Computer Architecture
Project - CPUSim

Group No. - 11

Members:

Name	ID Number
Aditya Agarwal	2017B1A71075H
Rudrajit Kargupta	2017B3A70452H
Raunak Mantri	2017B5A71340H
Amogh Saxena	2017B4A71731H
Jalaj Bansal	2017B3A71610H

Table of Contents

Introduction

We have implemented the architecture for the CPU described in our assignment using CPU Sim 4.0.11.

Implementation

Hardware Modules

The following pictures depict the hardware modules implemented as per the requirements of the assignment -

- RAM: As per the assignment requirements the RAM was implemented to have a length of 256 cells, with each cell being of the size 16 bits

Type of Module: RAM

name	length	cellSize
Main	256	16

- Registers: 6 registers were implemented:
  - ACC is the Accumulator register having a size of 16 bits
  - IR is the Instruction Register which would store the instructions (4 bits of opcode +4bits of EMPTY(ignored) + 8 bits for address or 4 bits opcode + 12 bits Unused)
  - MAR is the Memory Address Register (8 bits) to store the address for read/write operations
  - MDR is the Memory Data Register (16 bits) to store the data for reading/writing to the main memory (in this case, the RAM)
  - PC is the Program Counter register (8 bits) which stores the address of the current instruction being executed and is then incremented by 1 in order to proceed to the next instruction.
  - Status register (8 bits) is used like a flag register which is primarily used for implementing the carry, overflow and halt registers (and their functionalities)

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	☐

- Condition Bit: The halt box has been checked for the halt-bit, which indicates the execution to be halted when the halt bit is set.

Type of Module: ConditionBit

name	register	bit	halt
carry-bit	STATUS	1	☐
halt-bit	STATUS	0	☑
overflow	STATUS	2	☐

Micro instructions for the hardware modules

- Transfer from register to register
  - Accumulator to Memory data register
  - Instruction register to Memory Address (Only address portion of register, 8 bits)
  - Instruction register to Program Counter (Only address portion of register, 8 bits)
  - Memory data register to Memory address register (8 bits address)
  - Memory data register to Accumulator
  - Memory data register to Instruction register
  - Program counter to Memory address register (8 bits)

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)->m...	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

- Add and subtract arithmetic
  - Add data in Accumulator and MDR and store the result in the accumulator.
  - Subtract data in MDR from data in Accumulator and store the result in the accumulator.

Type of Microinstruction: Arithmetic

name	type	source1	source2	destination	overflowBit	carryBit
acc+mdr->...	ADD	ACC	MDR	ACC	overflow	carry-bit
acc-mdr->a...	SUBTRACT	ACC	MDR	ACC	overflow	carry-bit

- Test Accumulator (Self-explanatory)

Type of Microinstruction: Test

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

- Memory Access
  - RAM to Memory data register
  - Memory data register to RAM

Type of Microinstruction: MemoryAccess

name	direction	memory	data	address
Main[mar]->mdr	read	Main	MDR	MAR
mdr->Main[mar]	write	Main	MDR	MAR

- Increment Program Counter

Type of Microinstruction: Increment

name	register	overflowBit	carryBit	delta
Inc2-pc	PC	halt-bit	(none)	1

- Input to accumulator and Output from accumulator

Type of Microinstruction: IO

name	type	buffer	direction
input-int->acc	integer	ACC	input
output-acc->int	integer	ACC	output

- Decode instruction

Type of Microinstruction: Decode

name	ir
decode-ir	IR

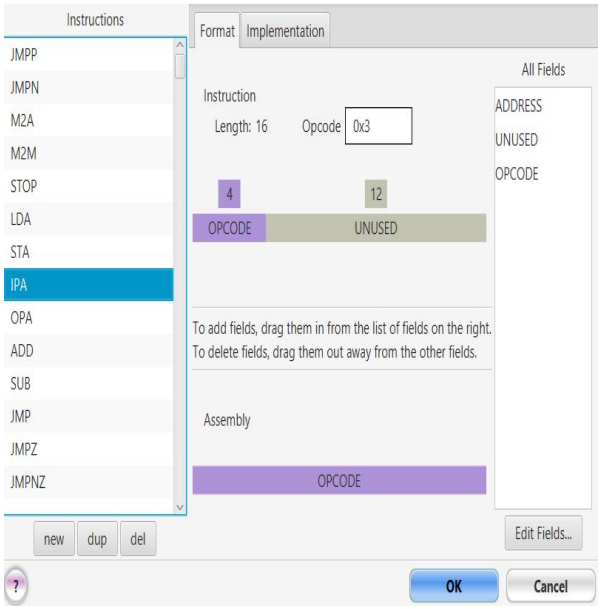
- Set condition bit
  - Halt bit

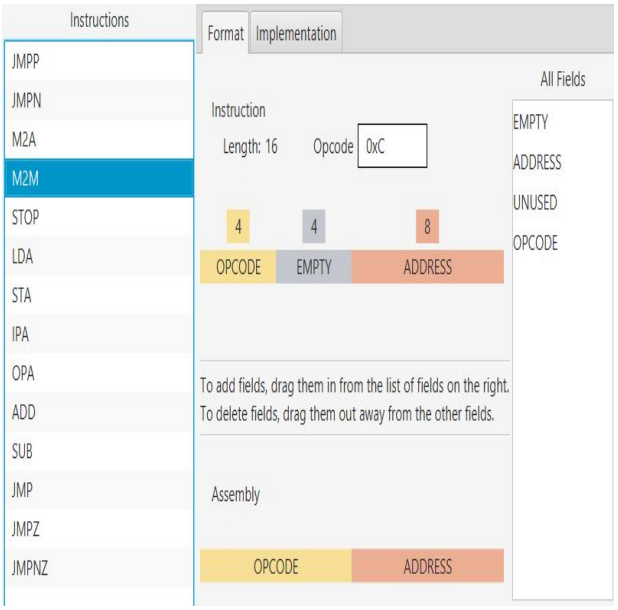
Type of Microinstruction: SetCondBit

name	bit	value
set-halt-bit	halt-bit	1

Defining Machine Instructions/Assembly Instructions

This is general format for our instructions -





Implementation of the instructions -

- Unconditional jump

    - JMP - Address of instruction from the Instruction register is loaded to the Program counter.

- Conditional Jumps
  (Here we use the omission value to skip that number of microinstructions to make the conditional assembly instructions).

    - JMPNZ - Test Accumulator value is non zero and load instruction from IR to PC.
    - JMPP - Test Accumulator value >0 and load instruction from IR to PC.
    - JMPN - Test Accumulator value <0 and load instruction from IR to PC.
    - JMPZ - Test Accumulator value equal to zero and load instruction from IR to PC.

- LOAD

    - First we load the address from the instruction register to the Memory address register.
    - Then we move the data stored at that address in the main memory (RAM) to the Memory data register.
    - Finally, we move the data to the accumulator.

- STORE

    - First we load the address from the instruction register to the Memory address register.
    - Then we move the data from the accumulator to the Memory data register.
    - Finally we move the data from the Memory data register to the particular address in main memory (RAM).

- READ INPUT

    - We take integer input and transfer it to the accumulator.

- WRITE OUTPUT

    - We move integer data from accumulator to output.

- ADD

    - First we load the address from the Instruction register to the Memory address register.
    - Then we retrieve data from the Main memory at that address and transfer to the Memory data register.
    - Finally, we add the data in the accumulator and the memory data register and store the result back into the accumulator.

- SUB
    - First we load the address from the Instruction register to the Memory address register.
    - Then we retrieve data from the Main memory at that address and transfer to the Memory data register.
    - Finally, we subtract the data in the memory data register from the data in the accumulator and store the result back into the accumulator.

- MEMORY TO ACCUMULATOR TRANSFER:

    - First, we load the address from the Instruction register to the Memory address register.
    - Then we retrieve data from the Main memory at that address and transfer to the Memory data register.
    - Now we transfer the address from the memory data register to the memory address register.
    - Thereafter we retrieve data from the Main memory at the address specified by the MAR, which we transferred in the previous step and store it into the memory data register.
    - Once the retrieval is done into the MDR, the data is then transferred to the Accumulator, thereby facilitating the Memory to Accumulator functionality.

- MEMORY TO MEMORY TRANSFER: This instruction transfers the contents of the accumulator to a memory location specified in the instruction. Which is achieved by the following steps.

    - First, we load the address from the Instruction register to the Memory address register.
    - Then we retrieve data from the Main memory at that address and transfer to the Memory data register.
    - Now we transfer the address from the memory data register to the memory address register.
    - Thereafter the content of the accumulator is transferred to the Memory Data Register.
    - Then the content of the Memory Data Register is transferred to the Main memory at the address specified by the Memory Address Register.

- STOP:
    - We just simply set the Halt bit here, which then leads to the termination of the program execution.

Code:

```
takeInput:
    IPA                     ;Take INPUT
    JMPZ loadStackAddress   ;if INPUT = 0, stop input
    JMPN takeInput          ;if INPUT is -ve, ignore the input (don't store)
    M2M stackAddress        ;if INPUT is +ve, continue and save the value from accumulator to memory value pointed by stackAddress
    LDA stackAddress        ;Load address value stored at stackAddress i.e 150 initially to accumulator
    ADD one                 ;Add one to the value so as to store the next value at next cell
    STA stackAddress        ;Store the new updated accumulator value(address) at stackAddress
    JMP takeInput           ;JUMP to take another input

loadStackAddress:
    LDA stackAddress        ;This step is to load the latest updated adress of the top of stack
    SUB one                 ;Since after the last step also one was added to this adress, we subtract one
    STA stackAddress        ;And store that address value from accumulator to the address

displayStack:
    M2A stackAddress        ;Move the value stored at address given by stackAddress's value(TOP of stack) to Accumulator
    OPA                     ;Print Accumulator Value
    LDA stackAddress        ;Load the address value stored at the given address
    SUB one                 ; Subract one to get the previos top of stack
    STA stackAddress        ; STore the updated adress value from accumulator to the address
    LDA stackAddress        ;Load the address value stored at the given address
    SUB address             ;Subtract with the inital reference address value
    JMPN exit               ;If that is zero, exit the program
    JMP displayStack        ;Else continue with the display of stack

exit: STOP                  ;To exit program

address: .data 1 150        ; Just for reference of the inital start address of Stack
one: .data 1 1              ; Just store one to be used in instructions
stackAddress: .data 1 150   ; Address of Top of Stack
```

PART B

Introduction

Opcode for every instruction has been changed to 5 bits and the length of unused portion in the instruction is reduced by 1 bit to maintain the instruction size to 16 bits. Instead of showing for every instruction we have shown the formats of similar instructions as one group.

M2m and m2a implementations have also been changed as the mdr->mar transfer path is not present in the current architecture. This has been handled by moving the address through a path of mdr->acc->d1->mar instead of mdr->mar.

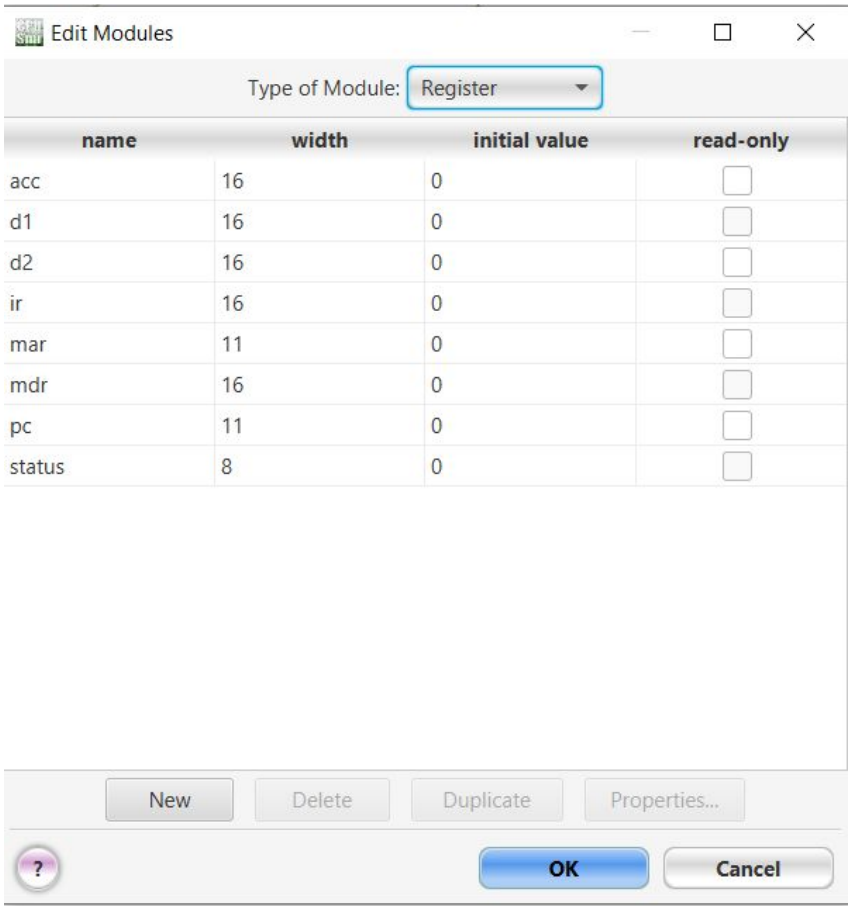
Implementation

1.) Hardware Modules

The following pictures depict the hardware modules that are added in this part as per the requirements of the assignment -

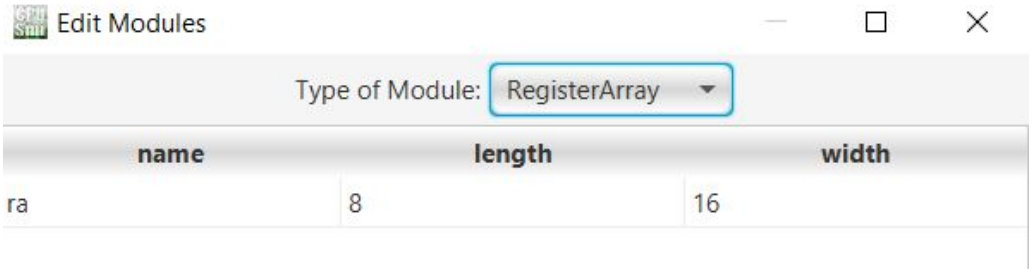
- **Registers:**

  The registers have been added according to the problem statement. Mar and pc are 11-bit registers as they are address registers and the opcode is of 5 bits. The status register is 8 bits and all other registers are 16 bits as per the given question.



- **RegisterArray:**

A register array named “ra” having 8 registers. Used EQUs to indicate each register in the RegisterArray by r1 to r8.



EQUs

Base: Dec

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

- **Condition Bits**:

The condition bits are set using the status register.

Edit Modules

Type of Module: ConditionBit

name	register	bit	halt
carry-bit	status	1	☐
halt-bit	status	0	☑
overflow-bit	status	2	☐

- **Memory :**

The total memory is 256 words.

Edit Modules

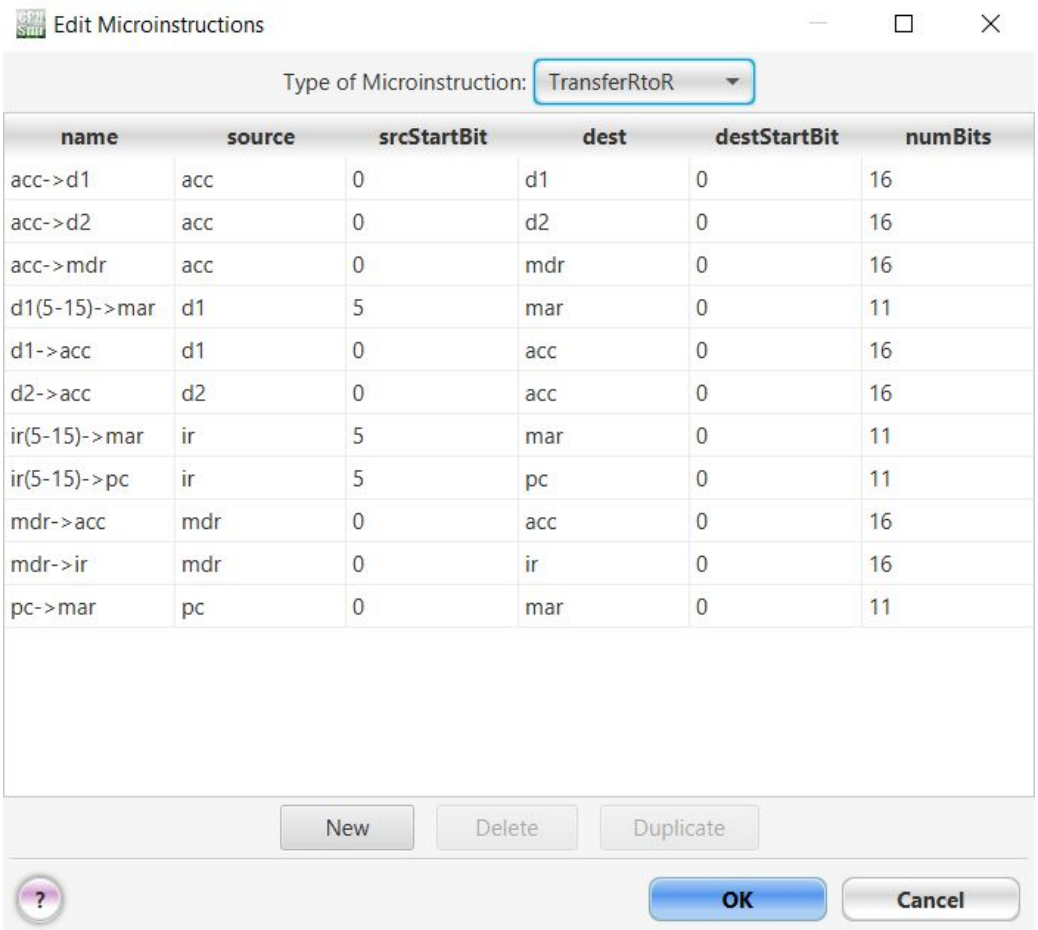
Type of Module: RAM

name	length	cellSize
RAM	256	16

2.) Micro instructions for the hardware modules

- **Transfer from register to register:**

This set of microinstructions represent the data buses between two registers. “Source” is the source of data. “Dest” is the destination of sats. The start and end indices of data transfer have been mentioned. The “numbits” represents the number of bits being transferred.



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(5-15)->mar	d1	5	mar	0	11
d1->acc	d1	0	acc	0	16
d2->acc	d2	0	acc	0	16
ir(5-15)->mar	ir	5	mar	0	11
ir(5-15)->pc	ir	5	pc	0	11
mdr->acc	mdr	0	acc	0	16
mdr->ir	mdr	0	ir	0	16
pc->mar	pc	0	mar	0	11

- **Transfer from Accumulator to Register**:

  This represents transfer from the register array to register.

Edit Microinstructions

Type of Microinstruction: TransferAtoR

name	source	srcstart...	dest	destSta...	numBits	index	indexSt...	indexN...
ra->d1	ra	0	d1	0	16	mar	0	3
ra->d2	ra	0	d2	0	16	mar	3	3

- **Transfer from Register to Accumulator:**

  This section represents the data transfers from register to register array. The data is being transferred to the register array “ra”. The bits indicating the index of the array to be accessed are represented using the “status” register.

Edit Microinstructions

Type of Microinstruction: TransferRtoA

name	source	srcStart...	dest	destSta...	numBits	index	indexSt...	indexN...
d1->ra	d1	0	ra	0	16	mar	0	3
d2->ra	d2	0	ra	0	16	mar	3	3

- **Arithmetic:**

  This set of instructions are used for arithmetic operations add/sub.

Edit Microinstructions

Type of Microinstruction: Arithmetic

name	type	source1	source2	destination	overflowBit	carryBit
acc+mdr->...	ADD	acc	mdr	acc	halt-bit	(none)
acc-mdr->a...	SUBTRACT	acc	mdr	acc	halt-bit	(none)

- **Test**

  This instruction set allows the user to skip 1 or 2 statements in the loop depending on the condition.

Edit Microinstructions

Type of Microinstruction: Test

name	register	start	numBits	comparison	value	omission
if(acc!=0)ski...	acc	0	16	NE	0	1
if(acc<0)ski...	acc	0	16	LT	0	2
if(acc==0)s...	acc	0	16	EQ	0	1
if(acc>0)ski...	acc	0	16	GT	0	2

- **Memory Access**

  This is used to access memory data at an address stored by “mar”.

Edit Microinstructions

Type of Microinstruction: MemoryAccess

name	direction	memory	data	address
RAM[mar]->mdr	read	RAM	mdr	mar
mdr->RAM[mar]	write	RAM	mdr	mar

- **Increment**

  This instruction is used to increment pc after reading each instruction.

Edit Microinstructions

Type of Microinstruction: Increment

name	register	overflowBit	carryBit	delta
Inc-pc	pc	halt-bit	(none)	1

- **I/O**

The I/O is carried on using acc register

Edit Microinstructions

Type of Microinstruction: IO

name	type	buffer	direction
input-int->acc	integer	acc	input
output-acc->int	integer	acc	output

- **Decode**

The instruction stored in IR needs to be decoded before usage.

Edit Microinstructions

Type of Microinstruction: Decode

name	ir
decode-ir	ir

- **Set condition bit**

This instruction is used to set the halt bit.

Edit Microinstructions

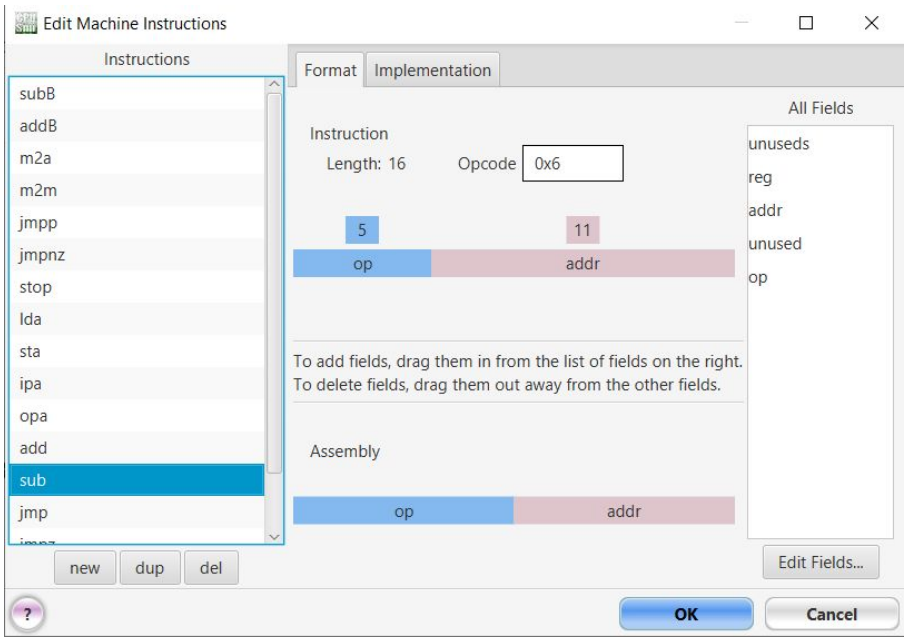
Type of Microinstruction: SetCondBit

name	bit	value
set-halt-bit	halt-bit	1

3) MACHINE INSTRUCTIONS

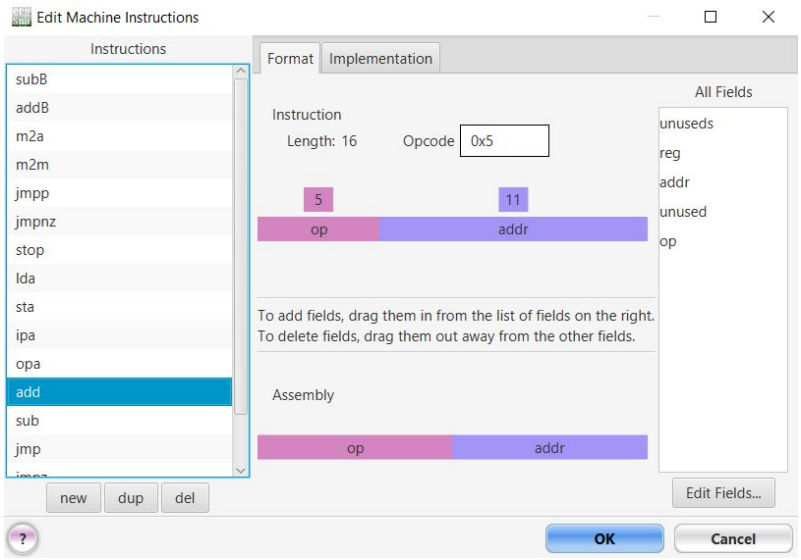
- **sub**

  This instruction holds the indices for the operands, retrieves those operands from the register array, performs subtraction and stores the value back in the accumulator.



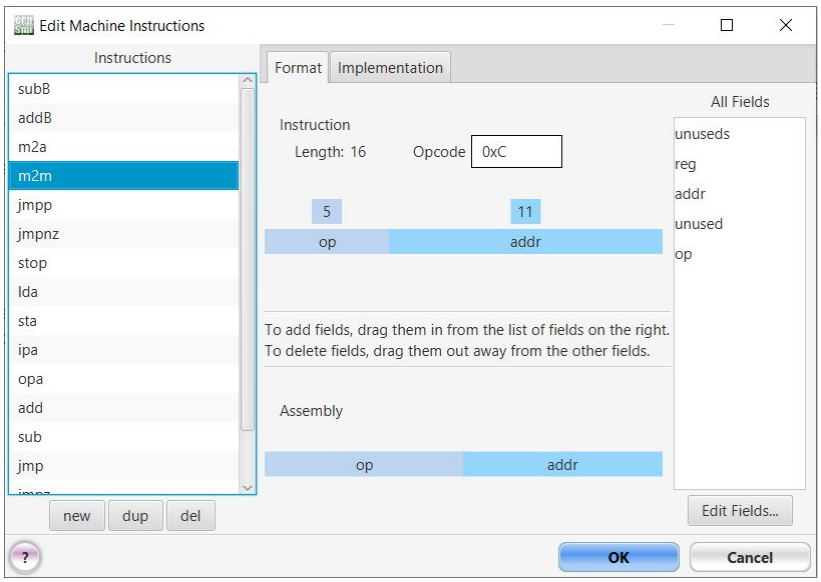
- **Add**

  This instruction holds the indices for the operands, retrieves those operands from the register array, performs addition and stores the value back in the accumulator.



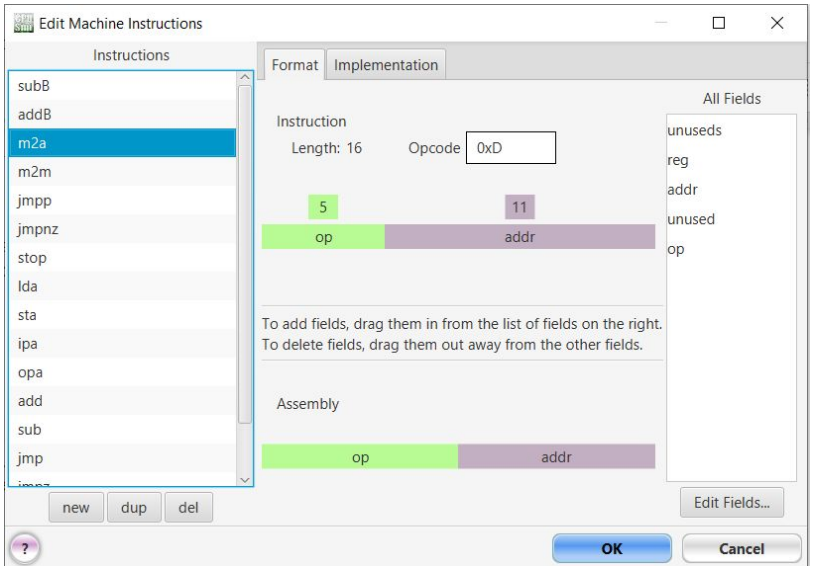
- **m2m**

This allows the information stored in the accumulator to be stored in a memory location whose value is stored in the memory address mentioned in the instruction.



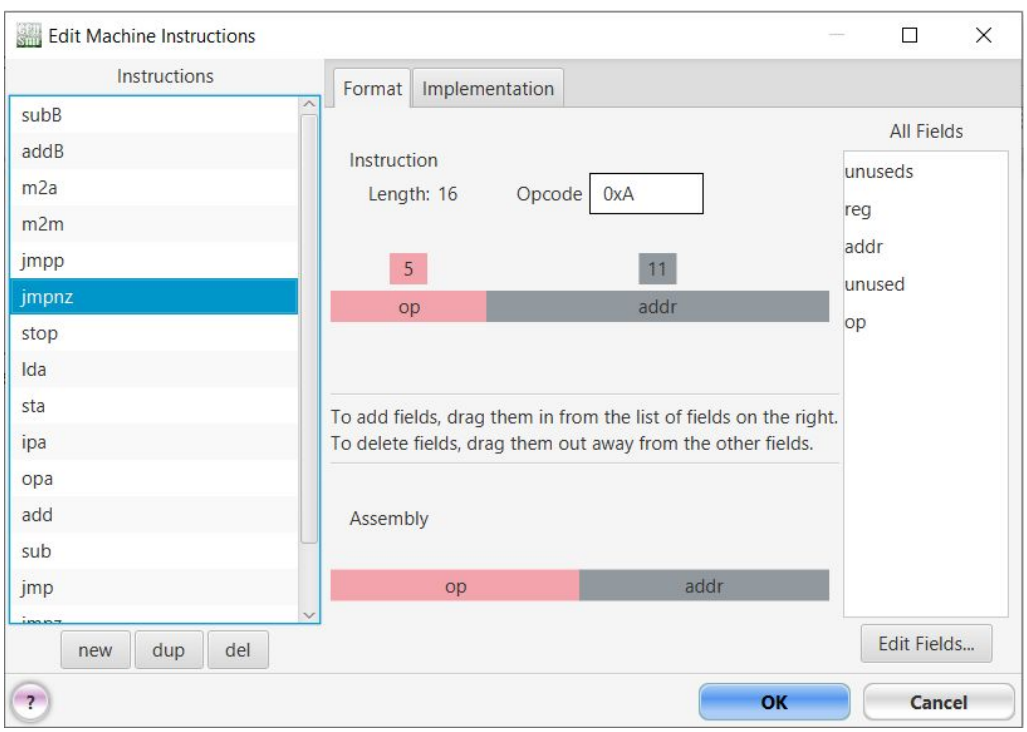
- **m2a**

This allows the information transfer to the accumulator. A memory location whose value is stored in the memory address mentioned in the instruction is retrieved and the accumulator value is stored in the retrieved memory address.

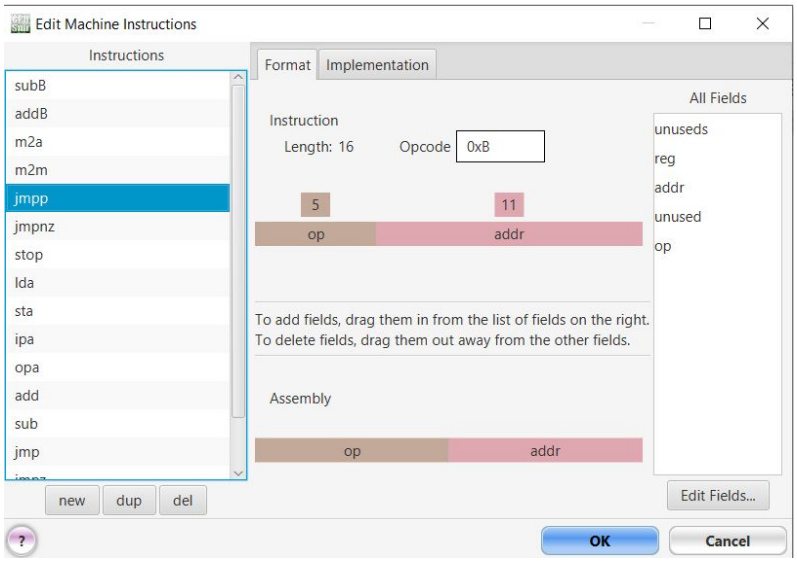


- **Jmpnz**

This instruction allows the user to jump to a new location to get the next instruction to be executed if the value in the accumulator is not equal to 0. If the accumulator value is not 0, the address of the new instruction is loaded into the program counter.

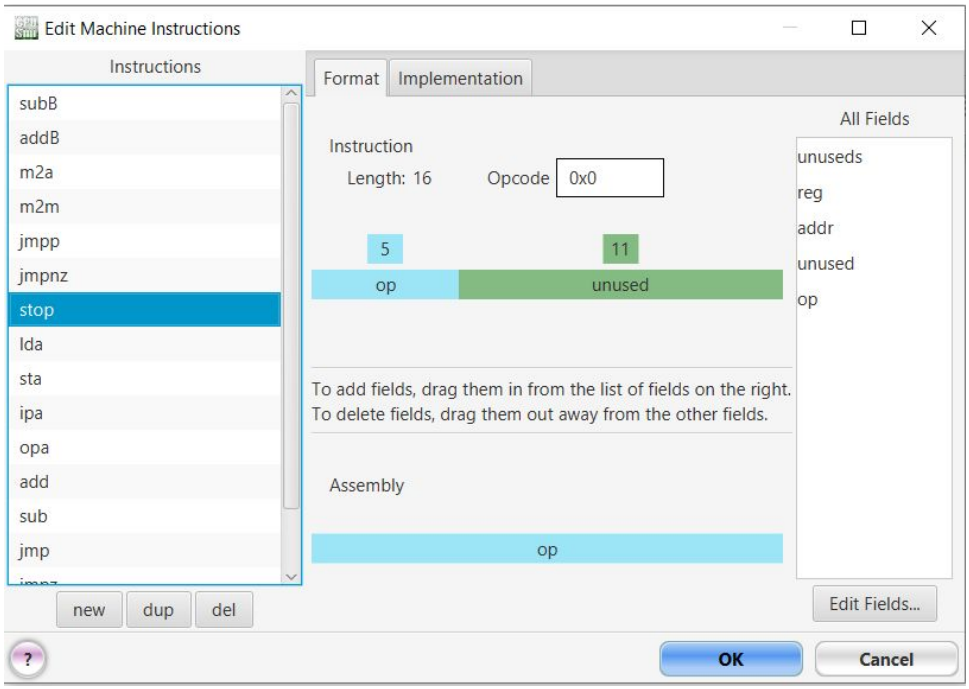


- **Jmpp**



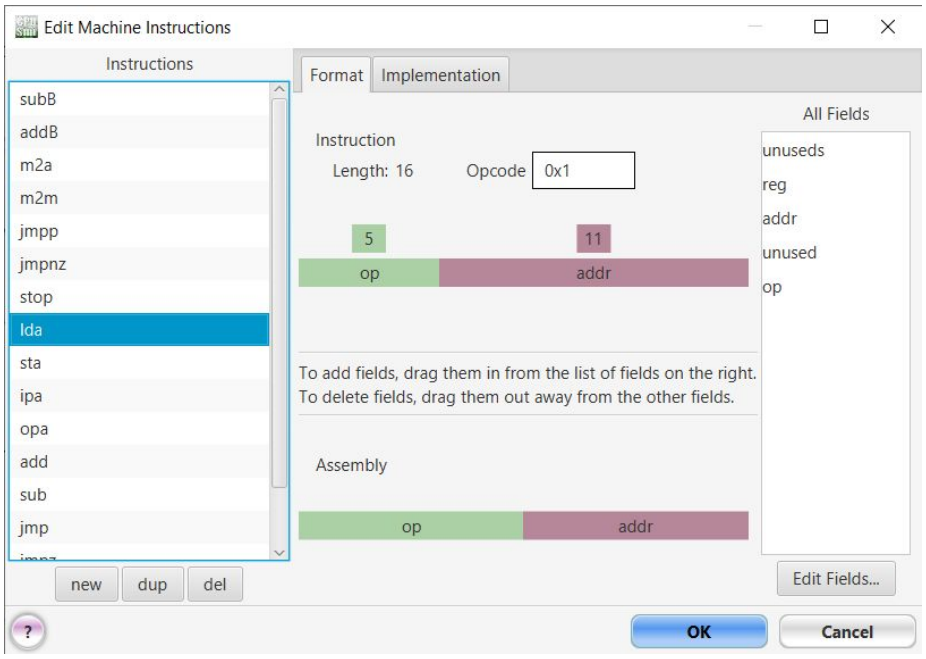
- **Stop**

This condition allows you to end the program by setting the halt bit.



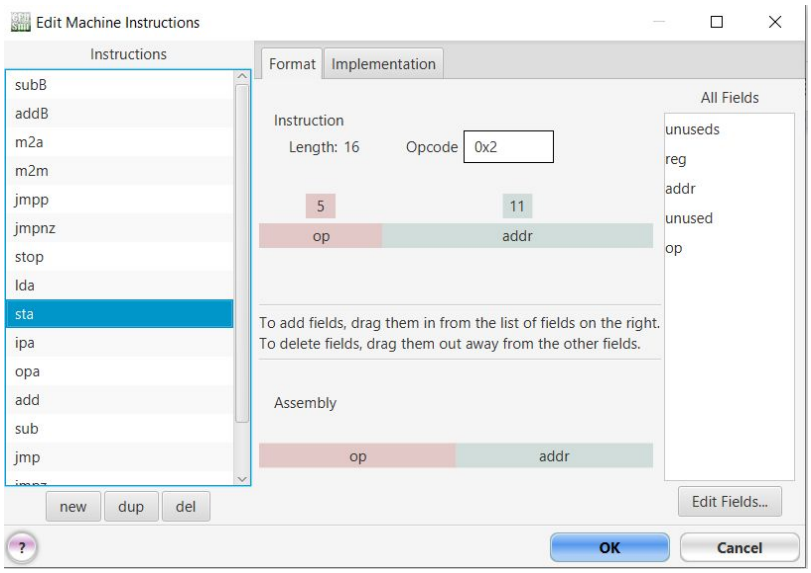
- **Lda**

This instruction retrieves data stored in a specific memory address and loads that value into the accumulator.

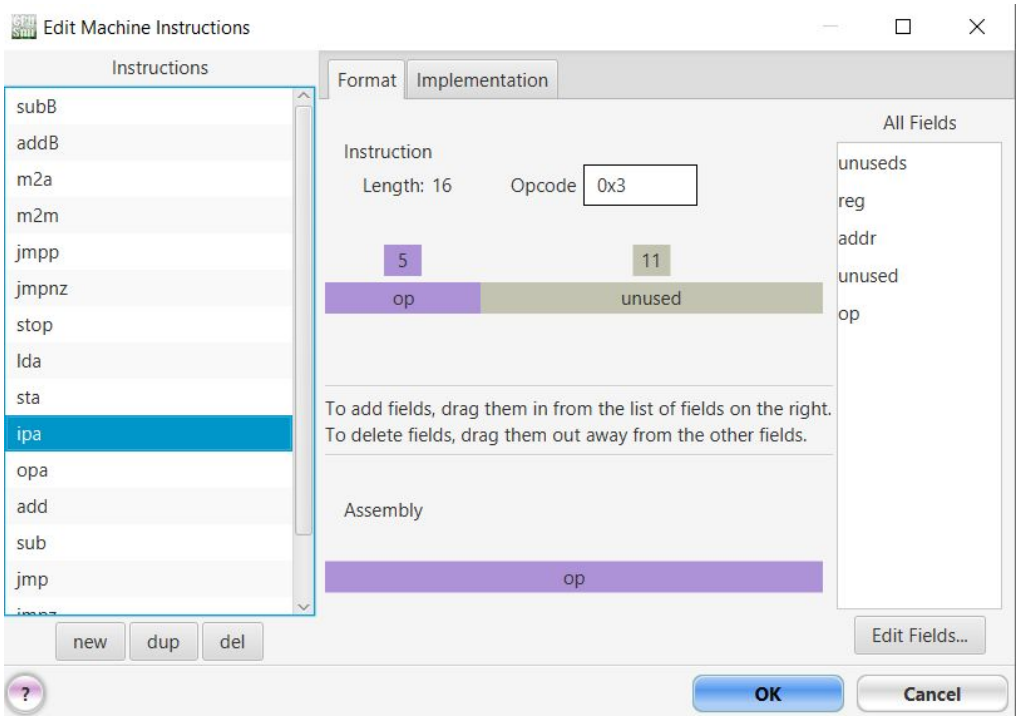


- **Sta**

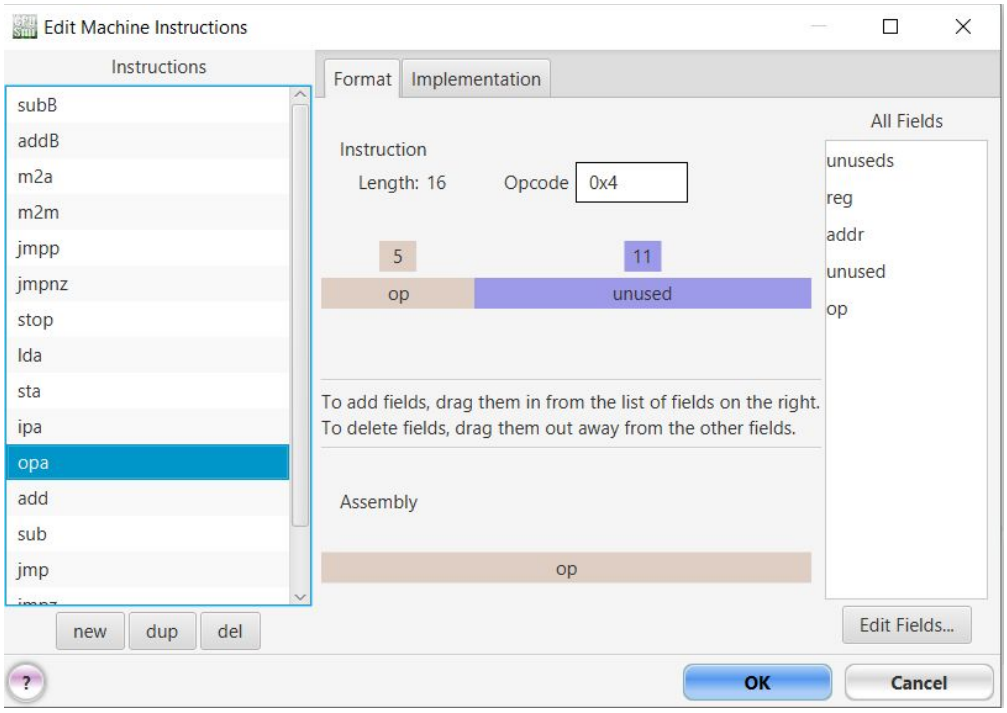
This instruction allows the data retrieved from the accumulator to be stored in a specific address in the memory.



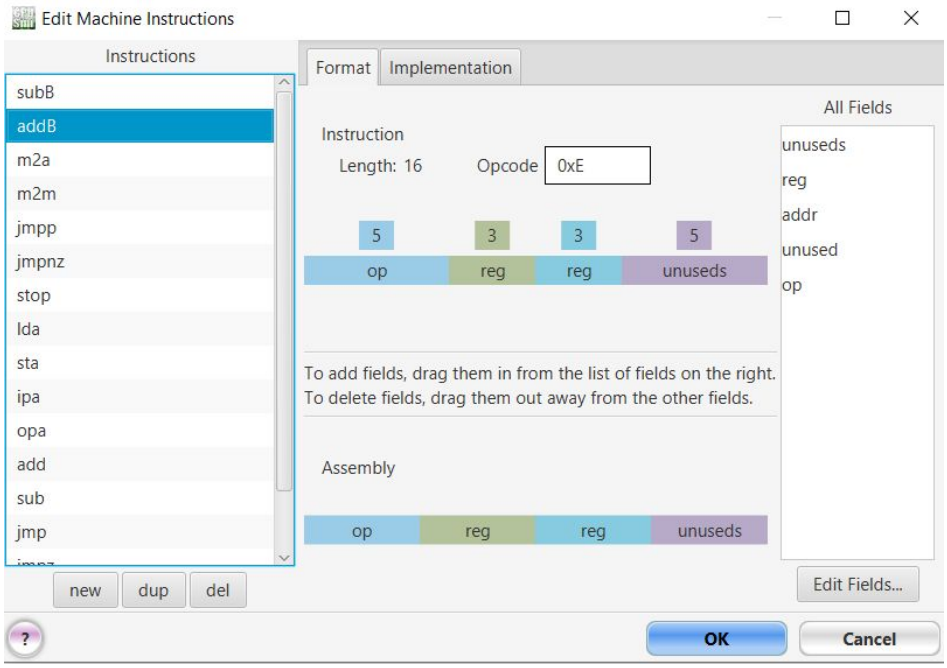
- **Ipa**


Edit Machine Instructions
Instructions
subB
addB
m2a
m2m
jmpp
jmpnz
stop
lda
sta
ipa
opa
add
sub
jmp
new
dup
del
Format
Implementation
Instruction
Length: 16
Opcode
0x3
5
11
op
unused
To add fields, drag them in from the list of fields on the right.
To delete fields, drag them out away from the other fields.
Assembly
op
All Fields
unuseds
reg
addr
unused
op
Edit Fields...
OK
Cancel

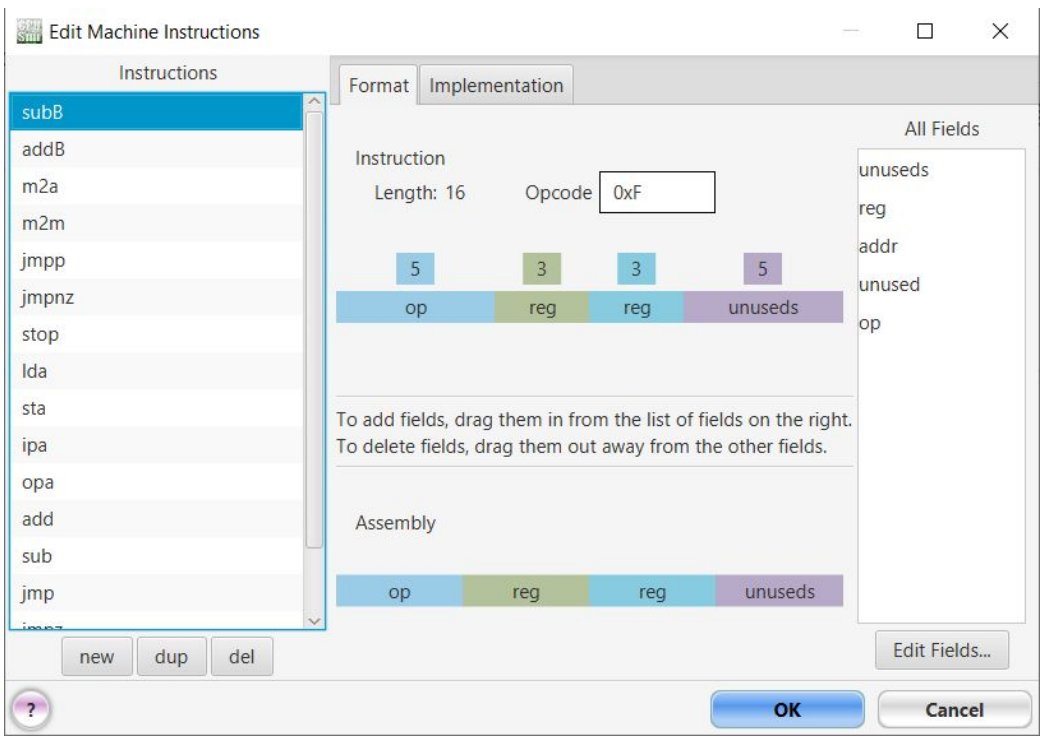
- **Opa**


Edit Machine Instructions
Instructions
subB
addB
m2a
m2m
jmpp
jmpnz
stop
lda
sta
ipa
opa
add
sub
jmp
new
dup
del
Format
Implementation
Instruction
Length: 16
Opcode
0x4
5
11
op
unused
To add fields, drag them in from the list of fields on the right.
To delete fields, drag them out away from the other fields.
Assembly
op
All Fields
unuseds
reg
addr
unused
op
Edit Fields...
OK
Cancel

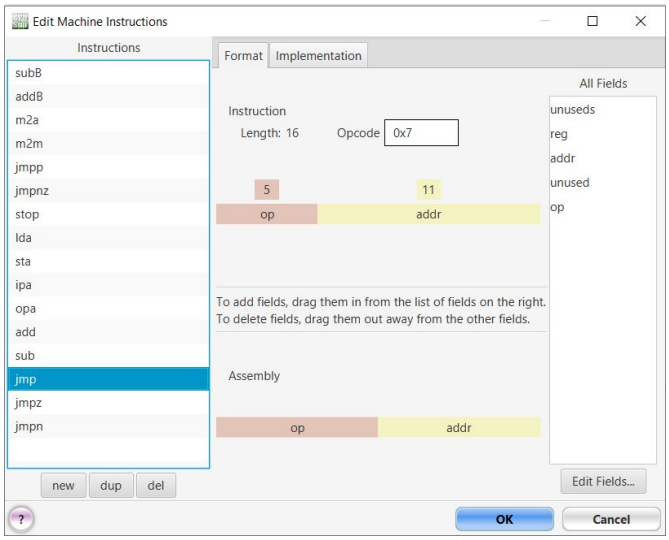
- **addB**


Edit Machine Instructions
Instructions
subB
addB
m2a
m2m
jmpp
jmpnz
stop
lda
sta
ipa
opa
add
sub
jmp
new
dup
del
Format
Implementation
Instruction
Length: 16
Opcode
0xE
5
3
3
5
op
reg
reg
unuseds
To add fields, drag them in from the list of fields on the right.
To delete fields, drag them out away from the other fields.
Assembly
op
reg
reg
unuseds
All Fields
unuseds
reg
addr
unused
op
Edit Fields...
OK
Cancel

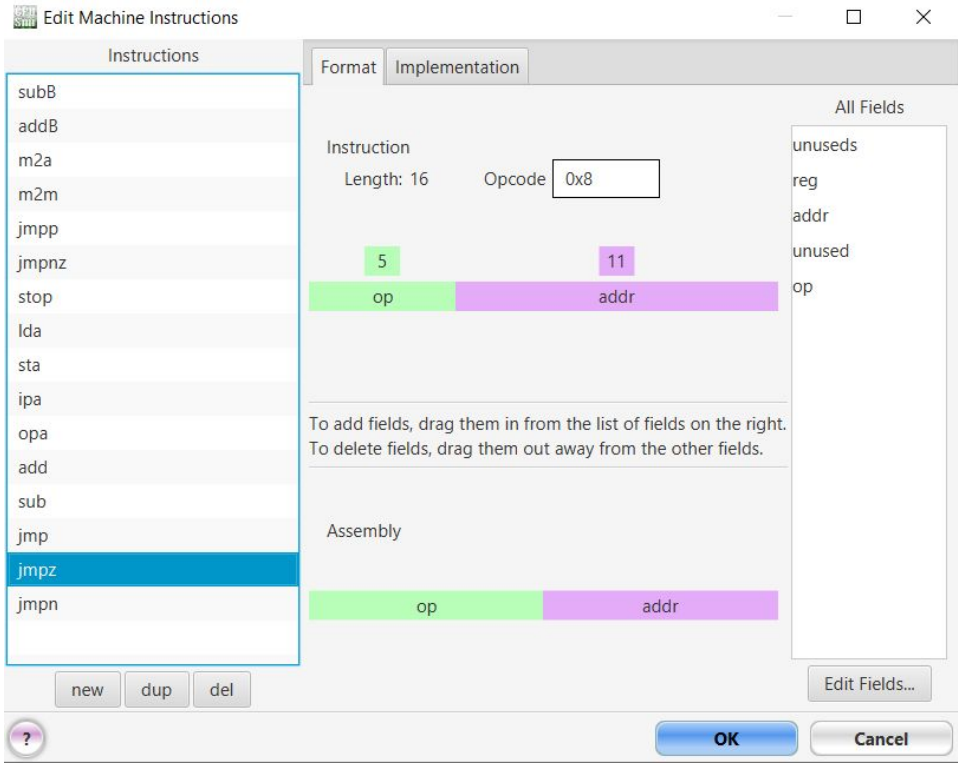
- **subB**


Edit Machine Instructions
Instructions
subB
addB
m2a
m2m
jmpp
jmpnz
stop
lda
sta
ipa
opa
add
sub
jmp
new
dup
del
Format
Implementation
Instruction
Length: 16
Opcode
0xF
5
3
3
5
op
reg
reg
unuseds
To add fields, drag them in from the list of fields on the right.
To delete fields, drag them out away from the other fields.
Assembly
op
reg
reg
unuseds
All Fields
unuseds
reg
addr
unused
op
Edit Fields...
OK
Cancel

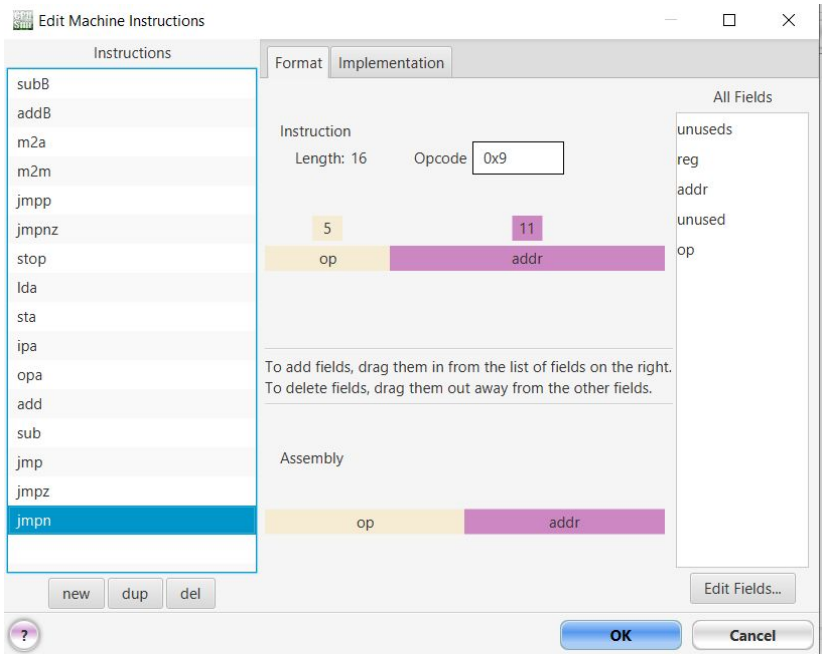
- **Jmp**


Edit Machine Instructions
Instructions
subB
addB
m2a
m2m
jmpp
jmpnz
stop
lda
sta
ipa
opa
add
sub
jmp
jmpz
jmpn
new
dup
del
Format
Implementation
Instruction
Length: 16
Opcode
0x7
5
11
op
addr
To add fields, drag them in from the list of fields on the right.
To delete fields, drag them out away from the other fields.
Assembly
op
addr
All Fields
unuseds
reg
addr
unused
op
Edit Fields...
OK
Cancel

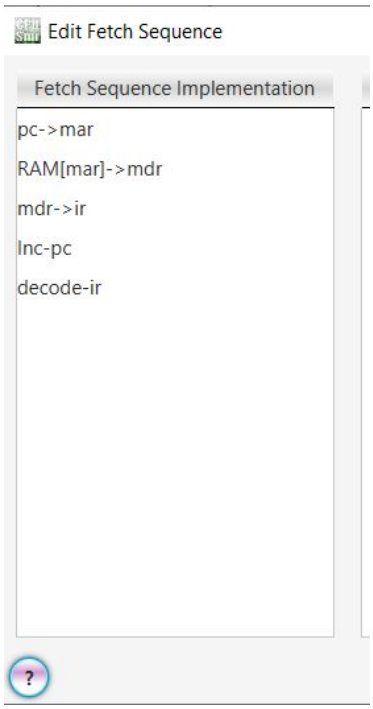
- **Jmpz**


Edit Machine Instructions
Instructions
subB
addB
m2a
m2m
jmpp
jmpnz
stop
lda
sta
ipa
opa
add
sub
jmp
jmpz
jmpn
new
dup
del
Format
Implementation
Instruction
Length: 16
Opcode
0x8
5
11
op
addr
To add fields, drag them in from the list of fields on the right.
To delete fields, drag them out away from the other fields.
Assembly
op
addr
All Fields
unuseds
reg
addr
unused
op
Edit Fields...
OK
Cancel

- **Jmpn**



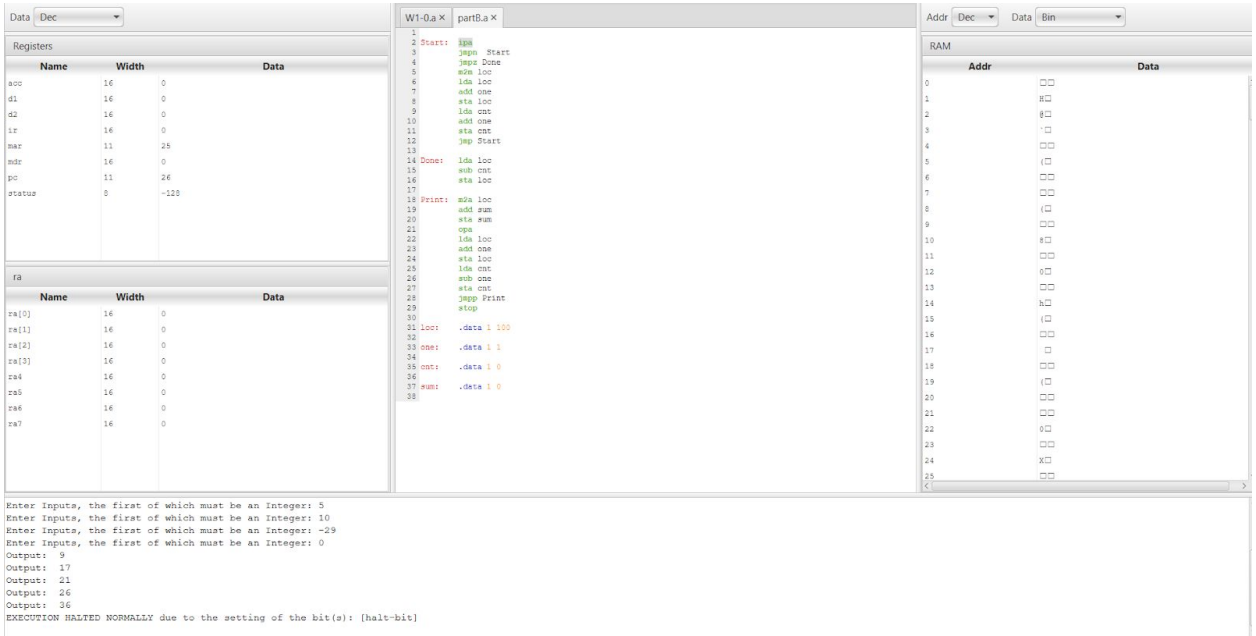
4) FETCH SEQUENCE



5) **PART B PROGRAM CODE:**

```
Start:  ipa
        jmpn  Start
        jmpz Done
        m2m loc
        lda loc
        add one
        sta loc
        lda cnt
        add one
        sta cnt
        jmp Start

Done:   lda loc
        sub cnt
        sta loc

Print:  m2a loc
        add sum
        sta sum
        opa
        lda loc
        add one
        sta loc
        lda cnt
        sub one
        sta cnt
        jmpp Print
        stop

loc:    .data 1 100

one:    .data 1 1

cnt:    .data 1 0

sum:    .data 1 0
```

6) FULL WINDOW AFTER PROGRAM EXECUTION:




```
Enter Inputs, the first of which must be an Integer: 5
Enter Inputs, the first of which must be an Integer: 10
Enter Inputs, the first of which must be an Integer: -29
Enter Inputs, the first of which must be an Integer: 0
Output:  9
Output:  17
Output:  21
Output:  26
Output:  36
EXECUTION HALTED NORMALLY due to the setting of the bit(s): [halt-bit]
```