A REPORT

ON

Computer Architecture Assignment

CPUSim

BY

Name	ID Number	Email-ID
Mudit Chaturvedi	2018A7PS0248H	f20180248@hyderabad.bits-pilani.ac.in
Hardik Parnami	2018A7PS0062H	f20180062@hyderabad.bits-pilani.ac.in
Kriti Jethlia	2018A7PS0223H	f20180223@hyderabad.bits-pilani.ac.in
Sristi Sharma	2018A7PS0299H	f20180299@hyderabad.bits-pilani.ac.in
Mayank Negi	2018A7PS0210H	f20180210@hyderabad.bits-pilani.ac.in

Group No. 21

Submitted to: Dr. Suvadip Batabyal

Under guidance of: Dr. Nikumani Choudhary, Dr. Rajib Ranjan Maiti

Birla Institute Of Technology And Science, Pilani(Hyderabad Campus)

November,2020

PART A

1. Hardware Modules:

Type of Module: RAM

name	length	cellSize
RAM	256	16

Type of Module: Register

name	width	initial value	read-only
ACC	16	0	☐
IR	16	0	☐
MAR	8	0	☐
MDR	16	0	☐
PC	8	0	☐
STATUS	8	0	☐

Type of Module: ConditionBit

name	register	bit	halt
CARRY-BIT	STATUS	1	☐
HALT-BIT	STATUS	0	☑
OVERFLOW	STATUS	2	☐

Explanation: The Main memory/RAM was specified to be 16 bits for each cell with a capacity for 256 words. Since **MDR** must contain content of Main memory, it was also set to 16 bits wide. Then, since the accumulator/**ACC** has a bi-directional connection with **MDR**, it was also set to 16 bits. The Main memory contains instructions which are transferred to the **IR** register, so the **IR** register must also be 16 bits wide. The **MAR** register holds the memory locations for Main memory, so the range of values it must be able to hold is equal to the number of cells in Main memory. The Main memory was specified to have 256 cells which is equal to 2 raised to the 8th power. Therefore, the **MAR** register needs to be at least 8 bits wide. Since the program counter, **PC** holds the location of the next instruction, it has been set to 8 bits as well.

Therefore, addressing registers of 8-bit width while data registers of 16-bit width.

A **STATUS** register is also created with 8-bit width.

2. **MicroInstructions**:

a. Register to Register Transfer:

Edit Microinstructions

Type of Microinstruction: TransferRtoR

name	source	srcStartBit	dest	destStartBit	numBits
ACC->MDR	ACC	0	MDR	0	16
IR(8-15)->MAR	IR	8	MAR	0	8
IR(8-15)->PC	IR	8	PC	0	8
MDR(8-15)->...	MDR	8	MAR	0	8
MDR->ACC	MDR	0	ACC	0	16
MDR->IR	MDR	0	IR	0	16
PC->MAR	PC	0	MAR	0	8

Explanation: The micro-instructions have been implemented as described in the assignment. For the cases where the destination register is smaller than the source register, the bits containing the address (this scenario occurs only in cases of moving the address), which is always the 8 most significant bits, are transferred.

Note: This also contains the **MDR** to **MAR** transfer.

b. Arithmetic Operations:

Type of Microinstruction: Arithmetic

name	type	source1	source2	destination	overflowBit	carryBit
ACC+MDR-...	ADD	ACC	MDR	ACC	HALT-BIT	(none)
ACC-MDR-...	SUBTRACT	ACC	MDR	ACC	HALT-BIT	(none)

add and **sub** operations are defined by adding and subtracting the values in **MDR** register to and from those in the accumulator (**ACC**).

c. Test:

Edit Microinstructions

Type of Microinstruction: Test

name	register	start	numBits	comparison	value	omission
IF (ACC != 0...	ACC	0	16	NE	0	1
IF (ACC < 0)...	ACC	0	16	LT	0	2
IF (ACC == ...	ACC	0	16	EQ	0	1
IF (ACC > 0)...	ACC	0	16	GT	0	2

Explanation: In these (Test) micro-instructions, the relation being evaluated is given in the comparison column (NE for !=, GT for >

and so on). CPU Simulator skips the number of lines given in the omission column when the operation specified in the comparison column evaluates to true. We want that if the relation specified in the name column evaluated to true, the line(s) following it is skipped.

The omission values depend on the actual use of the microinstructions when defining the machine instructions (in our case we've used omission value 2 for > and < test microinstructions and 1 others).

d. Main Memory Access:

Type of Microinstruction: MemoryAccess

name	direction	memory	data	address
MDR->RAM[MAR]	write	RAM	MDR	MAR
RAM[MAR]->MDR	read	RAM	MDR	MAR

Explanation: Main Memory/**RAM** is being indexed by **MAR** and values are being read from it and written to it by **MDR**

e. Increment PC:

Type of Microinstruction: Increment

name	register	overflowBit	carryBit	delta
INC-PC	PC	HALT-BIT	(none)	1

f. IO:

Type of Microinstruction: IO

name	type	buffer	direction
ACC->OUTPUT	integer	ACC	output
INPUT->ACC	integer	ACC	input

Explanation: Input is being written to accumulator and output for printing is also taken from accumulator.

g. Decode Instruction:

Type of Microinstruction: Decode

name	ir
DECODE-IR	IR

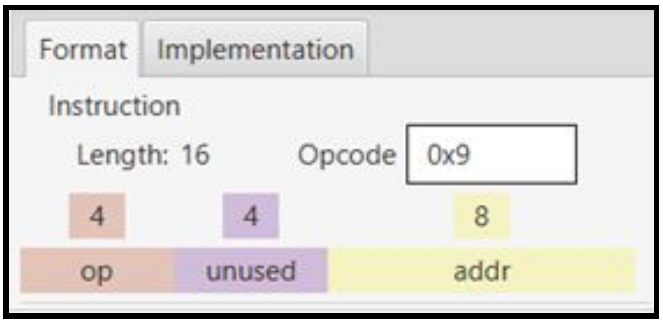
h. Set Condition Bit:

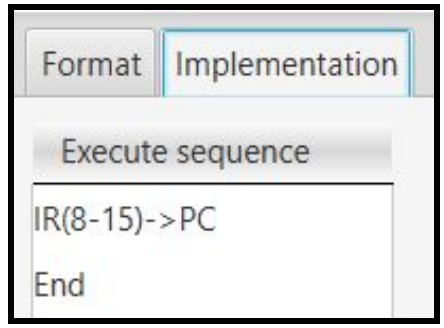
Type of Microinstruction: SetCondBit

name	bit	value
SET-HALT-BIT	HALT-BIT	1

3. **<u>Assembly Instructions</u>**:

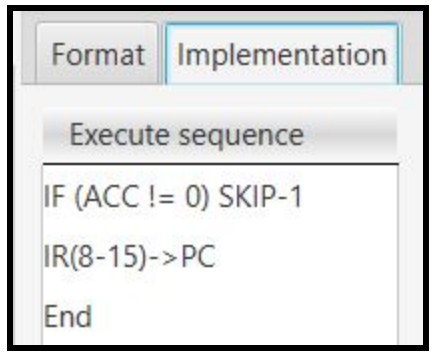
a. Unconditional Jump: **jmp**



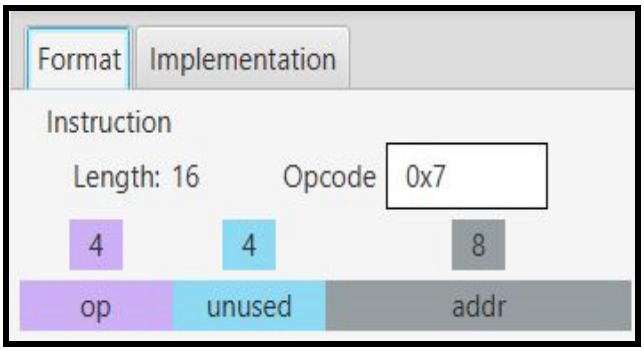


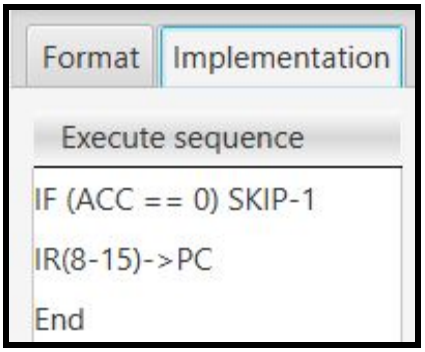
b. Conditional Jump:

- **jmpz**

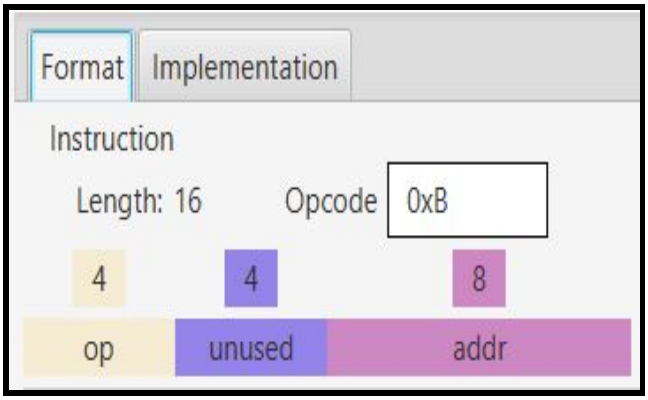


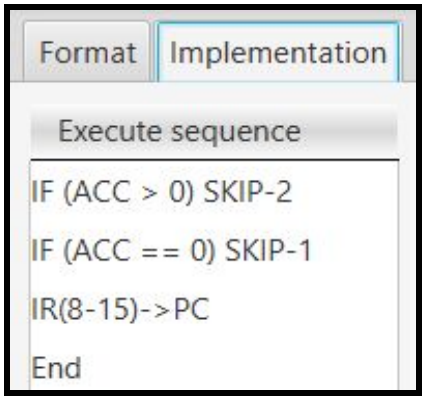
- **jmpnz**



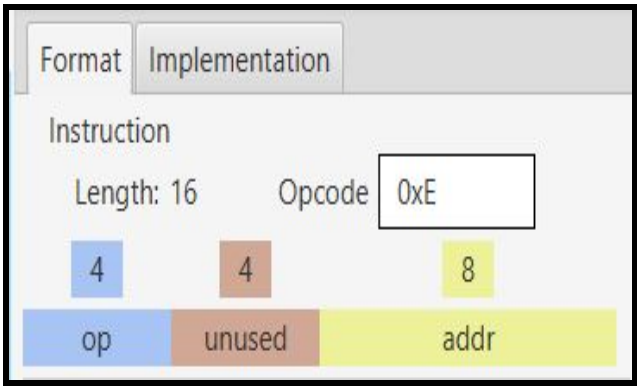


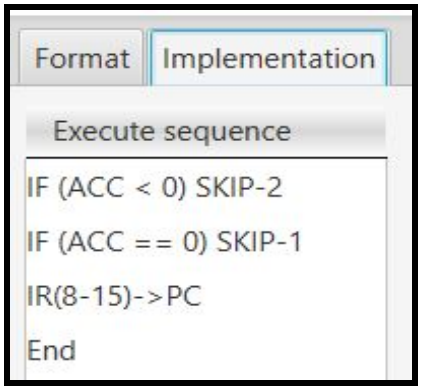
- **jmpn**



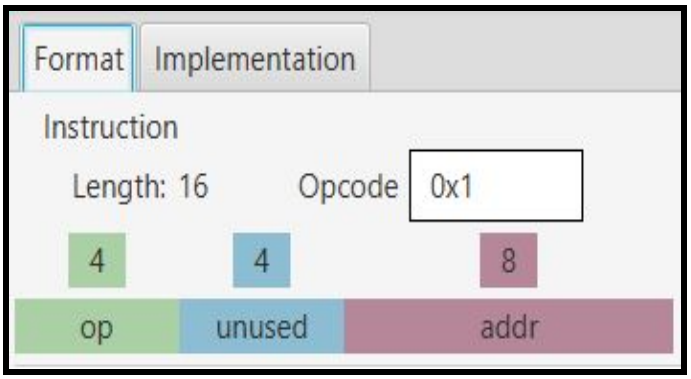


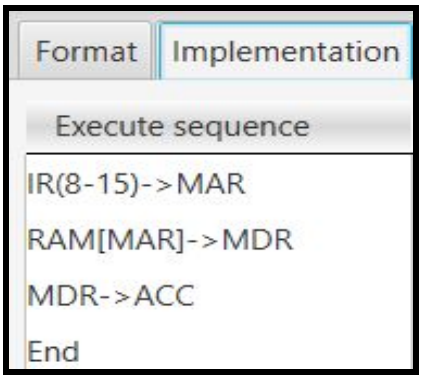
- **jmpp**



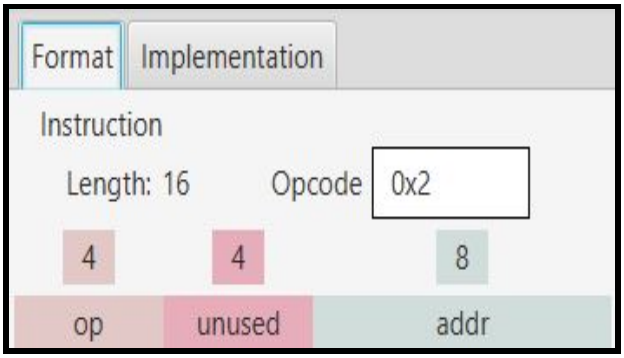


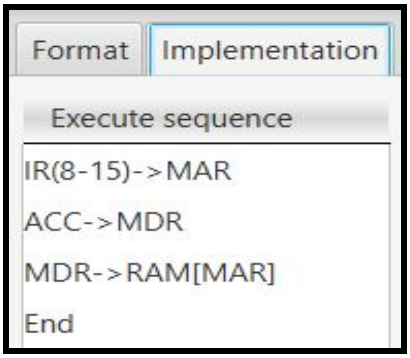
c. Load: **lda**



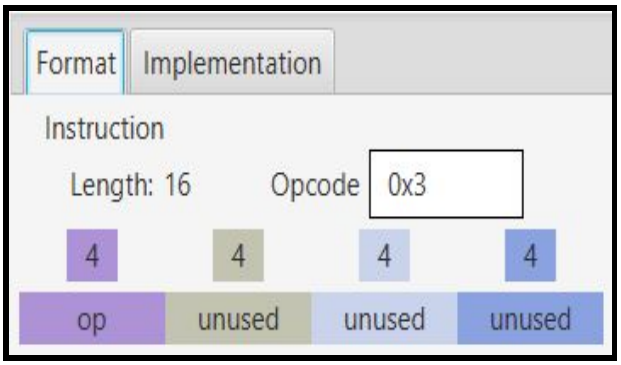


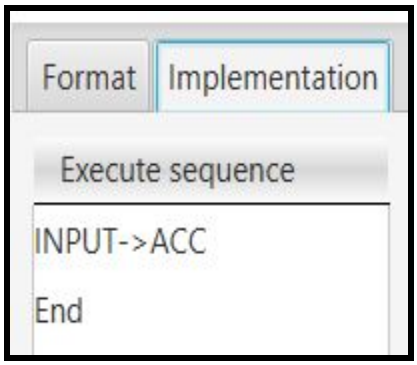
d. Store: **sta**



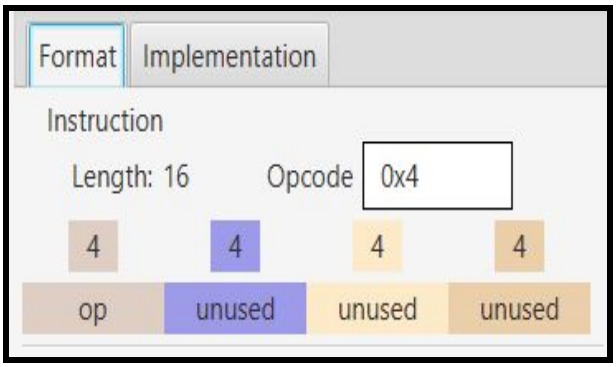


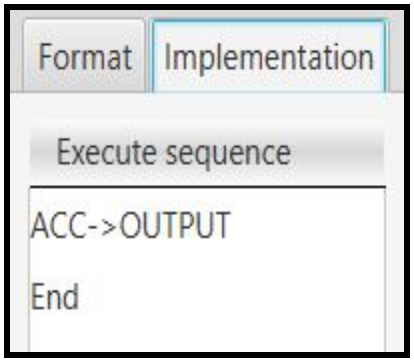
e. Input: **ipa**



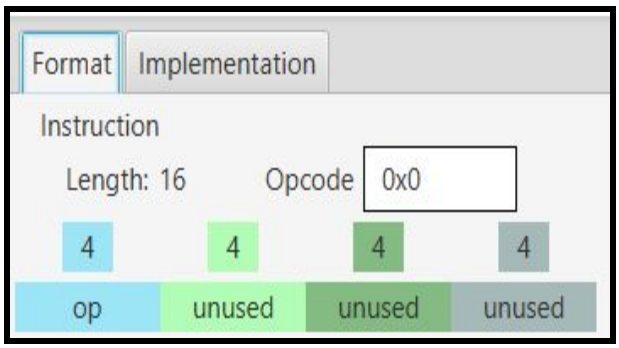


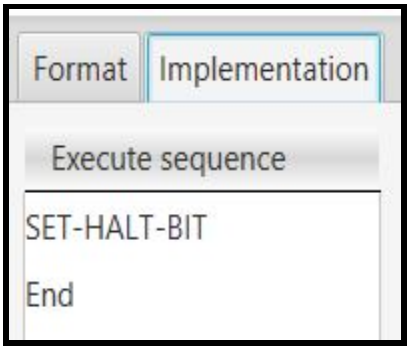
f. Output: **opa**



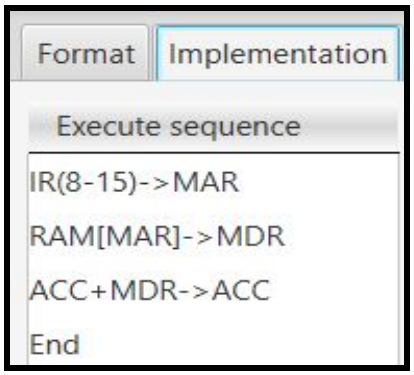


g. Stop: **stop**

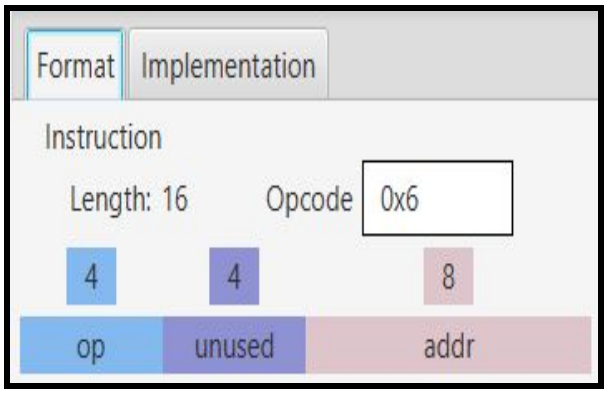


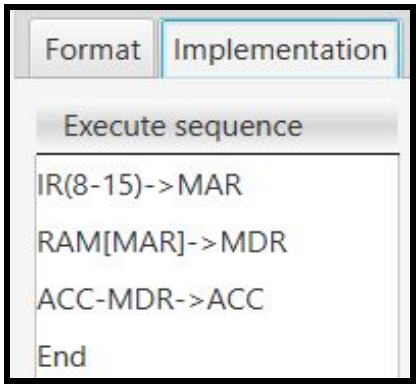


h. Add: **add**

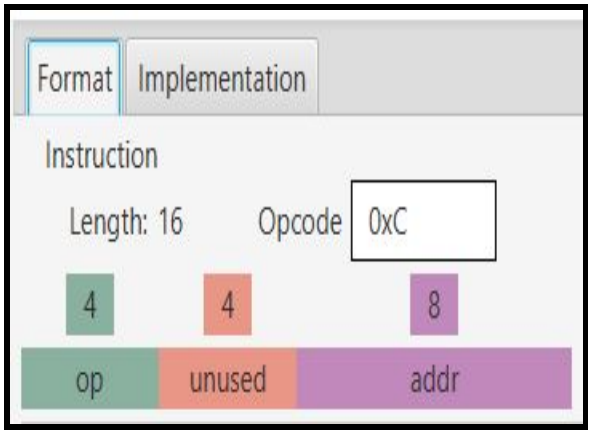


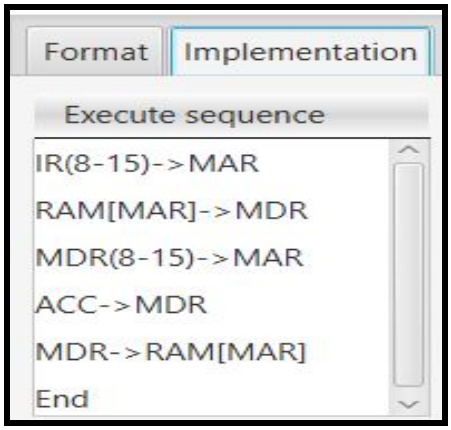
i. Subtract: **sub**





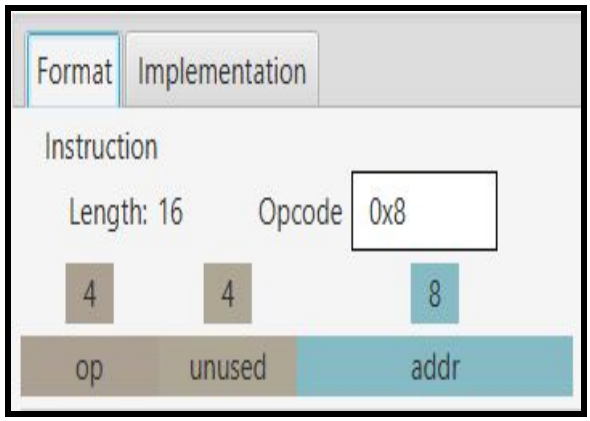
j. Memory to Memory Transfer: **m2m**

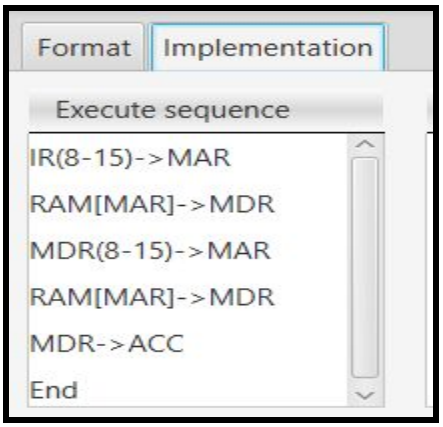




Explanation: First the address of **dataloc** is being passed to **MAR** using this we retrieve the address part of the value at **RAM[MAR]** into **MAR**. After this we write the data from **ACC/MDR** to the memory at **RAM[MAR].**

k. Transfer of content of a memory address stored in a location provided in the instruction to accumulator : **m2a**





Explanation: All the steps are same as **m2m** instruction except that final write is from **RAM[MAR]** to **ACC**.

4. **Fetch Sequence**:

Fetch Sequence Implementation
PC->MAR
RAM[MAR]->MDR
MDR->IR
INC-PC
DECODE-IR

5. **Code**:

partA.a ×

```
;Program to print positive integers in LIFO order
input_till_0:
        ;take input till 0 is entered
        ipa
        ;if negative don't save in stack and again take input
        jmpn input_till_0
        ;if 0 exit the loop
        jmpz if_input_0

        ;storing value in stack
        m2m dataloc

        ;adding one to stack pointer
        lda dataloc
        add one
        sta dataloc

        ;again take input
        jmp input_till_0

if_input_0:
        ;reducing the extra incremented stack pointer to point to the topmost element of stack
        lda dataloc
        sub one
        sta dataloc

output_in_LIFO:
        ;loading value pointed by the stack pointer to accumlator
        m2a dataloc
        ;printing output
        opa

        ;decrementing stack pointer
        lda dataloc
        sub one
        sta dataloc

        ;exit condition to check if we have reached the end of stack
        lda dataloc
        sub offset_dataloc
        jmpn done

        ;if not again printing output
        jmp output_in_LIFO
done:
        stop

dataloc:  .data 1 70 ;dataloc is the stack pointer pointing to the top of the stack
offset_dataloc:  .data 1 70 ; initial value of stack pointer
one:    .data 1 1 ; stores value 1
```

Explanation:

dataloc: stores the location where the positive numbers entered are to be stored, starting from 70 and increasing by 1 each time a number is written to the memory and decreasing by 1 each time a number is retrieved from memory for LIFO printing.

one: used to increment and decrement the values of variables by 1.

offset_dataloc: used to store initial location of **dataloc** and helpful in checking if all elements have been printed or not.

input_till_0 loop: takes input from user and if number entered is 0, exits to **output_in_LIFO.** If the number entered is positive, it stores the number in the location pointed by **dataloc** using **m2m** and increments **dataloc** by **one**.

if_input_0: Reduces **dataloc** by **one** and continues to **output_in_LIFO**.

output_in_LIFO: print all the positive numbers stored in **dataloc** and then decrements **dataloc** by **one**. If **dataloc** is equal to **offset_dataloc** i.e. all values have been printed then it exits to **done.**

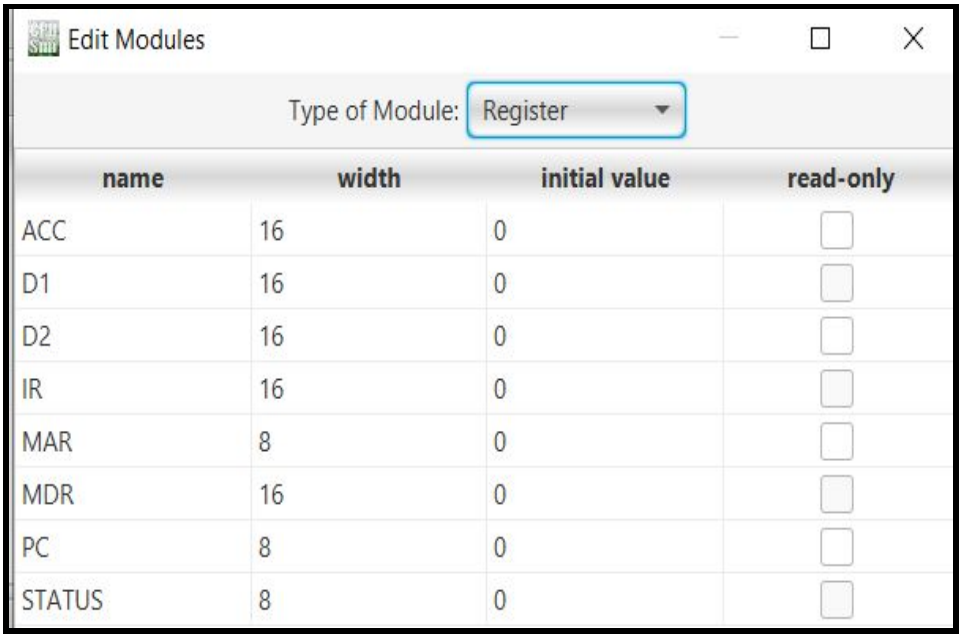
done: ends the program execution.

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 6 5 4 3 2 1 -5 -4 1 0
Output:  1
Output:  1
Output:  2
Output:  3
Output:  4
Output:  5
Output:  6
Output:  1
EXECUTION HALTED NORMALLY due to the setting of the bit(s): [HALT-BIT]
```

PART B

1. Hardware Modules:

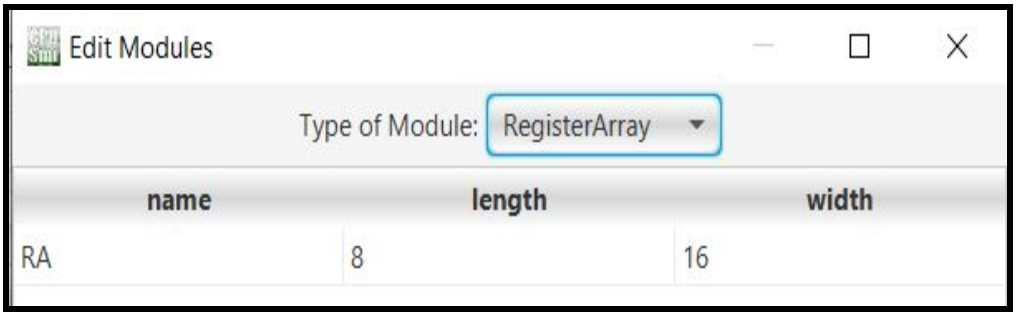
a. Registers:



name	width	initial value	read-only
ACC	16	0	☐
D1	16	0	☐
D2	16	0	☐
IR	16	0	☐
MAR	8	0	☐
MDR	16	0	☐
PC	8	0	☐
STATUS	8	0	☐

Two registers **D1** and **D2** of 16-bit width are added.

b. Register Array:



name	length	width
RA	8	16

Register Array of length 8 and width 16 bits is added.

2. **<u>MicroInstructions</u>**:

a. Data transfer from Register to Register Array:

Edit Microinstructions

Type of Microinstruction: TransferRtoA

name	source	srcStart...	dest	destSta...	numBits	index	indexSt...	indexN...
ACC->RA	ACC	0	RA	0	16	STATUS	5	3
D1->RA	D1	0	RA	0	16	STATUS	5	3
D2->RA	D2	0	RA	0	16	STATUS	5	3
MDR->RA	MDR	0	RA	0	16	STATUS	5	3

Indexing to **RA** is being done by **STATUS** register, in practice however, bits from **IR** are transferred to **STATUS.**
(Bits 5-7 and 8-11 of **IR** have been used as indices for transfer to and from from **RA** in machine instructions)
We are using two different sets of indices because in the machine instruction “add r1 r2” and sub r1 r2, 2 registers have to be specified. The length of the bits specified for each index is 3 bits because there are 8 registers in the register array.

b. Data transfer from Register Array to Register:

Edit Microinstructions

Type of Microinstruction: TransferAtoR

name	source	srcstart...	dest	destSta...	numBits	index	indexSt...	indexN...
RA->ACC	RA	0	ACC	0	16	STATUS	5	3
RA->D1	RA	0	D1	0	16	STATUS	5	3
RA->D2	RA	0	D2	0	16	STATUS	5	3
RA->MDR	RA	0	MDR	0	16	STATUS	5	3

c. Data transfer from Register to Register:

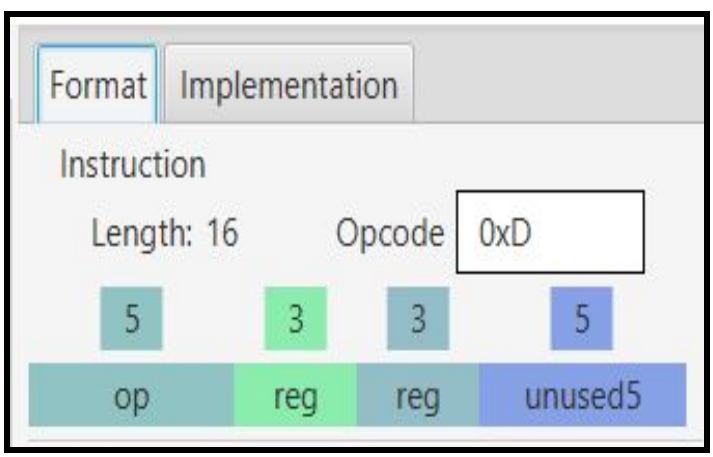
Edit Microinstructions

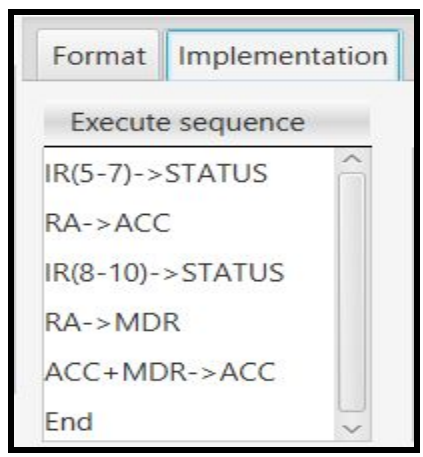
Type of Microinstruction: TransferRtoR

name	source	srcStartBit	dest	destStartBit	numBits
ACC->D1	ACC	0	D1	0	16
ACC->D2	ACC	0	D2	0	16
ACC->MDR	ACC	0	MDR	0	16
D1(8-15)->MAR	D1	8	MAR	0	8
D1->ACC	D1	0	ACC	0	16
D2->ACC	D2	0	ACC	0	16
IR(5-7)->STAT...	IR	5	STATUS	5	3
IR(8-10)->STA...	IR	8	STATUS	5	3
IR(8-15)->MAR	IR	8	MAR	0	8
IR(8-15)->PC	IR	8	PC	0	8
MDR->ACC	MDR	0	ACC	0	16
MDR->IR	MDR	0	IR	0	16
PC->MAR	PC	0	MAR	0	8

3. **<u>Assembly Instructions</u>**:

a. “add r1 r2” and similar: **add**

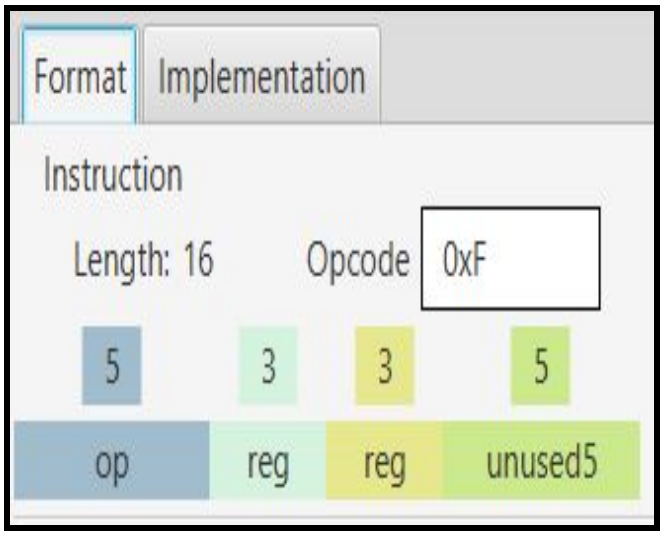


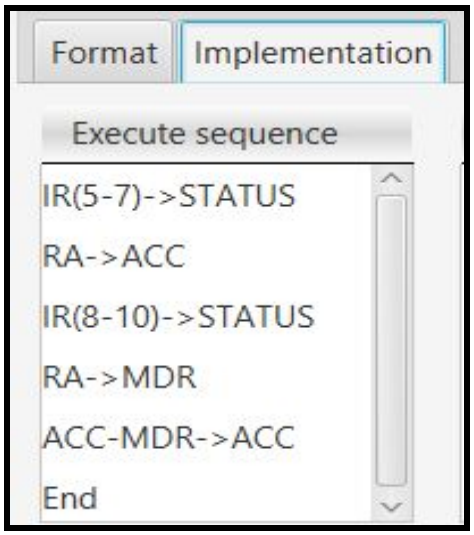


The contents of the first register are transferred to accumulator, followed by transfer of contents of second register to **MDR**, then addition occurs inside accumulator (**ACC**).

Note: normal **add** instruction from Part A has been renamed to **adda**.

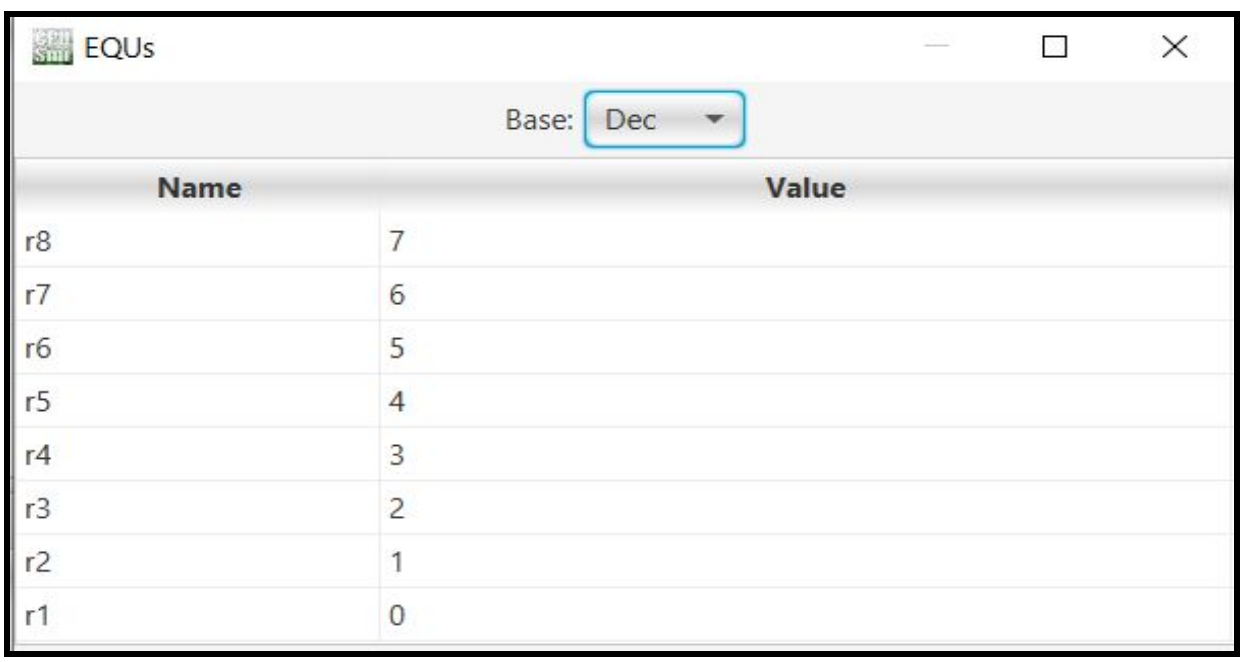
b. “sub r1 r2” and similar: **sub**





The contents of first register are being transferred to accumulator, followed by transfer of contents of second register to **MDR**, then subtraction occurs inside the accumulator (**ACC**).
Note: normal **sub** instruction from Part A has been renamed to **suba**.

EQUs :



Name	Value
r8	7
r7	6
r6	5
r5	4
r4	3
r3	2
r2	1
r1	0

The assignment required instructions of the form “add r1 r2” and “sub r1 r2” and the use of EQUs facilitates that. They replace the string “r1” by 0, “r2” by 1 and so on. This ensures that the correct indices are used in the instructions.

4. **<u>Code</u>**:

partB.a ×

```
;Program to take input till 0 is entered and then print the sum of elements entered so far
input_till_0: ; loop to take input
    ipa ; take input
    jmpn input_till_0 ; if negative number entered then don't store in memory
    jmpz if_input_0 ;if zero is entered then exit the loop

    m2m dataloc ; store the input number in memory

    lda dataloc ; increase the  memory pointer
    adda one
    sta dataloc

    lda count ; increase count of input numbers
    adda one
    sta count

    jmp input_till_0

if_input_0:
    lda dataloc ; reduce the memory pointer so it points to the start of the memory block used to store positive numbers
    suba count
    sta dataloc

output_sum_FIFO:
    m2a dataloc ;loads the value of memory in accumlator
    adda sum_till_count ;add value in the given memory location to accumlator
    sta sum_till_count ;store in a memory location from accumlator
    opa ; print value of the accumlator

    lda dataloc ;increase value of memory block pointer
    adda one
    sta dataloc

    lda count;decrease the value of count
    suba one
    sta count
    jmpp output_sum_FIFO

    stop

dataloc: .data 1 70 ; stores the memory address of the next loc where data can be written / is present
one: .data 1 1; constant value 1
count: .data 1 0 ;stores number of elements in the memory block
sum_till_count: .data 1 0 ;stores sum so far
```

Explanation:

dataloc: stores the location where the positive numbers entered are to be stored, starting from 70 and increasing by 1 each time a number is written to the memory and decreasing by 1 each time a number is retrieved from memory for LIFO printing.

one: used to increment and decrement the values of variables by 1.

count: used to keep track of the number of elements stored/printed.

sum_till_count: stores the sum of positive numbers entered till index **count**.

input_till_0 loop: takes input from user and if number entered is 0, exits to **output_sum_FIFO.** If the number entered is positive, it stores the number in the location pointed by **dataloc** using **m2m** and increments **dataloc** and **count** by **one**.

if_input_0: Reduces **dataloc** by **count** and continues to **output_sum_FIFO**. This makes **dataloc** point to the starting of the stored number sequence.

output_sum_FIFO: adds the number in **dataloc** to **sum_till_count** and prints **sum_till_count**, then it increments **dataloc** by **one** and decrements **count** by **one**. This happens till **count** becomes zero after which it exits the program.

```
EXECUTING...
Enter Inputs, the first of which must be an Integer: 1 3 4 2 -1 3 5 -2 2 0
Output:  1
Output:  4
Output:  8
Output:  10
Output:  13
Output:  18
Output:  20
EXECUTION HALTED NORMALLY due to the setting of the bit(s): [HALT-BIT]
```

CONTRIBUTION BY GROUP MEMBERS

1. Mudit Chaturvedi : Worked on Part A of the Assignment (Microinstructions and machine instructions), Part B of the Assignment (Microinstructions and coding).

2. Hardik Parnami : Worked on Part A of the Assignment (coding and testing), documentation of Part B of the assignment and on formatting of report and explanations of the first section of report.

3. Kriti Jethlia : Worked on Part A of the assignment (Hardware modules and opcodes), Part B of the assignment (Opcodes and coding) and on documentation of the codes for both parts.

4. Sristi Sharma : Worked on Part B of the assignment (Hardware modules and machine instructions), documentation of part A of the assignment and on formulation of the report.

5. Mayank Negi : Worked on Part A of the assignment (coding and debugging), Part B of the assignment (microinstructions and debugging) and on formatting of report and explanations of the first section of report.