CS6461 - COMPUTER ARCHITECTURE
PROJECT DESIGN NOTES - GROUP 5

The objective of the project is to develop a simulator of a small classical CISC computer based on assembly language. The phase 1 of the project will have the following characteristics:

- 4 general purpose registers (GPRs) – each 16 bits in length
- 3 Index registers – 16 bits in length
- 16-bit words
- Memory of 2048 words, expandable to 4096 words
- Word addressable

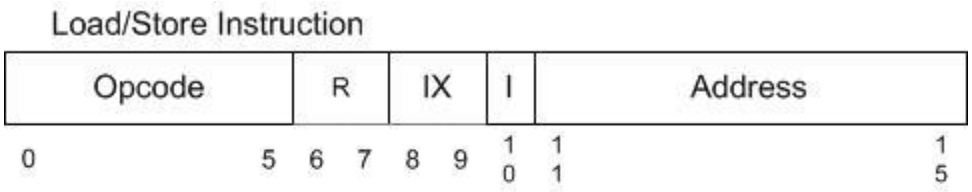
The four GPRs are named as: R0, R1, R2 and R3

The index registers are named as: X1, X2 and X3

Phase 1 implementation:

Load/store instruction

Every instruction should be in this format:



Opcode (6 bits) – Specifies one of 64 possible instructions (5 types in load and store)

IX (2 bits) – Specifies one of the three index registers (X1, X2 or X3), when IX is 0 there is no indexing

R (2 bits) – specifies one of the four general purpose registers (R0, R1, R2 or R3)

I (1 bit) – specifies indirect addressing when I=1 otherwise no indirect addressing

Address (7 bits) – specifies one of the 32 locations

The value of **IX** is used to select the indexing register in the following way:

00	No indexing
01	Index Register 1
10	Index Register 2
11	Index Register 3

The value of **I** denotes whether Indirect addressing is used or not:

0	No indirect addressing
1	Indirect addressing

CS6461 - COMPUTER ARCHITECTURE
PROJECT DESIGN NOTES - GROUP 5

To execute the different Load/Store instructions, the Effective Address(EA) should be computed as follows:

```
Effective Address (EA) =
    If I field=0
        If IX=00, contents of the address field
        If IX=1,2 or 3, c(Xj) + contents of the Address field, where j = c(IX)
    If I field=1
        if IX = 00, c(Address)
        if IX = 1,2 or 3, c(c(Xj) + Address), where j = c(IX)
```

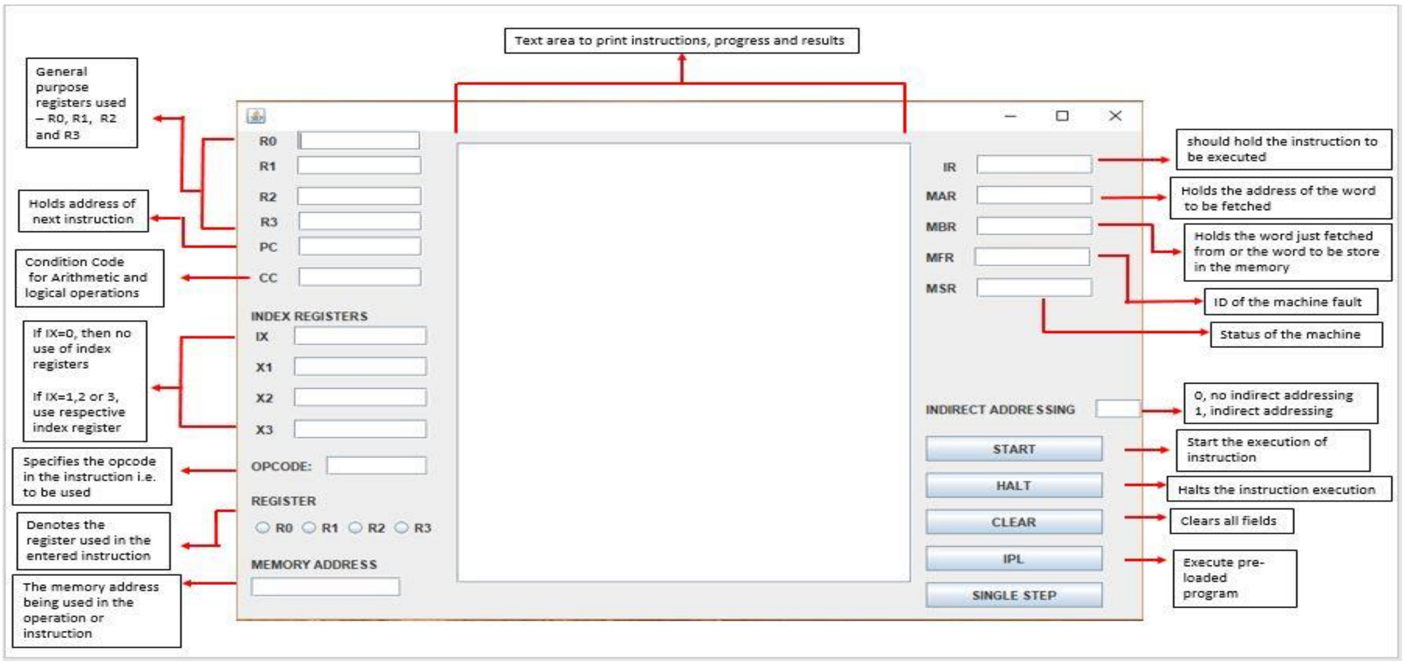
The 5 Load and Store instructions are as follows:

OpCode_8	Instruction	Description

01	LDR r, x, address[,I]	Load Register From Memory, r = 0..3
		r <- c(EA)
		r <- c(c(EA)), if I bit set

02	STR r, x, address[,I]	Store Register To Memory, r = 0..3 Memory(EA) <- c(r)
03	LDA r, x, address[,I]	Load Register with Address, r = 0..3 r <- EA
41	LDX x, address[,I]	Load Index Register from Memory, x = 1..3 Xx <- c(EA)
42	STX x, address[,I]	Store Index Register to Memory. X = 1..3 Memory(EA) <- c(Xx)

The GUI should be designed as follows:


Text area to print instructions, progress and results
General purpose registers used – R0, R1, R2 and R3
Holds address of next instruction
Condition Code for Arithmetic and logical operations
If IX=0, then no use of index registers
If IX=1,2 or 3, use respective index register
Specifies the opcode in the instruction i.e. to be used
Denotes the register used in the entered instruction
The memory address being used in the operation or instruction
R0
R1
R2
R3
PC
CC
INDEX REGISTERS
IX
X1
X2
X3
OPCODE:
REGISTER
R0 R1 R2 R3
MEMORY ADDRESS
IR
MAR
MBR
MFR
MSR
INDIRECT ADDRESSING
START
HALT
CLEAR
IPL
SINGLE STEP
should hold the instruction to be executed
Holds the address of the word to be fetched
Holds the word just fetched from or the word to be store in the memory
ID of the machine fault
Status of the machine
0, no indirect addressing
1, indirect addressing
Start the execution of instruction
Halts the instruction execution
Clears all fields
Execute pre-loaded program

CS6461 - COMPUTER ARCHITECTURE
PROJECT DESIGN NOTES - GROUP 5

The possible load and store instructions:

Opcode	**R**	**IX**	**I**	**Address**	
01		00	0(no indirect addressing)		Contents of Address field
01		X1/X2/X3(index ing) Index Register	0		C(Xi)+Contents of Address field
01		00	1(indirect addressing)		C(address)
01		01/02/03 (X1/X2/X3)	1(indirect addressing)		C(C(Xi)+address)
Load register from memory	R<-c(EA)	LDR r, x, address[,I]			
02		00	0		Contents of Address field
02		01/02/03 (X1/X2/X3)	0		C(Xi)+Contents of Address field
02		00	1		C(address)
02		01/02/03 (X1/X2/X3)	1		C(C(Xi)+ Contents of Address field))
Store register to memory	c(R)->Mem(EA)	STR r, x, address[,I]			
03		00	0		Contents of Address field
03		01/02/03 (X1/X2/X3)	0		C(Xi)+Contents of Address field
03		00	1		C(address)
03		01/02/03 (X1/X2/X3)	1		C(C(Xi)+address)
Load register with address	R<-EA	LDA r, x, address[,I]			
41		00	0		Contents of Address field
41		01/02/03	0		C(Xi)+Contents of Address field
41		00	1		C(address)
41		01/02/03	1		C(C(Xi)+address)
Load index register from memory	Xx<-c(EA)	LDX x, address[,I]			
42		00	0 (no indirect addressing)		Contents of address field
42		01/02/03 (X1/X2/X3 index register)	0		C(Xi)+content of address field
42		00	1(indirect addressing)		C(address)
42		01/02/03 (X1/X2/X3 index register)	1		C(C(Xi)+address)
Store index register to memory	c(Xi)->Mem(EA)	STX x, address[,I]			

CS6461 - COMPUTER ARCHITECTURE
PROJECT DESIGN NOTES - GROUP 5

PHASE 2 IMPLEMENTATION

Cache design and implementation:

The design for cache and cache line is as follows:

For example, if EA=(1000)decimal

1. Determine which block EA is in.
   1000/6=166
   166 is the number of block.
   1000%6=4
   4 presents that the word existing in the word[4] in a block array.

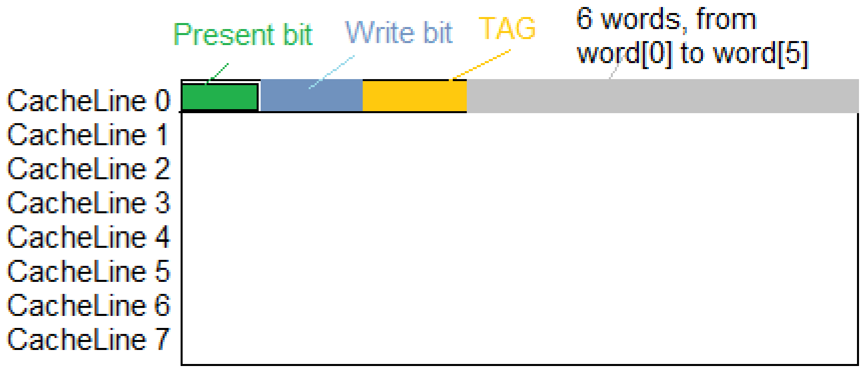
2. In direct-mapping, the address of block determines which cache line the word should be put in.

   (166)decimal=(0000000101000110)binary
   The last 3 bits = the number of cache line the word should be put in.
   The first 13 bits = TAG number.

3. Write bit
   Write bit =0, means no data in cache line has been modified.
   Write bit=1, means some data have been changed.

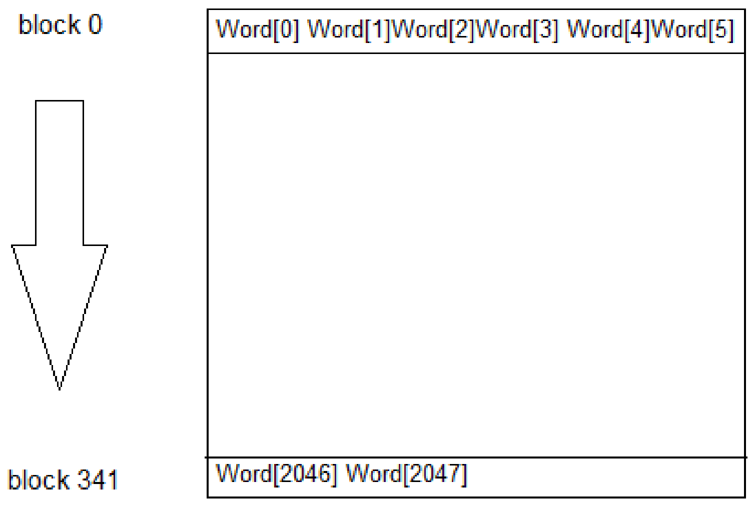
4. Present bit
   Present bit =1, means there are valid data in the line.
   Present bit=0, means there are not valid number in the line.

5. The cache:



Direct mapping

6. The memory:



Memory

The coding for cache class and cache line is included in the source code pdf.

Remaining instructions

The set of instructions included and tested for phase 2 are:

OpCode	Instruction	Description
010	JZ r, x, address[,I]	Jump If Zero: If c(r) = 0, then PC ⍰⍰ EA Else PC <- PC+1
011	JNE r, x, address[,I]	Jump If Not Equal: If c(r) != 0, then PC ⍰⍰- EA Else PC <- PC + 1
012	JCC cc, x, address[,I]	Jump If Condition Code cc replaces r for this instruction cc takes values 0, 1, 2, 3 as above and specifies the bit in the Condition Code Register to check; If cc bit = 1, PC <- EA Else PC <- PC + 1
013	JMA x, address[,I]	Unconditional Jump To Address PC <- EA, Note: r is ignored in this instruction

014	JSR x, address[,I]	Jump and Save Return Address:
		R3 <- PC+1;
		PC <- EA

		R0 should contain pointer to arguments Argument list should end with –1 (all 1s) value
015	RFS Immed	Return From Subroutine w/ return code as Immed portion (optional) stored in the instruction's address field. R0 <- Immed; PC <- c(R3) IX, I fields are ignored.

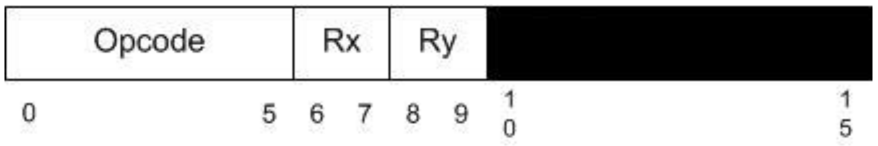
016	SOB r, x, address[,I]	Subtract One and Branch. R = 0..3
		r <-c(r) – 1
		If c(r) > 0, PC <- EA;
		Else PC <- PC + 1

017	JGE r,x, address[,I]	Jump Greater Than or Equal To: If c(r) >= 0, then PC <- EA Else PC <- PC + 1
004	AMR r, x, address[,I]	Add Memory To Register, r = 0..3 r<- c(r) + c(EA)
005	SMR r, x, address[,I]	Subtract Memory From Register, r = 0..3 r<- c(r) – c(EA)
006	AIR r, immed	Add Immediate to Register, r = 0..3 r <- c(r) + Immed Note: 1. if Immed = 0, does nothing 2. if c(r) = 0, loads r with Immed IX and I are ignored in this instruction
007	SIR r, immed	Subtract Immediate from Register, r = 0..3 r <- c(r) - Immed Note: 1. if Immed = 0, does nothing 2. if c(r) = 0, loads r1 with –(Immed) IX and I are ignored in this instruction
020	MLT rx,ry	Multiply Register by Register rx, rx+1 <- c(rx) * c(ry) rx must be 0 or 2 ry must be 0 or 2 rx contains the high order bits, rx+1 contains the low order bits of the result Set OVERFLOW flag, if overflow
021	DVD rx,ry	Divide Register by Register rx, rx+1 <- c(rx)/ c(ry) rx must be 0 or 2 rx contains the quotient; rx+1 contains the remainder ry must be 0 or 2 If c(ry) = 0, set cc(3) to 1 (set DIVZERO flag)
022	TRR rx, ry	Test the Equality of Register and Register

		If c(rx) = c(ry), set cc(4) <- 1; else, cc(4) <-0
023	AND rx, ry	Logical And of Register and Register c(rx) <-c(rx) AND c(ry)
024	ORR rx, ry	Logical Or of Register and Register c(rx) <- c(rx) OR c(ry)
025	NOT rx	Logical Not of Register To Register C(rx) <-NOT c(rx)
031	SRC r, count, L/R, A/L	Shift Register by Count c(r) is shifted left (L/R =1) or right (L/R = 0) either logically (A/L = 1) or arithmetically (A/L = 0) XX, XXX are ignored Count = 0…15 If Count = 0, no shift occurs

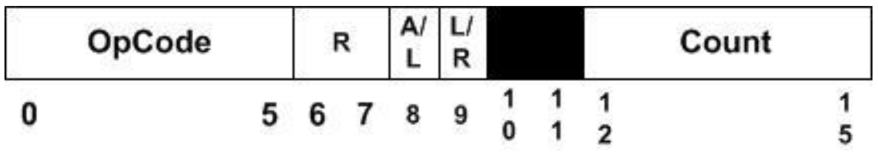
032	RRC r, count, L/R, A/L	Rotate Register by Count
		c(r) is rotated left (L/R = 1) or right (L/R =0) either logically (A/L =1)
		Count = 0…15
		If Count = 0, no rotate occurs

The instruction format for arithmetic and logical operations is different:



All the instructions are based on registers Rx and Ry. The lower order and higher order bits of the results are also stored in these registers.

The instruction format for shifting and rotating operations is as follows:



In this, count represents the extent to which the value should be shifted or rotated. A/L specifies whether it is arithmetic or logical shift/rotate and L/R specifies the direction i.e. left or right.

Program 1 implementation

Check the attracted text file “instruction.txt” for the machine code for executing Program 1

In the machine code, we have used instructions such as LDR, STR, SMR, ABS and CMP. These instructions call the functions relevant to the respective opcode.

Such as: when the instruction begins with “LDR”, the function “loadregister()” is called. Similarly the respective functions get called.

The above programs functionality is to get the 20 input numbers along with the search number and store it in continuous blocks of memory.

After storing the numbers, we find the difference between the search number and each input number to find the minimum difference.

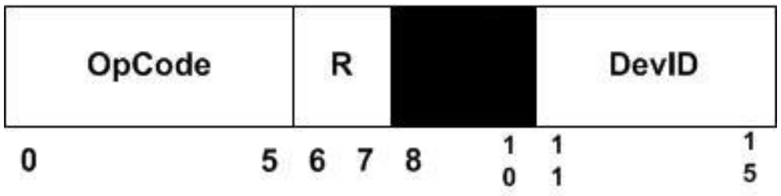
Finally, we compare the results and print the number closest to the search number.

The java code for Program 1 is included in the source code PDF.

PHASE 3 IMPLEMENTATION

I/O OPERATION

The instruction format for I/O operations is as follows:



The assigned DEVID in the simulator are:
0: Console keyboard
1: console printer
2: card reader
3 – 31: The registers and switches in the console

The I/O instructions are:

OpCode	Instruction	Description
061	IN r, devid	Input Character To Register from Device, r = 0..3
062	OUT r, devid	Output Character to Device from Register, r = 0..3
063	CHK r, devid	Check Device Status to Register, r = 0..3 c(r) <- device status

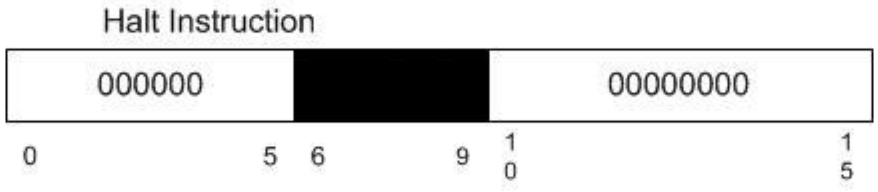
MACHINE FAULT

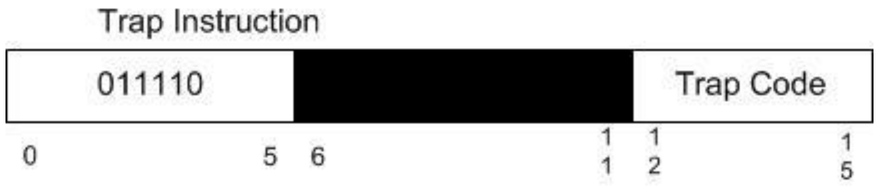
The possible machine faults that are predefined are:

ID	Fault
0	Illegal Memory Address to Reserved Locations
1	Illegal TRAP code
2	Illegal Operation Code
3	Illegal Memory Address beyond 2048 (memory installed)

So, whenever any of these faults occur, the MFR will be set to the respective fault ID.

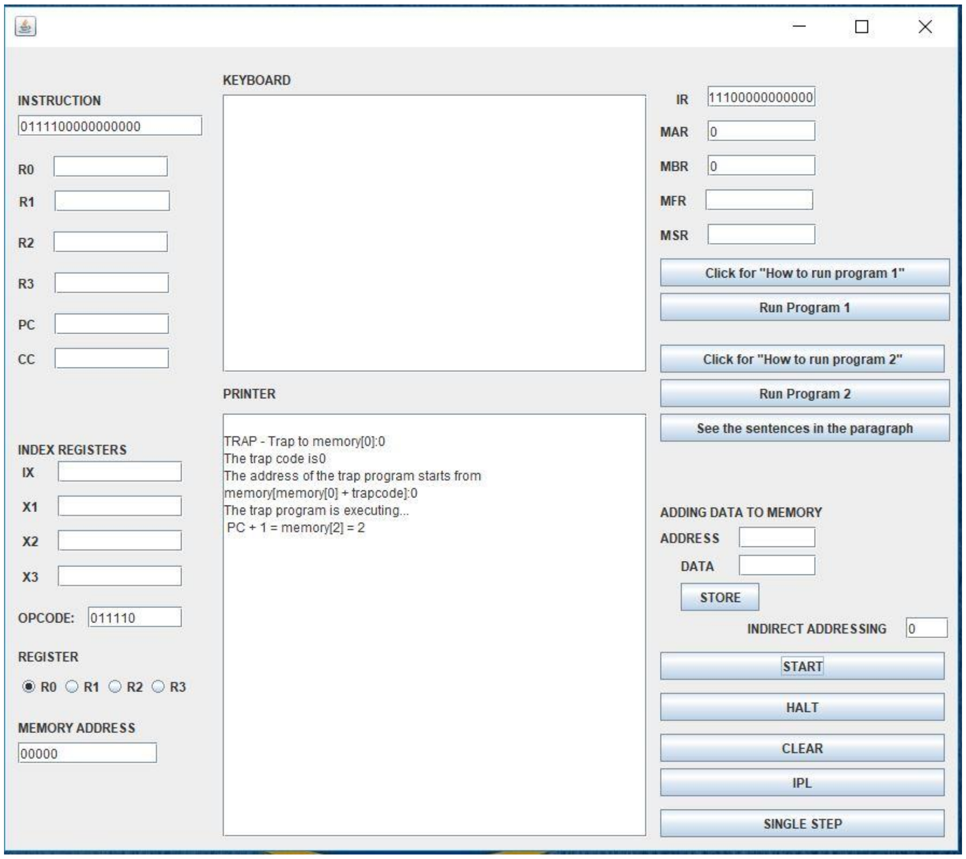
When a Trap instruction or a machine fault occurs, the processor saves the current PC and MSR contents to the locations specified above, then fetches the address from Location 0 (Trap) or 1 (Machine Fault) into the PC which becomes the next instruction to be executed.





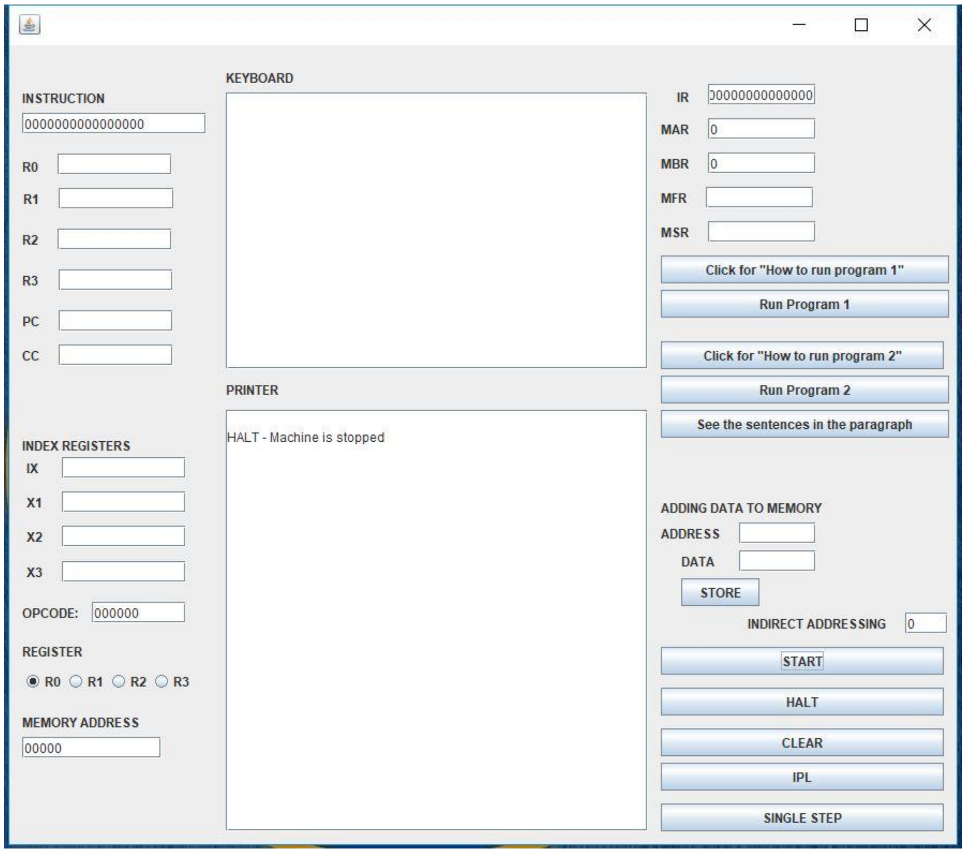
OpCode$_8$	**Instruction**	**Description**
000	HLT	Stops the machine.
036	TRAP code	Traps to memory address 0, which contains the address of a table in memory. Stores the PC+1 in memory location 2. The table can have a maximum of 16 entries representing 16 routines for user-specified instructions stored elsewhere in memory. Trap code contains an index into the table, e.g. it takes values 0 – 15. When a TRAP instruction is executed, it goes to the routine whose address is in memory location 0, executes those instructions, and returns to the instruction stored in memory location 2. The PC+1 of the TRAP instruction is stored in memory location 2.

CS6461 - COMPUTER ARCHITECTURE
PROJECT DESIGN NOTES - GROUP 5


KEYBOARD
PRINTER
Click for "How to run program 1"
Run Program 1
Click for "How to run program 2"
Run Program 2
See the sentences in the paragraph
ADDING DATA TO MEMORY
STORE
START
HALT
CLEAR
IPL
SINGLE STEP

INSTRUCTION 0111100000000000
R0
R1
R2
R3
PC
CC
INDEX REGISTERS
IX
X1
X2
X3
OPCODE: 011110
REGISTER
[✓ R0] R1 R2 R3
MEMORY ADDRESS 00000
IR 1110000000000
MAR 0
MBR 0
MFR
MSR
ADDRESS
DATA
INDIRECT ADDRESSING 0

```
TRAP - Trap to memory[0]:0
The trap code is0
The address of the trap program starts from
memory[memory[0] + trapcode]:0
The trap program is executing...
 PC + 1 = memory[2] = 2
```


KEYBOARD
PRINTER
Click for "How to run program 1"
Run Program 1
Click for "How to run program 2"
Run Program 2
See the sentences in the paragraph
ADDING DATA TO MEMORY
STORE
START
HALT
CLEAR
IPL
SINGLE STEP

INSTRUCTION 0000000000000000
R0
R1
R2
R3
PC
CC
INDEX REGISTERS
IX
X1
X2
X3
OPCODE: 000000
REGISTER
[✓ R0] R1 R2 R3
MEMORY ADDRESS 00000
IR 0000000000000
MAR 0
MBR 0
MFR
MSR
ADDRESS
DATA
INDIRECT ADDRESSING 0

```
HALT - Machine is stopped
```

PROGRAM 2 IMPLEMENTATION

Program 2 is a program that reads a set of a paragraph of 6 sentences from a file into memory. It prints the sentences on the console printer. It then asks the user for a word. It searches the paragraph to see if it contains the word. If so, it prints out the word, the sentence number, and the word number in the sentence

Register designations

// R0 - stores the current index which points to the reading character in the paragraph.

// R1 - stores the current index which points to the reading character in the searching word.

// R2 - Holds temporary value

// R3 -

Memory designations

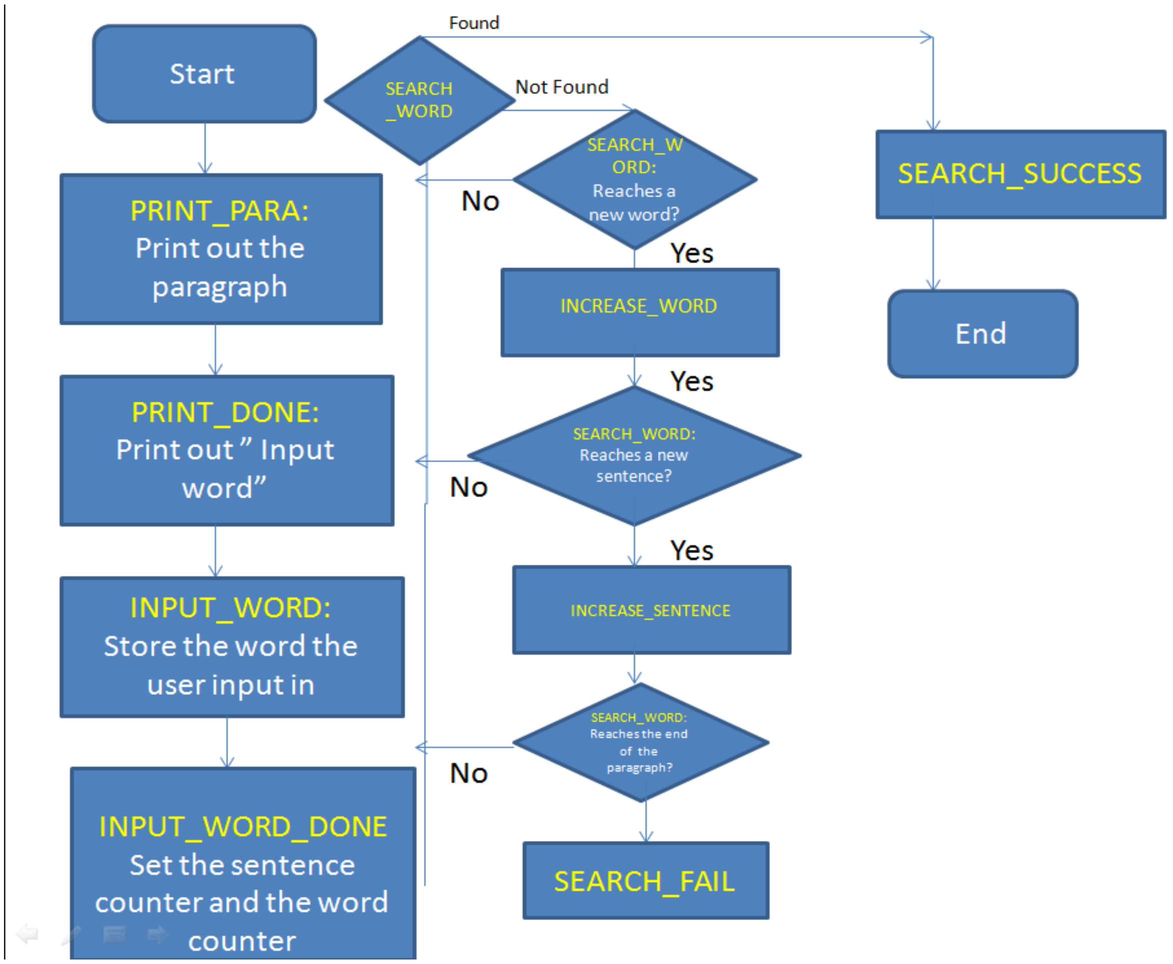
// $6 - Holds temporary value

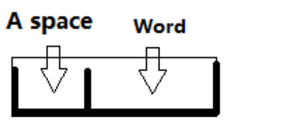
// $7 - Holds the base address of the paragraph ($1500), memory[7]=1500;

// $8 - Holds the base address of the searching word, memory[8]=300;

// $9 - sentence counter

// $10 - word counter


Start
PRINT_PARA: Print out the paragraph
PRINT_DONE: Print out " Input word"
INPUT_WORD: Store the word the user input in
INPUT_WORD_DONE Set the sentence counter and the word counter
SEARCH _WORD
Found
Not Found
SEARCH_WORD: Reaches a new word?
No
Yes
INCREASE_WORD
Yes
SEARCH_WORD: Reaches a new sentence?
No
Yes
INCREASE_SENTENCE
SEARCH_WORD: Reaches the end of the paragraph?
No
SEARCH_FAIL
SEARCH_SUCCESS
End



The word stored in memory.

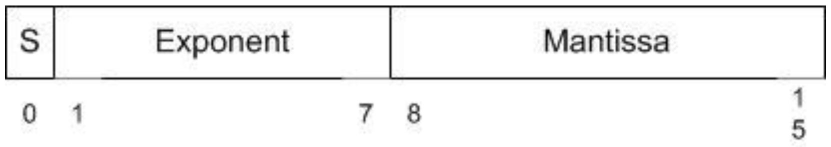
Case 1:	**Word in the paragraph**	**[a space]amaranth** ✗	**If the word does not end with a space, comma or period, the two words don't match.**
	Searching word	**[a space]am**	
Case 2:	**Word in the paragraph**	**[a comma]** **[a space]am[a space]** ✓ **[a period]**	
	Searching word	**[a space]am**	

PHASE 4 IMPLEMENTATION

FLOATING POINT INSTRUCTIONS

In order to implement floating point instructions, we need two float registers FR0 and FR1. The register field in the 16-bit instruction denotes which floating register to use.

The content of float registers is binary value based on the below format:



In the 16-bit binary,

First bit denotes the sign of float value, second 8 bits denotes the exponent value and last 8 bits denotes the mantissa. Using sign, exponent and mantissa the actual float value can be evaluated.

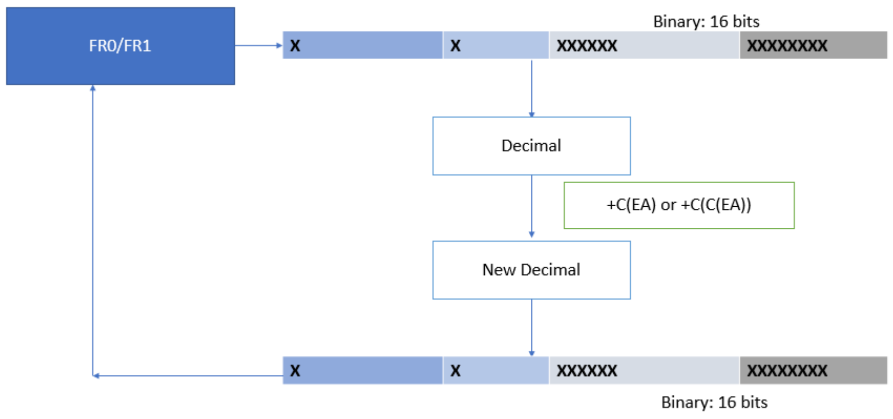
(1) The S bit (bit 0) is the sign of the entire floating-point number.
(2) There are 7 bits for the exponent, and the first bit of the exponent is its sign bit. The exponent ranges from –63 to 64, e.g., -2^6-1 to 2^6. We are using the complementary code to represent the exponent.
    If the value of the exponent number is positive, $X_{com} = (X)_2$;
    If the value of the exponent number is negative, $X_{com} = (X+2\^7)_2$;
(3) There 8 bits for the mantissa. For example, 10XXXXXX means 1.0XXXXXX; 01XXXXXX means 0.1XXXXXX.
(4) The decimal number of the representation Deci = Mantissa * 2^(Exponent)

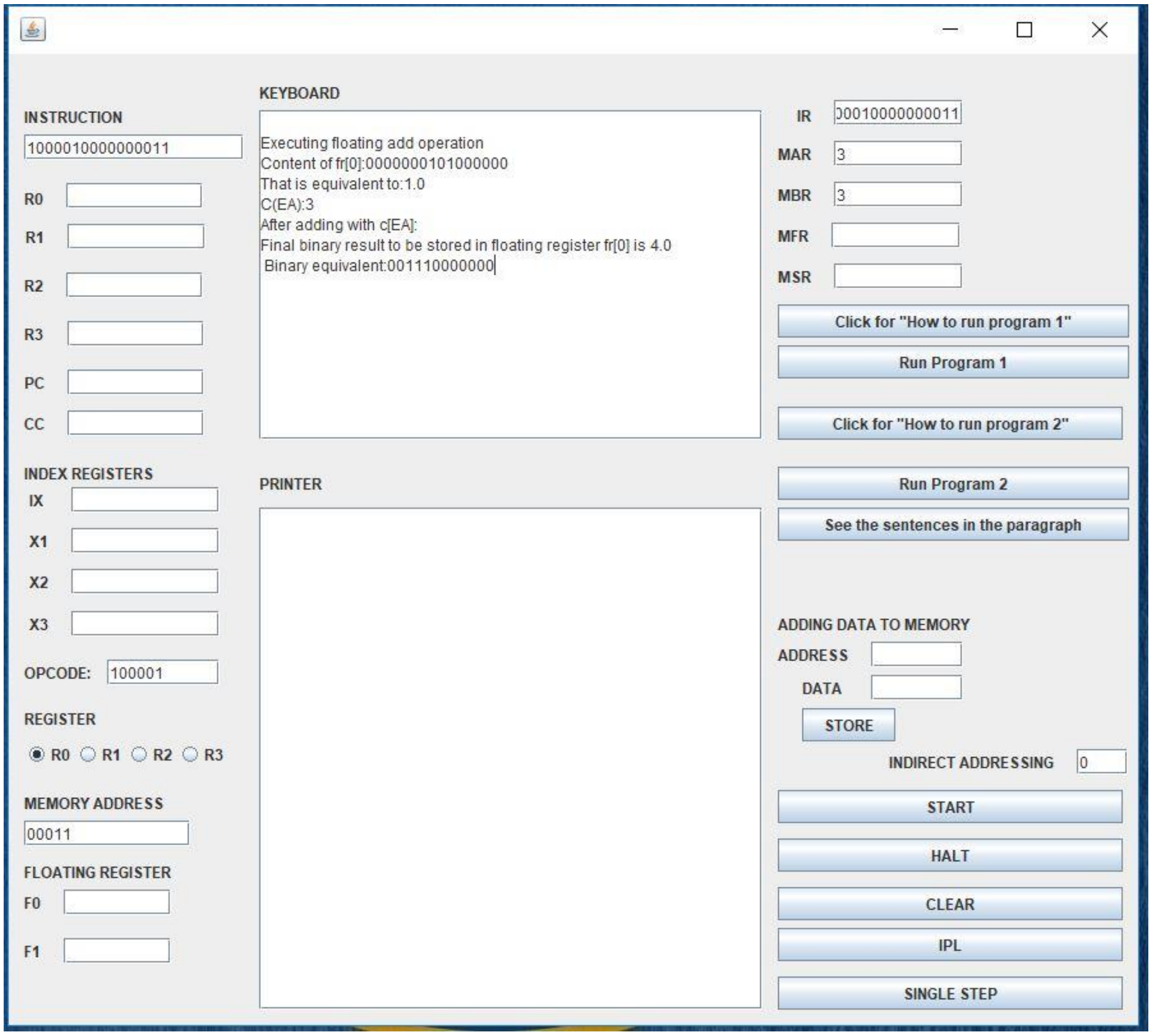
There are seven floating point instructions implemented in the simulator, as follows:

1. **FADD:**

OpCode	Instruction	Description
033	FADD fr, x, address[,I]	Floating Add Memory To Register c(fr) <– c(fr) + c(EA) c(fr) <– c(fr) + c(c(EA)), if I bit set fr must be 0 or 1. OVERFLOW may be set

FADD





INSTRUCTION: 1000010000000011

OPCODE: 100001

REGISTER: [✓ R0] R1 R2 R3

MEMORY ADDRESS: 00011

IR: 0001000000011

MAR: 3

MBR: 3

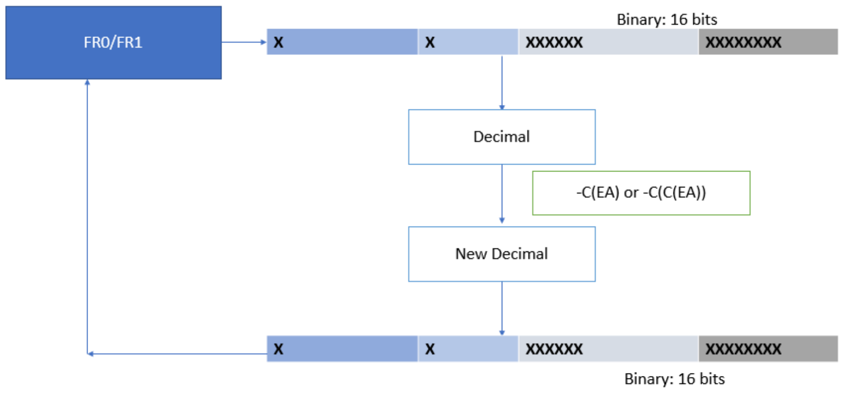
INDIRECT ADDRESSING: 0

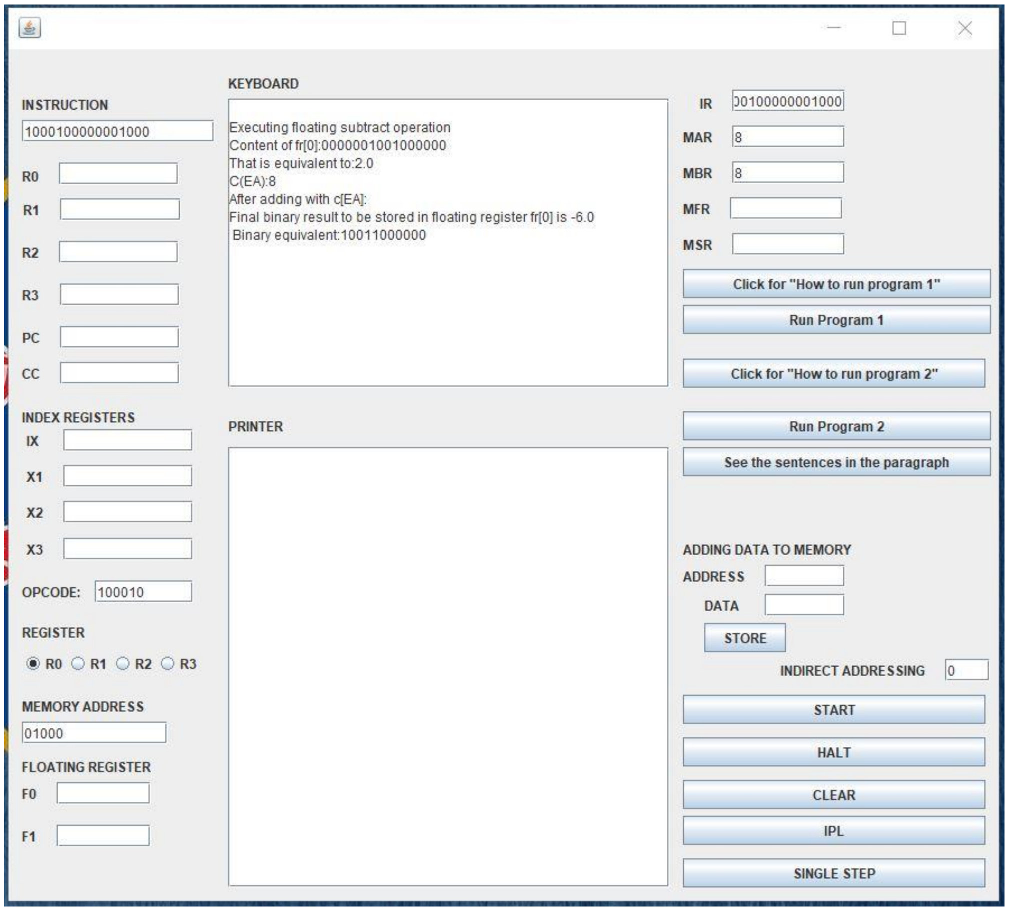
```
Executing floating add operation
Content of fr[0]:0000000101000000
That is equivalent to:1.0
C(EA):3
After adding with c[EA]:
Final binary result to be stored in floating register fr[0] is 4.0
 Binary equivalent:001110000000
```

2. FSUB:

c	Instruction	Description
034	FSUB fr, x, address[,I]	Floating Subtract Memory From Register c(fr) <– c(fr) - c(EA) c(fr) <– c(fr) - c(c(EA)), if I bit set fr must be 0 or 1 UNDERFLOW may be set

FSUB





INSTRUCTION: 1000100000001000

IR: 0100000001000

MAR: 8

MBR: 8

OPCODE: 100010

REGISTER: [✓ R0] R1 R2 R3

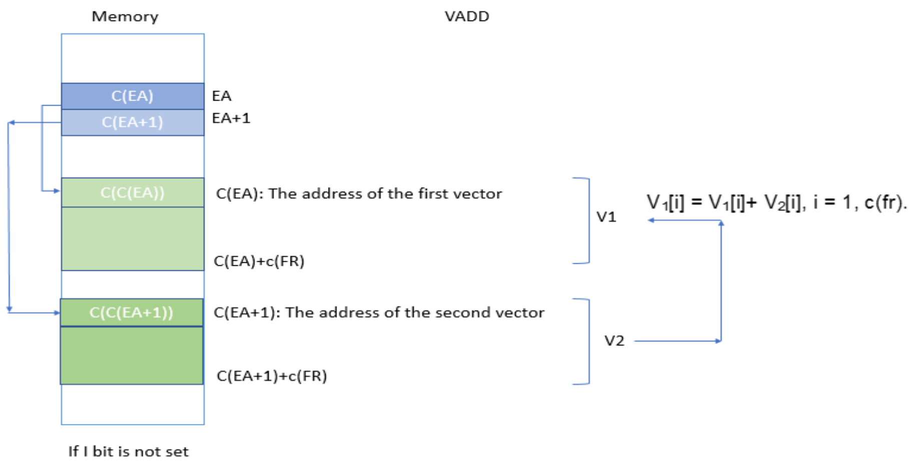
MEMORY ADDRESS: 01000

INDIRECT ADDRESSING: 0

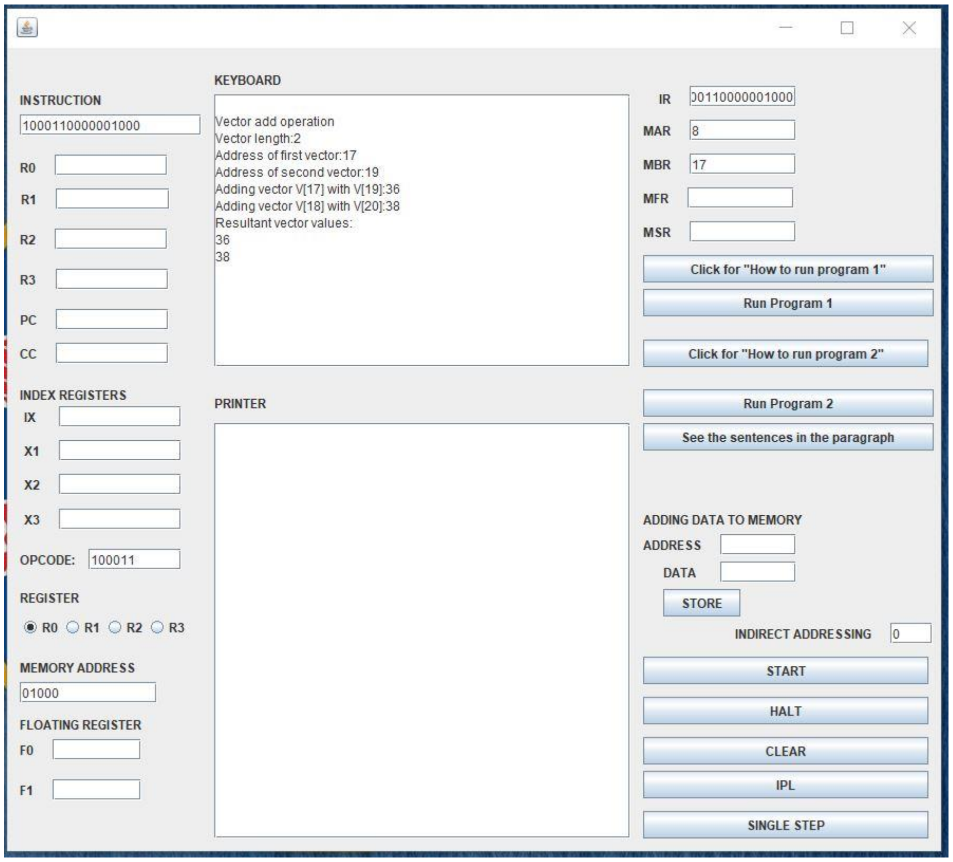
```
Executing floating subtract operation
Content of fr[0]:0000001001000000
That is equivalent to:2.0
C(EA):8
After adding with c[EA]:
Final binary result to be stored in floating register fr[0] is -6.0
 Binary equivalent:10011000000
```

3. VADD

035	VADD fr, x, address[,I]	Vector Add fr contains the length of the vectors c(EA) or c(c(EA)), if I bit set, is address of first vector c(EA+1) or c(c(EA+1)), if I bit set, is address of the second vector. Let V_1 be vector at address; Let V_2 be vector at address+1 Then, $V_1[i] = V_1[i] + V_2[i]$, i = 1, c(fr).


Memory
VADD
C(EA)
C(EA+1)
EA
EA+1
C(C(EA))
C(EA): The address of the first vector
C(EA)+c(FR)
V1
$V_1[i] = V_1[i] + V_2[i]$, i = 1, c(fr).
C(C(EA+1))
C(EA+1): The address of the second vector
C(EA+1)+c(FR)
V2
If I bit is not set

The above diagram shows the implementation of VADD in the simulator.



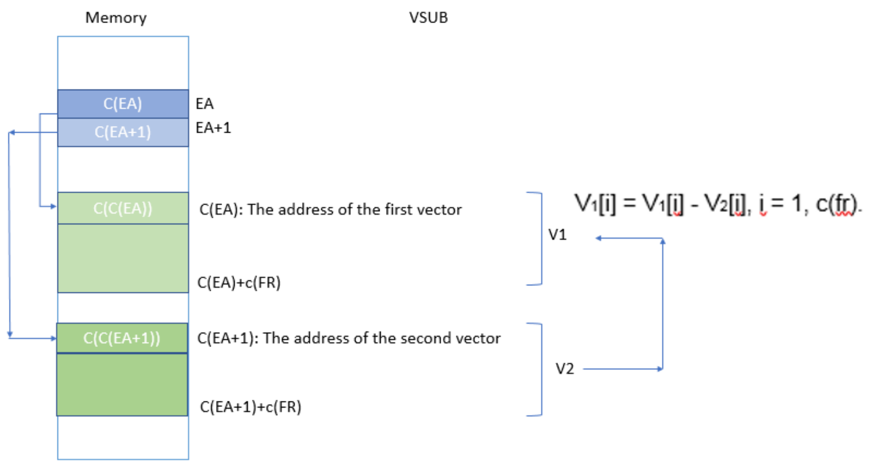
INSTRUCTION: 1000110000001000
KEYBOARD:

```
Vector add operation
Vector length:2
Address of first vector:17
Address of second vector:19
Adding vector V[17] with V[19]:36
Adding vector V[18] with V[20]:38
Resultant vector values:
36
38
```

IR: 0011000000001000
MAR: 8
MBR: 17
OPCODE: 100011
REGISTER: [✓ R0] R1 R2 R3
MEMORY ADDRESS: 01000
INDIRECT ADDRESSING: 0

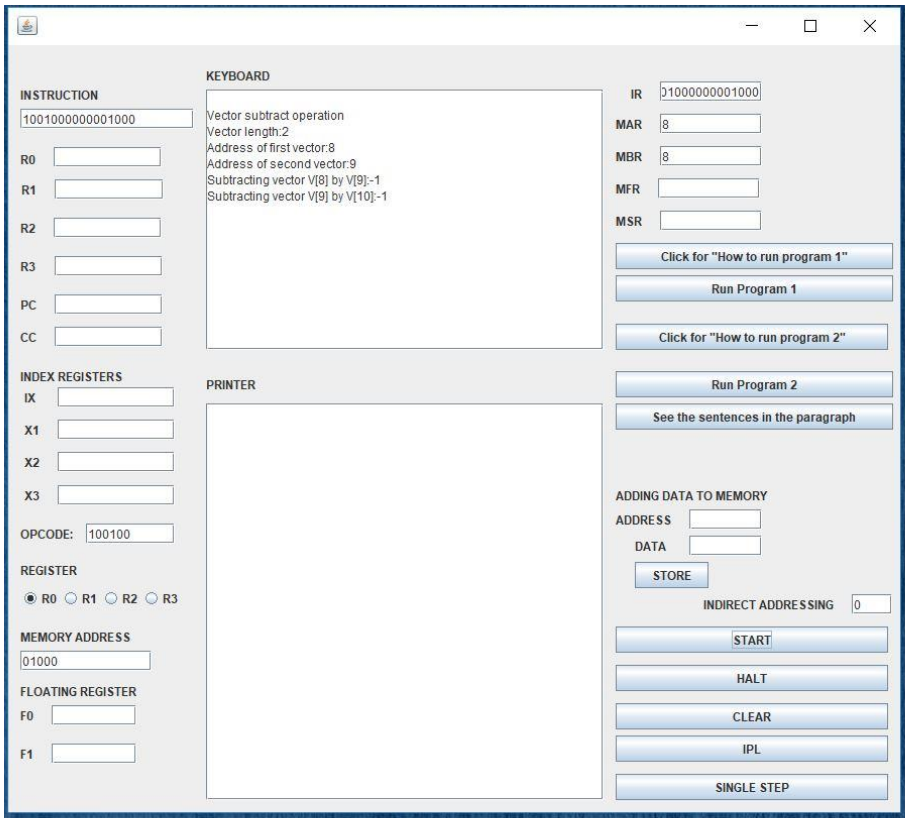
4. VSUB

036	VSUB fr, x, address[,I]	Vector Subtract fr contains the length of the vectors c(EA) or c(c(EA)), if I bit set is address of first vector c(EA+1) or c(c(EA+1)), if I bit set is address of the second vector Let V_1 be vector at address; Let V_2 be vector at address+1 Then, $V_1[i] = V_1[i] - V_2[i]$, i = 1, c(fr).


Memory
VSUB
C(EA)
EA
C(EA+1)
EA+1
C(C(EA))
C(EA): The address of the first vector
C(EA)+c(FR)
C(C(EA+1))
C(EA+1): The address of the second vector
C(EA+1)+c(FR)
V1
V2
V₁[i] = V₁[i] - V₂[i], i = 1, c(fr).

If I bit is not set

The above diagram sows the implementation of VSUB in the simulator.


KEYBOARD
PRINTER
Click for "How to run program 1"
Run Program 1
Click for "How to run program 2"
Run Program 2
See the sentences in the paragraph
ADDING DATA TO MEMORY
STORE
START
HALT
CLEAR
IPL
SINGLE STEP

INSTRUCTION: 1001000000001000
R0:
R1:
R2:
R3:
PC:
CC:
INDEX REGISTERS
IX:
X1:
X2:
X3:
OPCODE: 100100
REGISTER: (● R0) ○ R1 ○ R2 ○ R3
MEMORY ADDRESS: 01000
FLOATING REGISTER
F0:
F1:

```
Vector subtract operation
Vector length:2
Address of first vector:8
Address of second vector:9
Subtracting vector V[8] by V[9]:-1
Subtracting vector V[9] by V[10]:-1
```

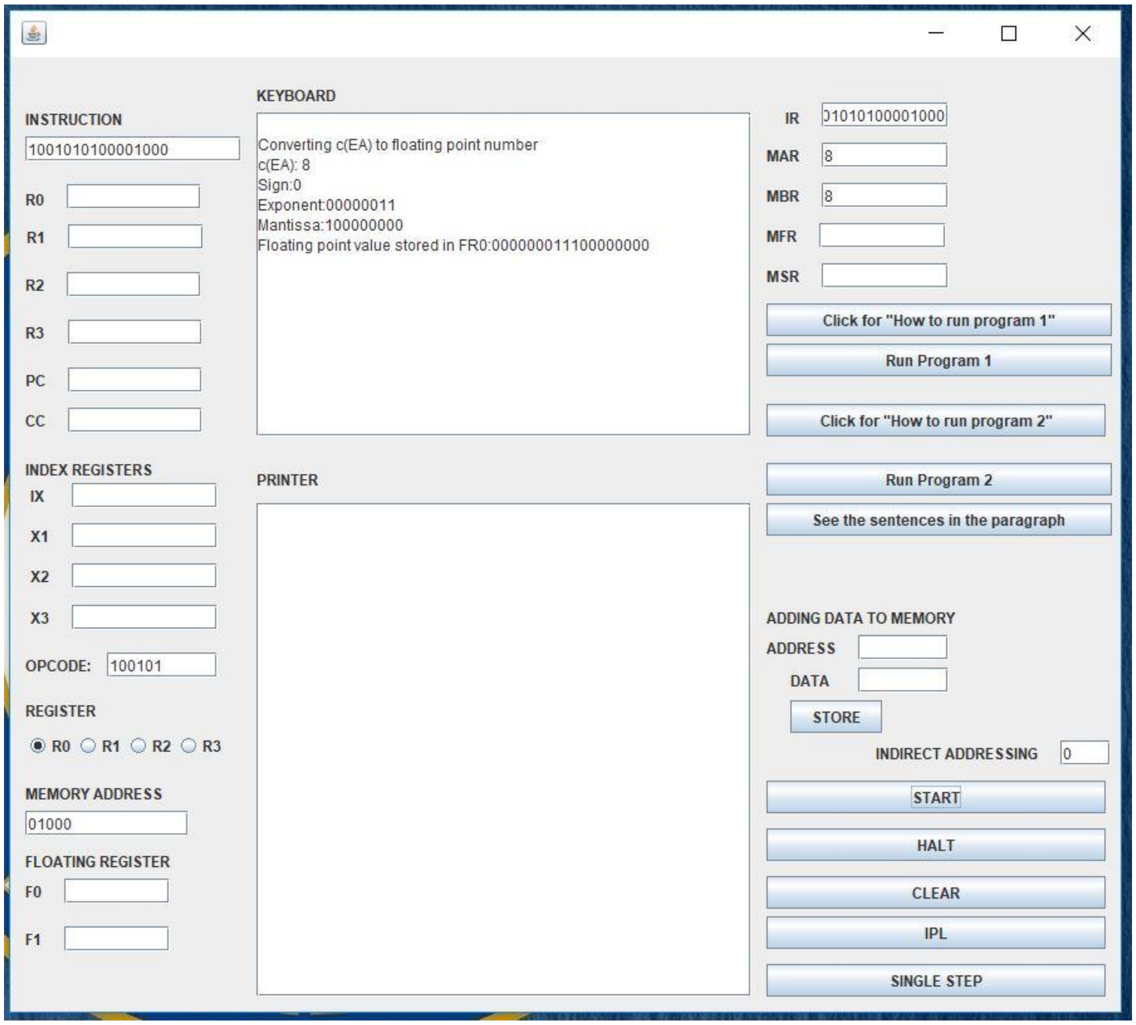
IR: 01000000001000
MAR: 8
MBR: 8
MFR:
MSR:
ADDRESS:
DATA:
INDIRECT ADDRESSING: 0

5. CNVRT

037	CNVRT r, x, address[,I]	Convert to Fixed/FloatingPoint: If F = 0, convert c(EA) to a fixed point number and store in r. If F = 1, convert c(EA) to a floating point number and store in FR0. The r register contains the value of F before the instruction is executed.

This instruction is used to convert the value found at the EA to either a fixed-point number (when F=0) or a floating-point number (when F=1).

When F=1, the value found at c(EA) is converted to its binary equivalent in order to evaluate sign, exponent and mantissa. Using these values, the floating-point value can be calculated. This result will be stored in the floating register.



INSTRUCTION: 1001010100001000

IR: 01010100001000

MAR: 8

MBR: 8

OPCODE: 100101

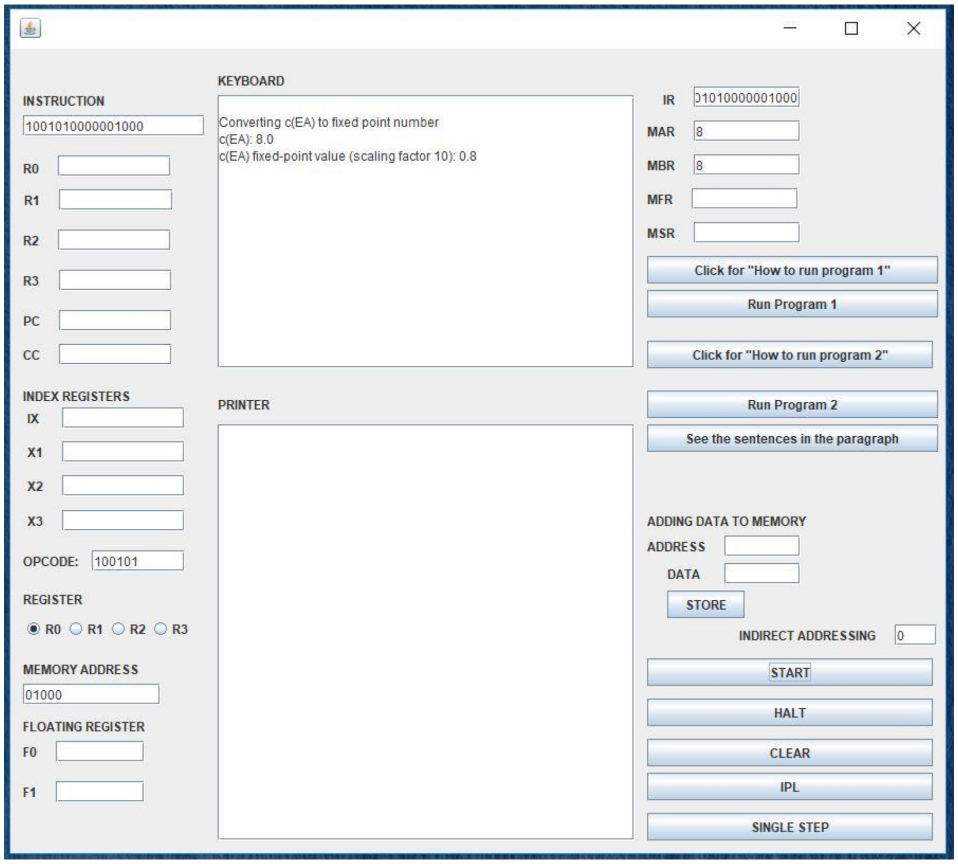
REGISTER: [● R0] ○ R1 ○ R2 ○ R3

MEMORY ADDRESS: 01000

INDIRECT ADDRESSING: 0

```
Converting c(EA) to floating point number
c(EA): 8
Sign:0
Exponent:00000011
Mantissa:100000000
Floating point value stored in FR0:000000011100000000
```

When F=0, the value found at c(EA) is converted to a fixed-point number of the required base and stored in the register r.



INSTRUCTION: 1001010000001000

KEYBOARD:

```
Converting c(EA) to fixed point number
c(EA): 8.0
c(EA) fixed-point value (scaling factor 10): 0.8
```

IR: 01010000001000
MAR: 8
MBR: 8

OPCODE: 100101

REGISTER: [✓ R0] R1 R2 R3

MEMORY ADDRESS: 01000

INDIRECT ADDRESSING: 0

6. LDFR

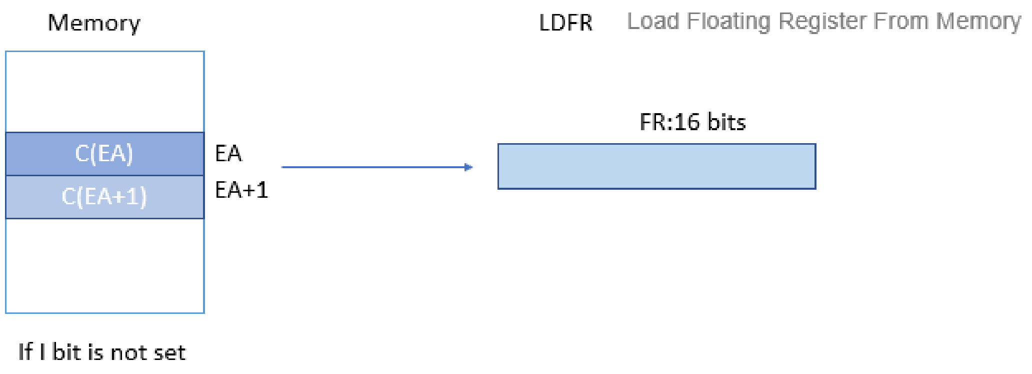
50	LDFR fr, x, address [,i]	Load Floating Register From Memory, fr = 0..1 fr <– c(EA), c(EA+1) fr <- c(c(EA), c(EA)+1), if I bit set

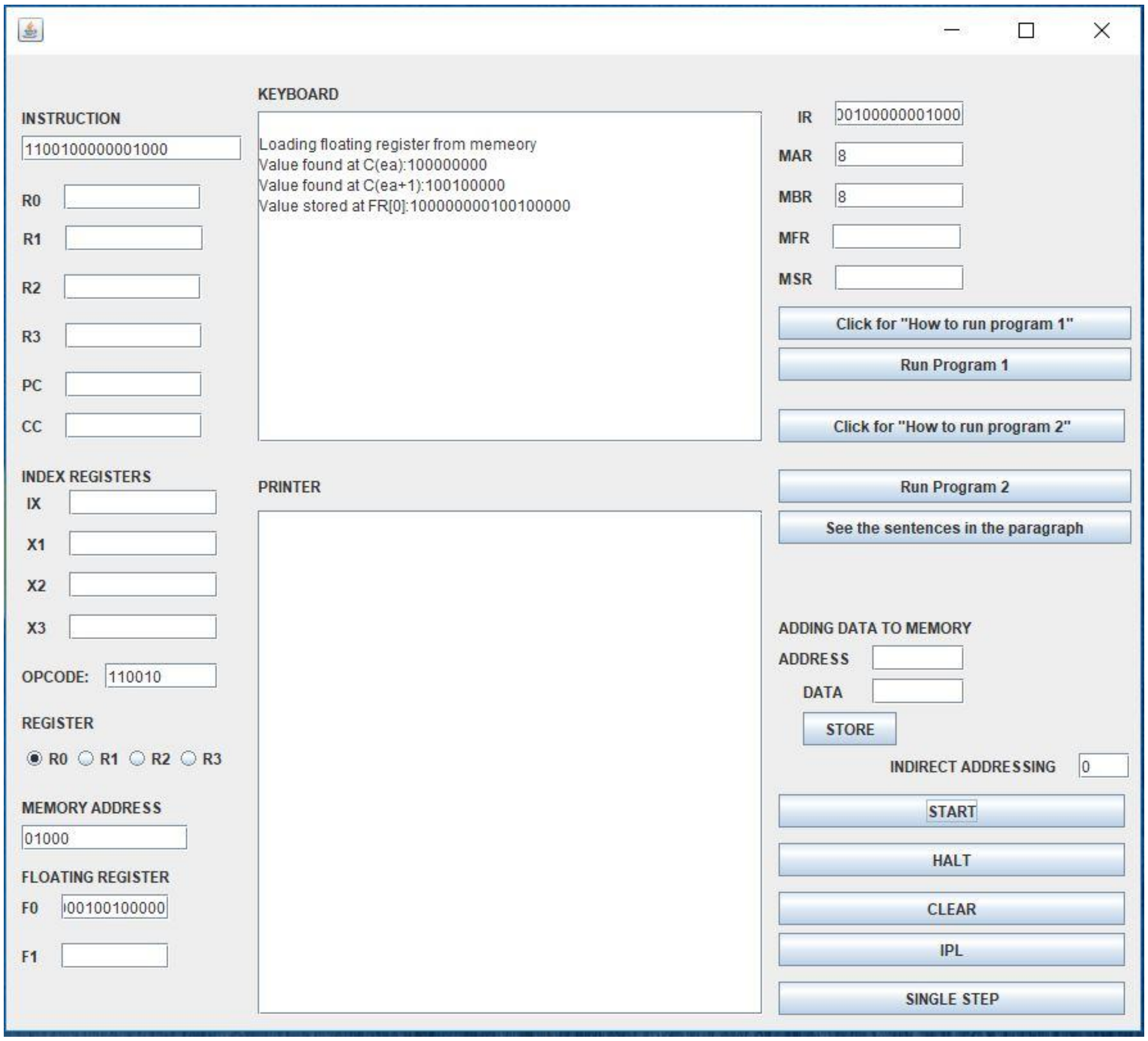
In this instruction, the content of memory at EA and EA+1 are merged and stored in FR. Since FR is 16-bit long and can store the combined values.

Similarly, when I value is set i.e. I=1 the content of memory at c(EA) and c(EA+1) are merged and stored in FR.

The R field in the instruction denotes whether to use FR0 and FR1.

CS6461 - COMPUTER ARCHITECTURE
PROJECT DESIGN NOTES - GROUP 5





INSTRUCTION: 1100100000001000
R0:
R1:
R2:
R3:
PC:
CC:
IX:
X1:
X2:
X3:
OPCODE: 110010
REGISTER: [✓ R0] R1 R2 R3
MEMORY ADDRESS: 01000
FLOATING REGISTER
F0: 00100100000
F1:

```
Loading floating register from memeory
Value found at C(ea):100000000
Value found at C(ea+1):100100000
Value stored at FR[0]:1000000000100100000
```

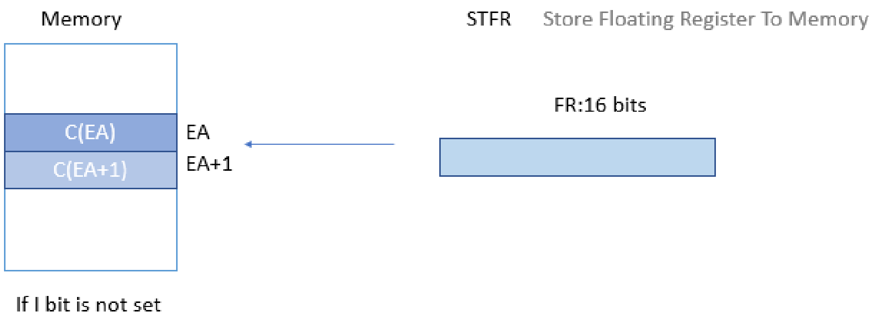
IR: 00100000001000
MAR: 8
MBR: 8
MFR:
MSR:
ADDRESS:
DATA:
INDIRECT ADDRESSING: 0

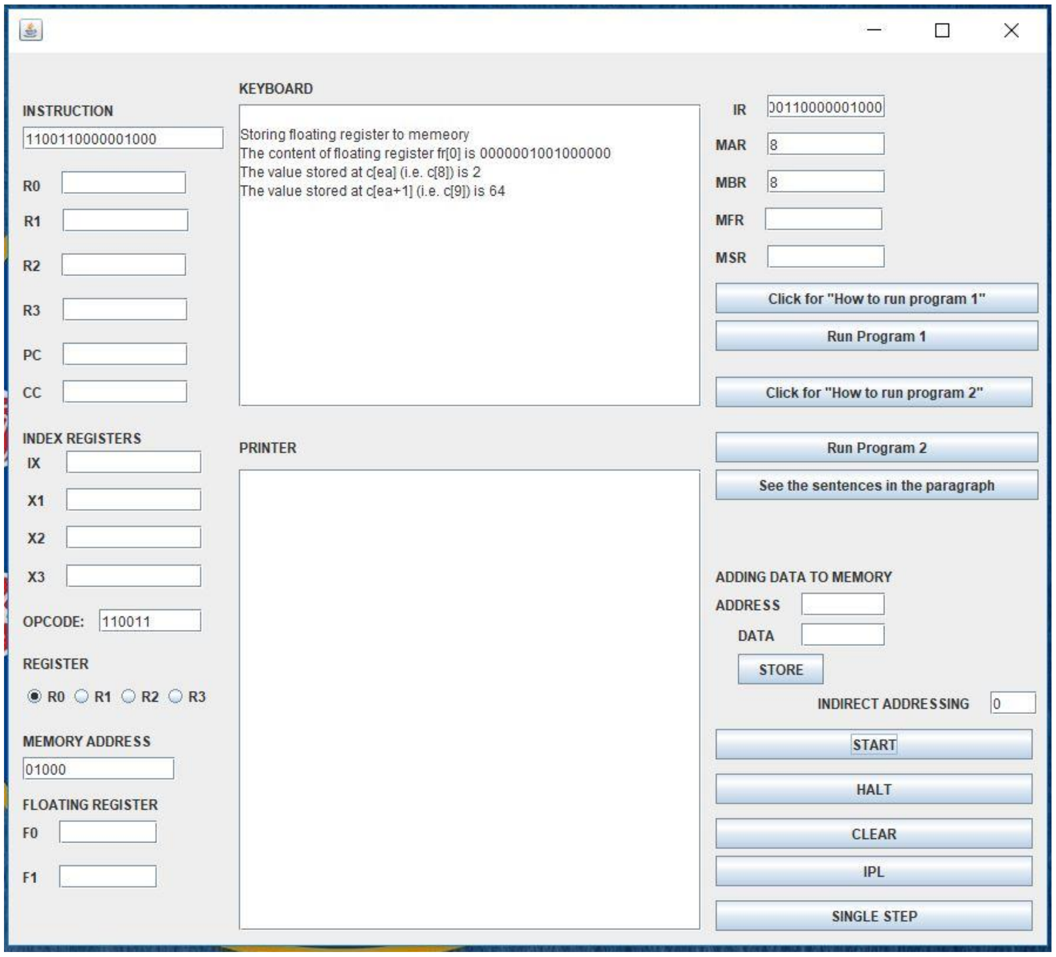
7. STFR

51	STFR fr, x, address [,i]	Store Floating Register To Memory, fr = 0..1 EA, EA+1 <– c(fr) c(EA), c(EA)+1 <- c(fr), if I-bit set

In this instruction, the content of float register is stored into the memory address specified in the instruction. The I-bit denotes the location, i.e. if I=0 then, content of float register is stored in location EA and EA+1.

If I=1 then, content of the float register is stored in the address found at c(EA) and c(EA+1).





INSTRUCTION: 1100110000001000

OPCODE: 110011

MEMORY ADDRESS: 01000

IR: 0011000000001000

MAR: 8

MBR: 8

INDIRECT ADDRESSING: 0

```
Storing floating register to memeory
The content of floating register fr[0] is 0000001001000000
The value stored at c[ea] (i.e. c[8]) is 2
The value stored at c[ea+1] (i.e. c[9]) is 64
```