# DESIGN NOTES

Boomika Karuppaiah

Sumanth Nandeti

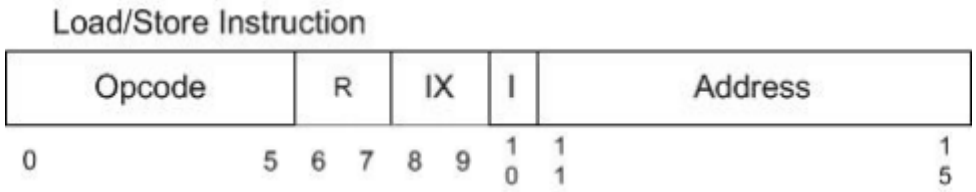
Waad Algorashy

Kamal Subedi

The project's goal is to create an assembly language-based simulator of a modest traditional CISCcomputer. The project's first phase will have the following features:

- 4 General Purpose Registers (GPRs) – each 16 bits in length **[R0, R1, R2, R3]**
- 3 Index Registers – 16 bits in length **[X1, X2, X3]**
- 16-bit words
- Memory of 2048 words, expandable to 4096 words
- Word addressable

**Load/Store Instruction:**




Opcode: 6 bits – Specifies one of 64 possible instructions; Phase 1 will have 5 Load/Store Opcodes.R : 2 bits – R0(00), R1(01), R2(10), R3(11) General Purpose Registers.

IX : 2 bits – X1(01), X2(10), X3(11) Indexed Registers.

I : 1 bit – I=0 specifies indirect addressing, or otherwise no indirect addressing.Address: 5 bits – Specifies one of 32 locations.

Effective address to execute various Load/Store instructions will be computed as follows:Effective Address (EA) =

I = 0:

IX = 00: content (address field)

IX = 01 or 10 or 11: content(IX) + content of address field

I = 1:

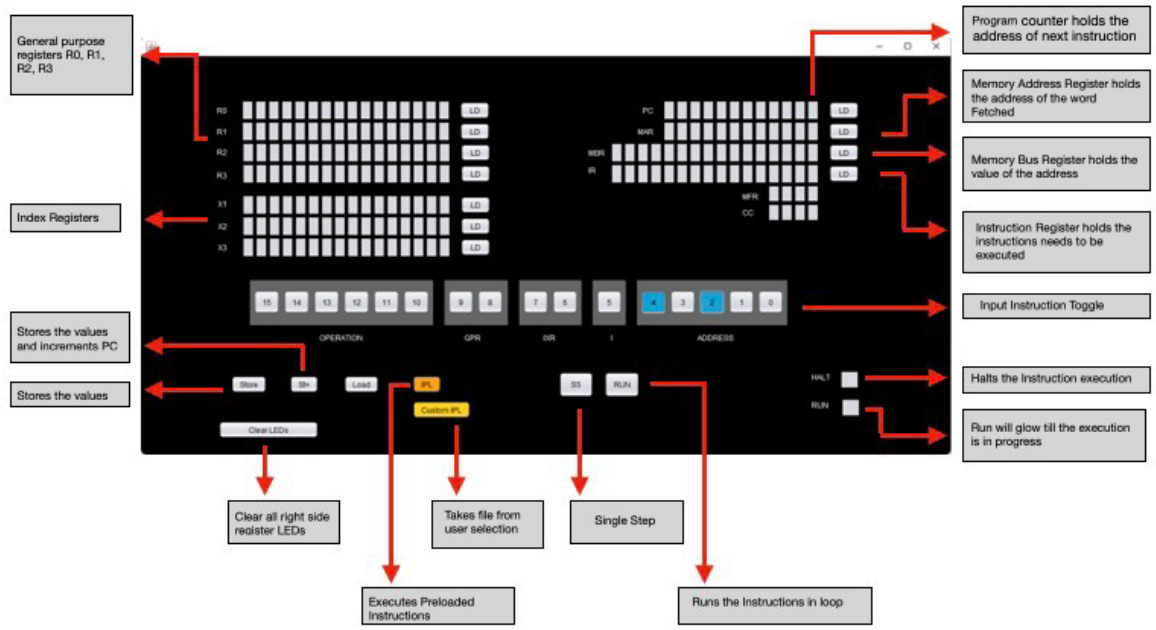
IX = 00: content (content (address field))

IX = 01 or 10 or 11: content(content(IX) + content of address field)

Following are the Load/Store instructions implemented in the Simulator (I is optional):

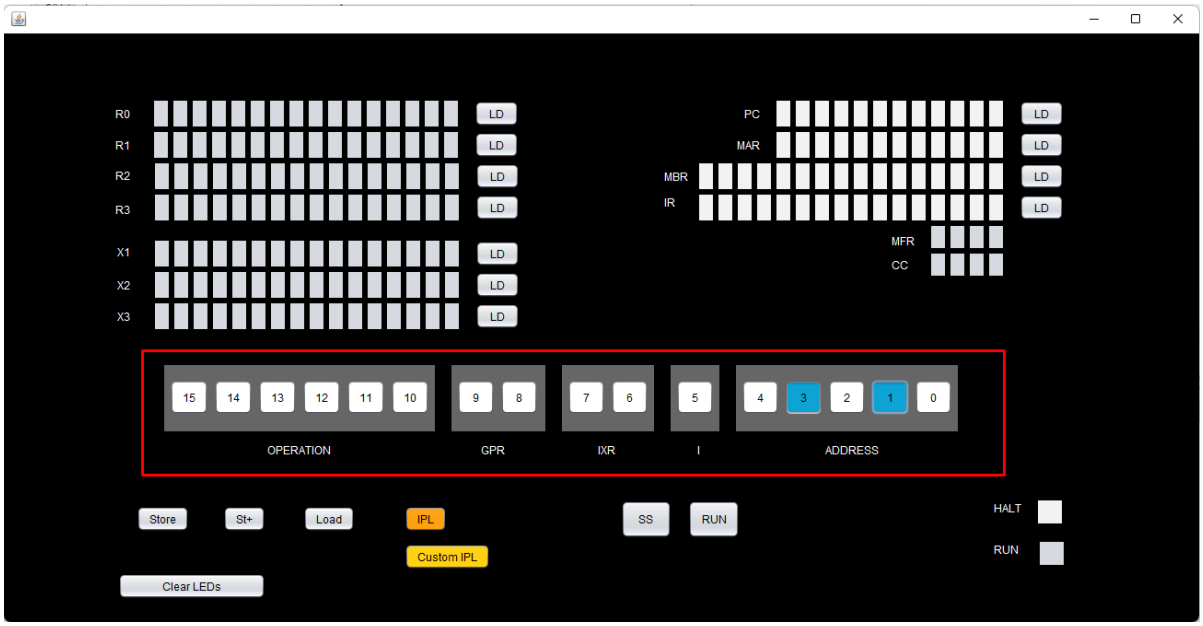
| OpCode | Instruction | Description |
|---|---|---|
| 01 | LDR r x I address | Load Register From Memory, r = 0..3<br>r <- c(EA) |
| 02 | STR r x I address | Store Register To Memory, r = 0..3<br>Memory(EA) <- c(r) |
| 03 | LDA r x I address | Load Register with Address, r = 0..3<br>r <- EA |
| 41 | LDX x I address | Load Index Register from Memory, x = 1..3<br>Xx <- c(EA) |
| 42 | STX x I address | Store Index Register to Memory. X = 1..3<br>Memory(EA) <- c(Xx) |

## Simulator Design:

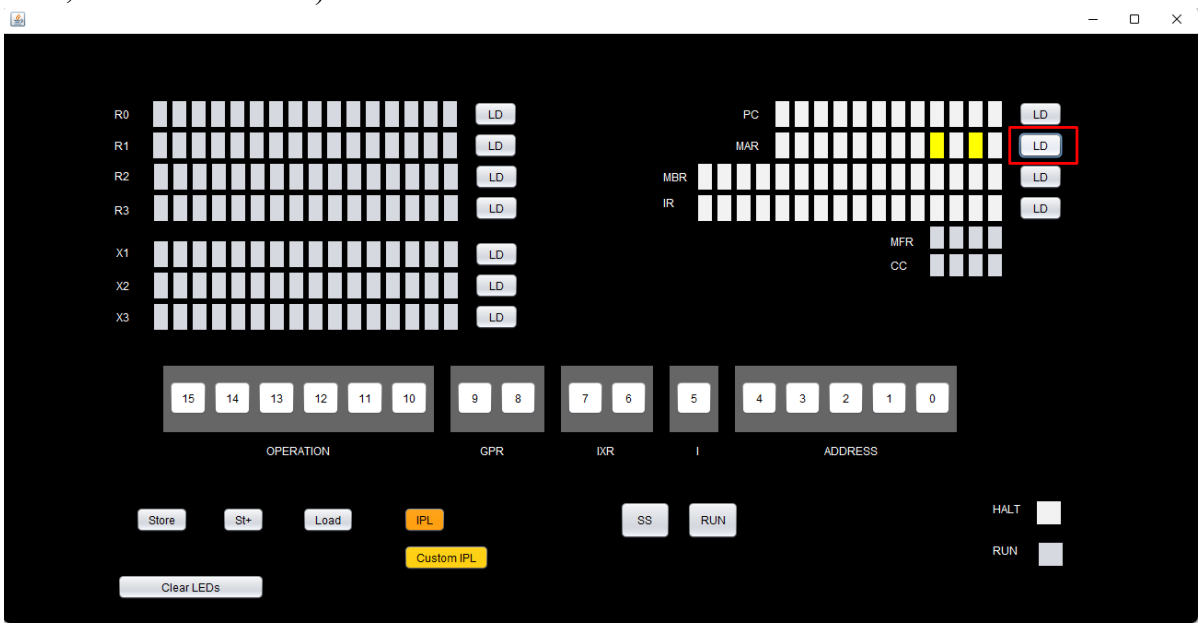


**How to add instruction to memory:**

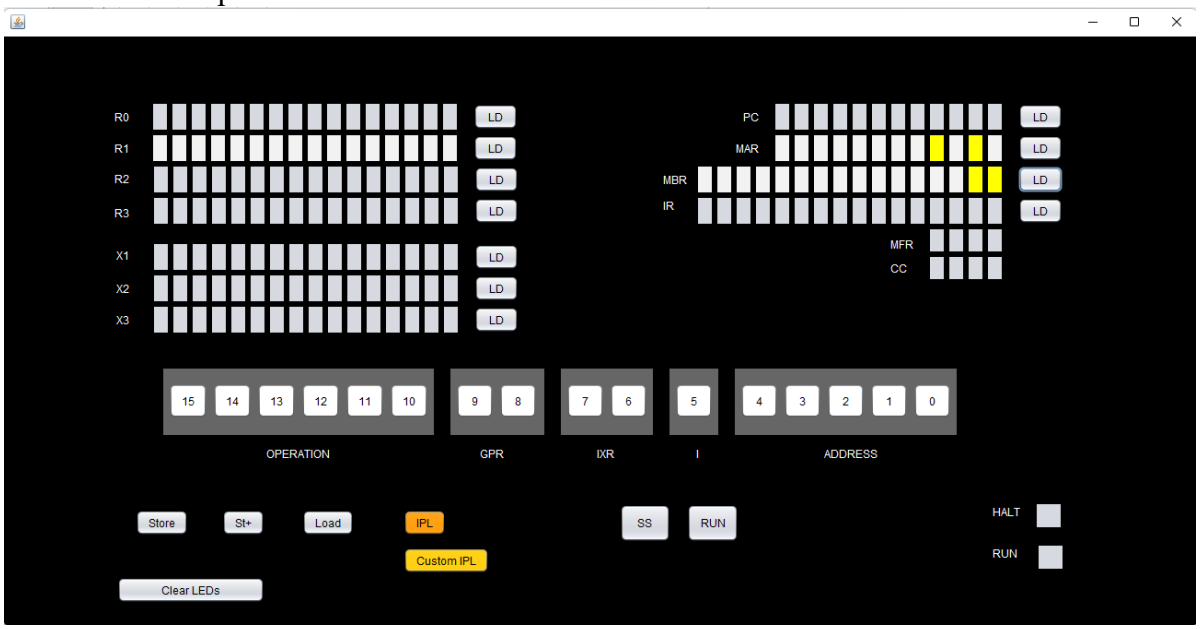
1. Input MAR instruction in Instruction Toggle, as highlighted below (White LED is 0 Blue LED is1:



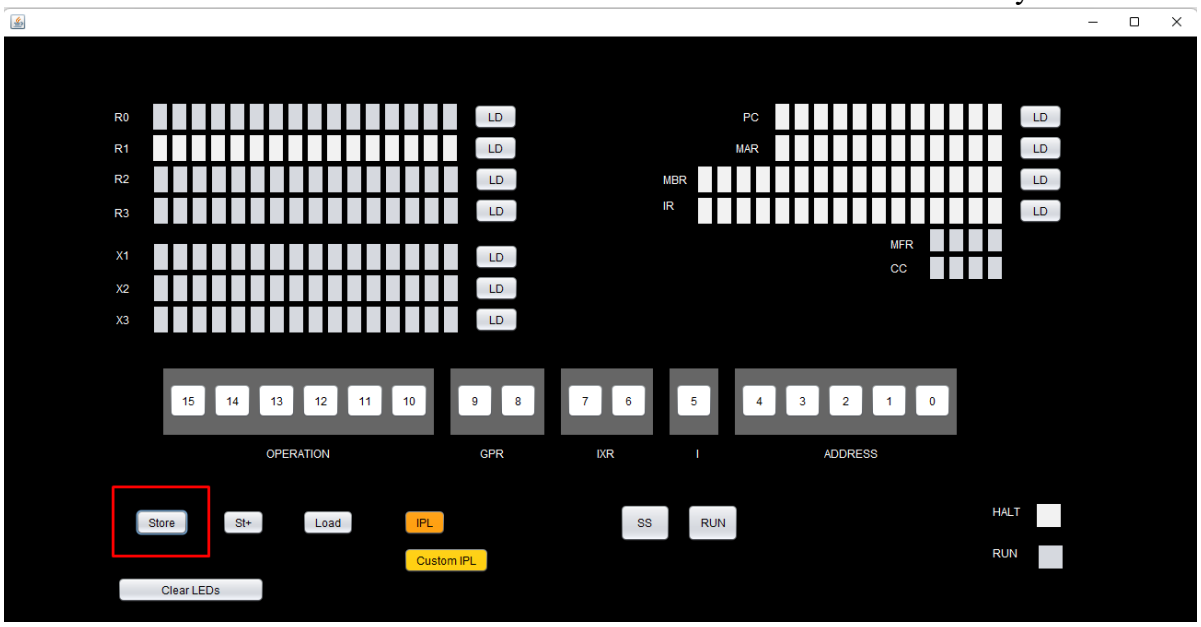

2. Click the “LD” button next to MAR to Load it to the MAR register (Yellow LED is for 1, whiteLED is for 0):



3. Do the above steps for MBR also:




4. Now Click the “Store” button to store the MBR instruction to the MAR memory location

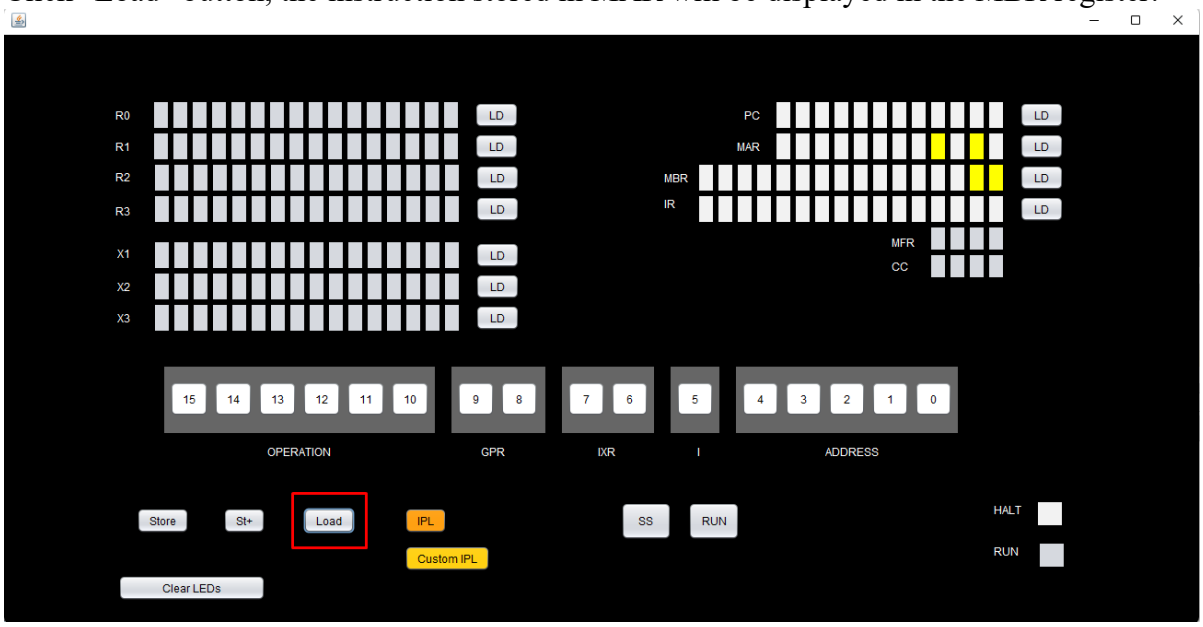



5. Store+ will work exactly like Store but with the added feature of incrementing the MARvalue, so that we don’t have to input it again for the next address.

**How to Load instruction from memory:**

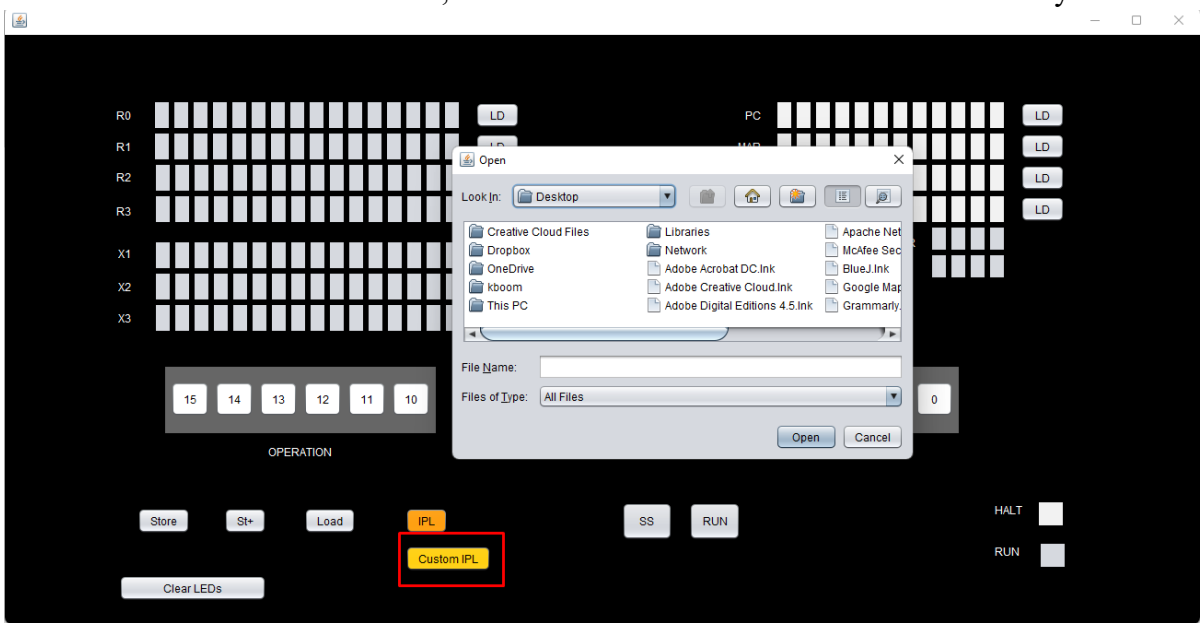
1. Click MAR instruction in the Instruction toggle and click “LD” to load it in the MAR register.

2. Click “Load” button, the instruction stored in MAR will be displayed in the MBR register:

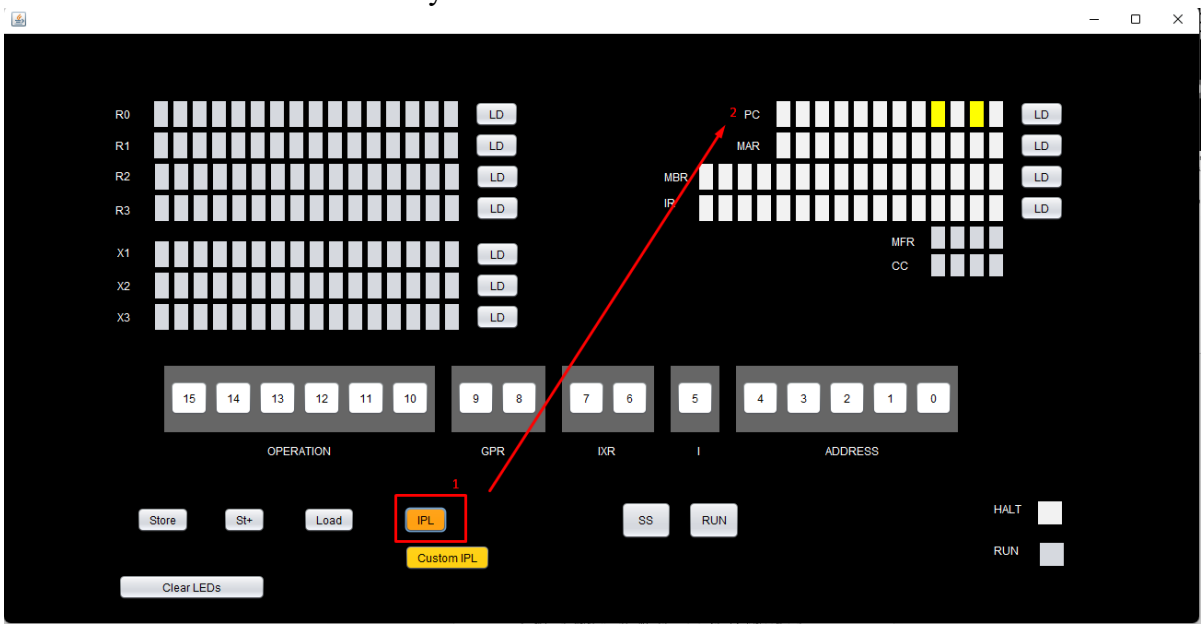



**How to Load IPL file or Custom IPL file:**

1. Click on the “IPL” button, this will load the contents of the default IPL.txt file to the memory.
2. Click on the “Custom IPL” button, to load the content of another txt file to memory.

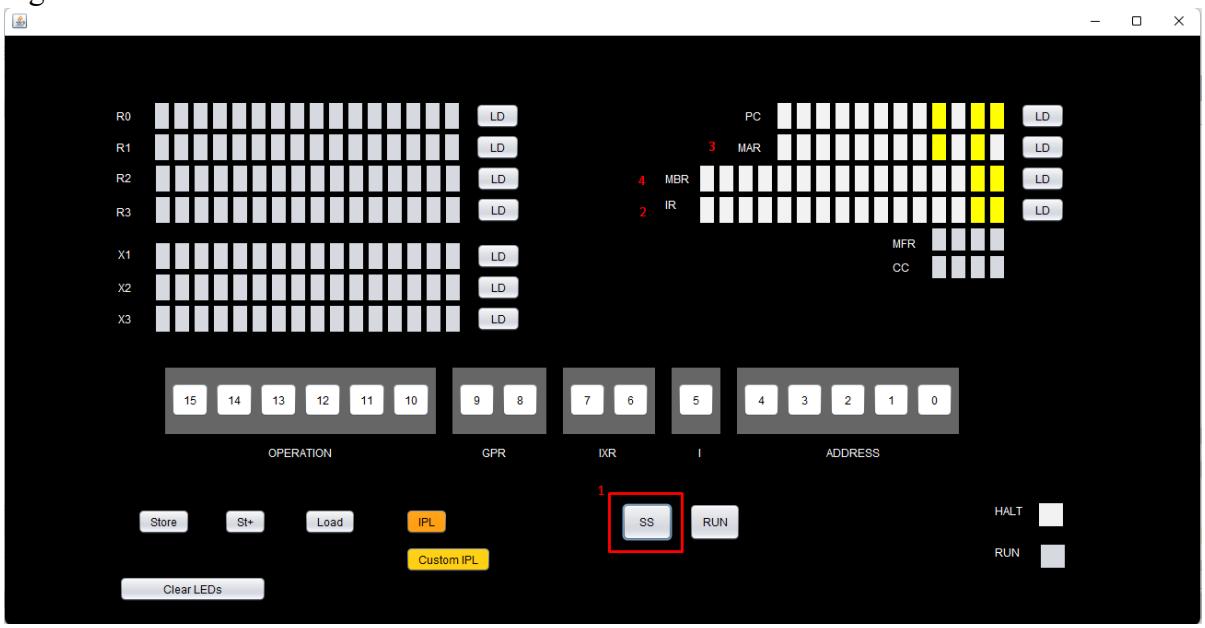


3. Once the “IPL” or “Custom IPL” button is clicked the PC will be loaded with the firstinstruction’s memory address:



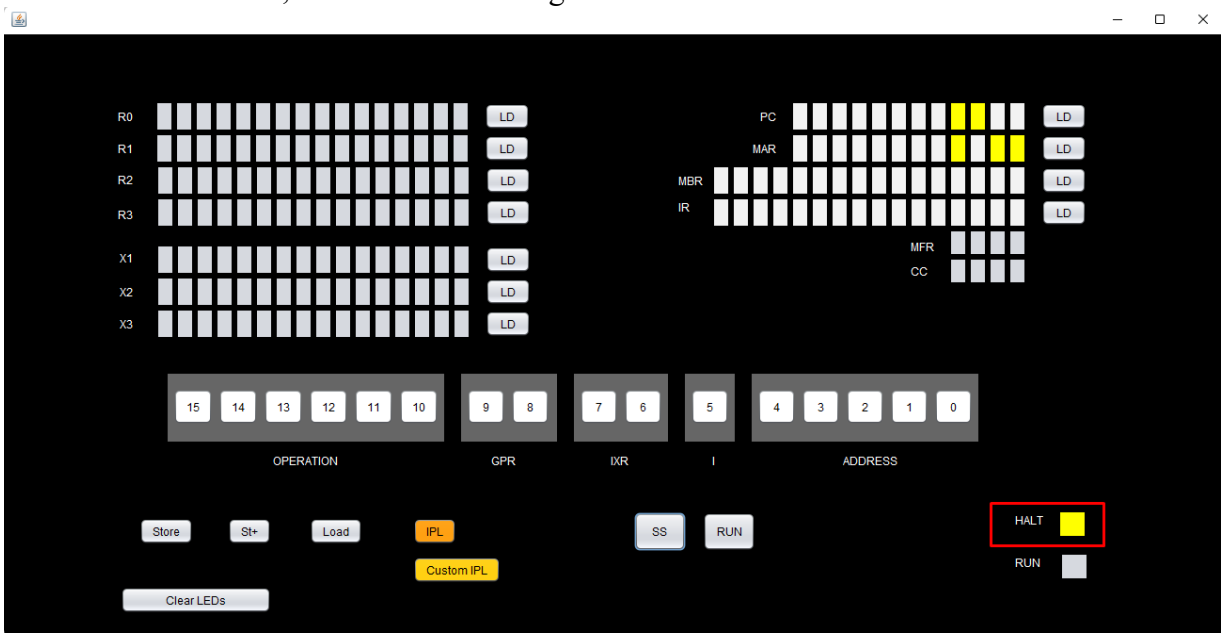

**How to Single Step:**

1. If “IPL” or “Custom IPL” was clicked, the PC would have automatically loaded, else if you want to try some other PC value you can click on the Instruction toggle buttons and click its“LD” to load the PC.
2. Click on the “SS” button to Single step, there will be a delay of 1 sec for registers to beloaded with their value.



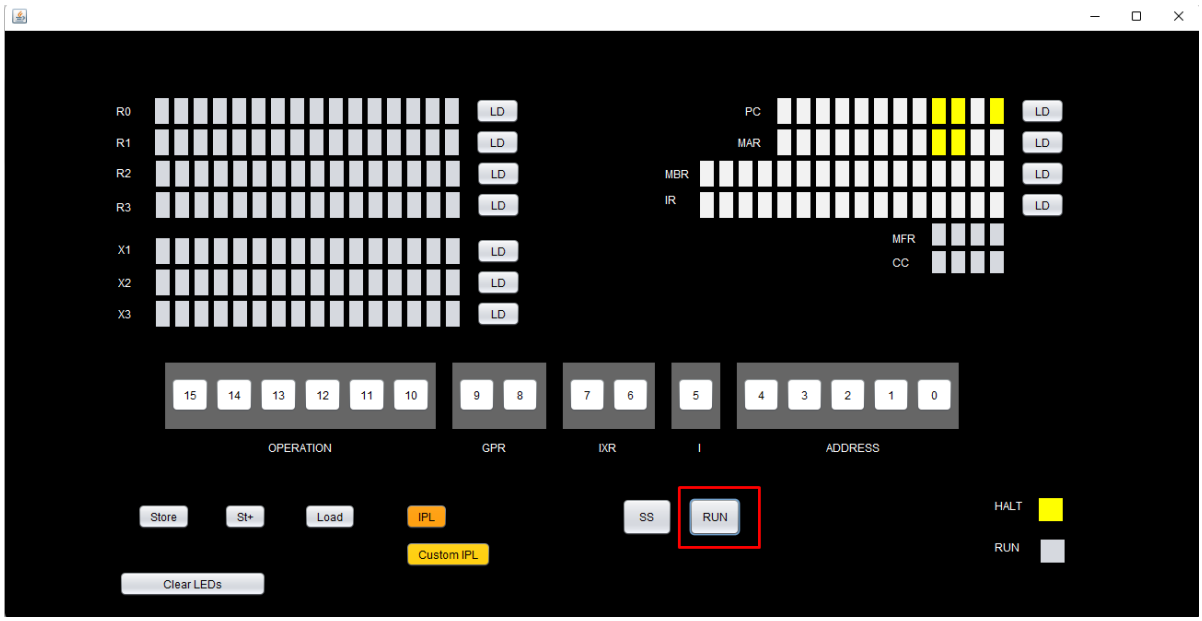

3. Once “SS” is clicked the registers will be loaded with their values in the following order: IR ->MAR -> MBR -> GPRs or IXs -> PC=PC+1.

4. If Halt is encountered, HALT LED would glow:



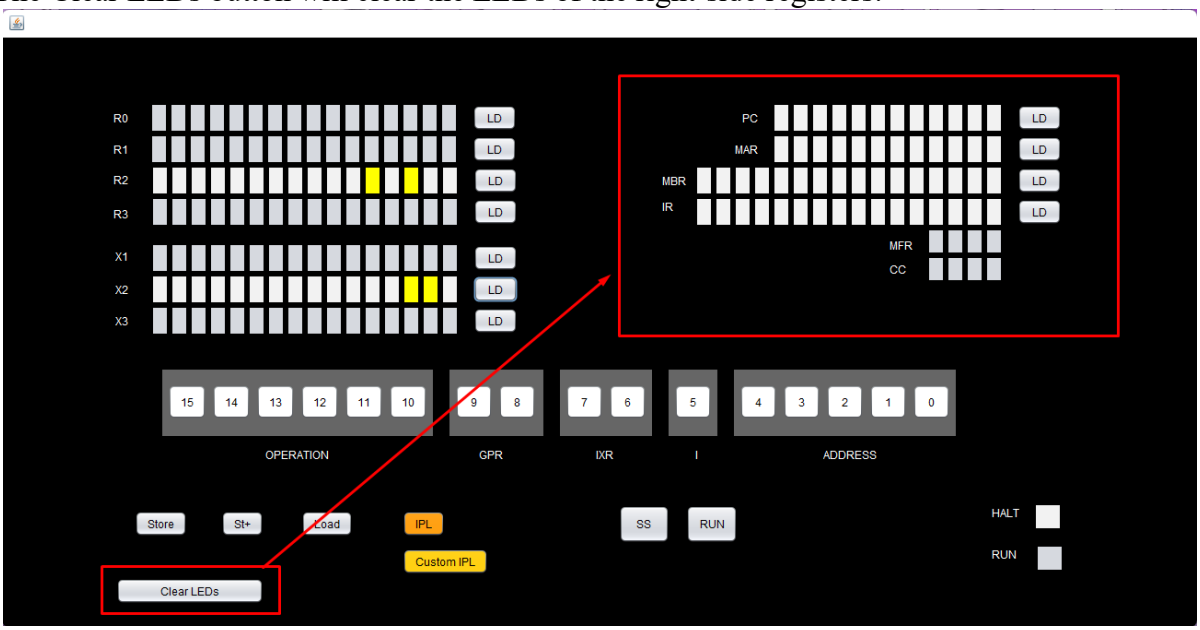

**How to Run:**

1. To do SS continuously till HALT is encountered, Click on the "RUN" button, the RUN LED willglow till the run is in progress and then turn to white when HALT is encountered.



**Clear LEDs:**

The Clear LEDs button will clear the LEDs of the right-side registers:



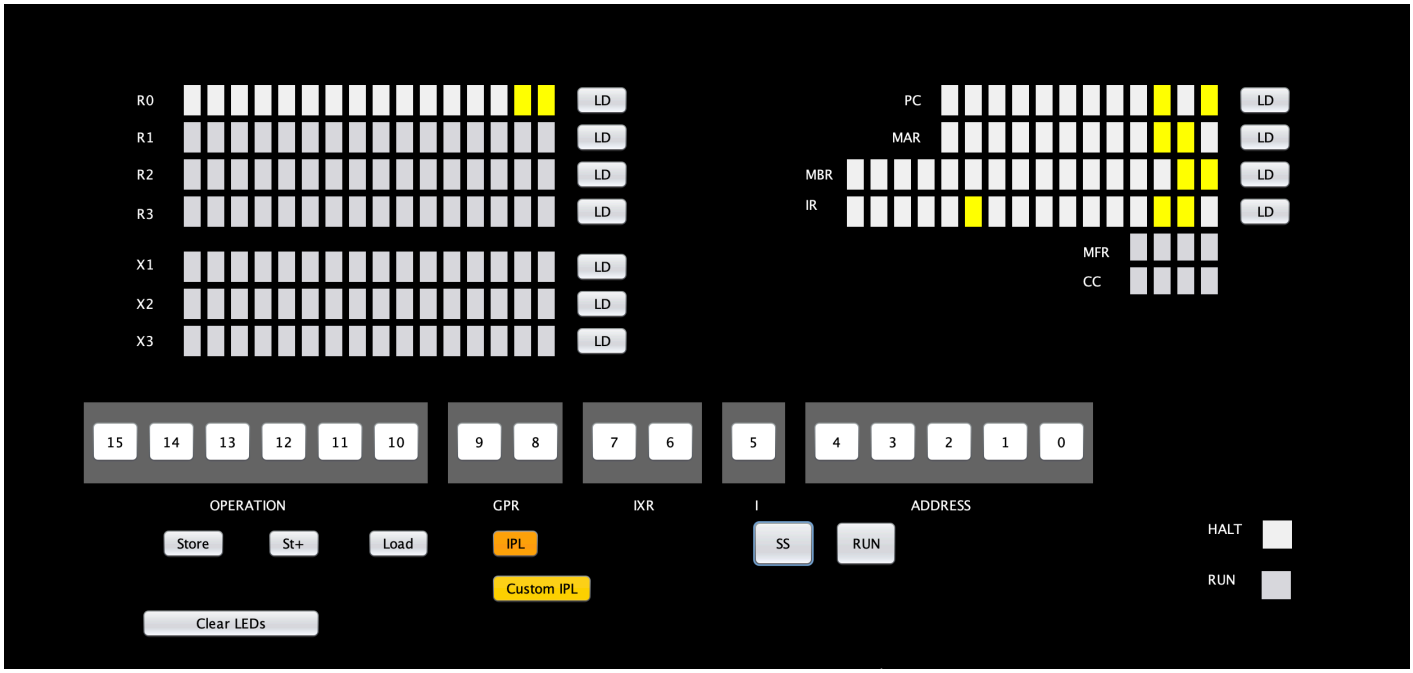

# Phase 2

**New UI elements we have added:**

- The Cache:
  - Used to
- Printer:
  - Used to
- Console:
  - Used to see output after every step in the instructions executed
- Keyboard
  - Used to take input from the user

**Executing Instructions:**
**Steps during each operation:**

1. **LDR r, x, address[,I] Load Register From Memory, r = 0..3 r <– c(EA) note that EA is computed as given above**:
   **Opcode: 01**
   - **i.** Load Address of 6 with value 3
   - **ii.** Load MBR Value, As the Opcode was one 0000010000000110
   - **iii.** Load MAR with 0000000000000100
   - **iv.** Load MBR with 0000000000000110
   - **v.** Store the Value and load PC as 0000000000000100
   - vi. Click on SS and then Pc will be incremented and GPR0 is loaded with the value with 1, which is the value at the effective address 6.




**2. STR r, x, address[,I] Store Register To Memory, r = 0..3 Memory(EA) <– c(r)**

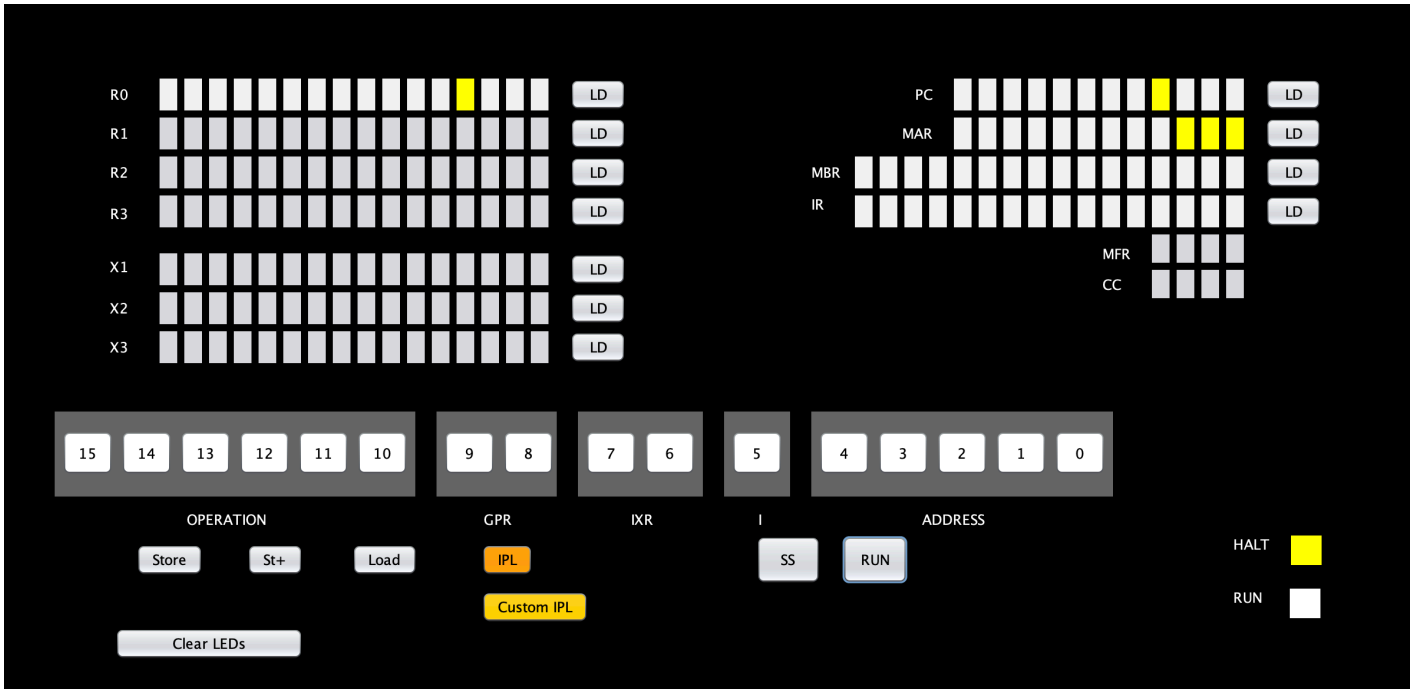
**Opcode: 02**

i. Load the GPR 0 to 0000000000000011
ii. Load MBR value as per the Opcode is 2 and GPR is 0 then 000010000000111
iii. Load MBR value to 0000000000001000
iv. Store the value and Load PC to 0000000000001000
v. We can see the values in the Panel Window

**3. LDA r, x, address[,I] Load Register with Address, r = 0..3 r <– EA**

**Opcode: 03**

**i.** Load MBR Value 0000110000001000 as the opcode is 3
**ii.** Give MAR value is 000000000110
**iii.** Store MAR Value and load PC to 000000000110
**iv.** Then GPR0 is loaded with effective address as below




**4. LDX x, address[,I] Load Index Register from Memory, x = 1..3 Xx <- c(EA)**

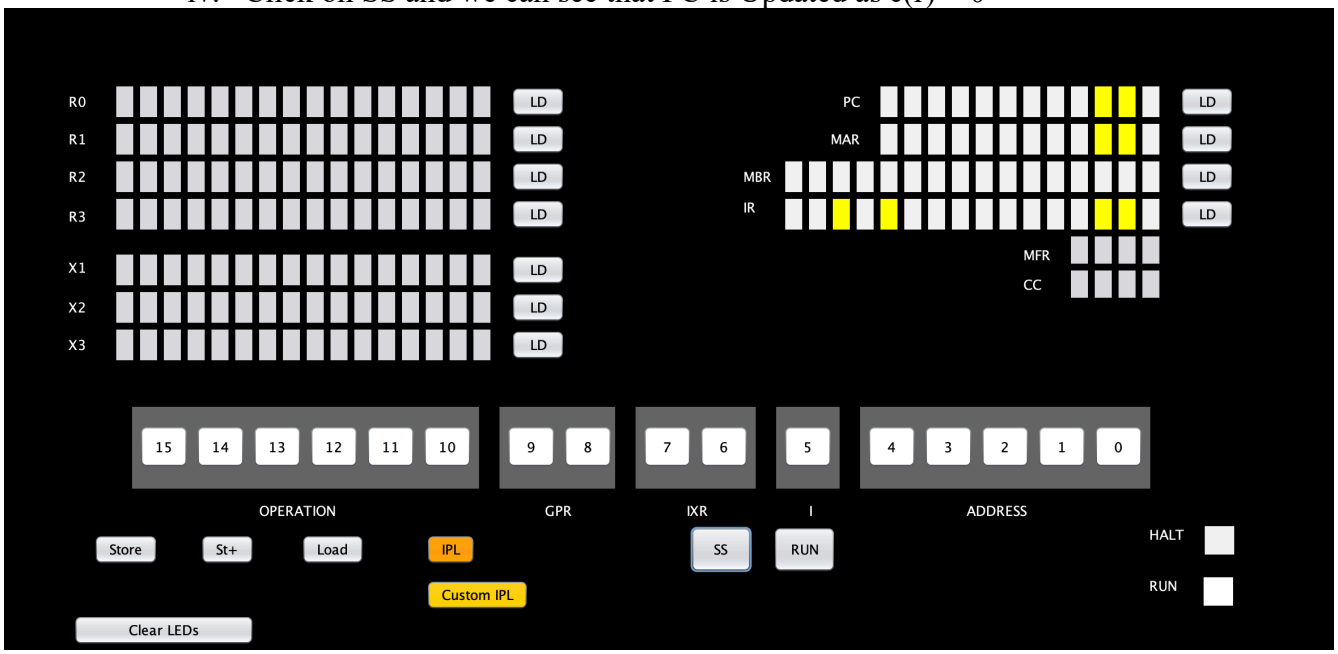
**Opcode: 41**

- **i.** Load the Value of address 7 with value 2
- **ii.** Load MBR value with 1010010001000111 and Load MAR Value with 000000000101
- **iii.** Store MAR and load pc to 000000000101
- **iv.** The IXR value is updated with value 1, which is the value of EA 2

**5. JZ r, x, address[,I] Jump If Zero: If c(r) = 0, then PC <– EA Else PC <- PC+1**

**Opcode: 10**

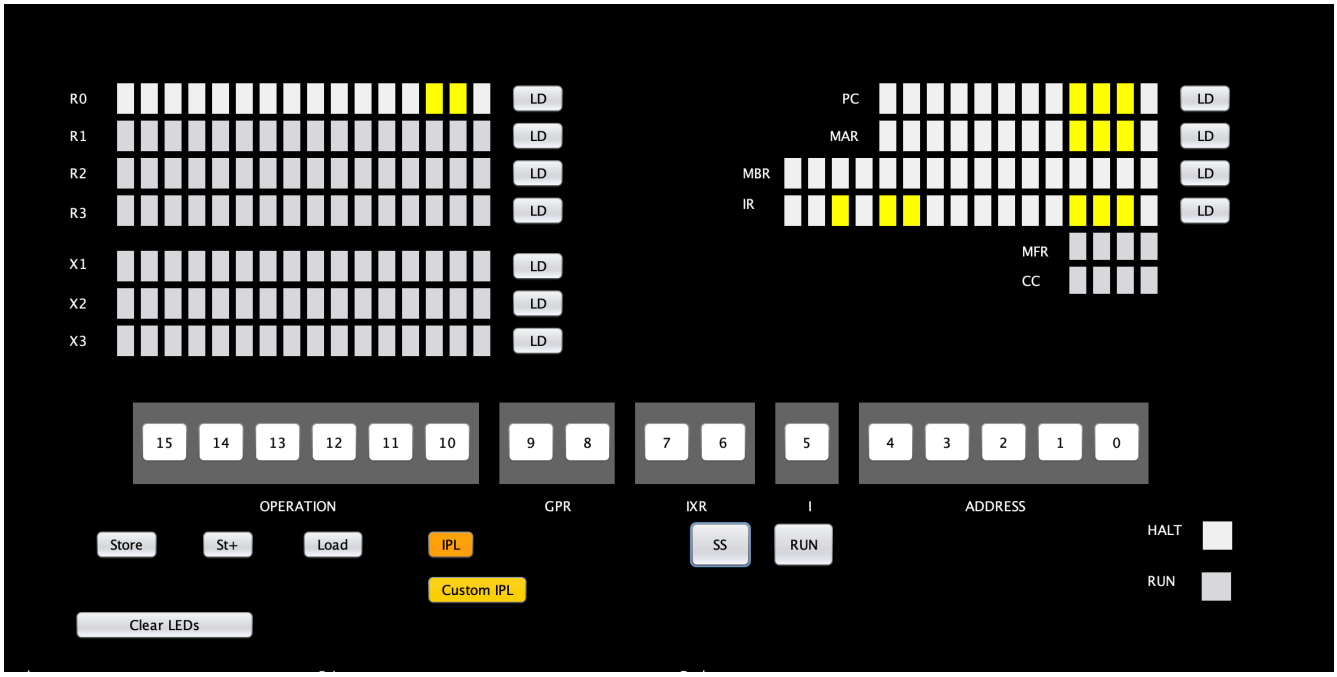
i. Load MBR value as 0010100000000110
ii. Load MAR value as 000000000100
iii. Load the value and load PC to 000000000100
iv. Click on SS and we can see that PC is Updated as c(r) = 0




**6. JNE r, x, address[,I] Jump If Not Equal: If c(r) != 0, then PC <–- EA Else PC <- PC + 1**

**Opcode: 11**

i. Load GPR 0 to 0000000000000110
ii. Load MBR value as the opcode was 11 to 0010110000001110
iii. Load MAR value as 000000000101
iv. Load the value and load PC to 000000000101
v. Click on SS and we can see that PC is Updated as c(r) != 0, then PC <– EA




7. **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**

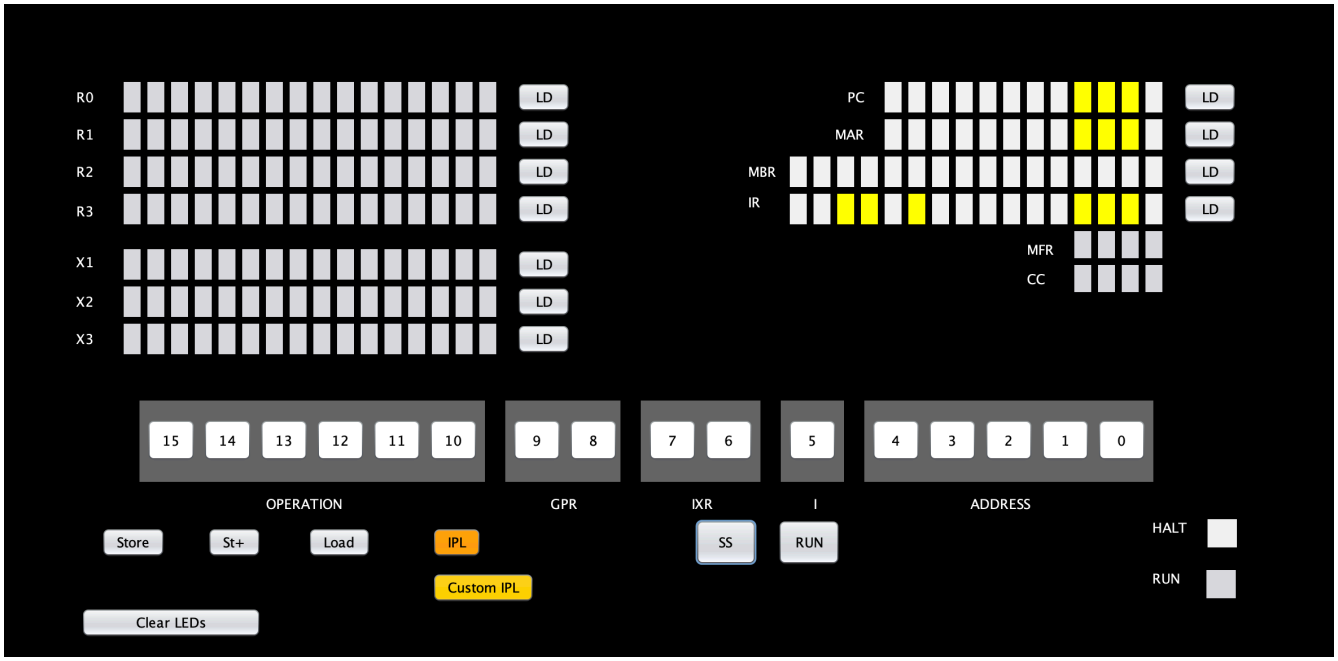
   **Opcode: 12**

8. **JMA x, address[,I] Unconditional Jump To Address PC <- EA, Note: r is ignored in this instruction**

   **Opcode: 13**

   i. Load MBR with 0011010000001110
   ii. Load MAR value as 000000000100
   iii. Load the value and load PC to 000000001100

Click on SS and we can see that PC is Updated as the effective address value to 000000001110

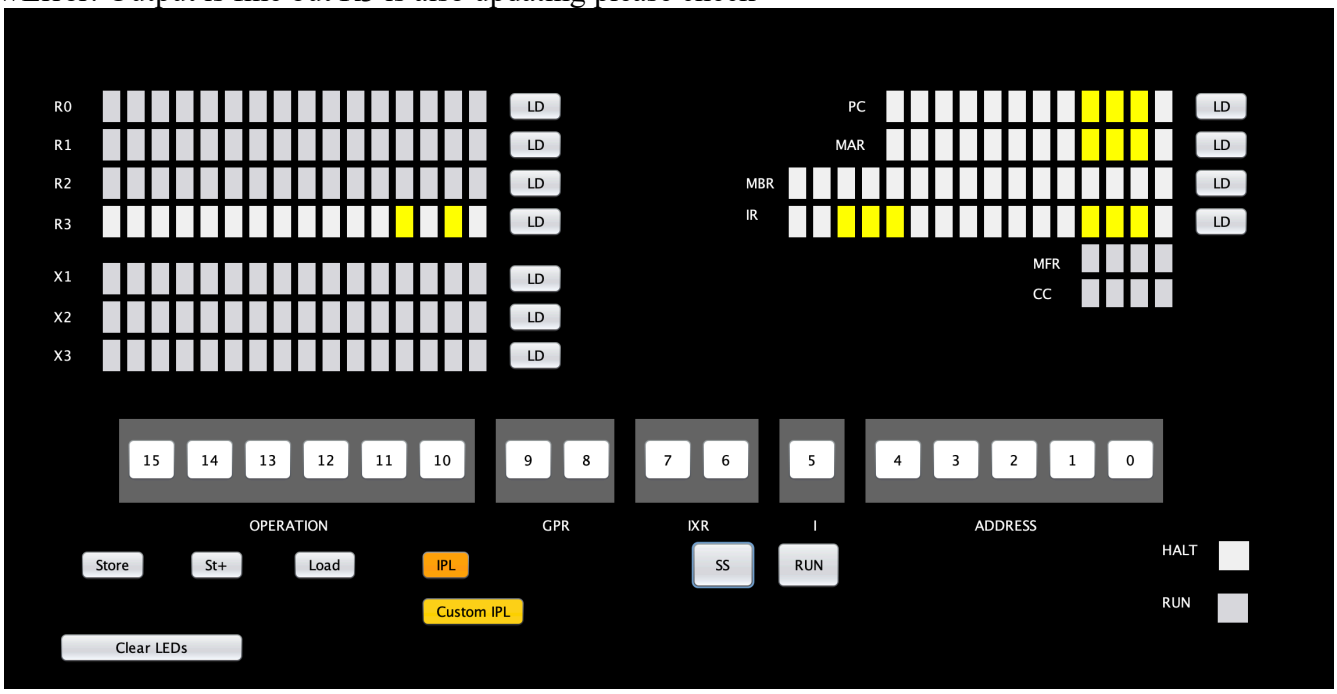



**9. JSR x, address[,I] Jump and Save Return Address: R3 <– PC+1; PC <– EA R0 should contain pointer to arguments Argument list should end with –1 (all 1s) value**

**Opcode: 14**

i. Load MBR with 0011100000001110
ii. Load MAR value as 000000001001
iii. Load the value and load PC to 000000001001
iv. Click on SS and we can see that PC is Updated as the effective address value to 000000001110

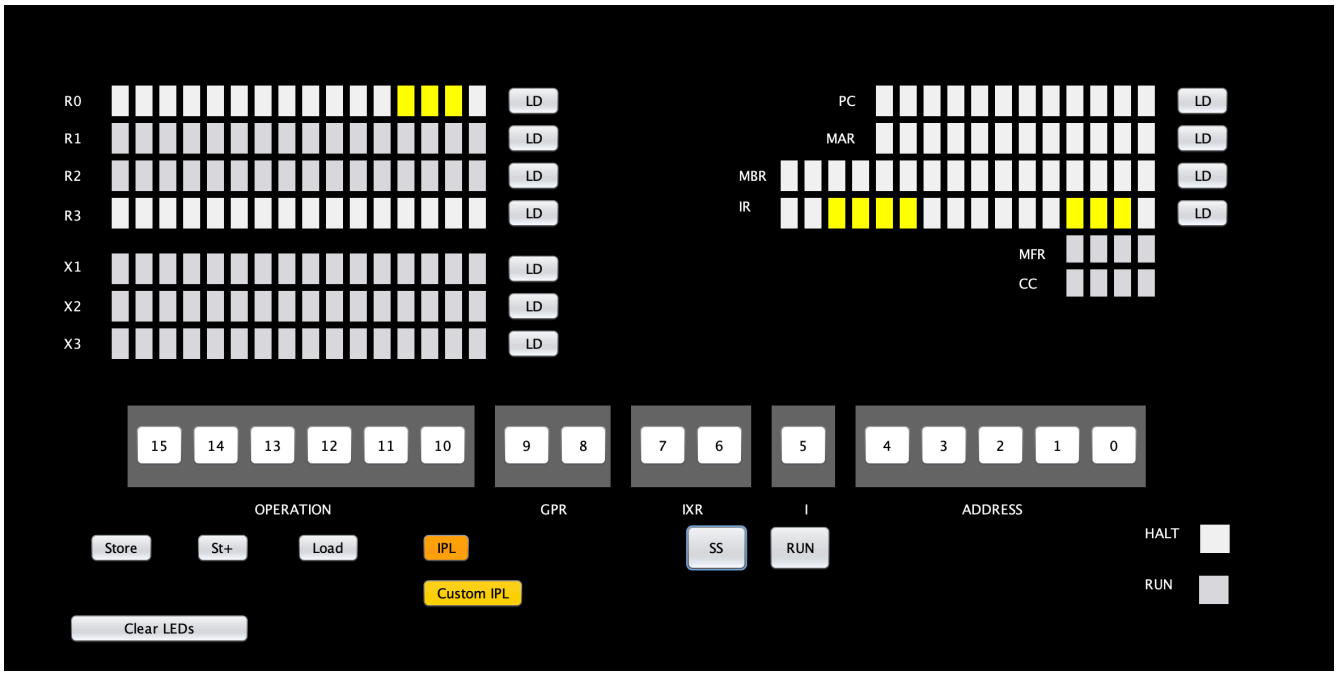
//Error: Output is fine but R3 is also updating please check



**10. 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.**

**Opcode: 15**

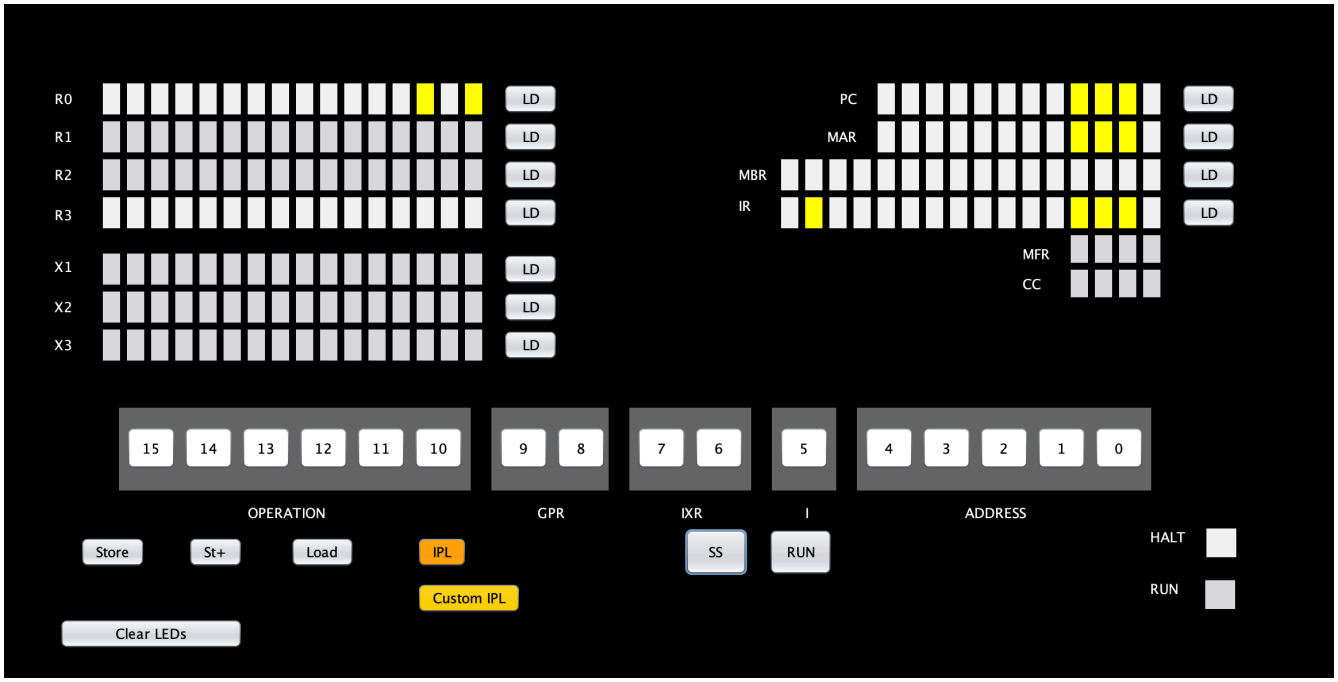
i. Load GPR3 with 0000000000000000
ii. Load MBR with opcode 15 to 0011110000001110
iii. Load MAR value as 000000000110
iv. Load the value and load PC to 000000000110
v. Click on SS and we can see that PC is Updated to 000000000000
vi. GPR 0 value was updated as 0000000000001110 which is EA.




**11. 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**

**Opcode: 16**

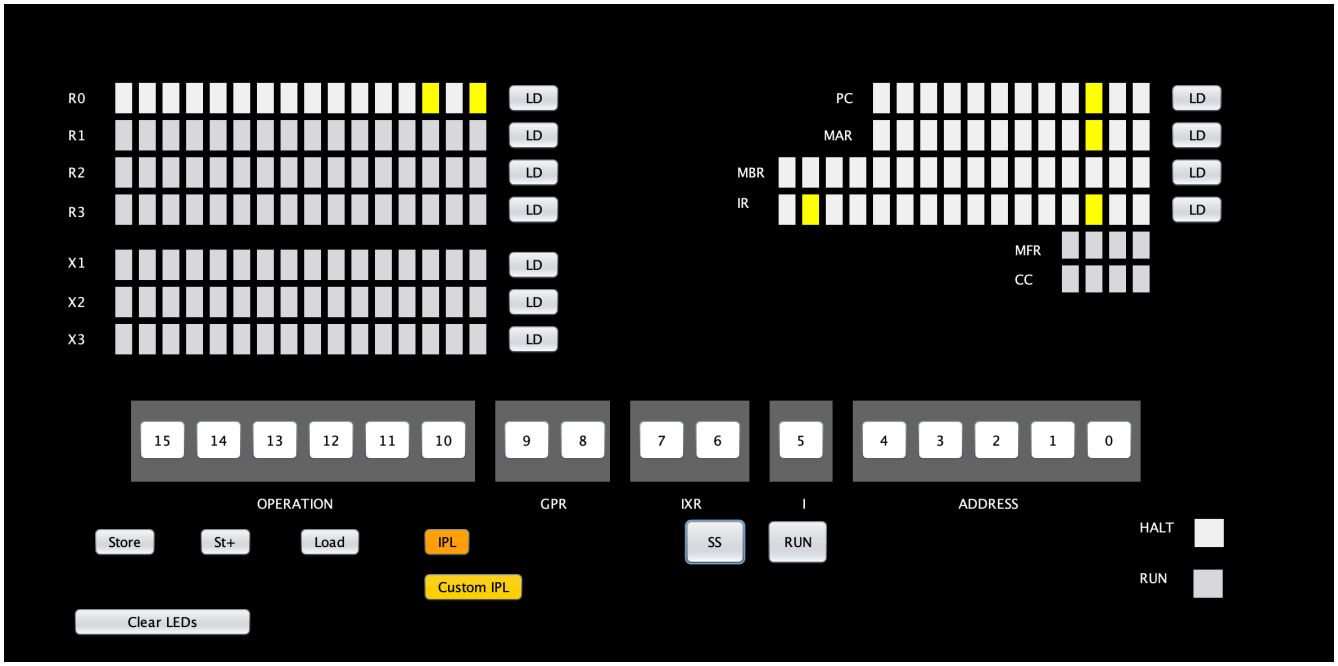
i. Load GPR0 with 0000000000000110
ii. Load MBR with 0100000000001110
iii. Load MAR value as 000000001000
iv. Store the value and load PC to 000000001000
v. Click on SS and we can see that PC is Updated EA to 000000001110
vi. GPR 3 was subtracted by one =0000000000000101




**12. JGE r,x, address[,I] Jump Greater Than or Equal To: If c(r) >= 0, then PC <- EA Else PC <- PC + 1**

**Opcode: 17**

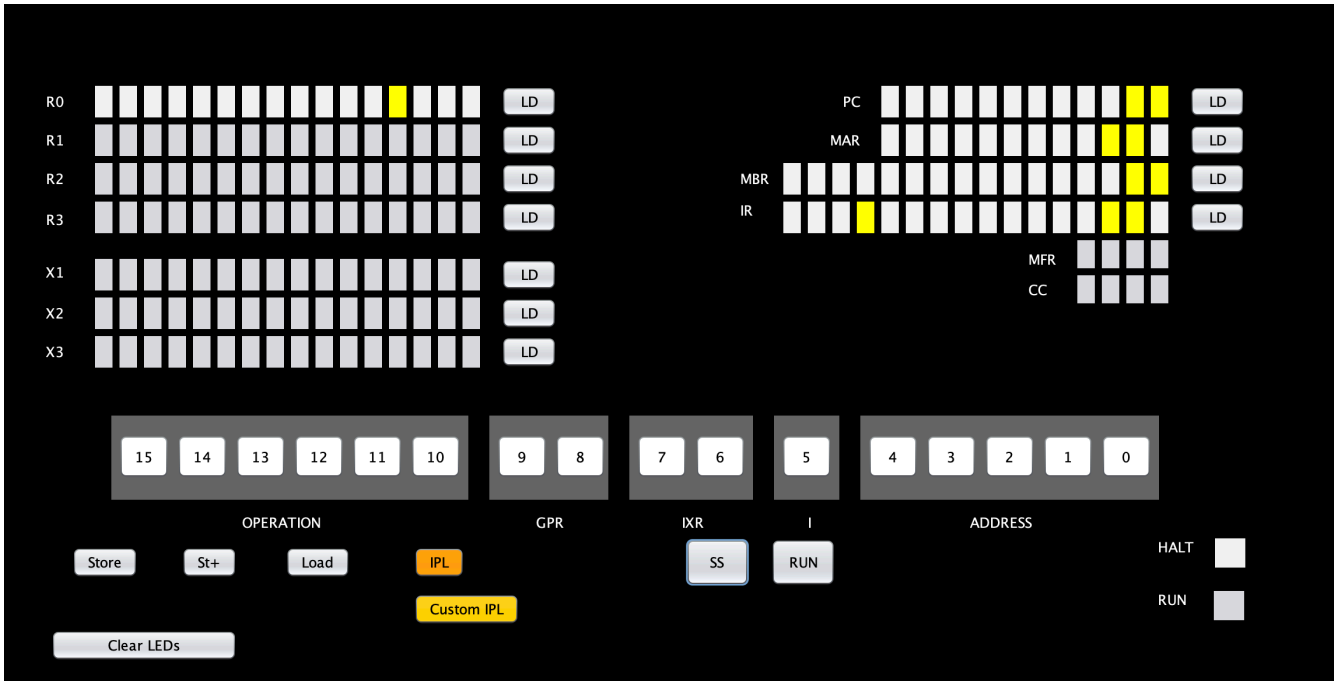
i. Load GPR0 with 0000000000000110
ii. Load MBR with 0100000000000100
iii. Load MAR value as 000000010000
iv. Load the value and load PC to 000000010000
v. Click on SS and we can see that PC is Updated EA to 000000000100




13. **AMR r, x, address[,I] Add Memory To Register, r = 0..3 r<– c(r) + c(EA)**

**Opcode: 04**

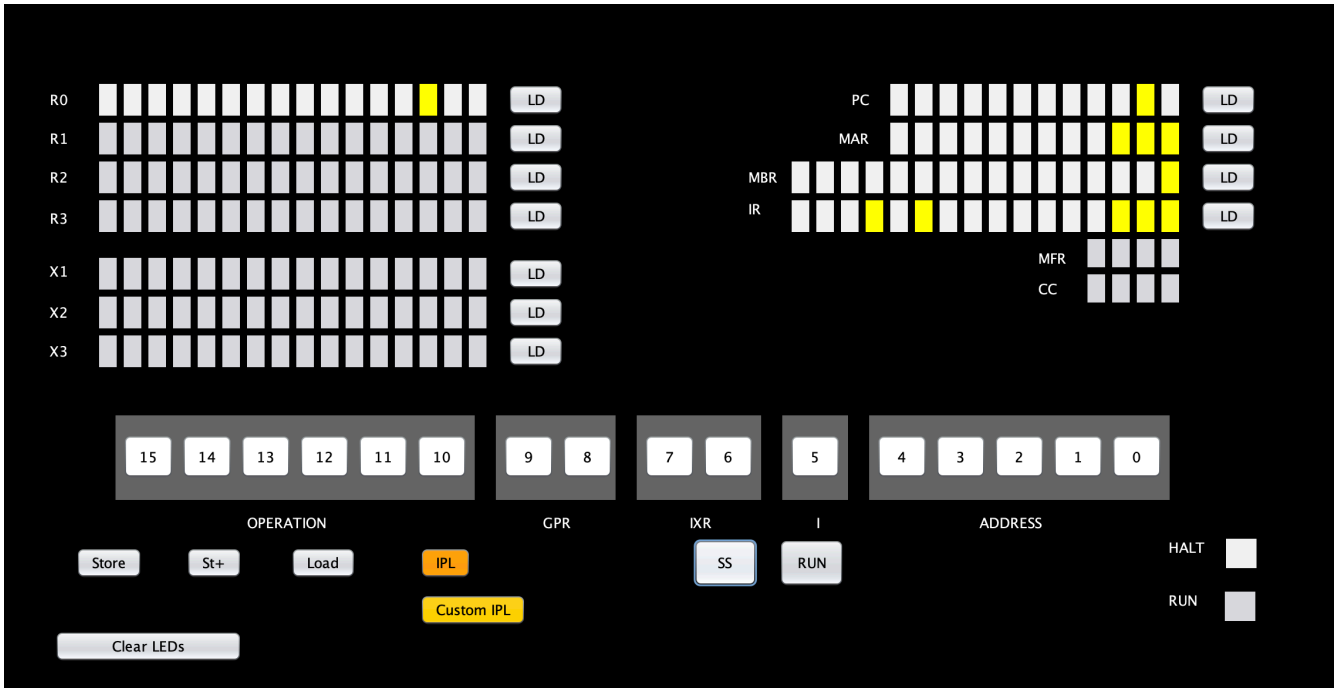
i. Load GPR0 with 0000000000000101
ii. Load MAR with location 6 which has value 3
iii. Load MBR with opcode 04 to 0001000000000110
iv. Load MAR value as 000000000010
v. Load the value and load PC to 000000000010
vi. GPR 0 is updated with value of ofc(r) + c(EA) i.e 000000001000




**14. SMR r, x, address[,I] Subtract Memory From Register, r = 0..3 r<– c(r) – c(EA)**

**Opcode: 05**

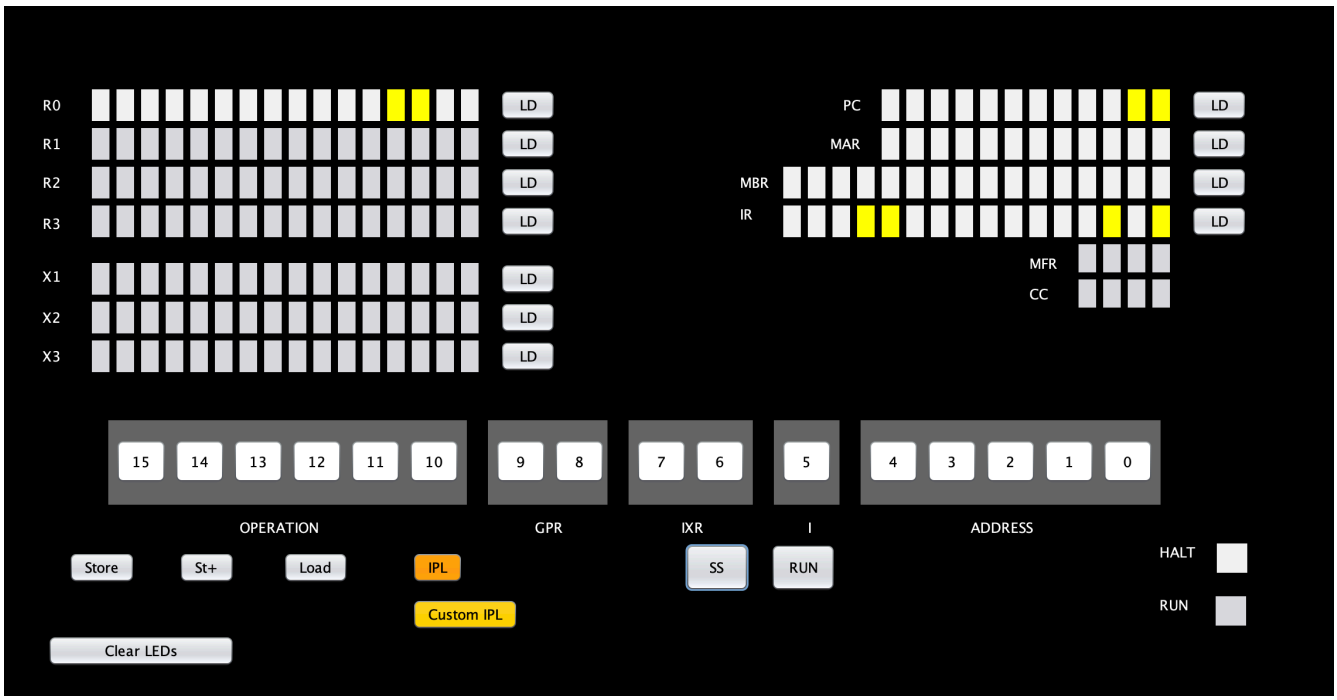
i. Load GPR0 with 0000000000000101
ii. Load MAR with location 7 which has value 1
iii. Load MBR with 0001010000000111
iv. Load MAR value as 000000000001
v. Load the value and load PC to 000000000001
vi. GPR 0 is updated with value of fc(r) - c(EA) = 2




**15. 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**

**Opcode: 06**

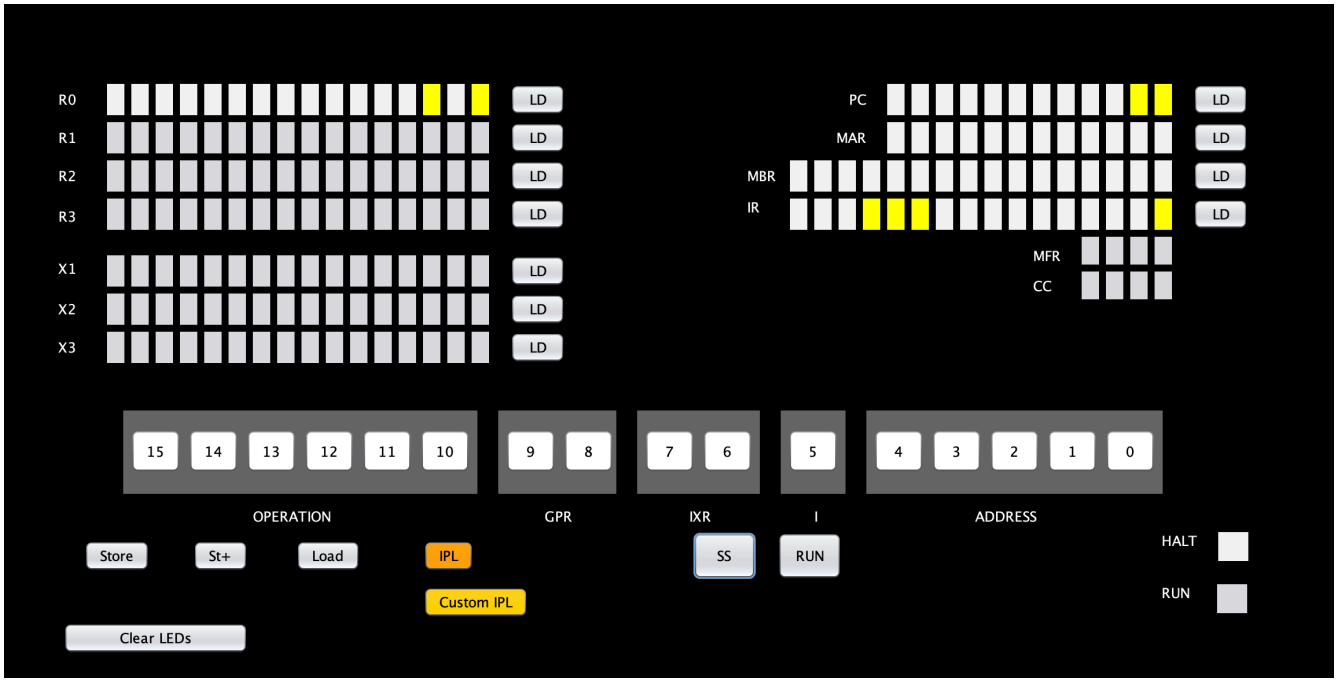
i. Load GPR0 with 0000000000000111
ii. Load MBR with 0001100000000101
iii. Load MAR value as 000000000010
iv. Load the value and load PC to 000000000010
v. GPR 0 is updated with value of (r) + Immed.= 0000000000001100



**16. 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**

**Opcode: 07**

i. Load GPR0 with 0000000000000110
ii. Load MBR with 0001110000000001
iii. Load MAR value as 000000000010
iv. Load the value and load PC to 000000000010
v. GPR 0 is updated with value of c(r) - Immed.= 0000000000000101




**17. 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**

**Opcode: 20**

i. Load GPR0 with
ii. Load GPR1 with
iii. Load MBR with opcode 20 to 0101000001000000
iv. Load MAR value as
v. Load the value and load PC to
vi. We have Overflow during Multiplication we set cc(0) as 1

**18. 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(2) to 1 (set DIVZERO flag)**

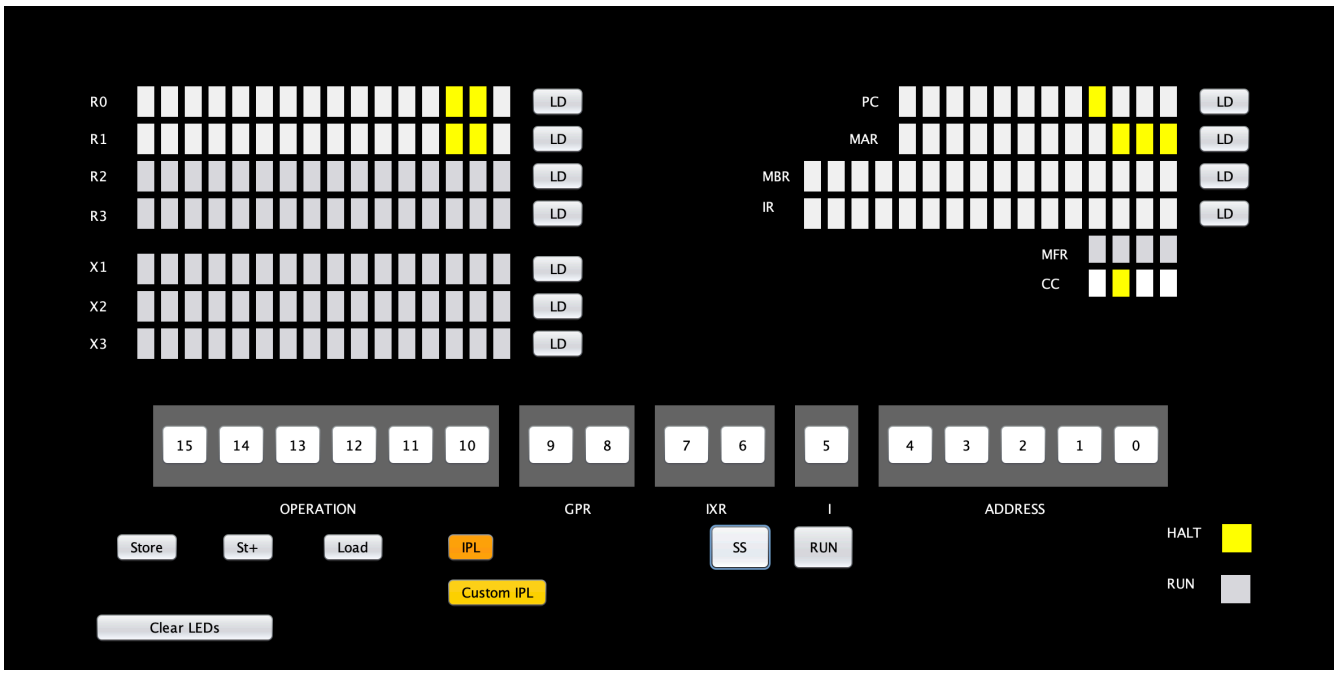
**Opcode: 21**

i. Load GPR0 with 0000000000000101
ii. Load GPR2 with 1000000000000010
iii. Load MBR with 0101010010000000
iv. Load MAR value as 000000000110
v. Load the value and load PC to 000000000110

**19. TRR rx, ry Test the Equality of Register and Register If c(rx) = c(ry), set cc(4) <– 1; else, cc(4) <– 0**

**Opcode: 22**

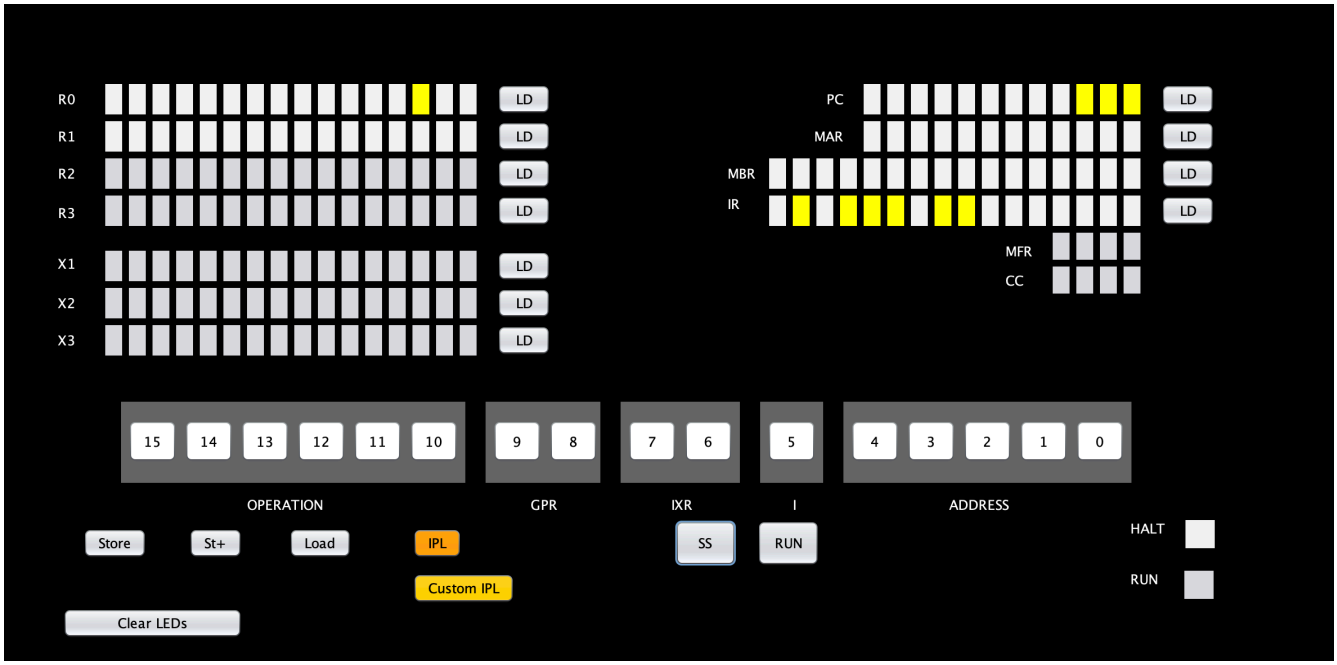
i. Load GPR0 with 0000000000000110
ii. Load GPR2 with 1000000000000110
iii. Load MBR with 0101010010000000
iv. Load MAR value as 000000000010
v. Load the value and load PC to 000000000010




**20. AND rx, ry Logical And of Register and Register c(rx) <– c(rx) AND c(ry)**

**Opcode: 23**

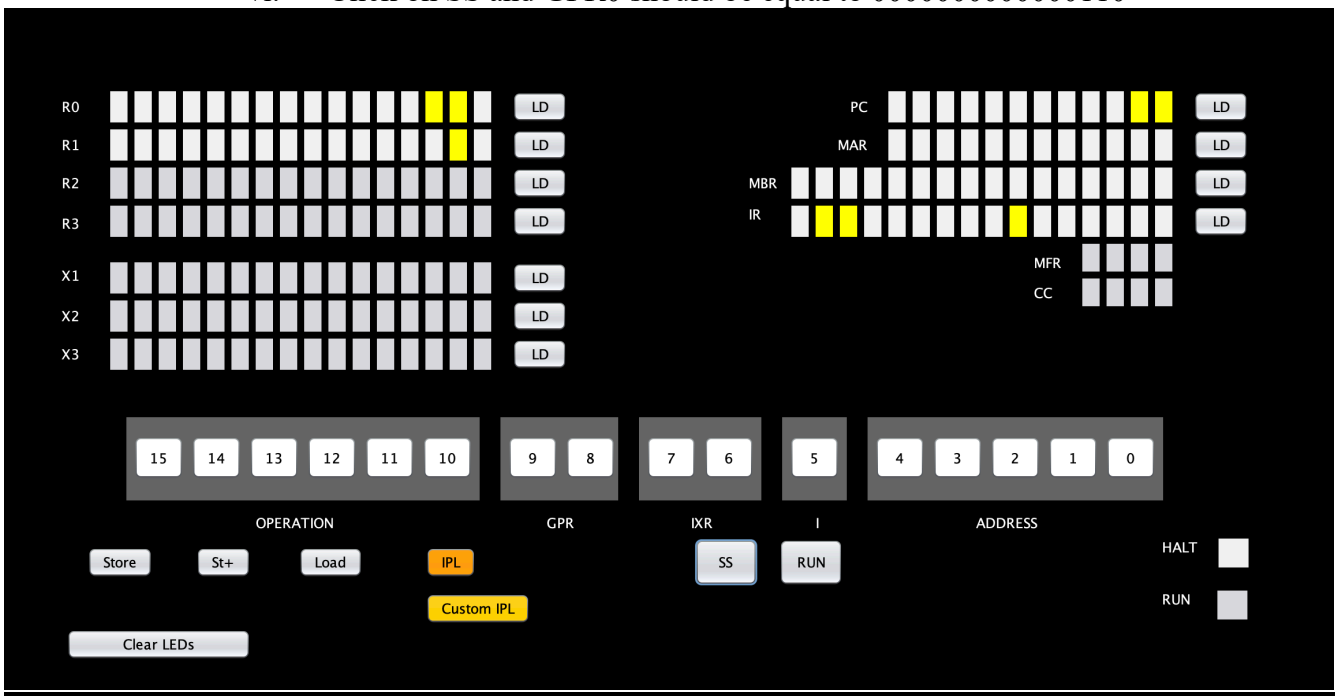
i. Load GPR1 with 0000000000000100
ii. Load GPR 2 with 0000000000001101
iii. Load MBR with 0101110110000000
iv. Load MAR value with 000000000110
v. Store and Load Pc with 000000000110
vi. Click on SS and GPR1 should be equal to 0000000000000100




**21. ORR rx, ry Logical Or of Register and Register c(rx) <– c(rx) OR c(ry)**

**Opcode: 24**

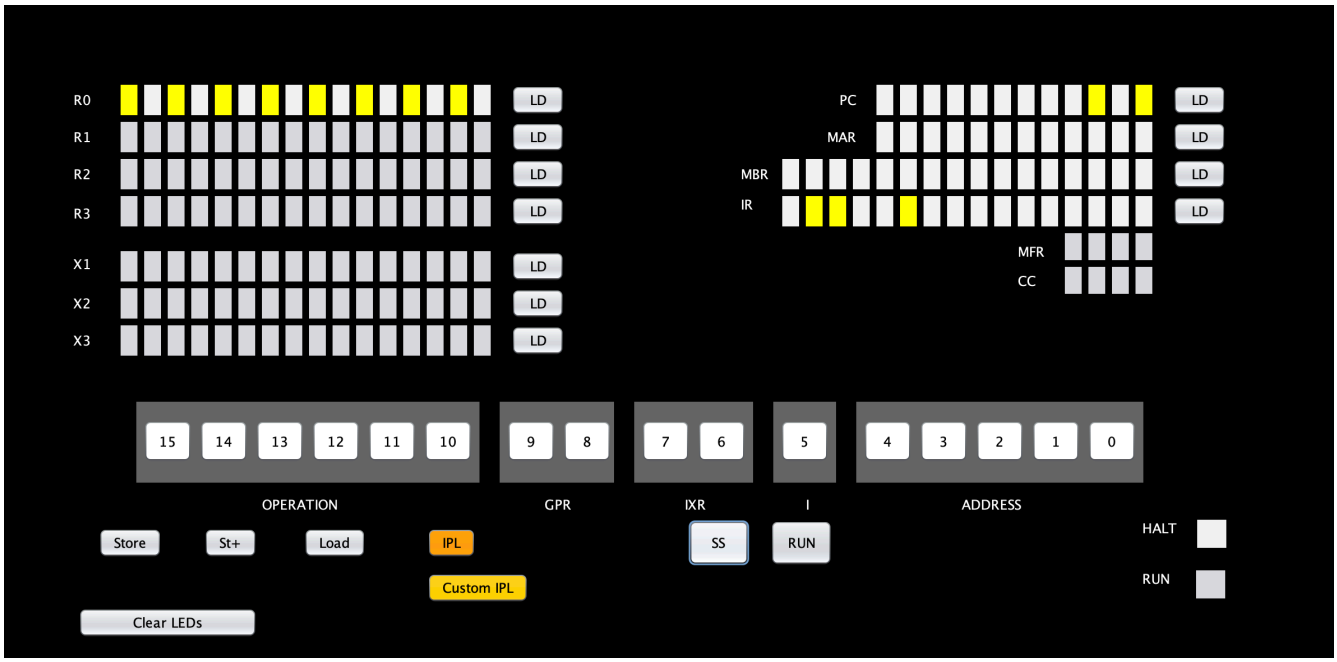
i. Load GPR0 with 0000000000000100
ii. Load GPR 1 with 0000000000000010
iii. Load MBR with 0110000001000000
iv. Load MAR value with 000000000010
v. Store and Load Pc with 000000000010
vi. Click on SS and GPR0 should be equal to 0000000000000110



**22. NOT rx Logical Not of Register To Register C(rx) <– NOT c(rx)**

**Opcode: 25**

i. Load GPR0 with 0101010101010101
ii. Load MBR with 0110010000000000
iii. Load MAR with 000000000100 and Store it
iv. Load PC with 000000000100
v. Click on ss the GPR0 will be 1010101010101010




**23. 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**

**Opcode: 31**

i. Load GPR1 With 0000000000000100
ii. Load MBR 0111110100000001
iii. Load MAR 000000000111 and Store it
iv. Load .PC→000000000111
v. Here, we have taken (L/R = 0) and arithmetically (A/L = 0) , so we can see that those are shifted right arithmetically and the resultant value is displayed in the GPR1

**24. 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) XX, XXX is ignored Count = 0…15 If Count = 0, no rotate occurs**

**Opcode: 32**

**i.** Load GPR1 with 0000000000000010
**ii.** Load MBR with 1000000100000001
**iii.** Load MAR with 000000000111

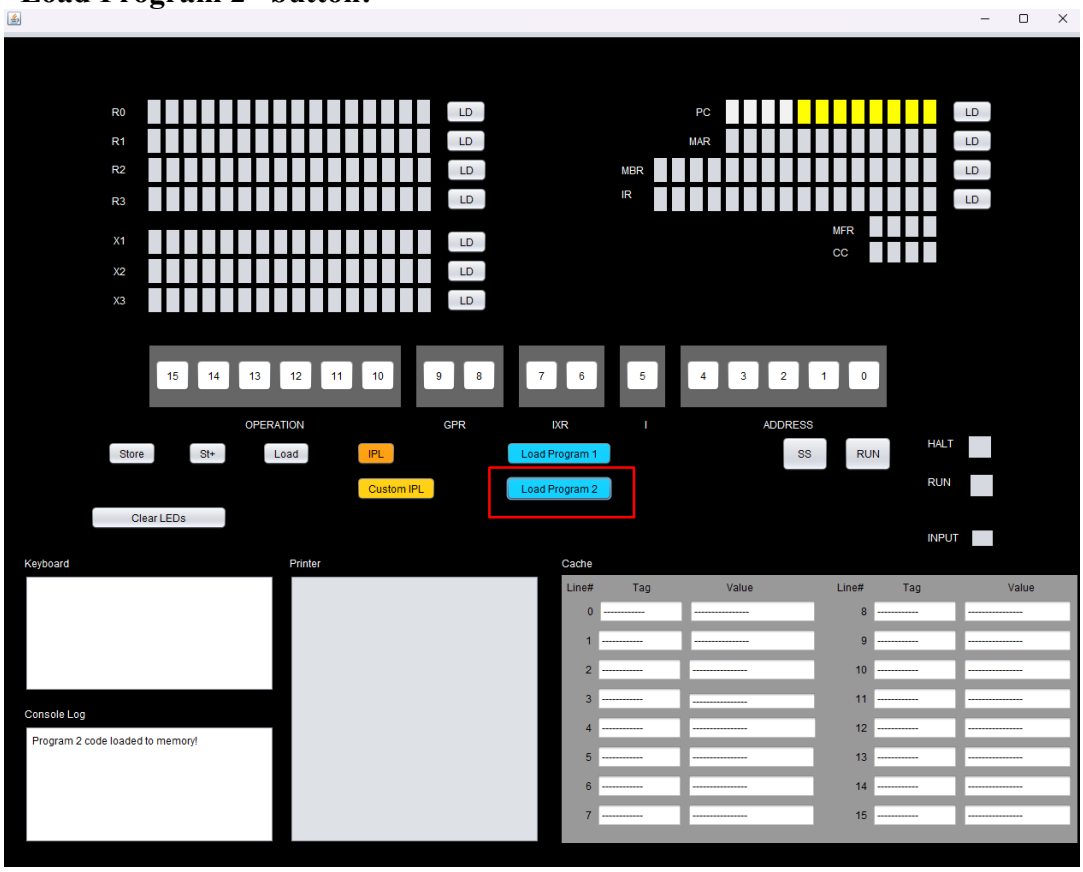
**iv.** Store it and Load PC with 000000000111

v. Here, we have taken (L/R = 1) and arithmetically (A/L = 1) , and count=1so we can see that those are shifted left logically and the resultant value is displayed in the GPR1.

# Phase 3

## New UI elements we have added:

### "Load Program 2" button:




On clicking the "Load Program 2" button, our console will load the content of Program2_text.txt to memory location 0x0258 to the length of paragraph containing 6 sentences, and the PC will be loaded with the first instruction address for Program 2.

On clicking "RUN" button, the Program2.txt instructions will execute.

First it will print the content of the 6 sentences in the printer, then it will prompt the user to input the word to be searched.

Each character of the sentence will be compared to each character in the word, if match is found "Word found!" will be printed along with the line # and word #, else "Word not found!" will be printed.

We have added a check for space and '.' to count the number of words and lines respectively.

If the word is found in the paragraph but it is found as a substring, a condition has been added to disregard that as a valid find and move on to the next word to continue the search.

**Traps and Machine Faults**

| OpCode | Instruction | Description |
|---|---|---|
| **0** | **HALT** | Stops the machine. |
| **30** | **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. |

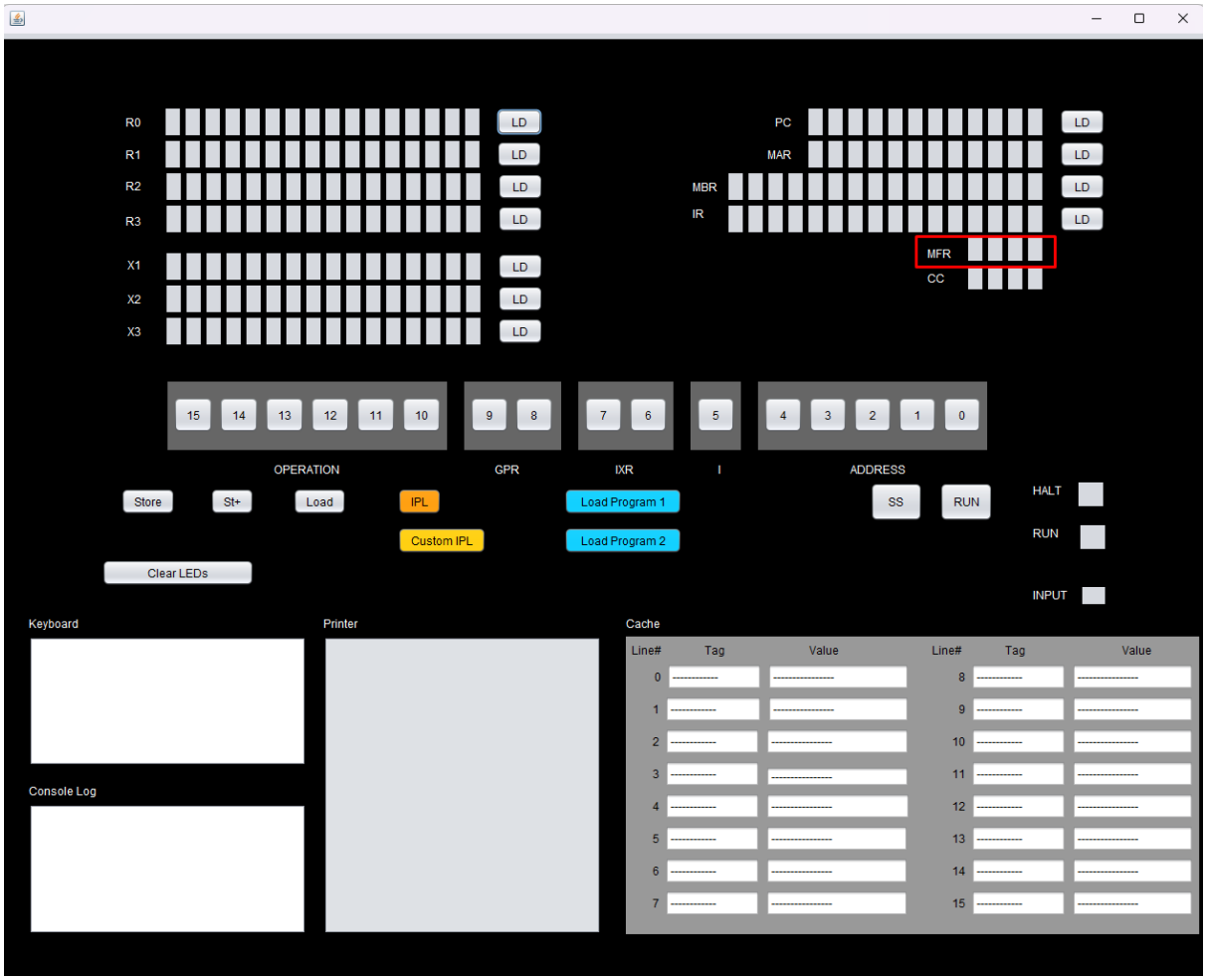
Following is the Reserved Locations:

| **Memory Address** | **Usage** |
|---|---|
| 0 | Reserved for the Trap instruction for Part III. |
| 1 | Reserved for a machine fault (see below). |
| 2 | Store PC for Trap |
| 3 | Not Used |
| 4 | Store PC for Machine Fault |
| 5 | Not Used |

There are checks added in code for below possible machine fault conditions:

| **ID** | **Fault** |
|---|---|
| 0 | Illegal Memory Address to Reserved Locations MFR set to binary 0001 |
| 1 | Illegal TRAP code MFR set to binary 0010 |
| 2 | Illegal Operation Code MFR set to 0100 |
| 3 | Illegal Memory Address beyond 2048 (memory installed) MFR set to binary 1000 |

On any machine fault condition, the corresponding MFR bit will glow:

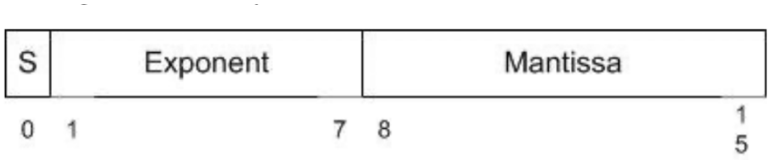


# Phase 4

**Floating point/Vector Operations:**

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:



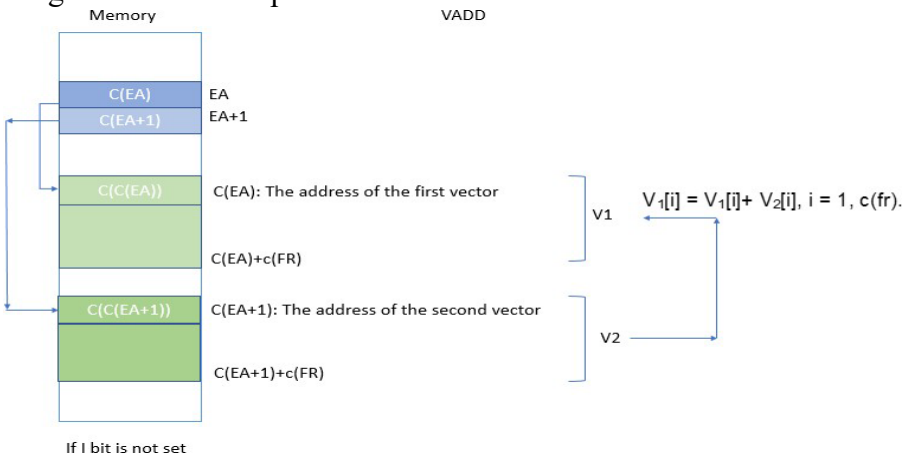

In the 16-bit binary, first bit denotes the sign of float value, second 8 bits denotes the exponent value and last 8 bits denotes the mantissa. Using sign, exponent and mantissa the actual float value can be evaluated.

| OpCode | Instruction | Description |
|---|---|---|
| 33 | FADD fr, x, address[,I] | Floating Add Memory To Register<br>c(fr) <- c(fr) + c(EA)<br>c(fr) <- c(fr) + c(c(EA)), if I bit setfr must be 0 or 1.<br>OVERFLOW may be set |
| 34 | FSUB fr, x, address[,I] | Floating Subtract Memory From Register<br>c(fr) <- c(fr) - c(EA)<br>c(fr) <- c(fr) - c(c(EA)), if I bit setfr must be 0 or 1<br>UNDERFLOW may be set |

**VADD:**

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

The below diagram sows the implementation of VADD in the simulator.




## VSUB:

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

The diagram below shows the implementation of VSUB: