CSA Lab Assignment

Harsh Bamotra AC-1216

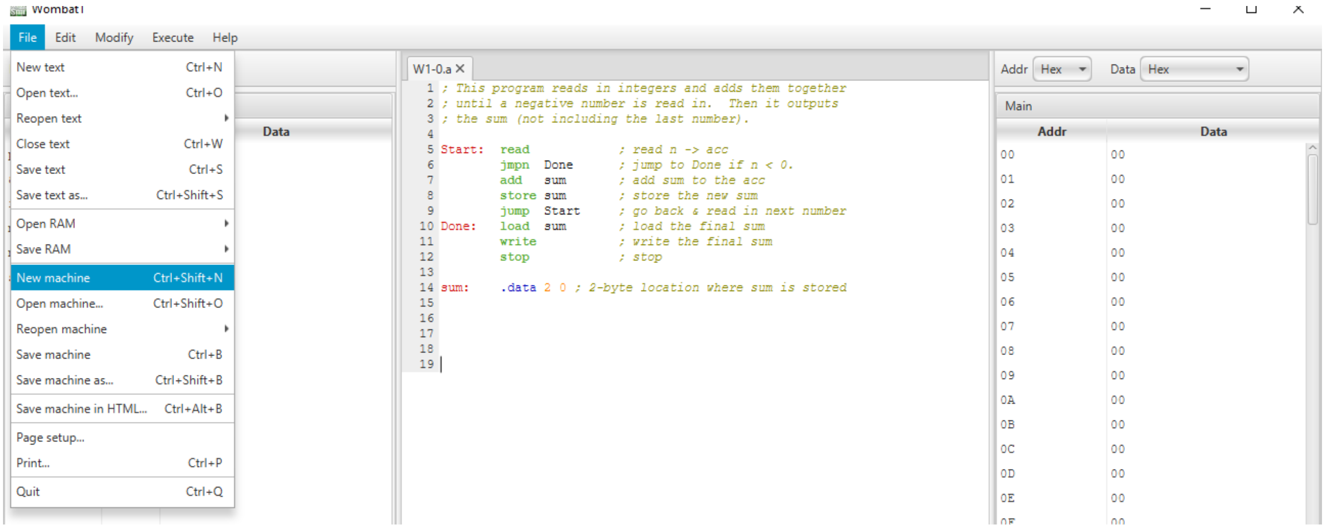
CPU Simulator Step-Wise Notes

Task → Multiplying two user input numbers

Step 1 : Creating a new machine and saving it.

Click on the file option you will find it in the top of the application and then click on “New Machine” .

Then after creating a new machine save it with “.cpu” extension by clicking on the save file from the file.

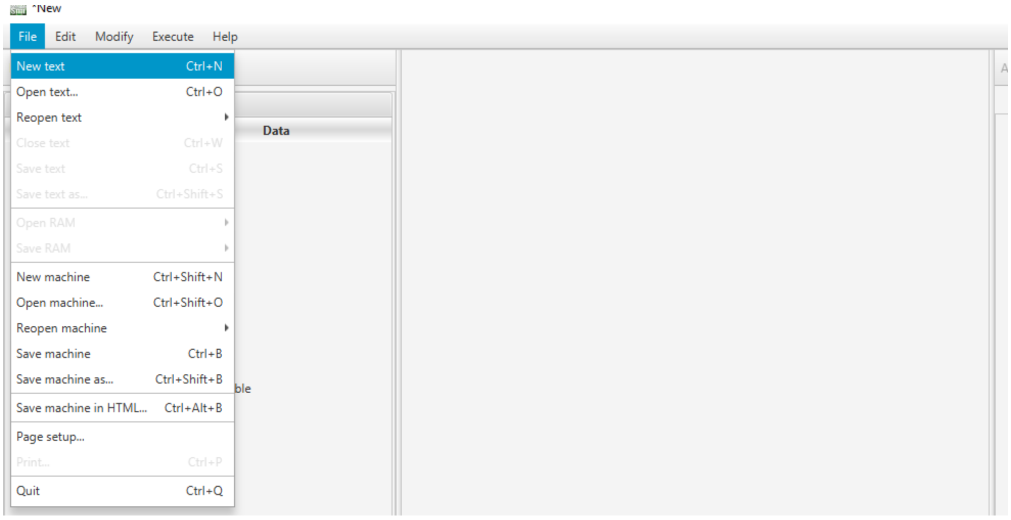


```
; This program reads in integers and adds them together
; until a negative number is read in.  Then it outputs
; the sum (not including the last number).

Start:  read            ; read n -> acc
        jmpn  Done      ; jump to Done if n < 0.
        add   sum       ; add sum to the acc
        store sum       ; store the new sum
        jump  Start     ; go back & read in next number
Done:   load  sum       ; load the final sum
        write           ; write the final sum
        stop            ; stop

sum:    .data 2 0 ; 2-byte location where sum is stored
```

Addr	Data
00	00
01	00
02	00
03	00
04	00
05	00
06	00
07	00
08	00
09	00
0A	00
0B	00
0C	00
0D	00
0E	00
0F	00

Step 2:Creating text file to write commands

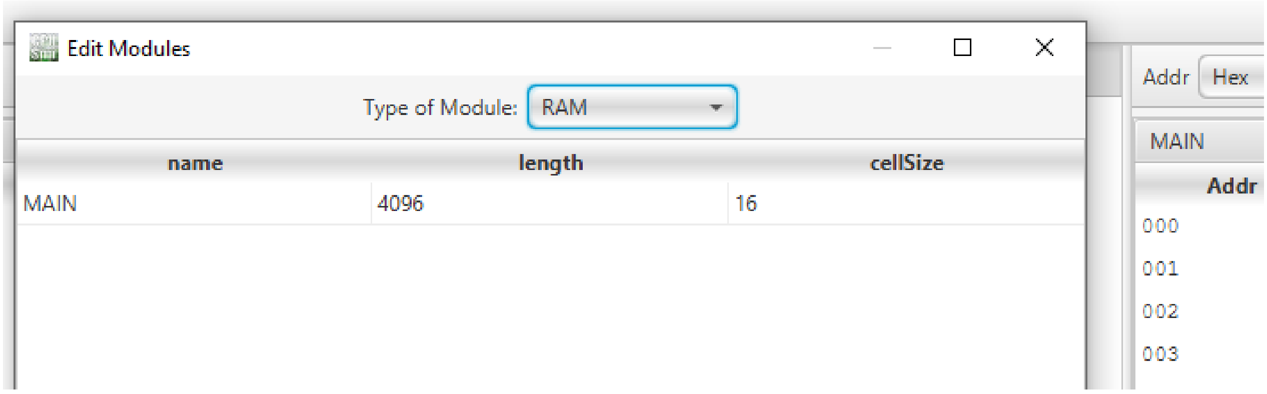
Click on the new text option from the file menu and create a new text and save it with “.a” extension .



Step 3: Creating Memory

To create memory go to **MODIFY → HARDWARE MODULES**

Now create MAIN memory in the RAM module.



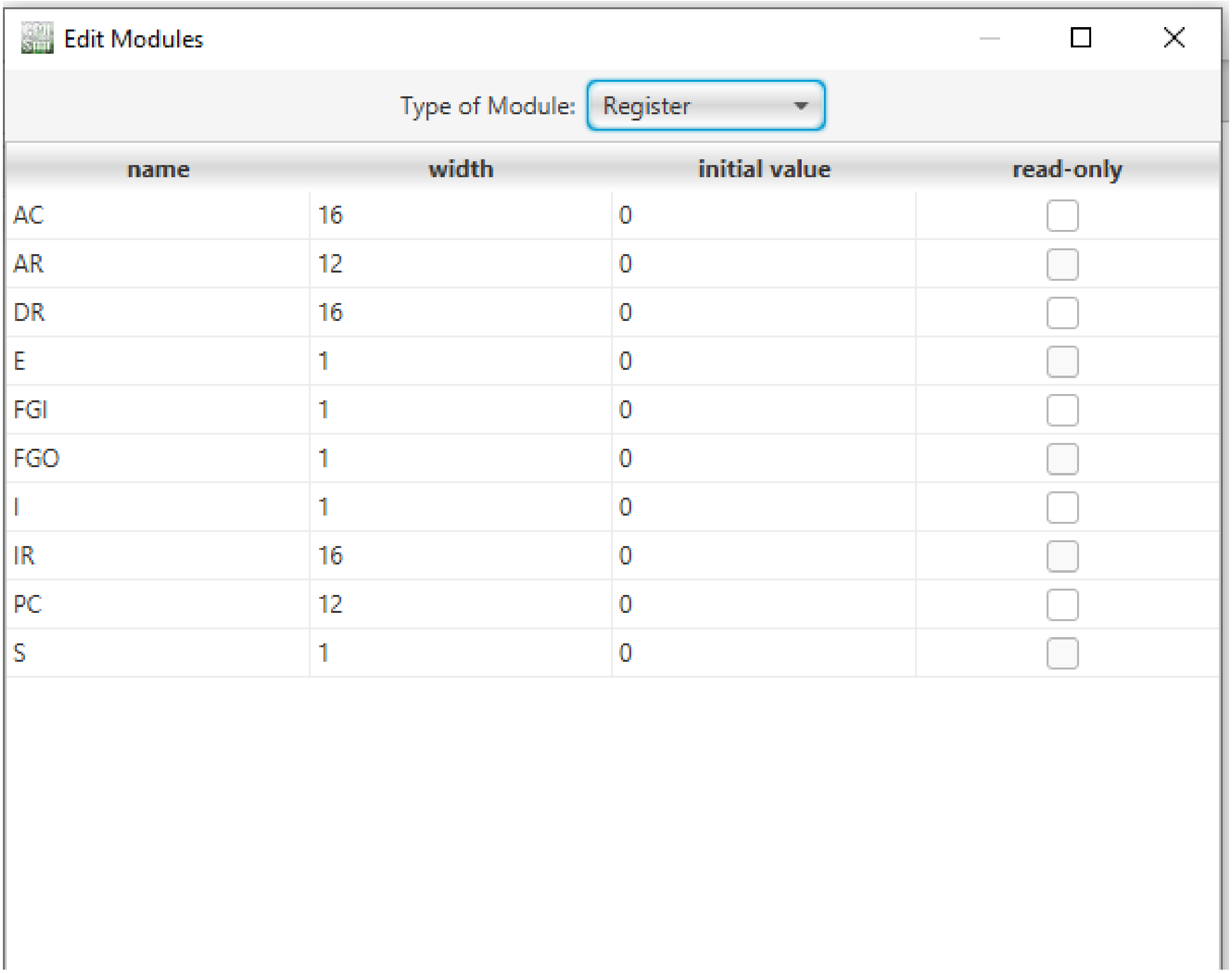
Type of Module: RAM

name	length	cellSize
MAIN	4096	16

Step 4:Creating Registers

To create registers go to **MODIFY → HARDWARE MODULES → REGISTER MODULE**

Now create the following Registers as shown in the picture below.



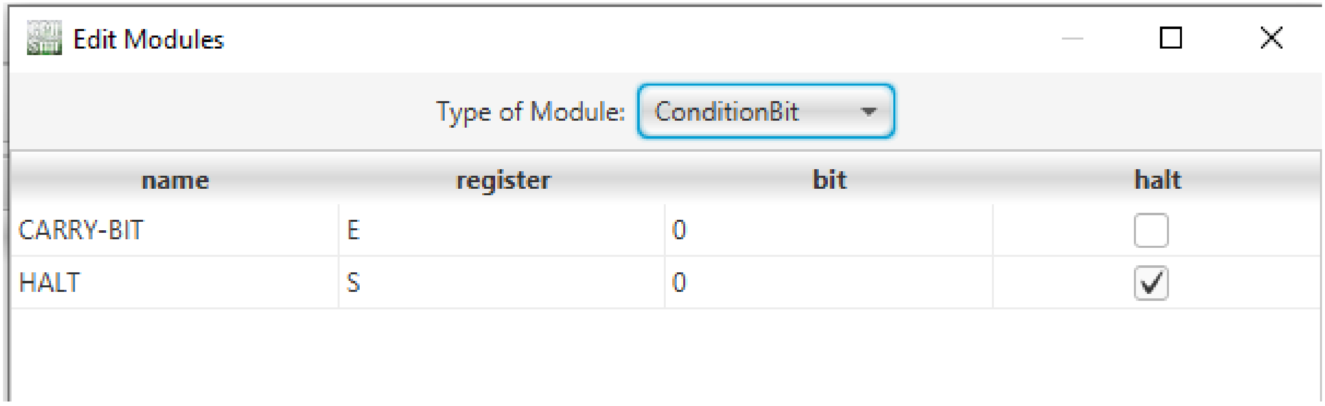
Type of Module: Register

name	width	initial value	read-only
AC	16	0	□
AR	12	0	□
DR	16	0	□
E	1	0	□
FGI	1	0	□
FGO	1	0	□
I	1	0	□
IR	16	0	□
PC	12	0	□
S	1	0	□

Step 5: Creating Condition Bit

To create condition bit go to **MODIFY→ HARDWARE MODULE**

Now you have create the following Condition Bit in the Condition Bit module as shown in the pictures.



Edit Modules

Type of Module: ConditionBit

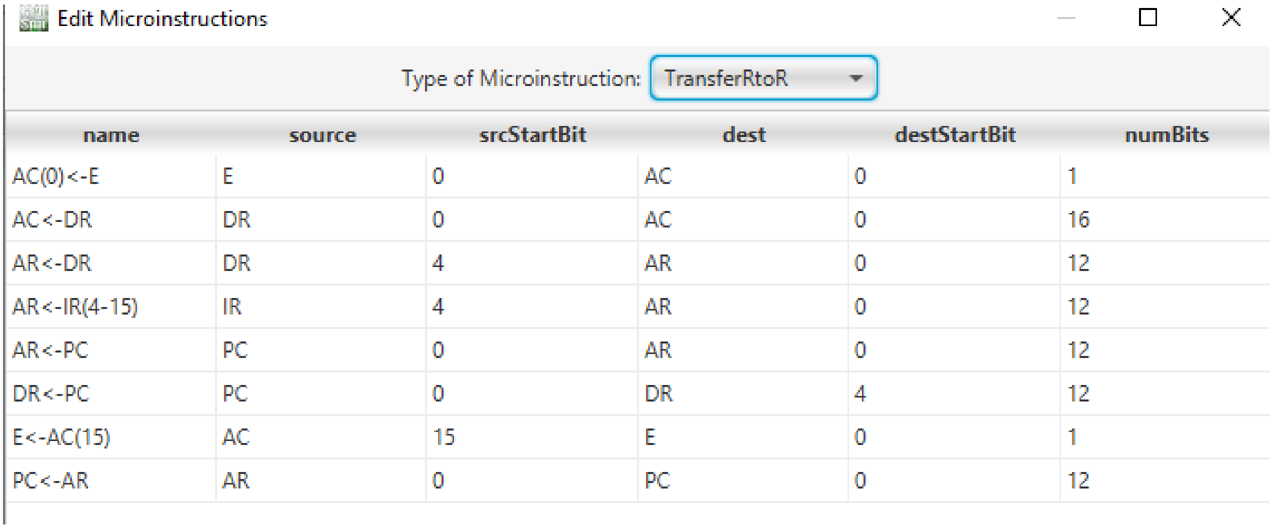
name	register	bit	halt
CARRY-BIT	E	0	☐
HALT	S	0	☑

Step 6: Creating Microinstructions

Now to create Microinstructions go to **MODIFY → MICROINSTRUCTIONS**

You have to create the following Microinstructions on your machine.

1.TranferRtoR

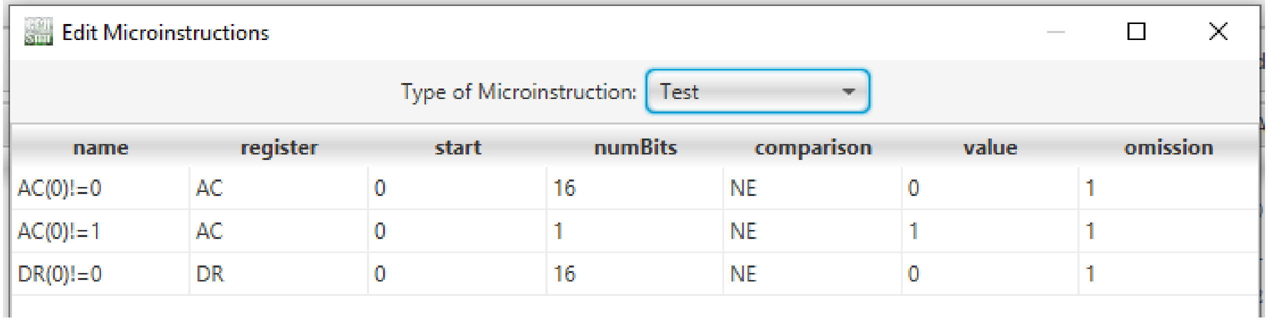


Edit Microinstructions

Type of Microinstruction: TransferRtoR

name	source	srcStartBit	dest	destStartBit	numBits
AC(0)<-E	E	0	AC	0	1
AC<-DR	DR	0	AC	0	16
AR<-DR	DR	4	AR	0	12
AR<-IR(4-15)	IR	4	AR	0	12
AR<-PC	PC	0	AR	0	12
DR<-PC	PC	0	DR	4	12
E<-AC(15)	AC	15	E	0	1
PC<-AR	AR	0	PC	0	12

2. Test

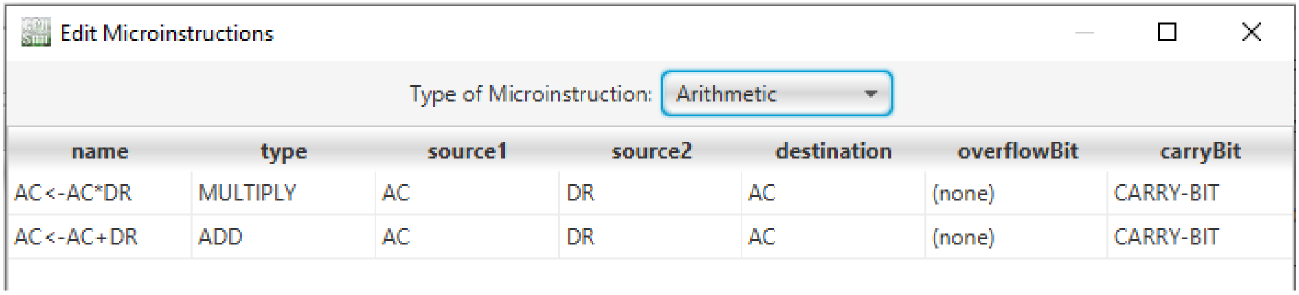


Edit Microinstructions

Type of Microinstruction: Test

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

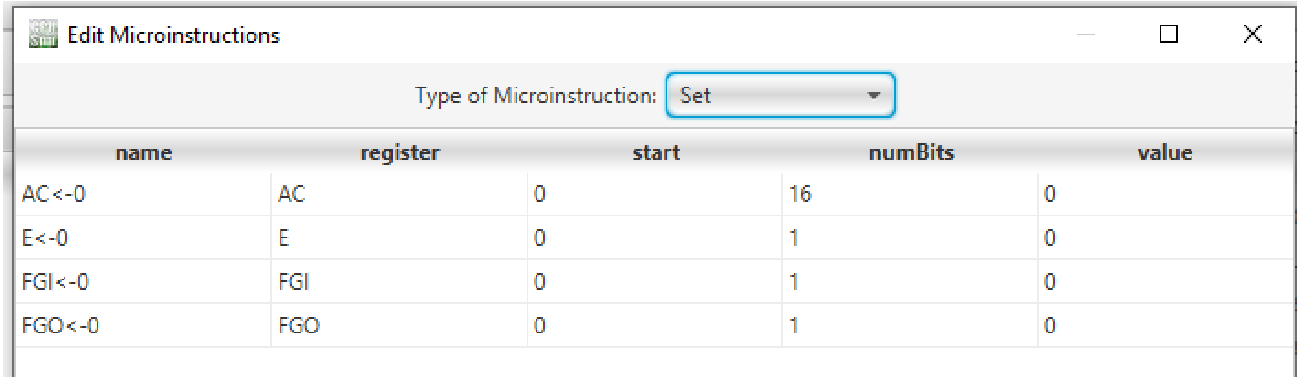
3.Arithmatic



Type of Microinstruction: Arithmetic

name	type	source1	source2	destination	overflowBit	carryBit
AC<-AC*DR	MULTIPLY	AC	DR	AC	(none)	CARRY-BIT
AC<-AC+DR	ADD	AC	DR	AC	(none)	CARRY-BIT

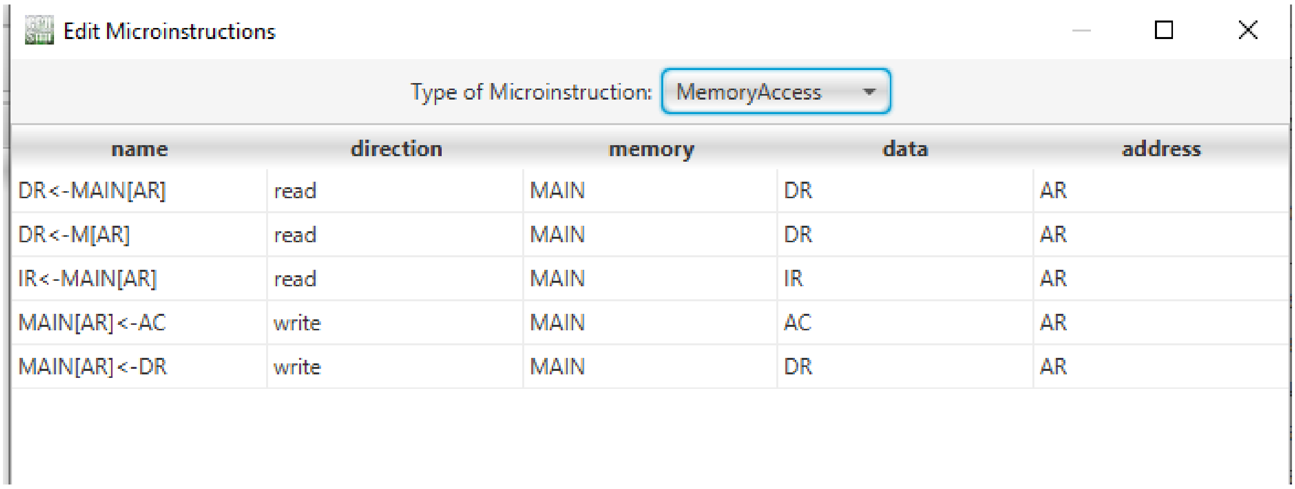
4.Set



Type of Microinstruction: Set

name	register	start	numBits	value
AC<-0	AC	0	16	0
E<-0	E	0	1	0
FGI<-0	FGI	0	1	0
FGO<-0	FGO	0	1	0

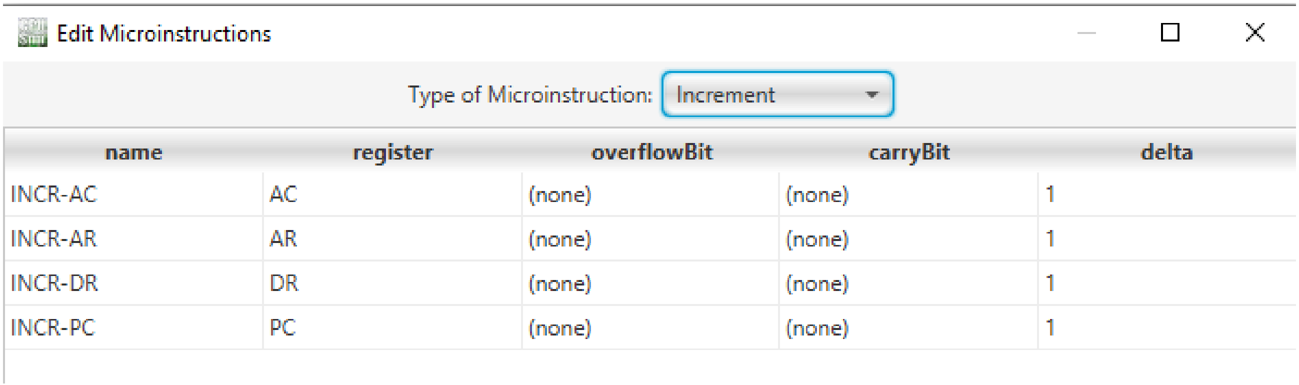
5.MemoryAccess



Type of Microinstruction: MemoryAccess

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

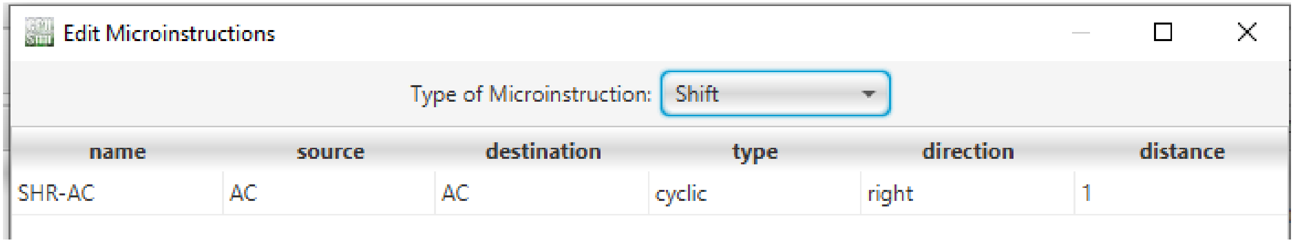
6. Increment



Type of Microinstruction: Increment

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

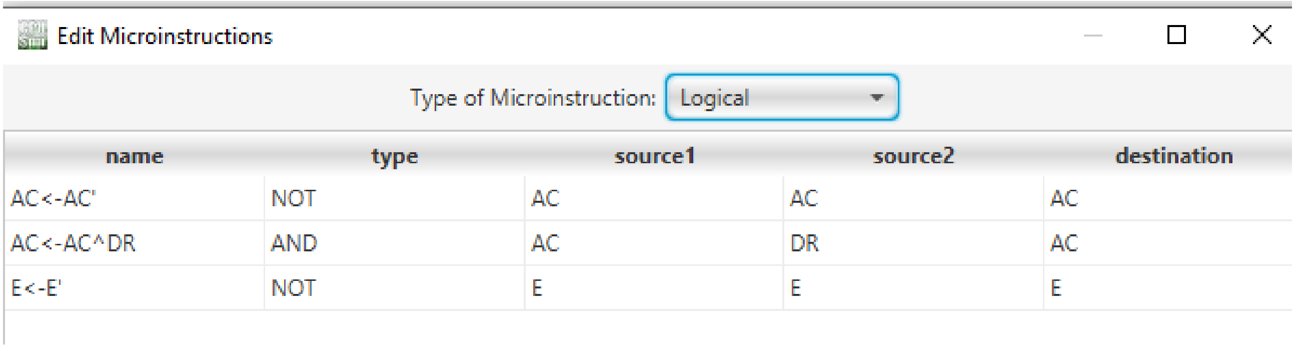
7. Shift



Type of Microinstruction: Shift

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

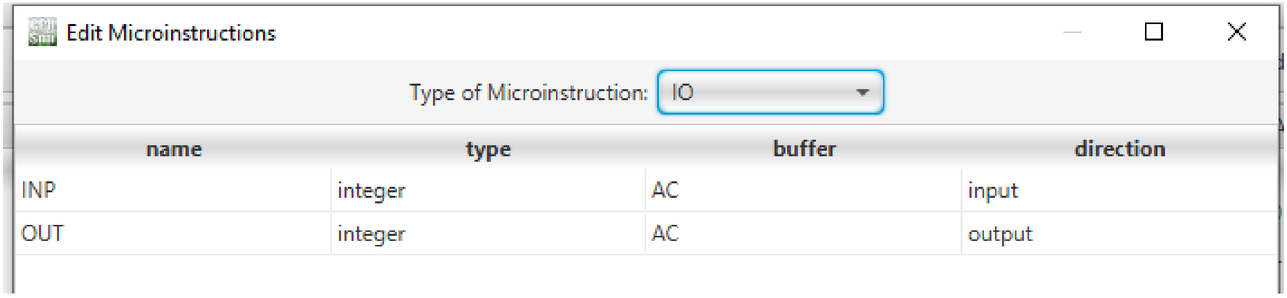
8.Logical



Type of Microinstruction: Logical

name	type	source1	source2	destination
AC<-AC'	NOT	AC	AC	AC
AC<-AC^DR	AND	AC	DR	AC
E<-E'	NOT	E	E	E

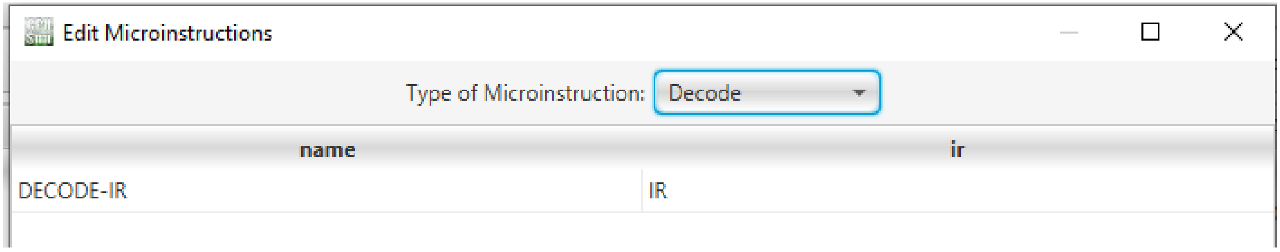
9. IO



Type of Microinstruction: IO

name	type	buffer	direction
INP	integer	AC	input
OUT	integer	AC	output

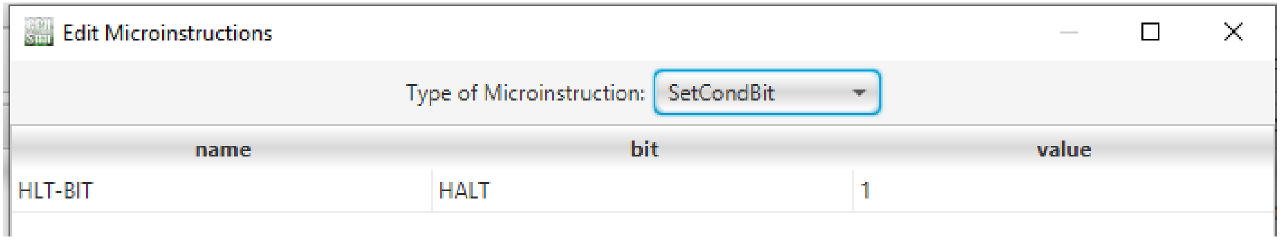
10. Decode



Type of Microinstruction: Decode

name	ir
DECODE-IR	IR

11.SetCondBit



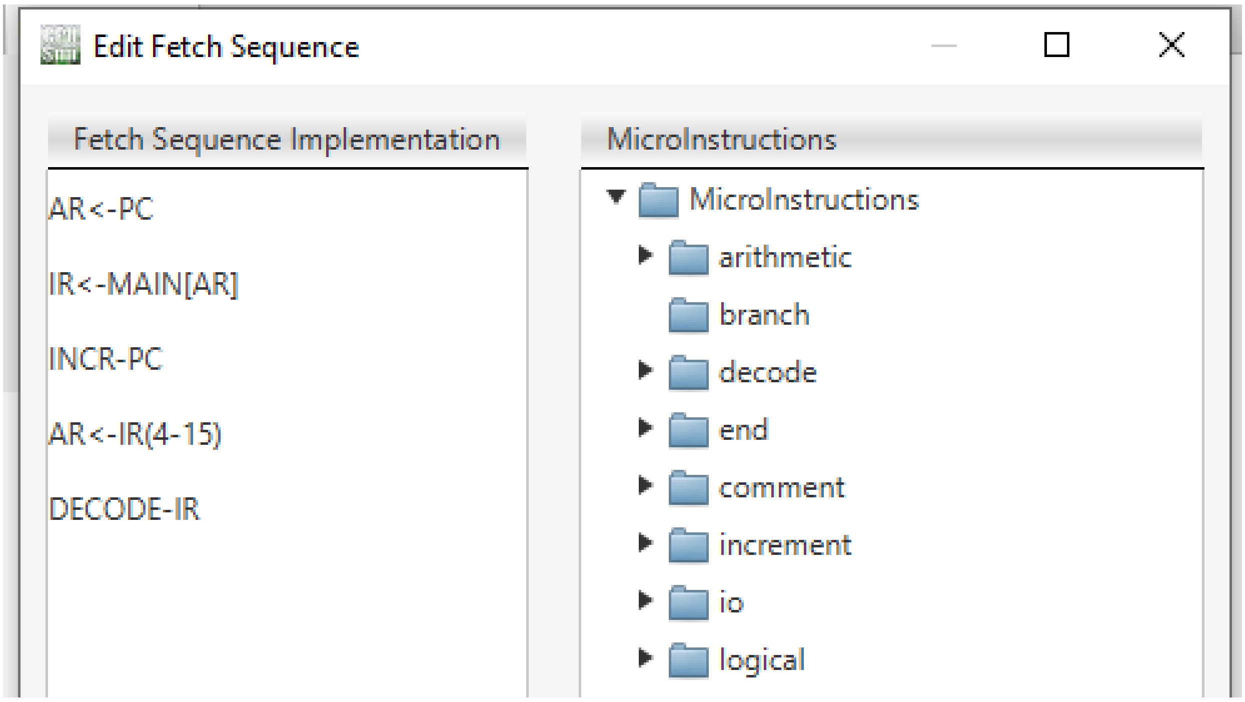
Type of Microinstruction: SetCondBit

name	bit	value
HLT-BIT	HALT	1

Step 7: Implementing Fetch Sequence

In order to implement the fetch sequence for our machine go to **MODIFY→FETCH SEQUENCE.**

Now create the following fetch sequence as shown in the figure.



AR<-PC

IR<-MAIN[AR]

INCR-PC

AR<-IR(4-15)

DECODE-IR

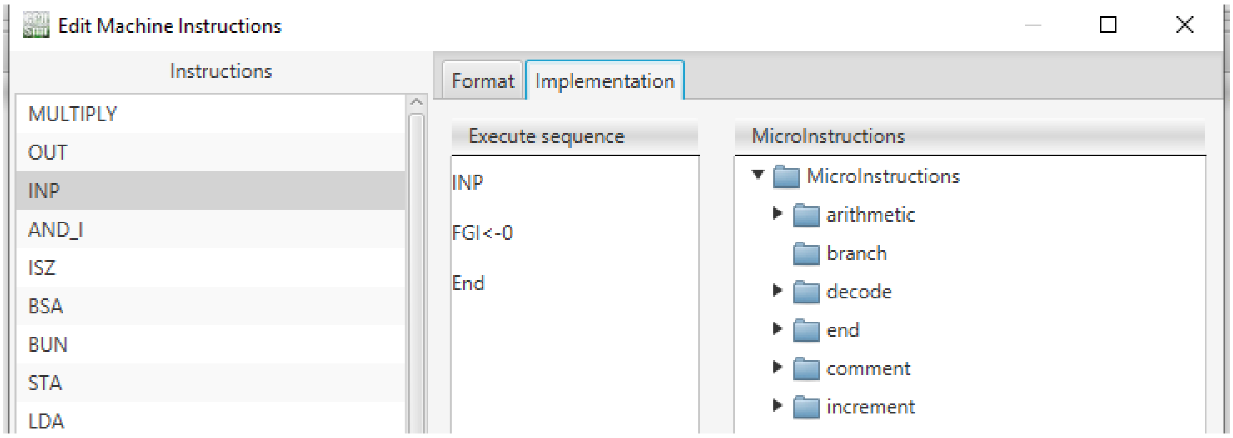
Step 8: Creating Instructions

In order to create set instruction just go to **MODIFY→ MACHINE INSTUCTIONS** .

Now create the following instructions

1. **INP**
   Opcode-0xF800

   Field-REGISTER

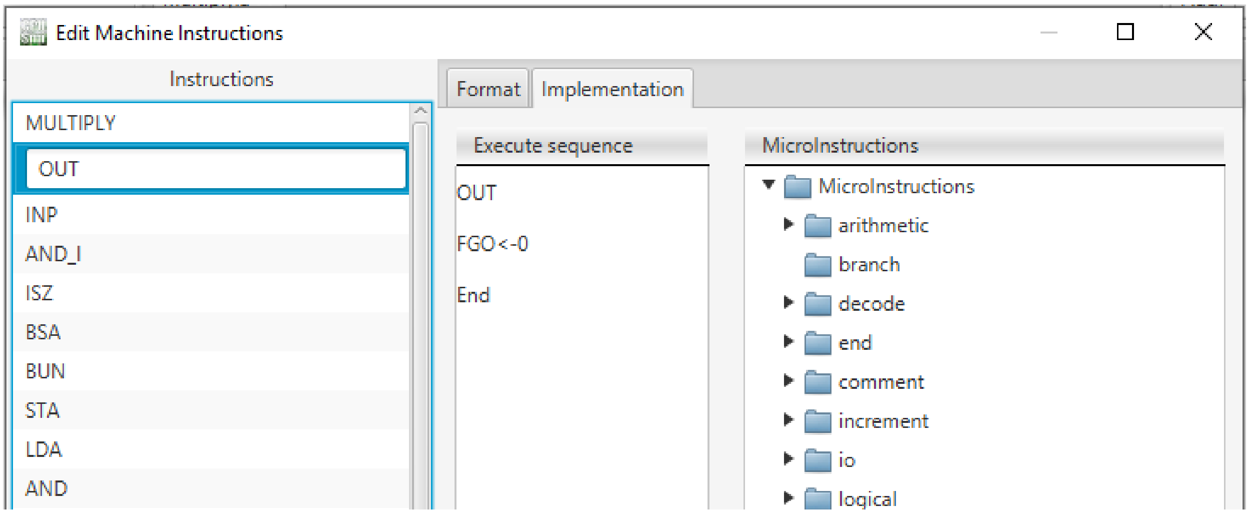


INP

FGI<-0

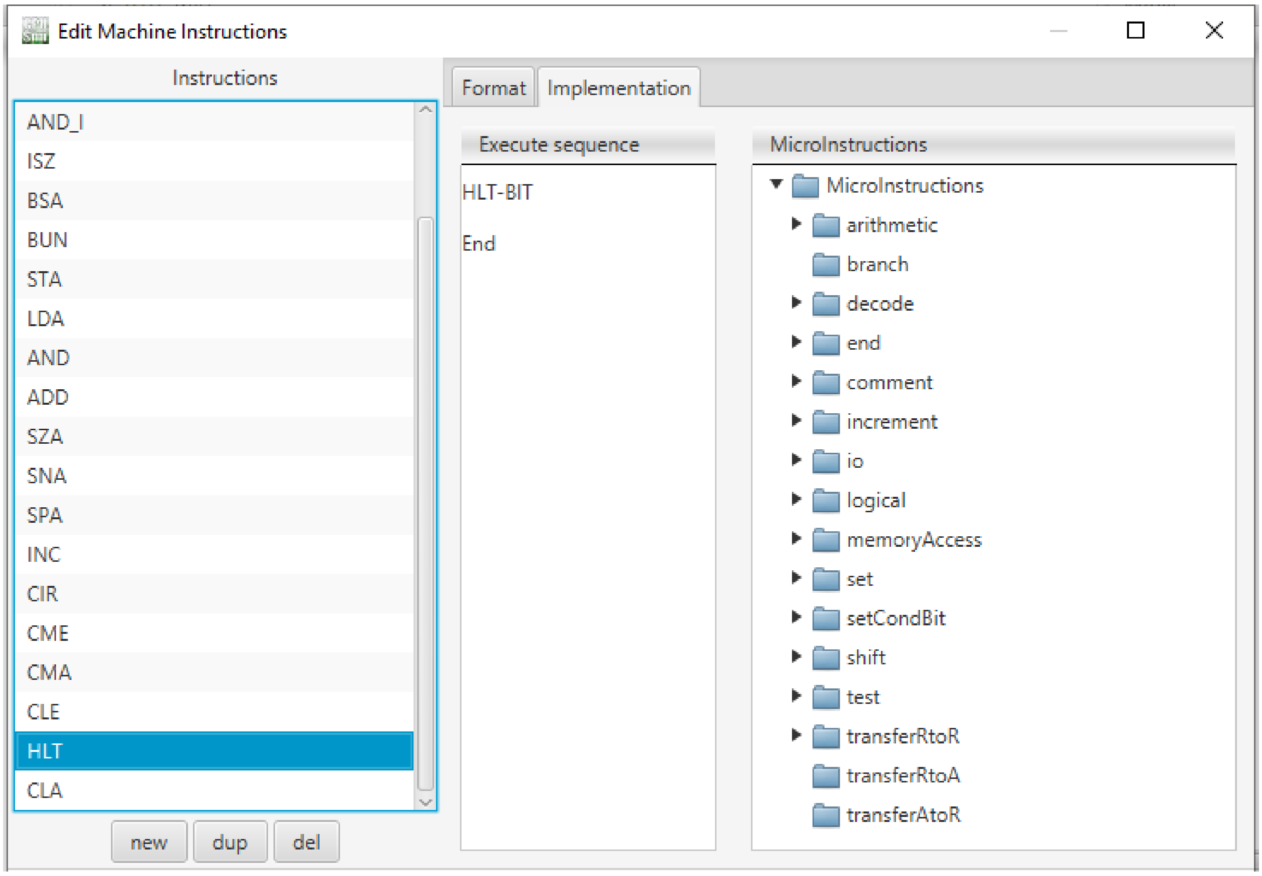
End

2. OUT
Opcode-0xF400
Field-REGISTER



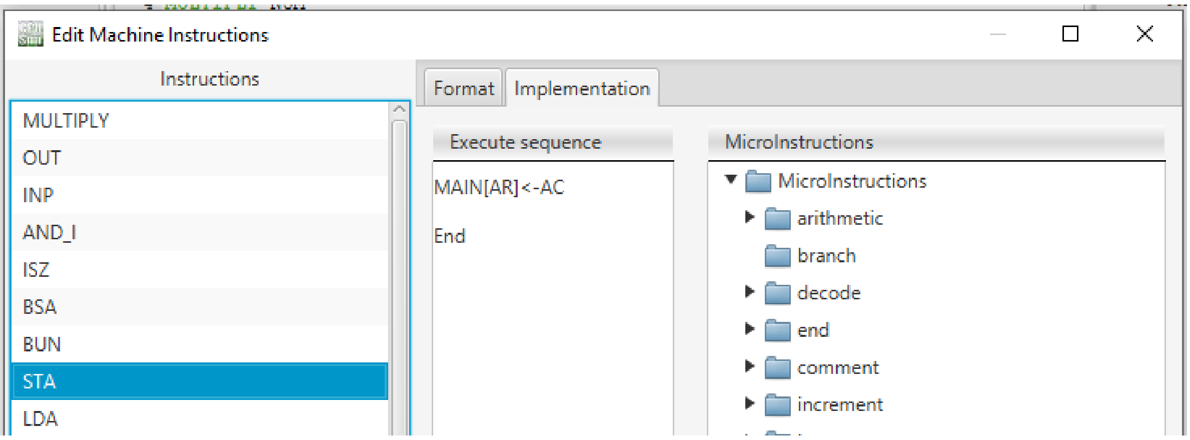
```
OUT
FGO<-0
End
```

3. HLT
Opcode-0xE001
Field-REGISTER



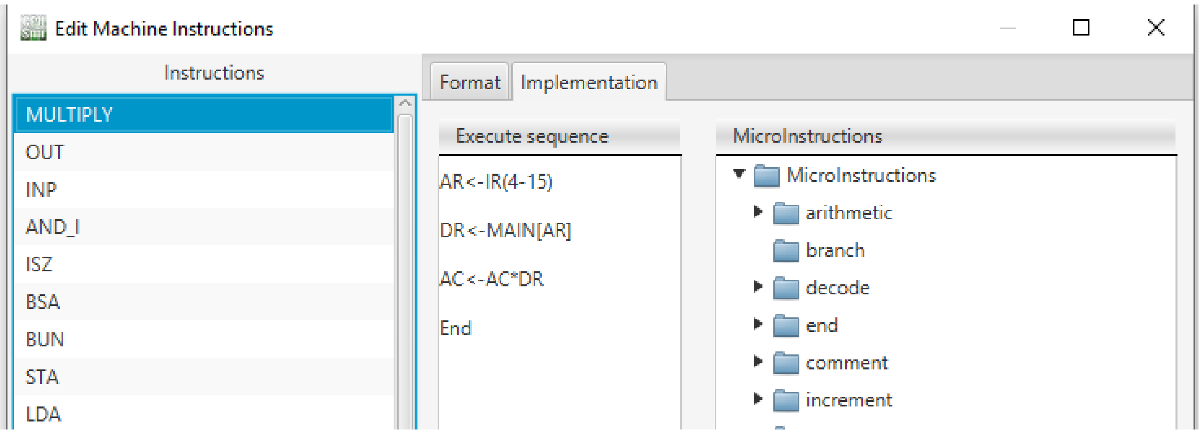
```
HLT-BIT
End
```

4. **STA**
   Opcode-0x6
   Field-OP , ADDR



```
MAIN[AR]<-AC
End
```

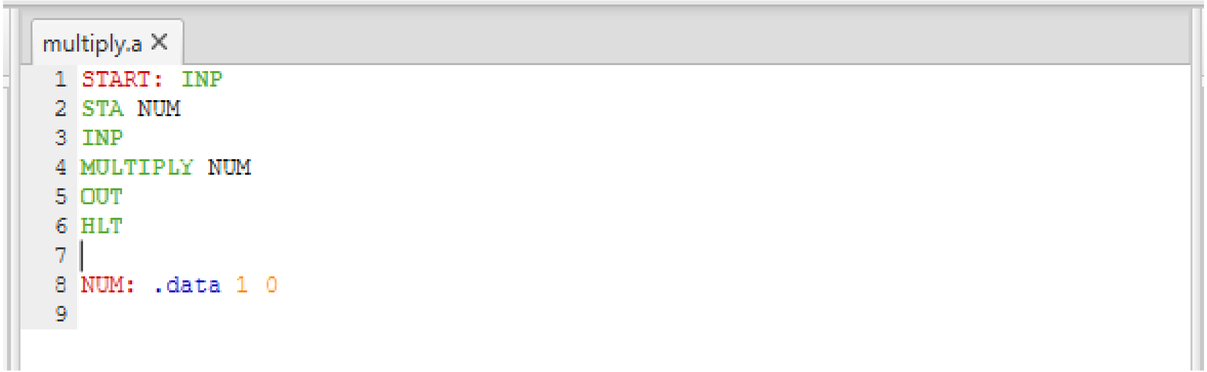
5. **MULTIPLY**
   Opcode-0x7
   Field-OP , ADDR



```
AR<-IR(4-15)
DR<-MAIN[AR]
AC<-AC*DR
End
```

Step 9: Writing Commands

Now just type the following commands in the text box we saved earlier as multipy.a as shown in the picture given below.

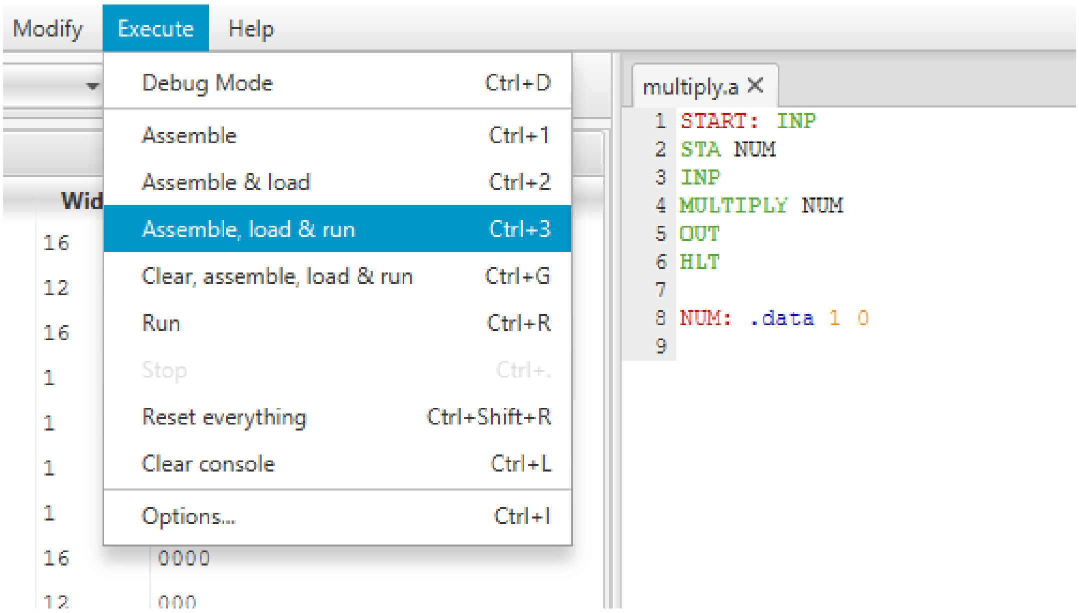


```
START:  INP
STA NUM
INP
MULTIPLY NUM
OUT
HLT

NUM: .data 1 0

```

Step 10: Running the commands to perform the task

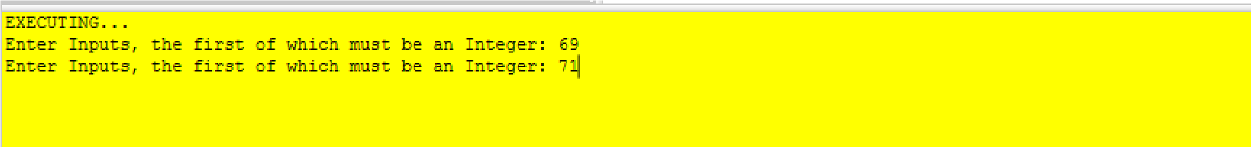
In order to run the commands go to **EXECUTE** and click on **RESET EVERYTHING.**

Now just select the **ASSEMBLE , LOAD , RUN** from the **EXECUTE** menu.



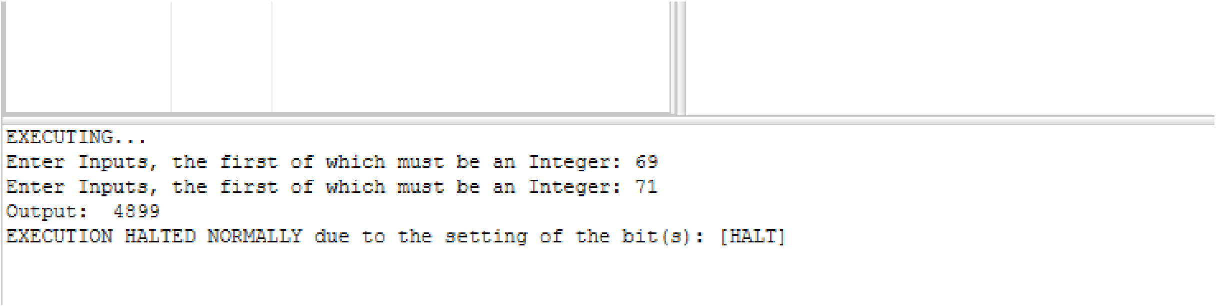
```
START: INP
STA NUM
INP
MULTIPLY NUM
OUT
HLT

NUM: .data 1 0
```

Now as soon you will click it you will see that the lower part of the window becomes yellow and now you just have to input the integers which you want to multiply .



```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 69
Enter Inputs, the first of which must be an Integer: 71
```

Now as soon you will give the input and press enter you will see that the result is printed as shown in the figure below.



```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 69
Enter Inputs, the first of which must be an Integer: 71
Output:  4899
EXECUTION HALTED NORMALLY due to the setting of the bit(s): [HALT]
```