflowBit
carryBit
AC <- AC + DR
ADD
AC
DR
AC
(none)
CARRY-BIT



3. Execute the instructions



4. Enter inputs of numbers

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 3
Enter Inputs, the first of which must be an Integer: 4
Output:  7
EXECUTION HALTED NORMALLY due to the setting of the bit(s): [HALT-BIT]
```

LAB 03

Introduction:

This lab explores subtraction in low-level programming using the Two's Complement method. Students will perform bitwise operations to carry out subtraction and understand its underlying logic. The step-by-step execution of subtraction instructions in assembly will be analyzed. Through hands-on practice, learners will strengthen their grasp of arithmetic operations at the machine level.

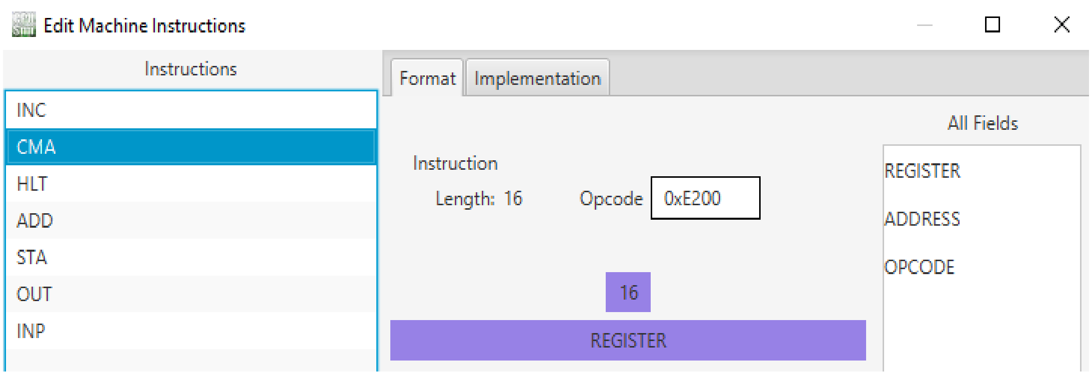
Program

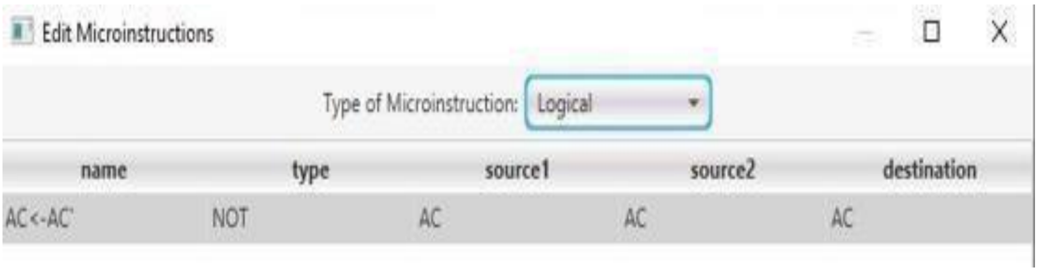
```
START:
INP
STA NUM
INP
CMA         ;Complement (invert) all bits of the second input
            ;(Two's complement preparation for subtraction)

INC         ; Add 1 to the complemented value (
            ;Completing Two's complement to get negative of the second input)

ADD NUM     ; Add the stored first input (NUM) with the negated second input
            ;(Effectively performing subtraction: First Input - Second Input)
OUT
HLT
NUM: .data 1 0
```

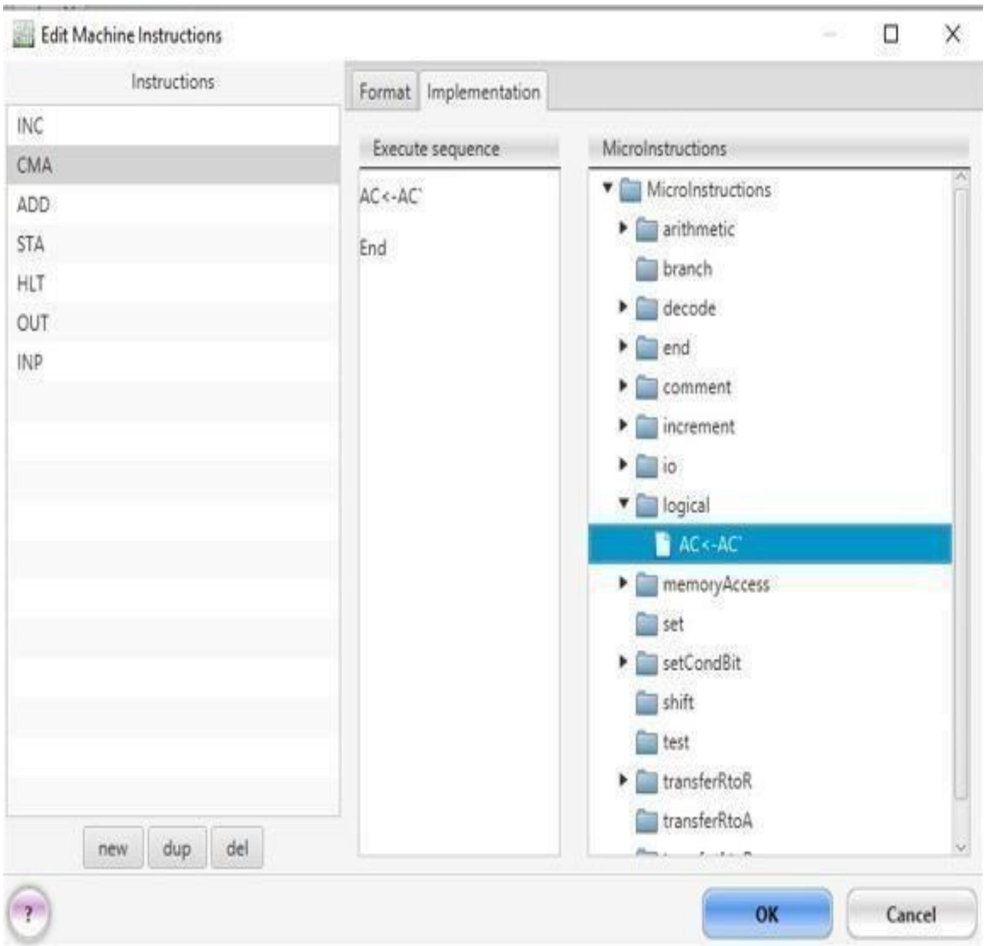
Edit machine instructions

1. CMA instruction:

- Set the **Opcode** for CMA as **E200**.
- Set it as a **Register Instruction**.
- Go to **Implementation** and select **Logical Instruction** with the operation **AC** ← **AC'**.

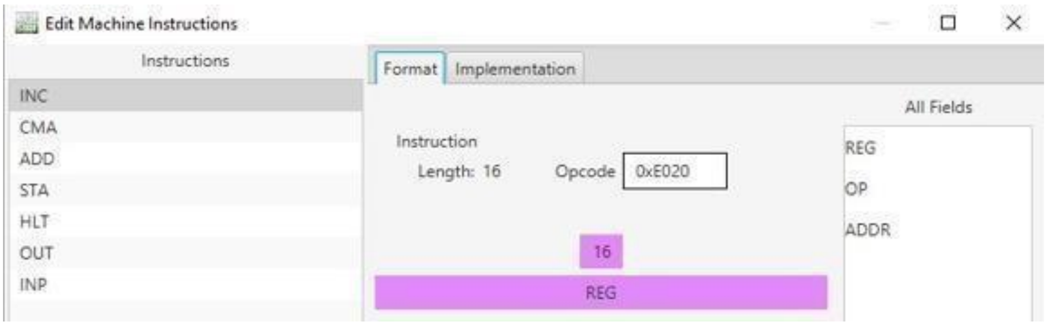


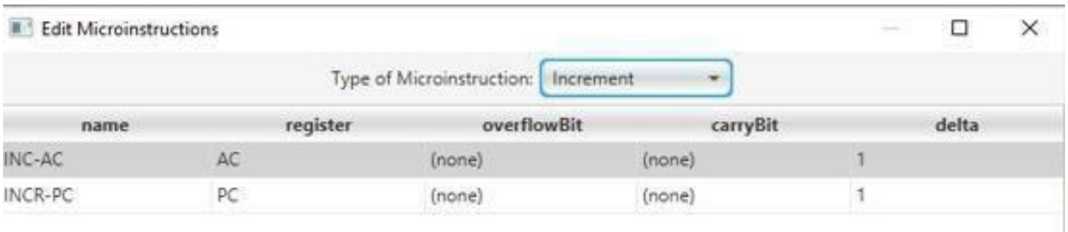

Edit Microinstructions
Type of Microinstruction:
Logical
name
type
source1
source2
destination
AC<-AC'
NOT
AC
AC
AC

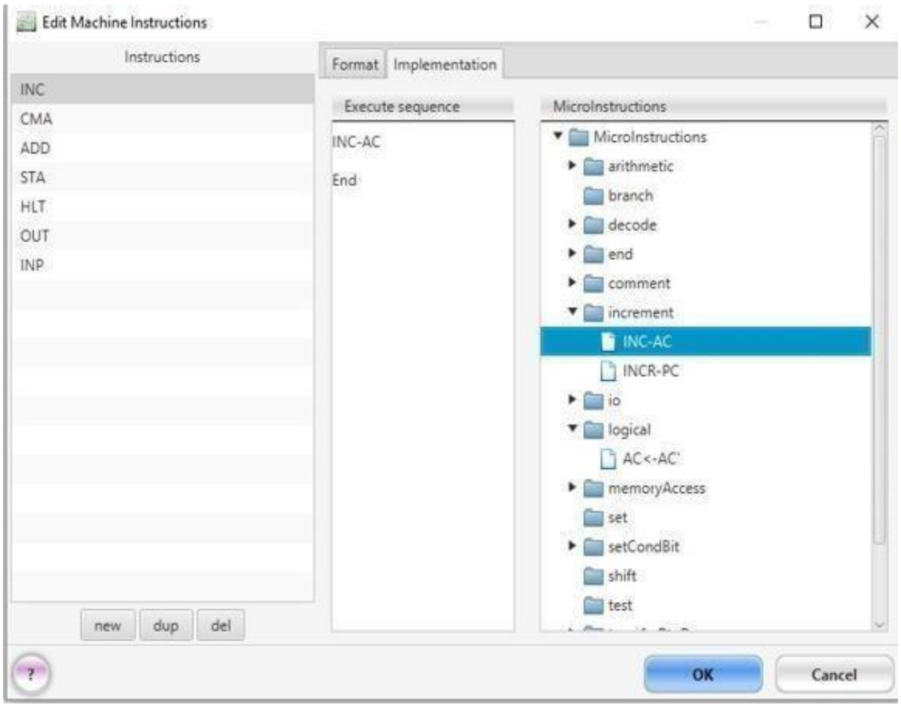

Edit Machine Instructions
Instructions
INC
CMA
ADD
STA
HLT
OUT
INP
new
dup
del
Format
Implementation
Execute sequence
AC<-AC'
End
MicroInstructions
MicroInstructions
arithmetic
branch
decode
end
comment
increment
io
logical
AC<-AC'
memoryAccess
set
setCondBit
shift
test
transferRtoR
transferRtoA
?
OK
Cancel

2. **INC instruction:**

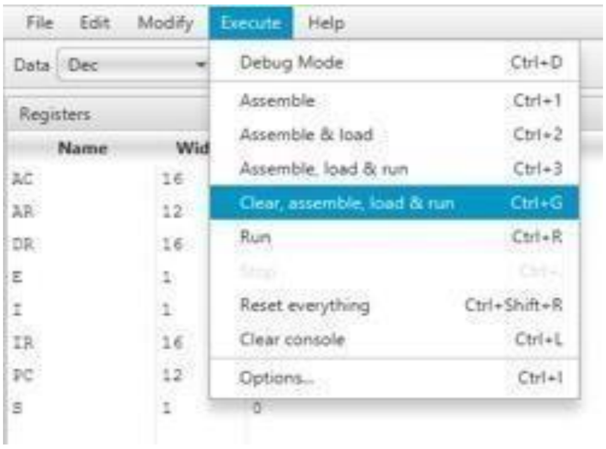
- Set the **Opcode** for INC as **E020**.
- Set it as a **Register Instruction**.
- Select **Increment Instruction** with operation **AC ← AC + 1**.



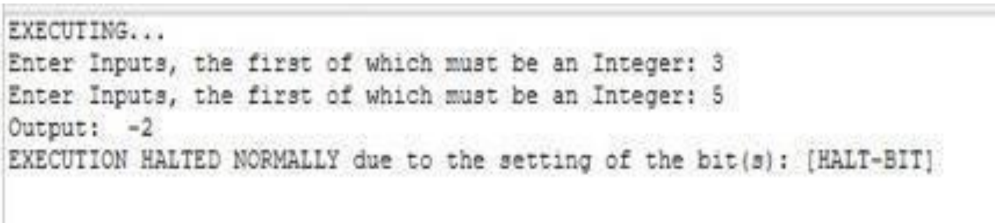




3. Execute the instructions



4. Enter inputs of numbers



```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 3
Enter Inputs, the first of which must be an Integer: 5
Output:  -2
EXECUTION HALTED NORMALLY due to the setting of the bit(s): [HALT-BIT]
```

LAB 04

Introduction:

This lab introduces the use of fundamental bitwise operators essential in digital logic and low-level programming. Students will perform operations using AND, OR, NOT, NAND, NOR, and XOR to manipulate binary data. Each operator's behavior will be explored through practical examples and simulations. This hands-on approach enhances understanding of how binary logic supports system operations.

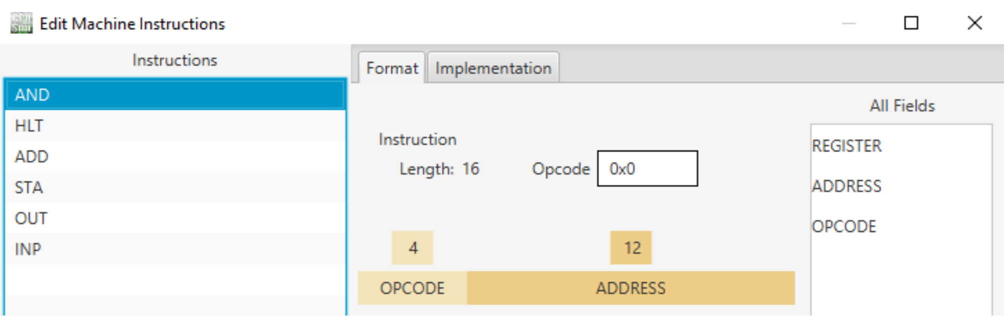
Bitwise AND

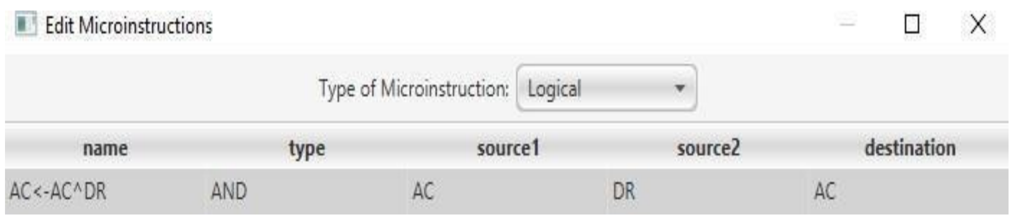
```
START:
INP        ; Input a value and store it in the accumulator
STA NUM    ; Store the accumulator value in memory location "NUM"
INP        ; Input another value
AND NUM    ; use logical shift AND operator
OUT        ; Output the result
HLT        ; Halt execution
NUM: .data 1 0   ; Memory location labeled "NUM", initialized with 0
```

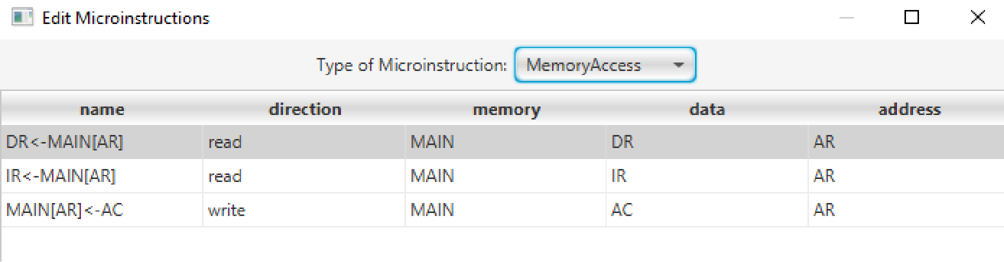
1. **AND instruction:**

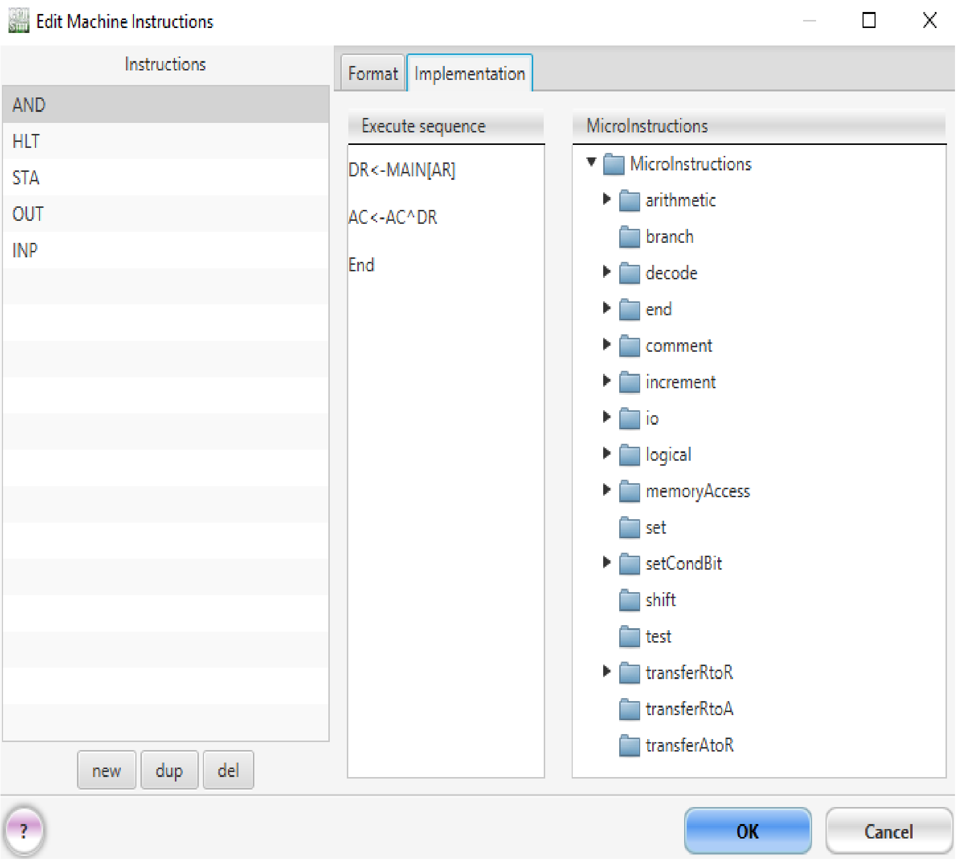
- Set the **Opcode** for AND as **0**.
- Set it as **opcode** and **address**.
- Go to **Implementation** and select **Logical Instruction** with the operation

  AC<-AC^DR
- **Accumulator (AC) and a memory location (M).**
- **Define a Sequence Instruction for execution**









Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 1
Enter Inputs, the first of which must be an Integer: 1
Output:  1
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

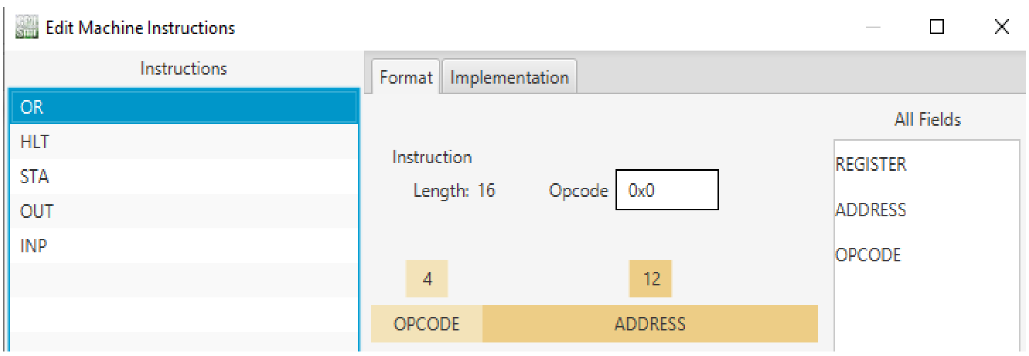
Bitwise OR

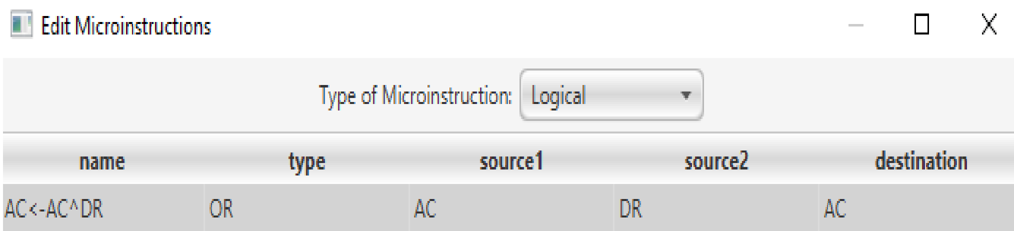
```
START:
INP        ; Input a value and store it in the accumulator
STA NUM    ; Store the accumulator value in memory location "NUM"
INP        ; Input another value
OR NUM   ; use logical shift OR operator
OUT        ; Output the result
HLT        ; Halt execution
NUM: .data 1 0   ; Memory location labeled "NUM", initialized with 0
```

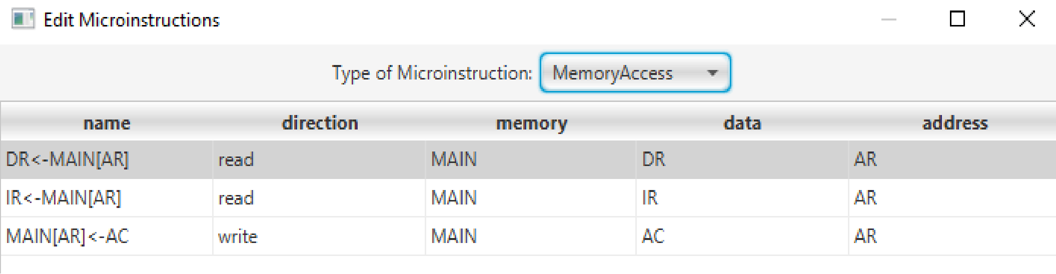
2. OR instruction:

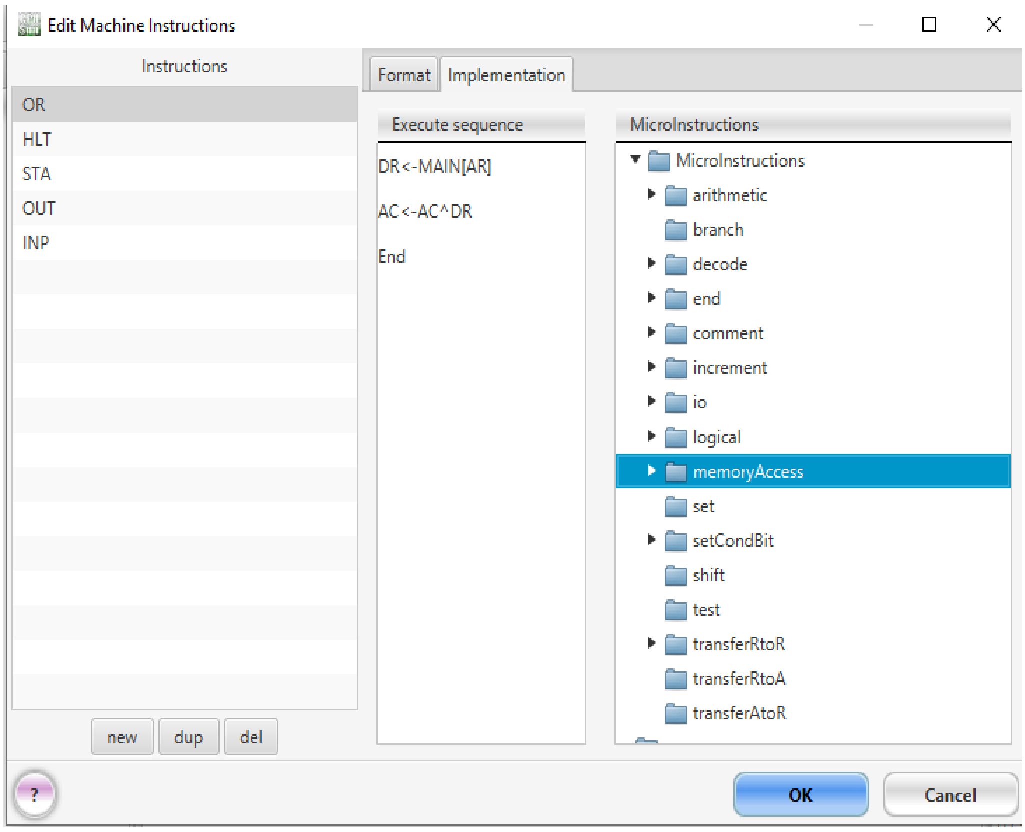
- Set the **Opcode** for OR as **0**.
- Set it as **opcode** and **address**.
- Go to **Implementation** and select **Logical Instruction** with the operation

  AC<-AC^DR
- **Accumulator (AC) and a memory location (M).**
- **Define a Sequence Instruction for execution.**









Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 0
Enter Inputs, the first of which must be an Integer: 1
Output:  1
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

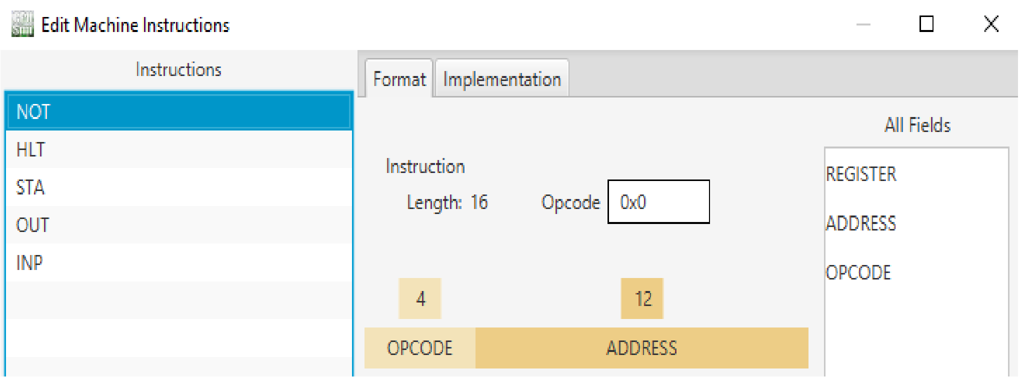
Bitwise NOT

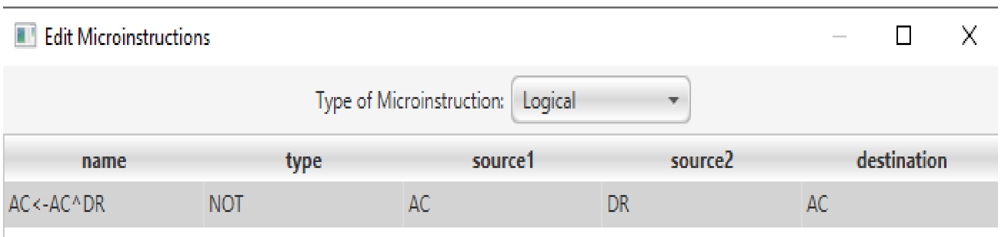
```
START:
INP       ; Input a value and store it in the accumulator
STA NUM   ; Store the accumulator value in memory location "NUM"
NOT NUM   ; use logical shift NOT operator
OUT       ; Output the result
HLT       ; Halt execution
NUM: .data 1 0  ; Memory location labeled "NUM", initialized with 0

```

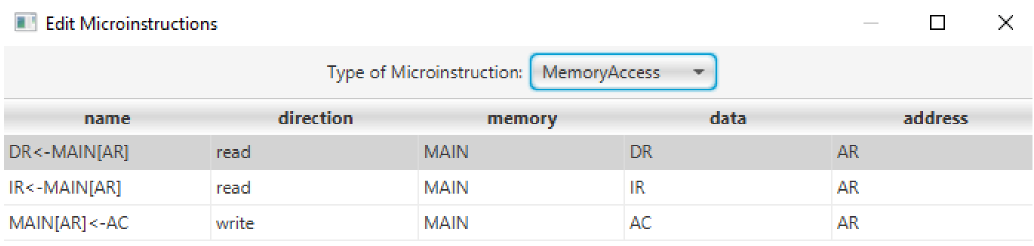
3. **NOT instruction:**

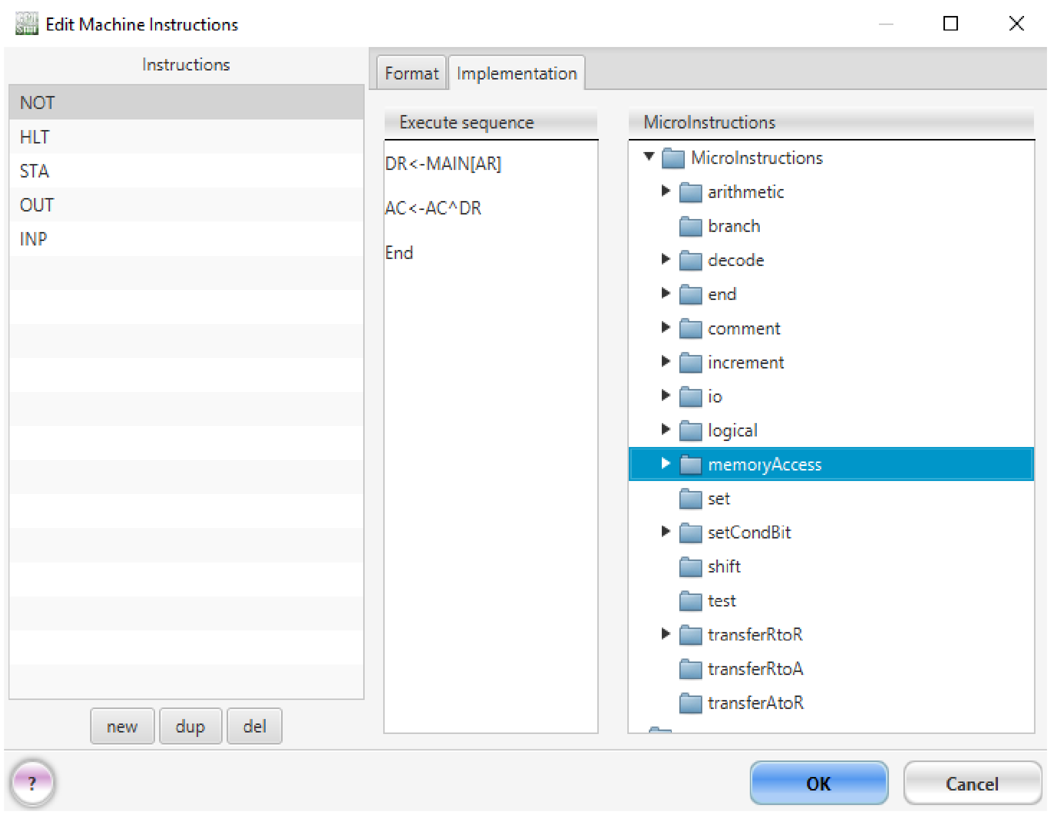
- Set the **Opcode** for NOT as **0**.
- Set it as **opcode** and **address**.
- Go to **Implementation** and select **Logical Instruction** with the operation

  AC<-AC^DR
- **Accumulator (AC) and a memory location (M).**
- **Define a Sequence Instruction for execution**









Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 1
Output:  -2
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

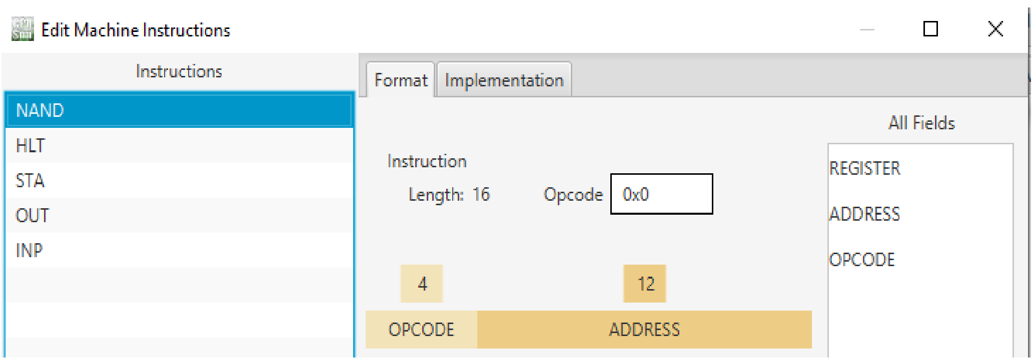
NAND Operator

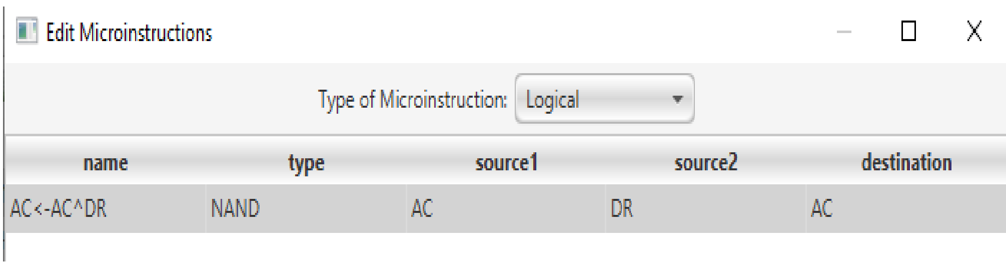
```
START:
INP       ; Input a value and store it in the accumulator
STA NUM   ; Store the accumulator value in memory location "NUM"
INP       ; Input another value
NAND NUM   ; use logical shift NAND operator
OUT       ; Output the result
HLT       ; Halt execution
NUM: .data 1 0  ; Memory location labeled "NUM", initialized with 0

```

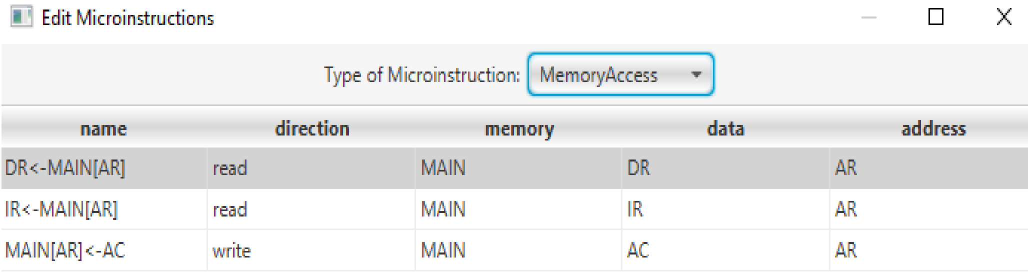
4. **NAND instruction:**

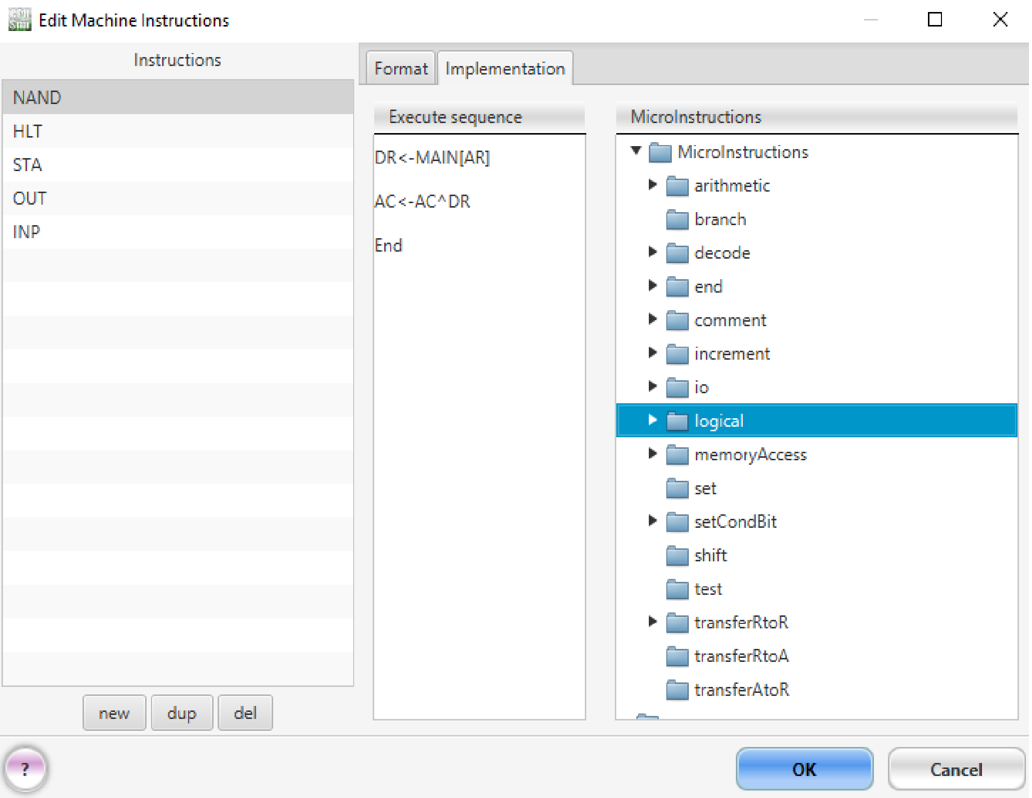
- Set the **Opcode** for NAND as **0**.
- Set it as **opcode** and **address**.
- Go to **Implementation** and select **Logical Instruction** with the operation

  AC<-AC^DR
- **Accumulator (AC) and a memory location (M).**
- **Define a Sequence Instruction for execution.**









Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 3
Enter Inputs, the first of which must be an Integer: 2
Output:  -3
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

NOR Operator

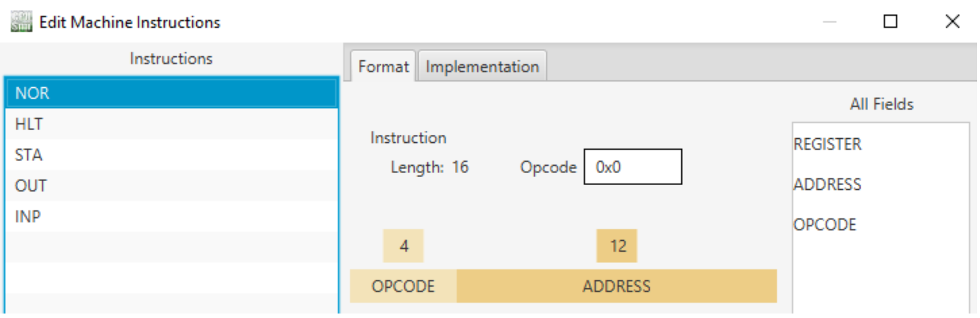
```
START:
INP       ; Input a value and store it in the accumulator
STA NUM   ; Store the accumulator value in memory location "NUM"
INP       ; Input another value
NOR NUM   ; use logical shift NOR operator
OUT       ; Output the result
HLT       ; Halt execution
NUM: .data 1 0   ; Memory location labeled "NUM", initialized with 0
```

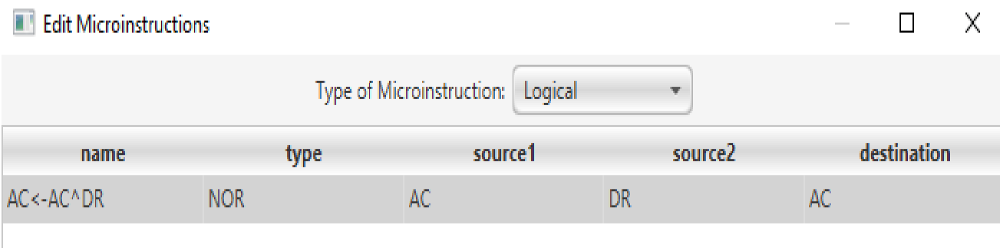
5. **NOR instruction:**

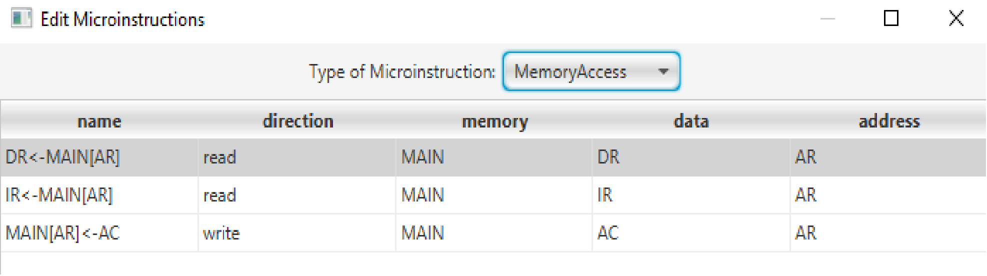
- Set the **Opcode** for NOR as **0**.
- Set it as **opcode** and **address**.
- Go to **Implementation** and select **Logical Instruction** with the operation

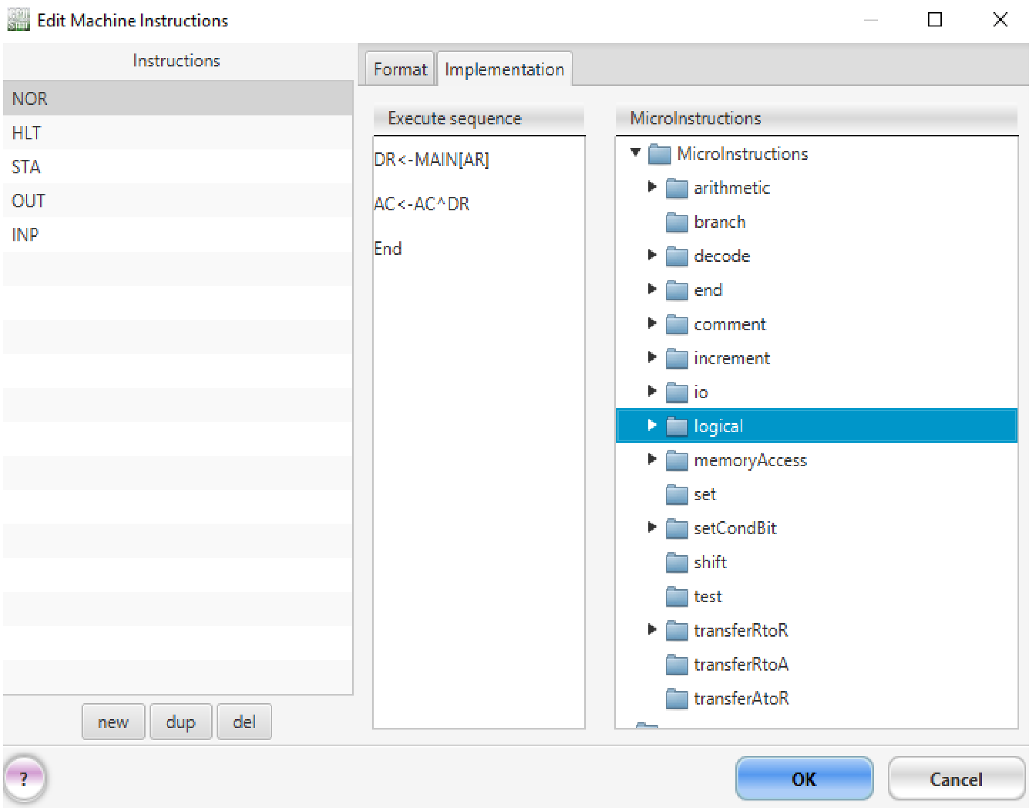
  AC<-AC^DR

- **Accumulator (AC) and a memory location (M).**
- **Define a Sequence Instruction for execution.**









Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 7
Enter Inputs, the first of which must be an Integer: 5
Output:  -8
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

XOR Operator

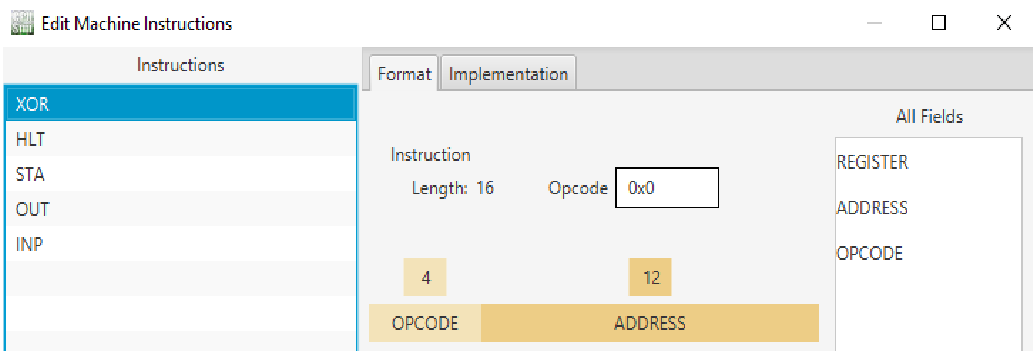
```
START:
INP       ; Input a value and store it in the accumulator
STA NUM   ; Store the accumulator value in memory location "NUM"
INP       ; Input another value
XOR NUM   ; use logical shift XOR operator
OUT       ; Output the result
HLT       ; Halt execution
NUM: .data 1 0  ; Memory location labeled "NUM", initialized with 0
```

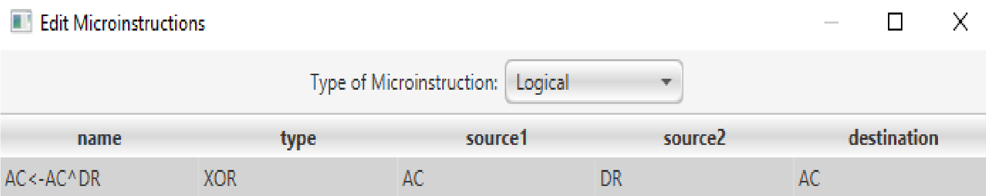
6. **XOR instruction:**

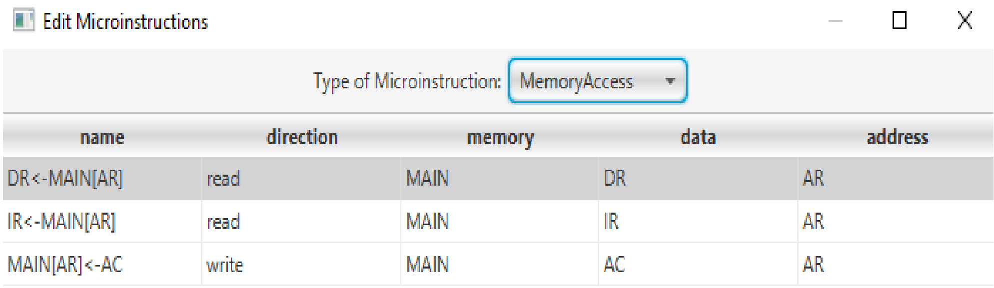
- Set the **Opcode** for XOR as **0**.
- Set it as **opcode** and **address**.
- Go to **Implementation** and select **Logical Instruction** with the operation

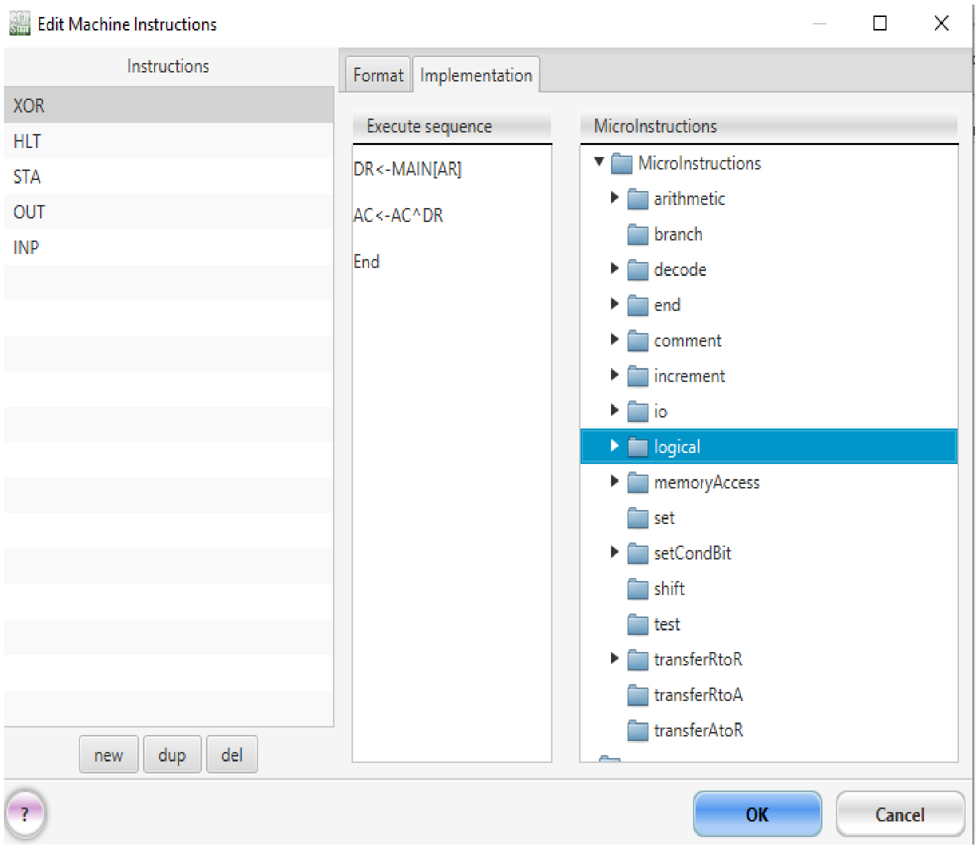
  AC<-AC^DR

- **Accumulator (AC) and a memory location (M).**
- **Define a Sequence Instruction for execution**









Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 7
Enter Inputs, the first of which must be an Integer: 5
Output:  2
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

LAB 05

Introduction:

This lab focuses on implementing multiplication using basic assembly instructions. Students will learn how to take user input, store it in memory with the STA instruction, and retrieve it using LDA. The use of the BUN instruction will demonstrate control flow by jumping to specified memory locations. Through these exercises, learners will deepen their understanding of data handling.

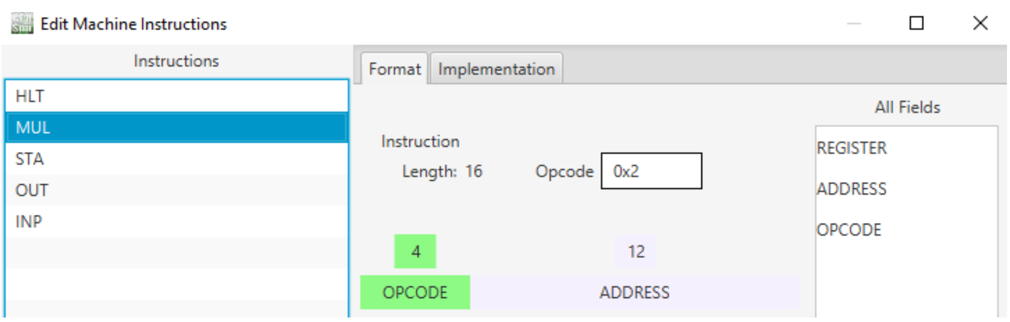
Multiplication

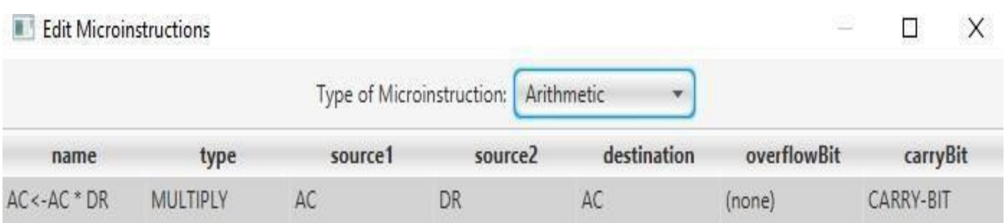
```
START:
INP       ; Input a value and store it in the accumulator
STA NUM   ; Store the accumulator value in memory location "NUM"
INP       ; Input another value
MUL NUM   ; multiply the value stored in "NUM" to the accumulator
OUT       ; Output the result
HLT       ; Halt execution
NUM: .data 1 0   ; Memory location labeled "NUM", initialized with 0

```

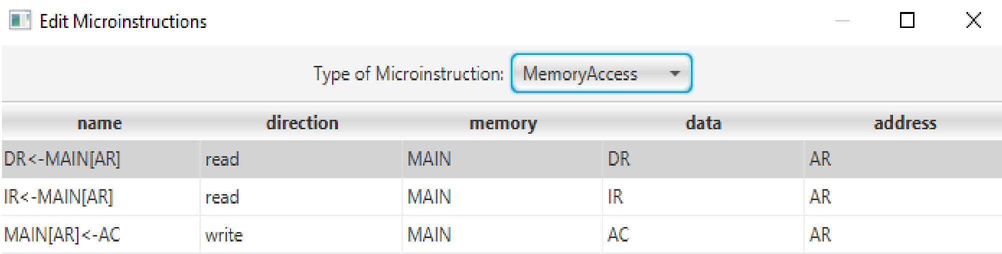
1. MUL instruction:

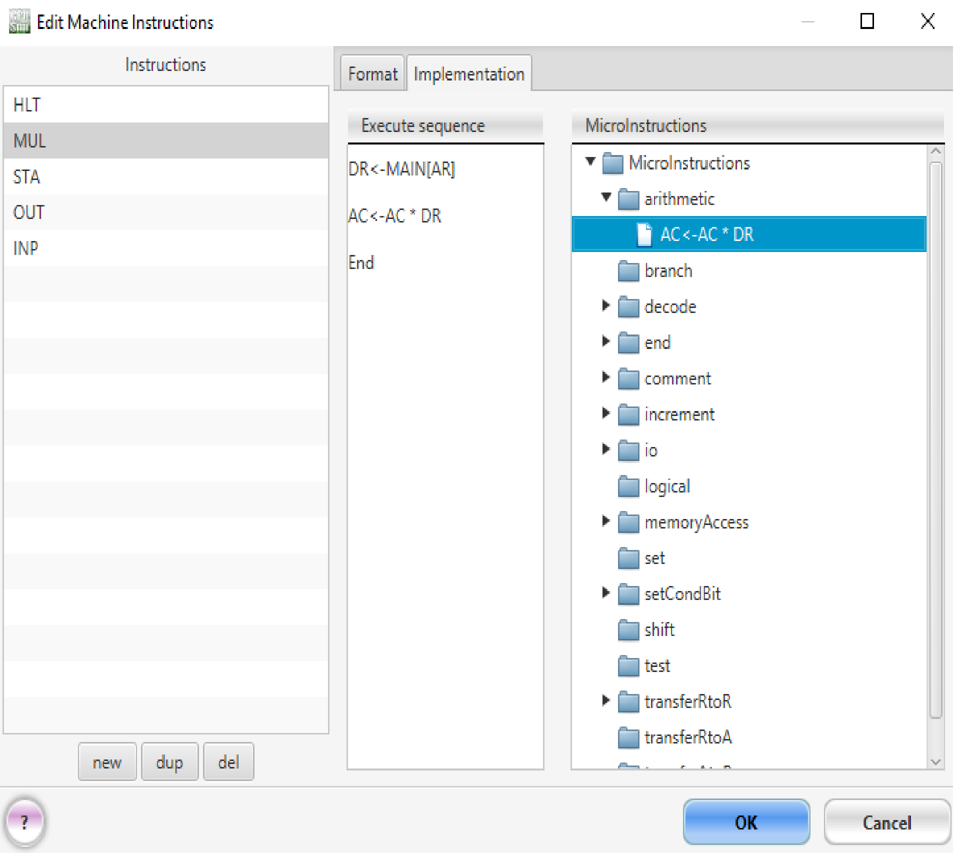
- Set the **Opcode** for MUL as **2**.
- Set it as **opcode** and **address**.
- Go to **Implementation** and select **Arithmetic Instruction** with the operation

  AC<-AC*DR
- **Accumulator (AC) and a memory location (M).**
- **Define a Sequence Instruction for execution**









Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 2
Enter Inputs, the first of which must be an Integer: 3
Output:  6
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

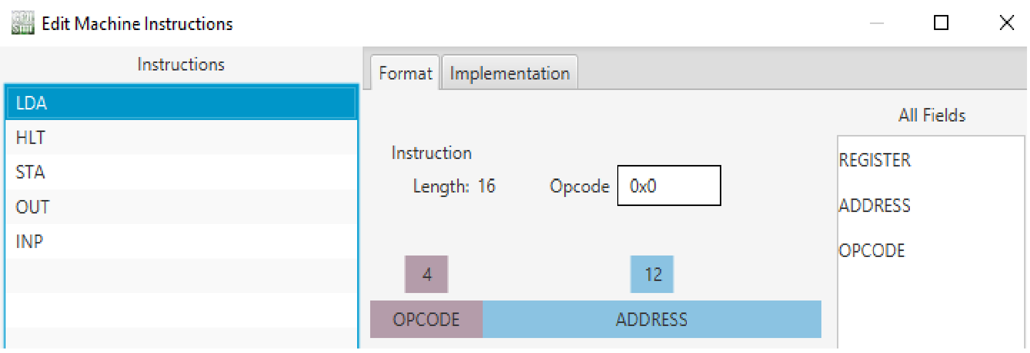
LDA

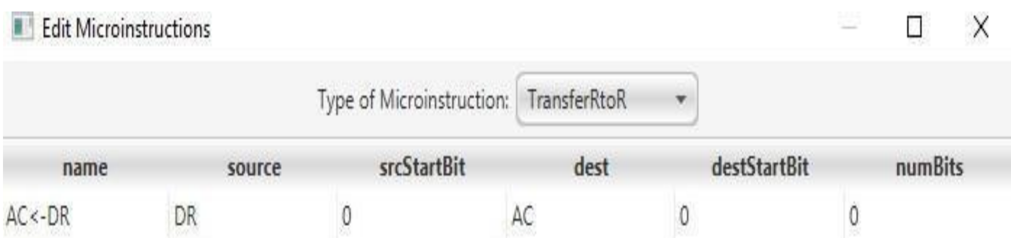
```
START:
INP       ; Input a value and store it in the accumulator
STA NUM   ; Store the accumulator value in memory location "NUM"
LDA NUM ;
OUT       ; Output the result
HLT       ; Halt execution
NUM: .data 1 0   ; Memory location labeled "NUM", initialized with 0

```

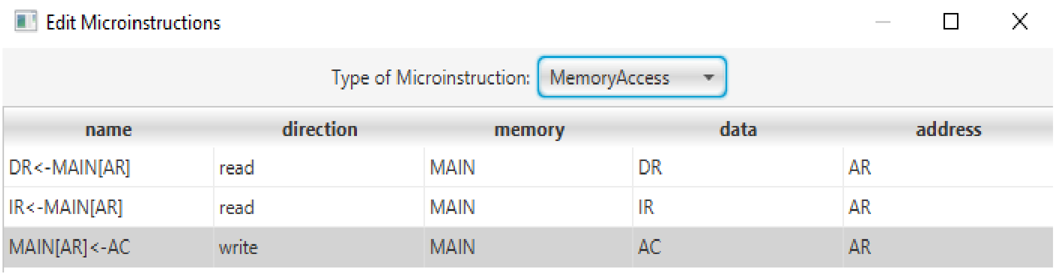
2. **LDA instruction:**

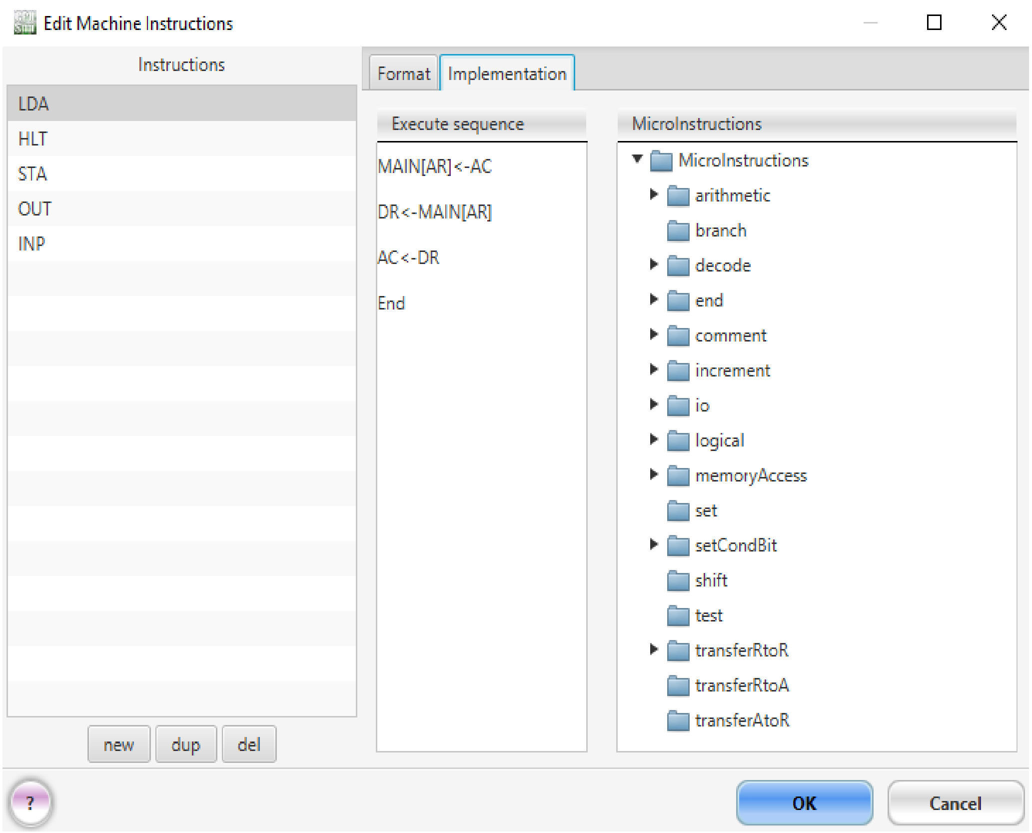
- Set the **Opcode** for LDA as **0**.
- Set it as **opcode** and **address**.
- Go to **Implementation** and select **transferRtoR Instruction** with the operation

  AC<-DR
- **Accumulator (AC) and a memory location (M).**
- **Define a Sequence Instruction for execution.**









Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 3
Output:  3
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

Branch (BUN)

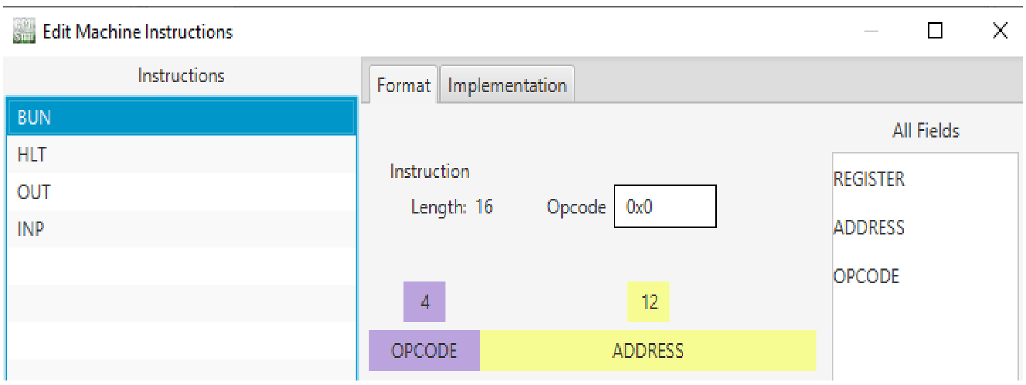
```
START:
INP         ; Input a value and store it in the accumulator
BUN K       ; jump to K block
INP
K: OUT         ; Output the result
HLT         ; Halt execution
NUM: .data 1 0   ; Memory location labeled "NUM", initialized with 0

```

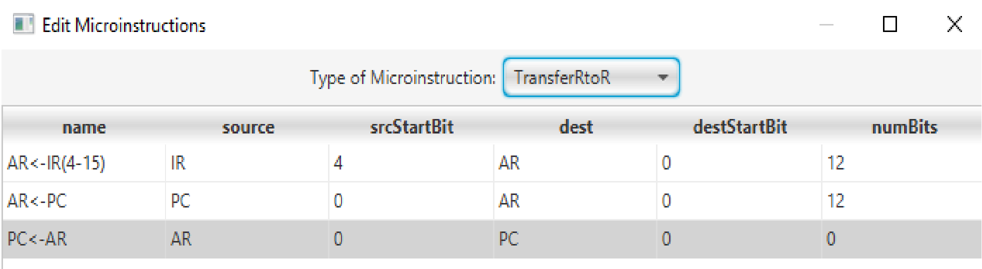
3. BUN instruction:

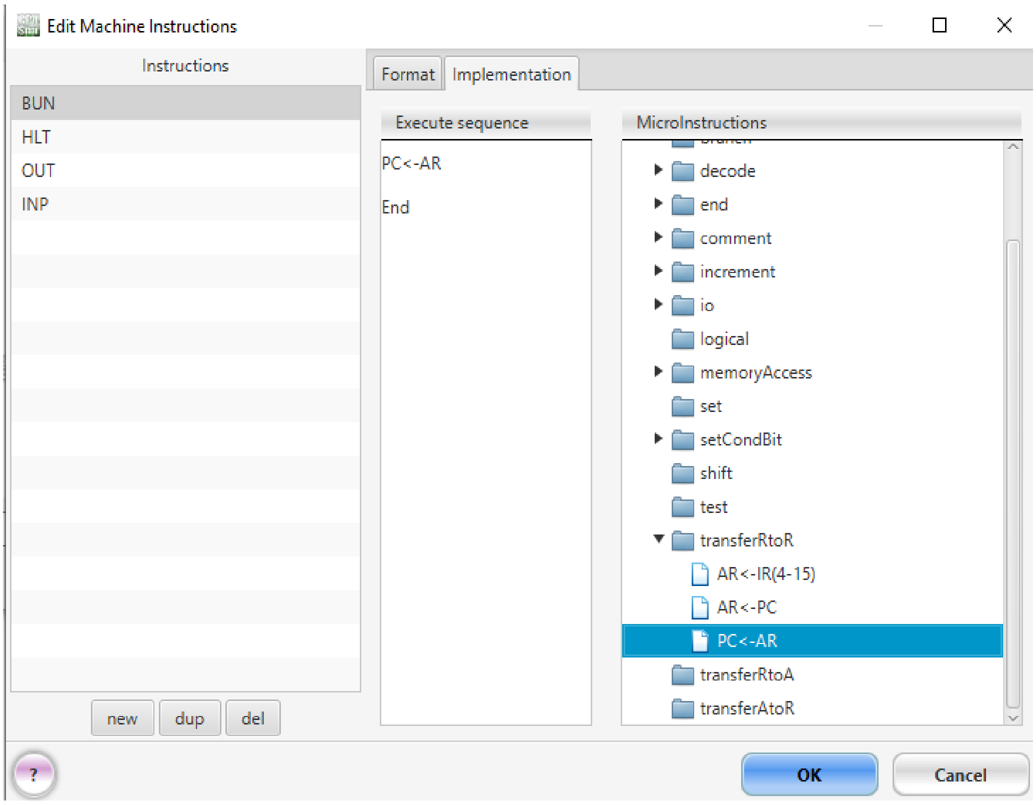
- It jumps to another part of the program without any condition.
- Set the **Opcode** for BUN as **0**.
- Set it as **opcode** and **address**.
- Go to **Implementation** and select **transferRtoR Instruction** with the operation

  PC<-AR

- **Define a Sequence Instruction for execution**







Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 2
Output:  2
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

LAB 06

Introduction:

This lab introduces the Increment and Skip if Zero (ISZ) instruction used in assembly language programming. Students will explore how ISZ operates by incrementing a memory value and conditionally skipping the next instruction. The lab highlights its usefulness in loop control and conditional execution. Through these, learners will gain insight into efficient low-level program design.

ISZ (Increment and Skip if Zero) Operation

The **ISZ (Increment and Skip if Zero)** instruction increments the value stored at a specified memory location. If the result after incrementing is **zero**, the next instruction in the program is **skipped**.

Program

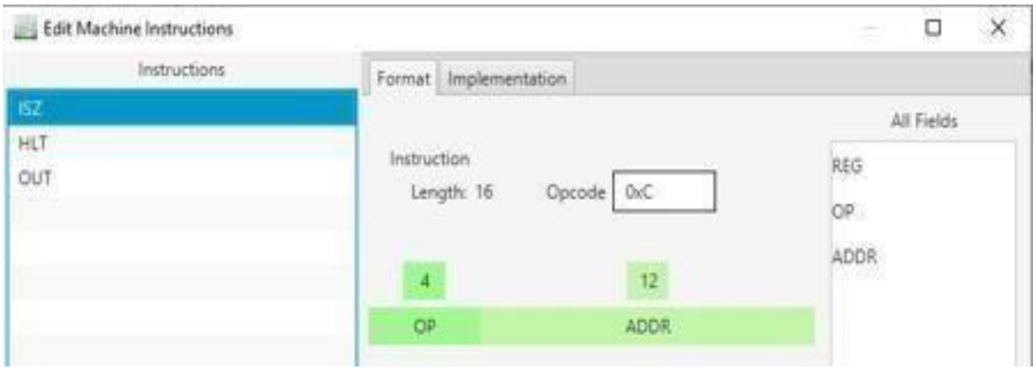
```
START:
ISZ 008
OUT
HLT
NUM: .data 1 0

```

Step-by-Step Execution:

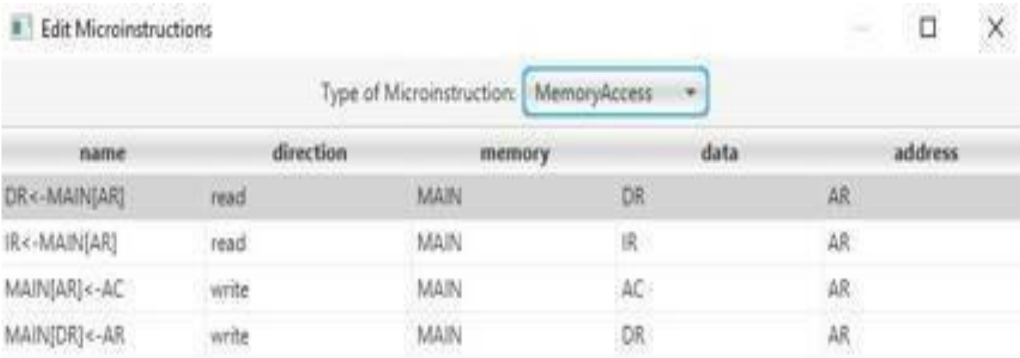
1. **ISZ 009 (Increment and Skip if Zero)**
   - **Reads the value stored at memory location 009.**
   - **Increments it by 1.**
   - **If the new value becomes 0, it skips the next instruction (OUT).**
   - **Otherwise, it proceeds to the next instruction.**
2. **OUT (Output the value in AC)**
   - **If ISZ does not skip, this instruction executes.**
   - **Outputs the value in the accumulator (AC) to the display/output device.**
3. **HLT (Halt Execution)**
   - **Stops program execution.**

Machine instructions

- **Set the upcode and format.**



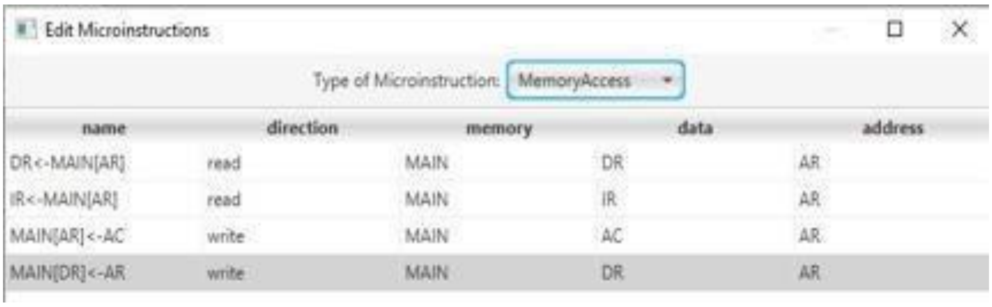
- **Fetch the value from memory (Address Register - AR) into Data Register (DR).**



- Increment the value in DR.

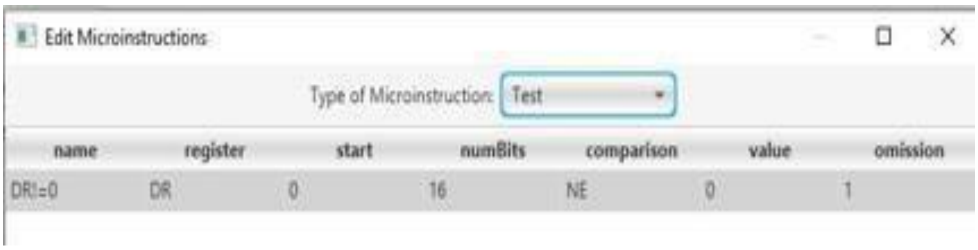
name	direction	memory	data	address
DR<-MAIN[AR]	read	MAIN	DR	AR
IR<-MAIN[AR]	read	MAIN	IR	AR
MAIN[AR]<-AC	write	MAIN	AC	AR
MAIN[DR]<-AR	write	MAIN	DR	AR

- Store the updated value back into memory.



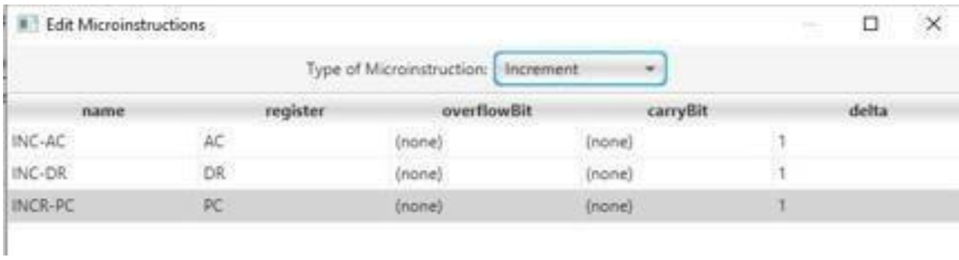
name	direction	memory	data	address
DR<-MAIN[AR]	read	MAIN	DR	AR
IR<-MAIN[AR]	read	MAIN	IR	AR
MAIN[AR]<-AC	write	MAIN	AC	AR
MAIN[DR]<-AR	write	MAIN	DR	AR

- Check if the new value in DR is zero.

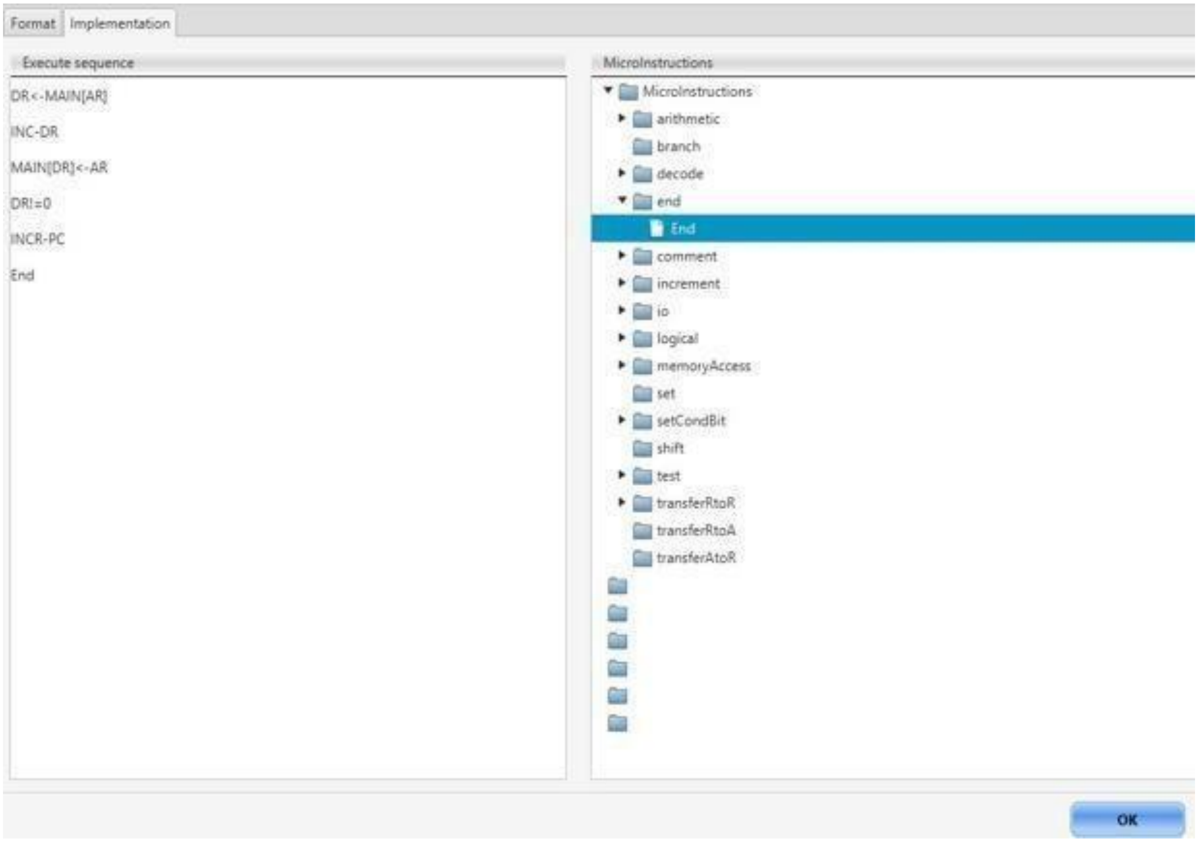


name	register	start	numBits	comparison	value	omission
DR!=0	DR	0	16	NE	0	1

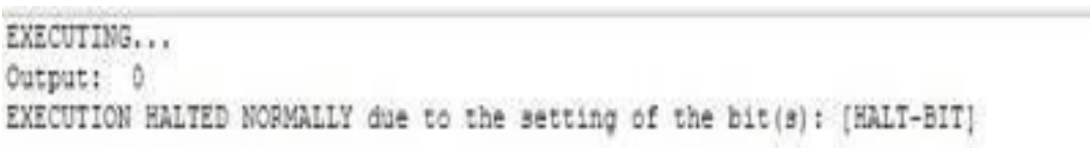
- If the value is zero, increment the Program Counter (PC) to skip the next instruction.



- End the microinstruction sequence



Result



LAB 07

Introduction:

This lab focuses on understanding key instructions such as CLA, CMA, SPA, and SNA. These operations are essential for performing logical and conditional tasks in assembly programming. Through hands-on practice, learners will enhance their ability to write efficient code.

ADD 3 Numbers

ADD instruction is used in performing this task just like the addition of 2 numbers using this **instruction**.

Program

```
START:
INP       ; Input a value and store it in the accumulator
STA NUM   ; Store the accumulator value in memory location "NUM"
INP       ; Input another value
ADD NUM   ; Add the value stored in "NUM" to the accumulator
STA NUM
INP
ADD NUM
OUT       ; Output the result
HLT       ; Halt execution
NUM: .data 1 0   ; Memory location labeled "NUM", initialized with 0
```

Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 3
Enter Inputs, the first of which must be an Integer: 3
Enter Inputs, the first of which must be an Integer: 1
Output:  7
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

CLA Instruction

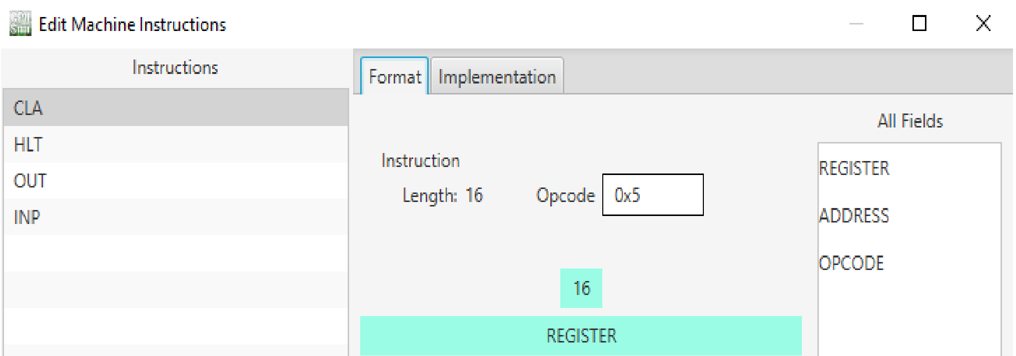
CLA stands for **Clear Accumulator**. It is used to set the value of the accumulator to zero.

Program

```
START:
INP      ; Input a value and store it in the accumulator
CLA      ;Clear the accumulator
OUT      ; Output the result
HLT      ; Halt execution
NUM: .data 1 0  ; Memory location labeled "NUM", initialized with 0
```

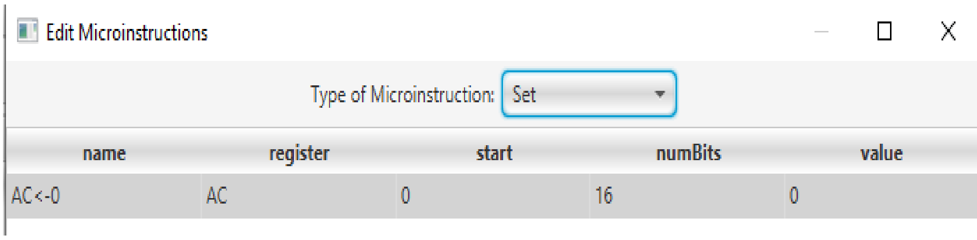
Machine instructions

- **Set the upcode and format.**

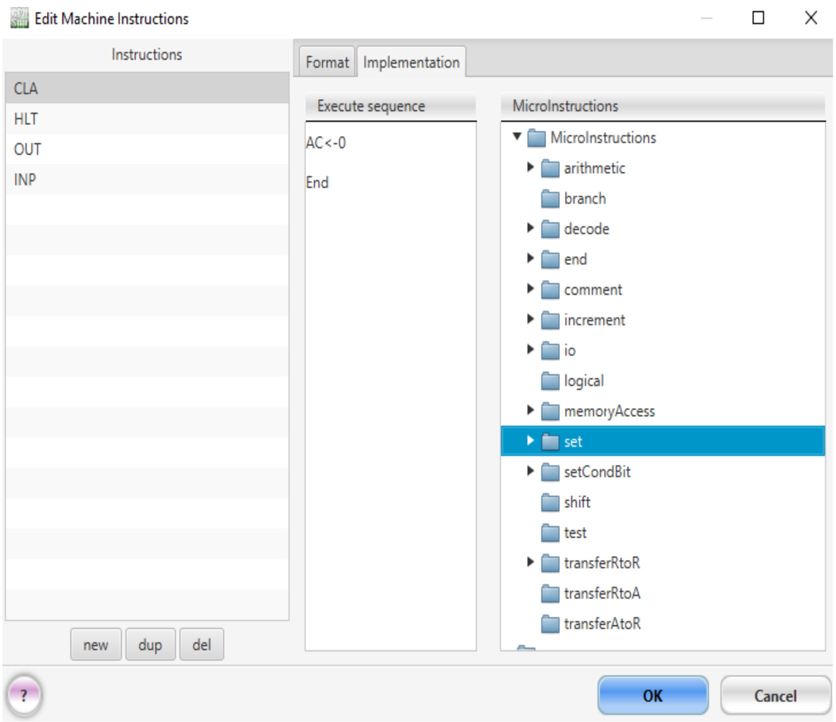


- Go to **Implementation** and select **Set** with the operation.

  AC<- 0



- Define a **Sequence Instruction** for execution.



Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 2
Output:  0
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

CMA Instruction

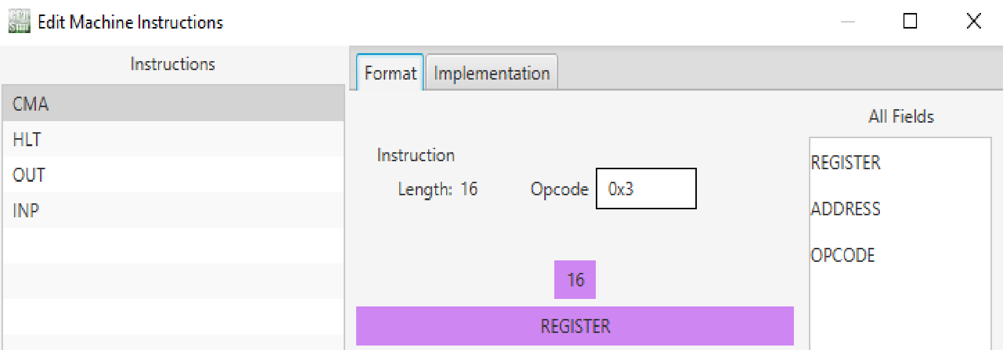
CMA stands for **Complement Accumulator**. It **is an instruction** that performs the operation of bitwise NOT (1's complement) on the accumulator.

Program

```
START:
INP       ; Input a value and store it in the accumulator
CMA       ;Complements the accumulator value
OUT       ; Output the result
HLT       ; Halt execution
NUM: .data 1 0  ; Memory location labeled "NUM", initialized with 0
```

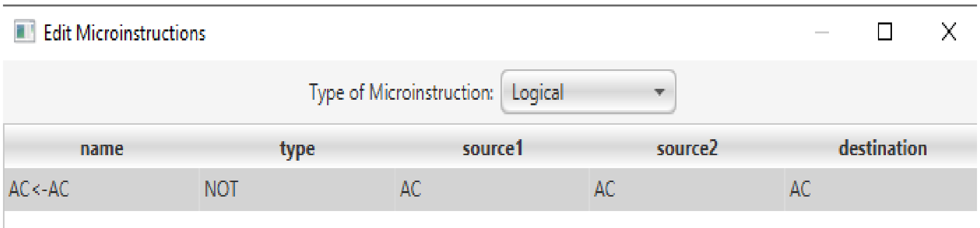
Machine instructions

- **Set the upcode and format.**

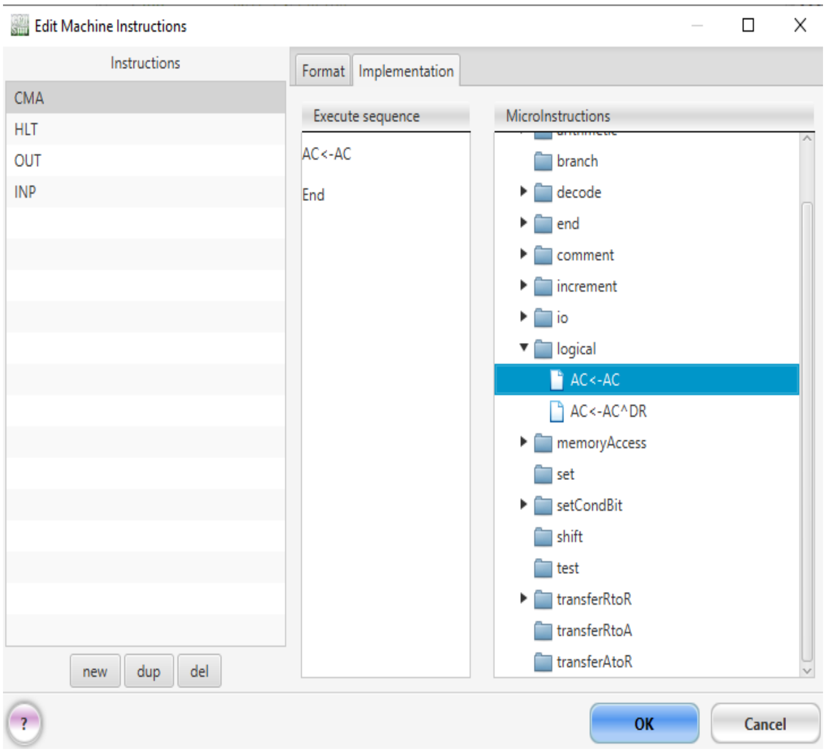


- **Go to Implementation** and select **Logical Instruction** with the operation.

  AC< - AC



- Define a **Sequence Instruction** for execution**.**



Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 5
Output:  -2
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

SPA Instruction

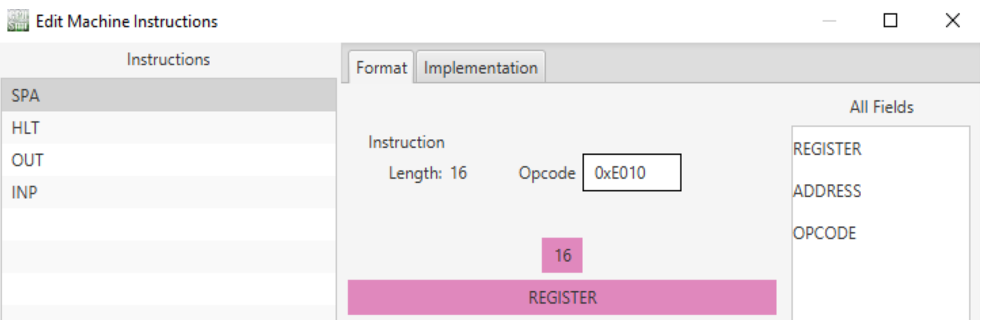
SNA stands for **Skip if Positive Accumulator**. It skips the next instruction if the accumulator holds a Positive value.

Program

```
START:
INP
SPA
OUT
HLT
NUM: .data 1 0

```

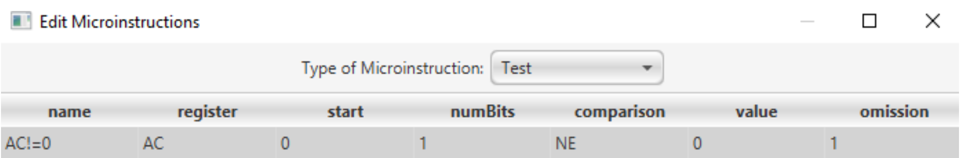
Machine instructions

- **Set the upcode and format.**



- **Go to Implementation** and select **Test** with the operation.

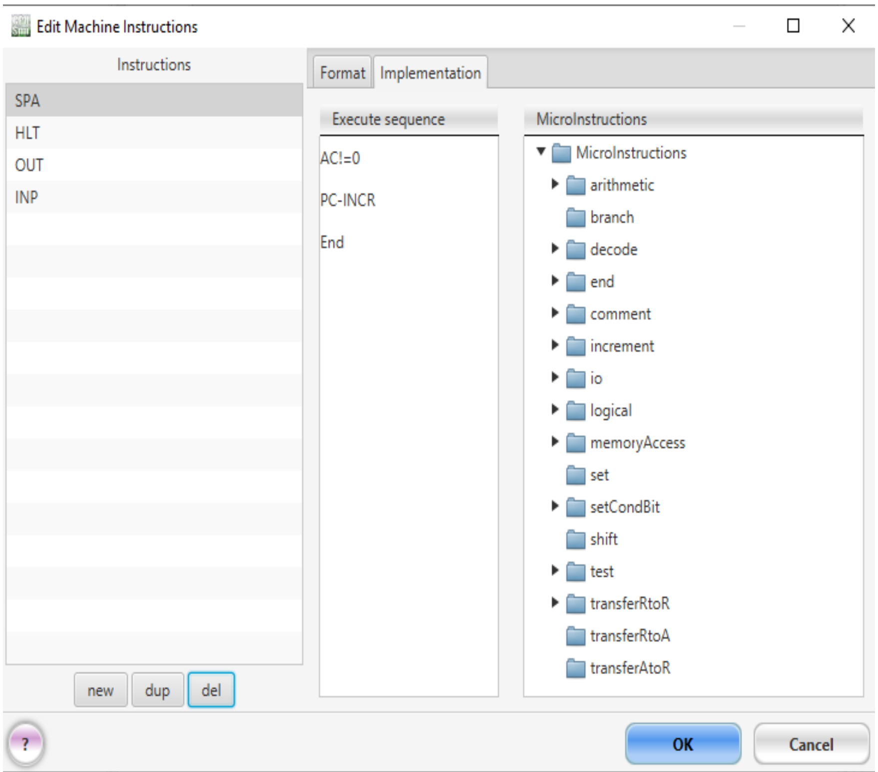
  AC != 0



- **Go to Implementation** and select **Instruction** with the operation.

  PC - INCR

- Define a **Sequence Instruction** for execution**.**



Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 3
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

SNA Instruction

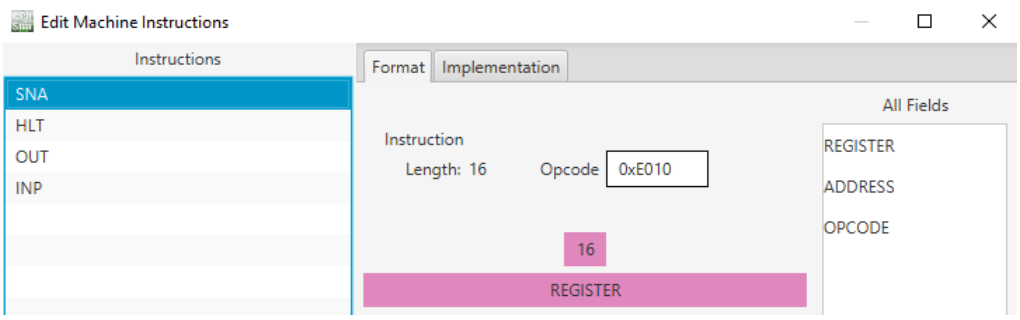
SNA stands for **Skip if Negative Accumulator**. It skips the next instruction if the accumulator holds a negative value.

Program

```
START:
INP
SNA
OUT
HLT
NUM: .data 1 0

```

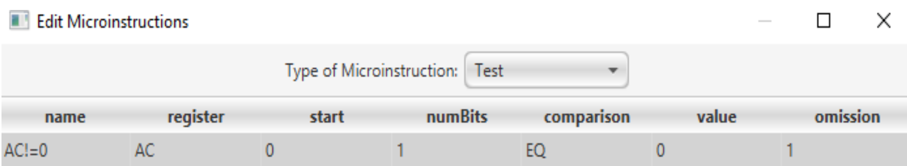
Machine instructions

- **Set the upcode and format.**



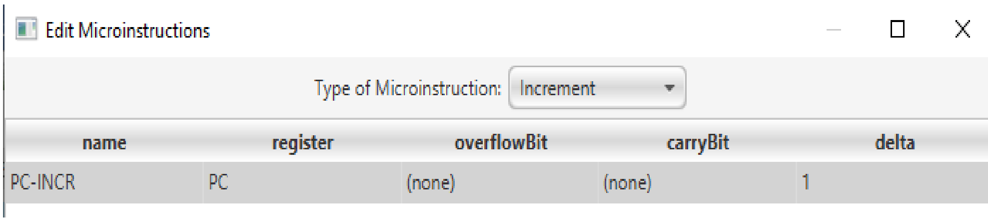
- **Go to Implementation** and select **Test** with the operation.

AC != 0

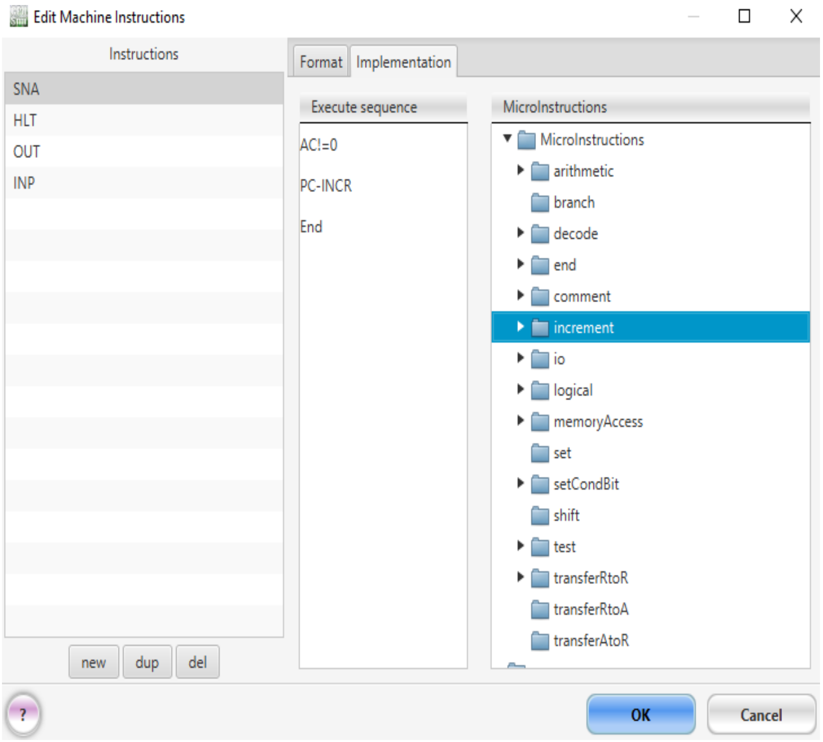


- **Go to Implementation** and select **Increment** with the operation.

  PC - INCR



- Define a **Sequence Instruction** for execution**.**



Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: -2
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

LAB 08

Introduction:

This lab explores the use of SZA, CIR, and CIL instructions in assembly programming. Students will learn how these instructions manipulate data and control program flow. The lab also introduces the JUMPN instruction to create loops that sum input values until a negative number is entered. These tasks provide practical insight into conditional execution and bit-level operations.

SZA Instruction

SZA stands for **Skip if Zero Accumulator.**

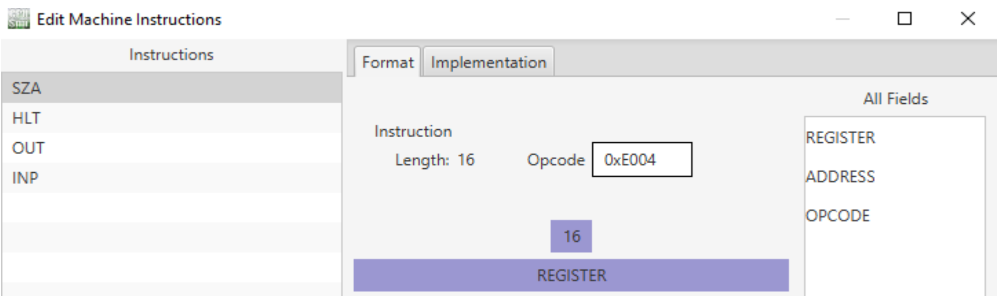
- It's a conditional skip instruction**.**
- It checks if the accumulator **(AC) is zero**, and if so, it **skips the next instruction**.

Program

```
START:
INP   ;Take input from the user
SZA   ;Skip the next instruction if the value in AC is zero
OUT   ;Output the value in AC (only if AC ≠ 0)
HLT   ;Halt the program
NUM: .data 1 0
```

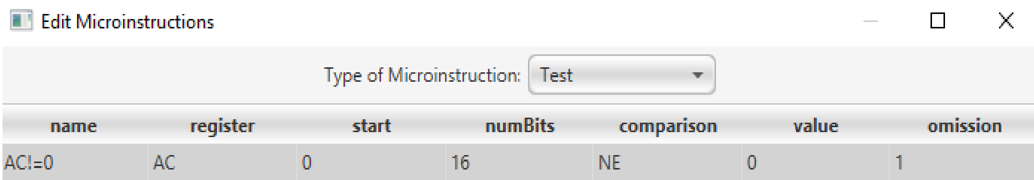
Machine instructions

- **Set the upcode and format.**

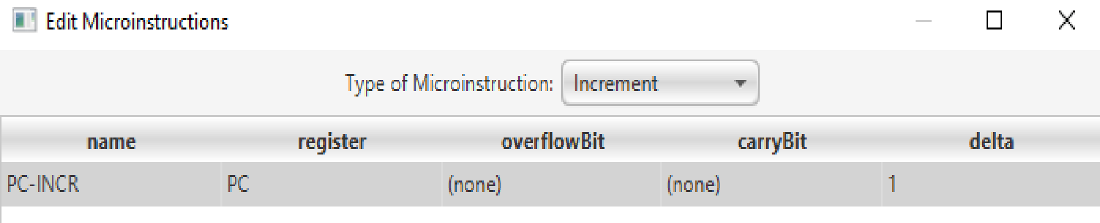


- Go to **Implementation** and select **Test** with the operation.

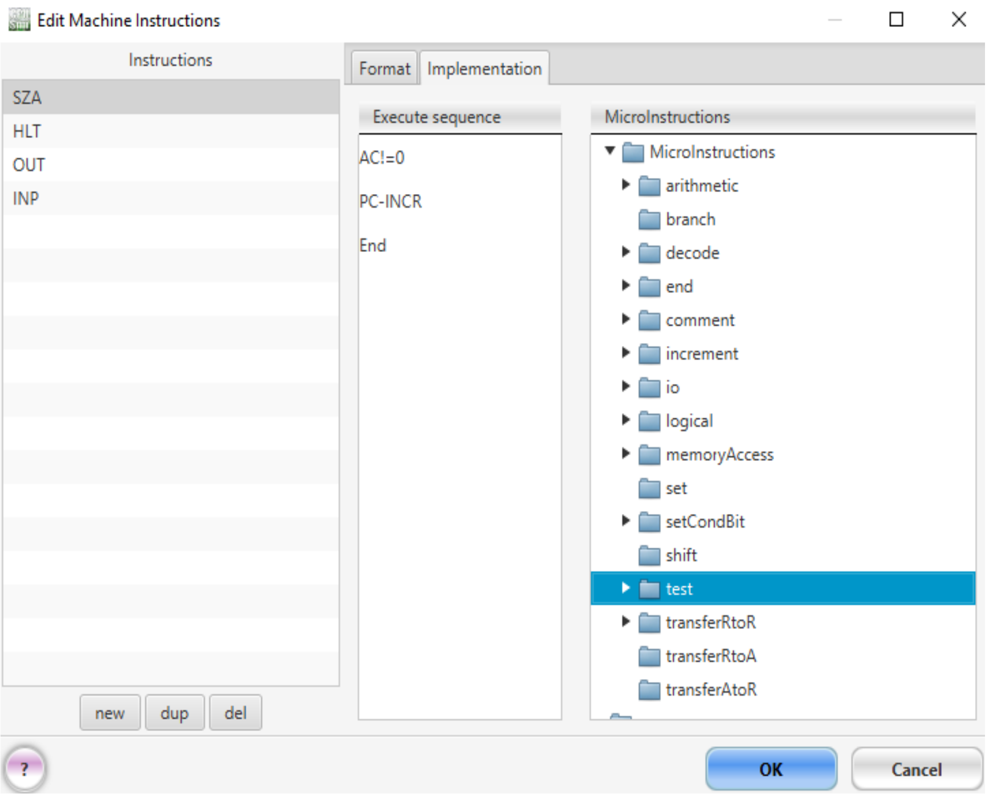
  AC! = 0



- Increment the value in PC.



- Define a **Sequence Instruction** for execution.



Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 0
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

CIR Instruction

CIR stands for **Circular Rotate Right**. It rotates the bits in the **AC** to the right **by one bit**.

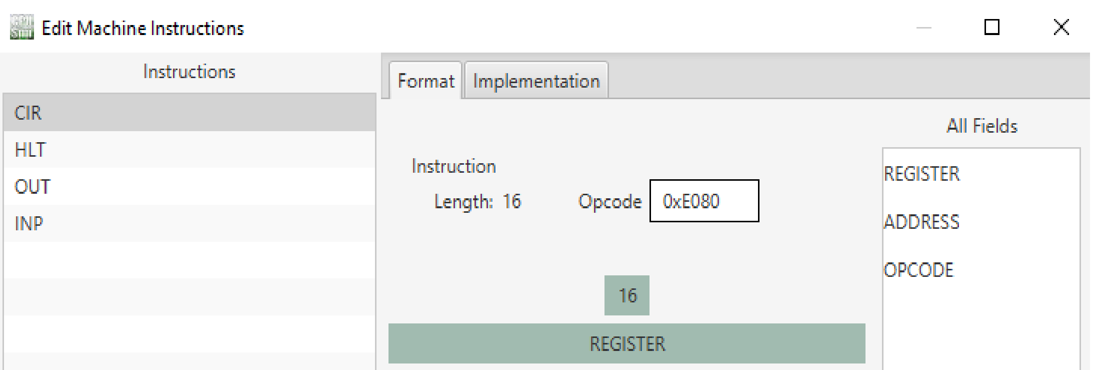
- In CIR, least significant bit **(LSB)** moves to the most significant bit **(MSB)** position.
- This is mathematically similar to **dividing** by **2.**
- It gives correct division results only for even numbers**.**

Program

```
START:
INP   ;Take input from user
CIR   ;Perform a Circular Rotate Right on AC
OUT   ;Output the value in AC
HLT   ;Halt execution
NUM: .data 1 0
```

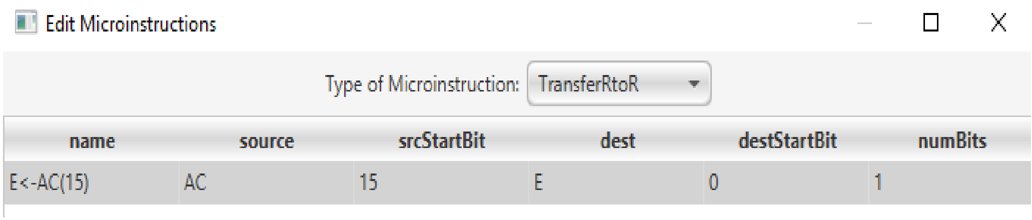
Machine instructions

- **Set the upcode and format.**



- Go to Implementation and select **TransferRtoR** with the operation.

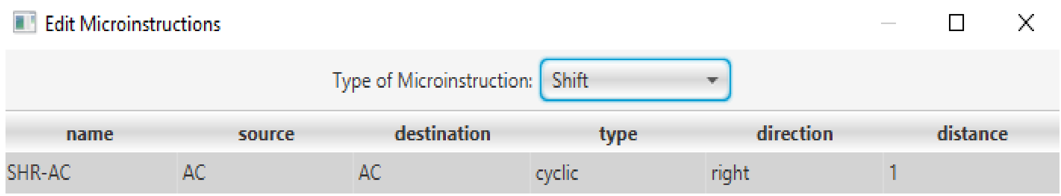
E – AC(15)



name	source	srcStartBit	dest	destStartBit	numBits
E<-AC(15)	AC	15	E	0	1

- Go to Implementation and select **Shift** with the operation.

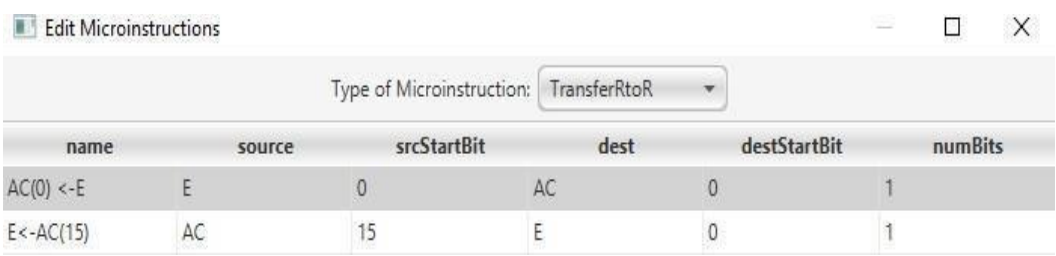
SHR - AC



name	source	destination	type	direction	distance
SHR-AC	AC	AC	cyclic	right	1

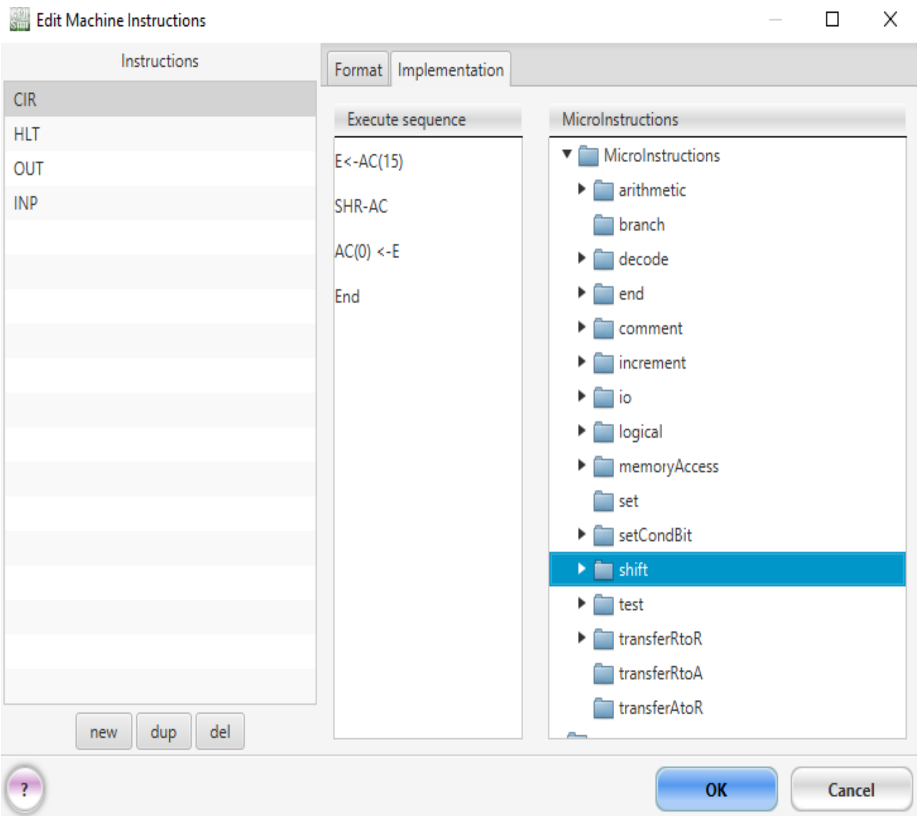
- Go to Implementation and select **TransferRtoR** with the operation.

AC(0) <- E



name	source	srcStartBit	dest	destStartBit	numBits
AC(0) <-E	E	0	AC	0	1
E<-AC(15)	AC	15	E	0	1

- Define a **Sequence Instruction** for execution.



Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 6
Output:  3
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

CIL Instruction

CIL stands for **Circular Rotate Left.** It rotates the bits in the **AC** to the Left **by one bit**.

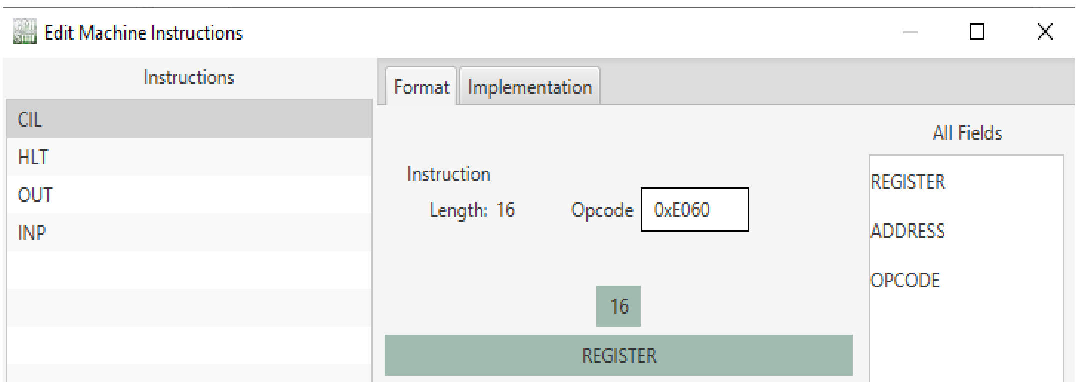
- In CIL, most significant bit **(MSB)** moves to the least significant bit **(LSB)** position.
- This is mathematically similar to **multiplying** by **2.**
- It gives correct multiplication results only for even numbers**.**

Program

```
START:
INP    ;Take input from user
CIL    ;Perform a Circular Rotate Left on AC
OUT    ;Output the value in AC
HLT    ;Halt execution
NUM: .data 1 0

```

Machine instructions

- **Set the upcode and format.**



- Go to Implementation and select **TransferRtoR** with the operation.

E – AC(15)

Edit Microinstructions

Type of Microinstruction: TransferRtoR

name	source	srcStartBit	dest	destStartBit	numBits
E<-AC(15)	AC	15	E	0	1

- Go to Implementation and select **Shift** with the operation.

SHL - AC

Edit Microinstructions

Type of Microinstruction: Shift

name	source	destination	type	direction	distance
SHL-AC	AC	AC	cyclic	left	1

- Go to Implementation and select **TransferRtoR** with the operation.

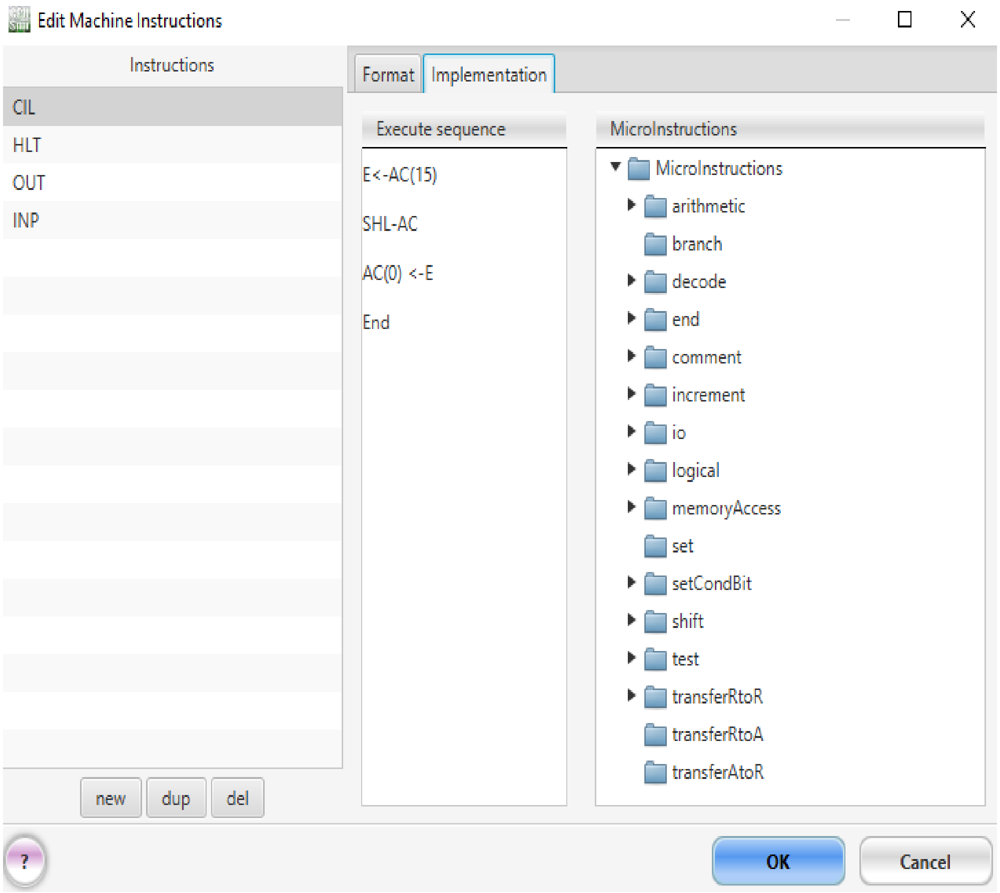
AC(0) <- E

Edit Microinstructions

Type of Microinstruction: TransferRtoR

name	source	srcStartBit	dest	destStartBit	numBits
AC(0) <-E	E	0	AC	0	1
E<-AC(15)	AC	15	E	0	1

- Define a **Sequence Instruction** for execution.



Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 2
Output:  4
EXECUTION HALTED DUE TO AN EXCEPTION: The step is out of range
at step 0 of HLT.
```

JUMPN Instruction

JUMPN stands for **Jump if Negative**.

- It checks the value in the **Accumulator (AC).**
- If the value is **negative**, it **jumps to the specified address/label**.
- If the value is **zero or positive**, it **continues with the next instruction**.

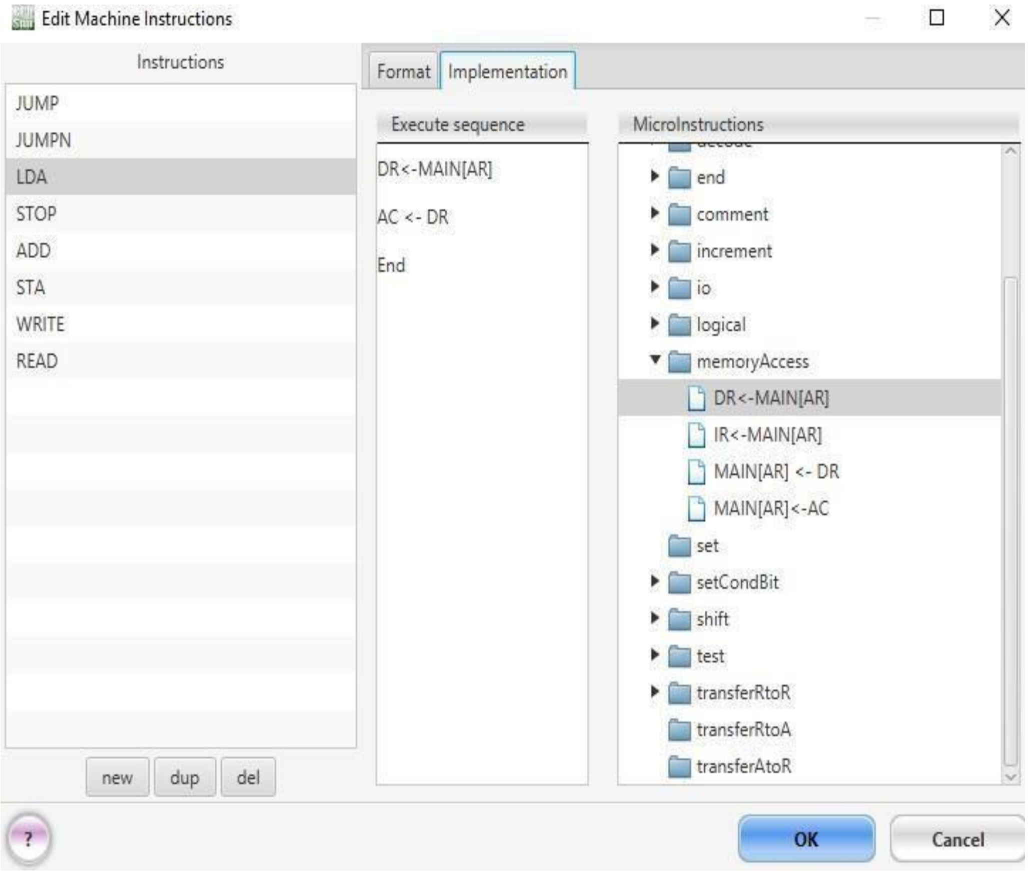
Program

```
START:
  READ            ;Read a number from the user
  JUMPN DONE      ;If the number is negative, jump to DONE
  ADD SUM         ;Add the value
  STA SUM
  JUMP START      ;jump to Start

DONE:
  LDA SUM        ;Load the final sum into AC
  WRITE          ;Output
  STOP           ;End the program

SUM: .data 2 0
```

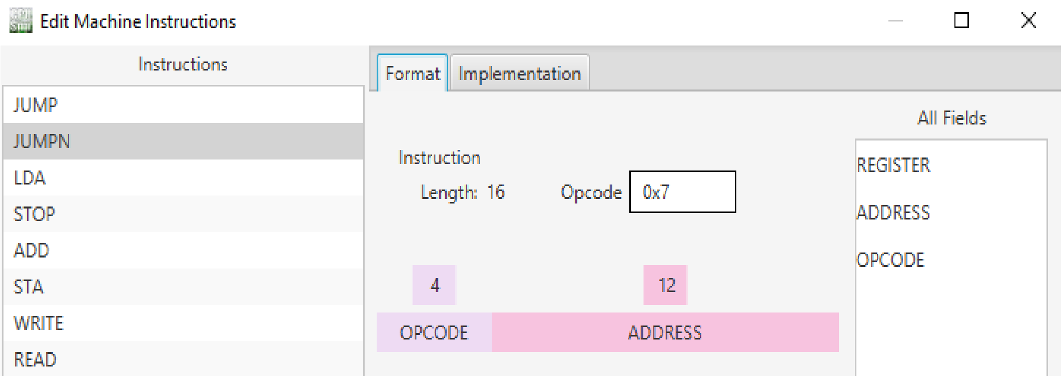
Machine instructions

- Rename **INP to Read.**
- Rename **OUT to Write.**
- Rename **HLT to STOP**
- **STA** remains same.
- **ADD** remains same.
- **JUMP** also remains same.

- Define a **Sequence Instruction** for execution for **LDA.**



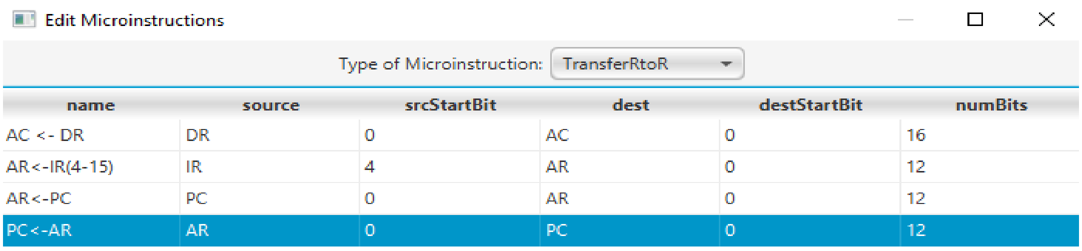
- **Set the upcode and format for JUMPN.**



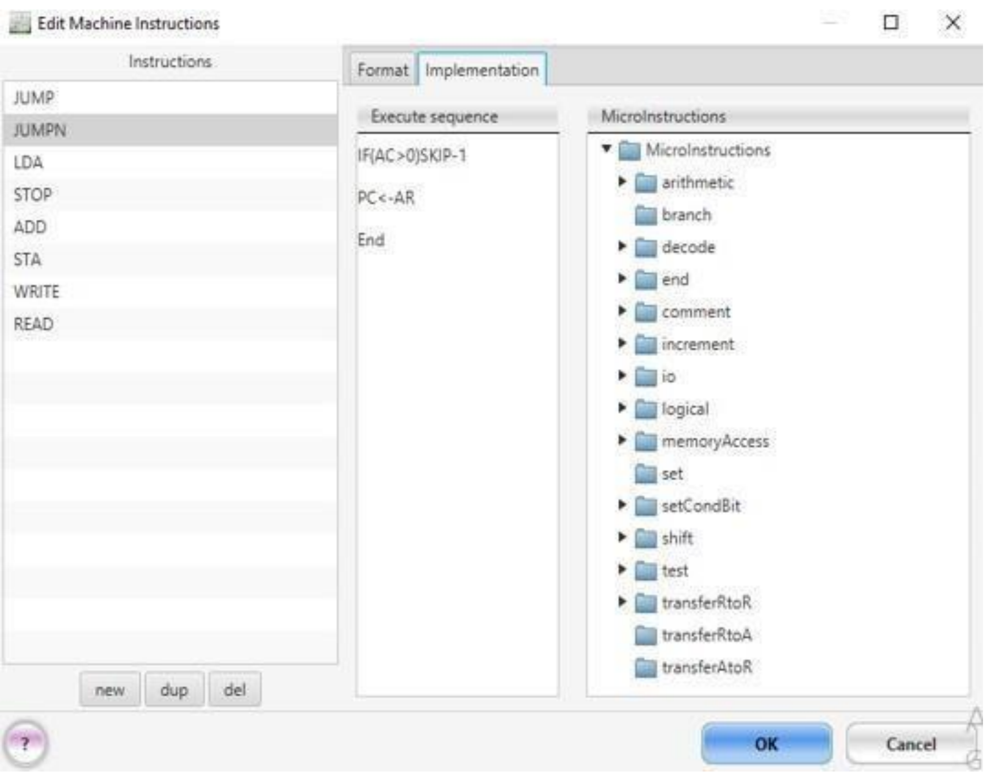
- Go to **Implementation** and select **Test** with the operation.

  IF(AC>0)SKIP - 1

- Go to **Implementation** and select **TransferRtoR** with the operation. **PC< - AR**



- Define a **Sequence Instruction** for execution.



Result

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 2
Enter Inputs, the first of which must be an Integer: 2
Enter Inputs, the first of which must be an Integer: 2
Enter Inputs, the first of which must be an Integer: -3
Output:  6
EXECUTION HALTED NORMALLY due to the setting of the bit(s): [HALT-BIT]
```