1. **What we did in project 2:**

In project 2, we implemented all instructions except Part IV. We also wrote the test1 program which can be executed on our simulator based on our instruction set.

1. **Which instructions we implemented:**

We implemented the following instructions:

**opcode Instruction**

010(8) JZ r,x,address[,I]

011(9) JNE r,x,address[,I]

012(10) JCC cc,x,address[,I]

013(11) JMA x, address[,I]

014(12) JSR x, address[,I]

015(13) RFS Immed

016(14) SOB r,x,address[,I]

017(15) JGE r,x,address[,I]

004 AMR r,x, address[,I]

005 SMR r,x, address[,I]

006 AIR r, immed

007 SIR r, immed

020(16) MLT rx, ry

021(17) DVD rx, ry

022(18) TRR rx, ry

023(19) AND rx, ry

024(20) ORR rx, ry

025(21) NOT rx

031(25) SRC r, count, L/R, A/L

032(26) RRC r, count, L/R, A/L

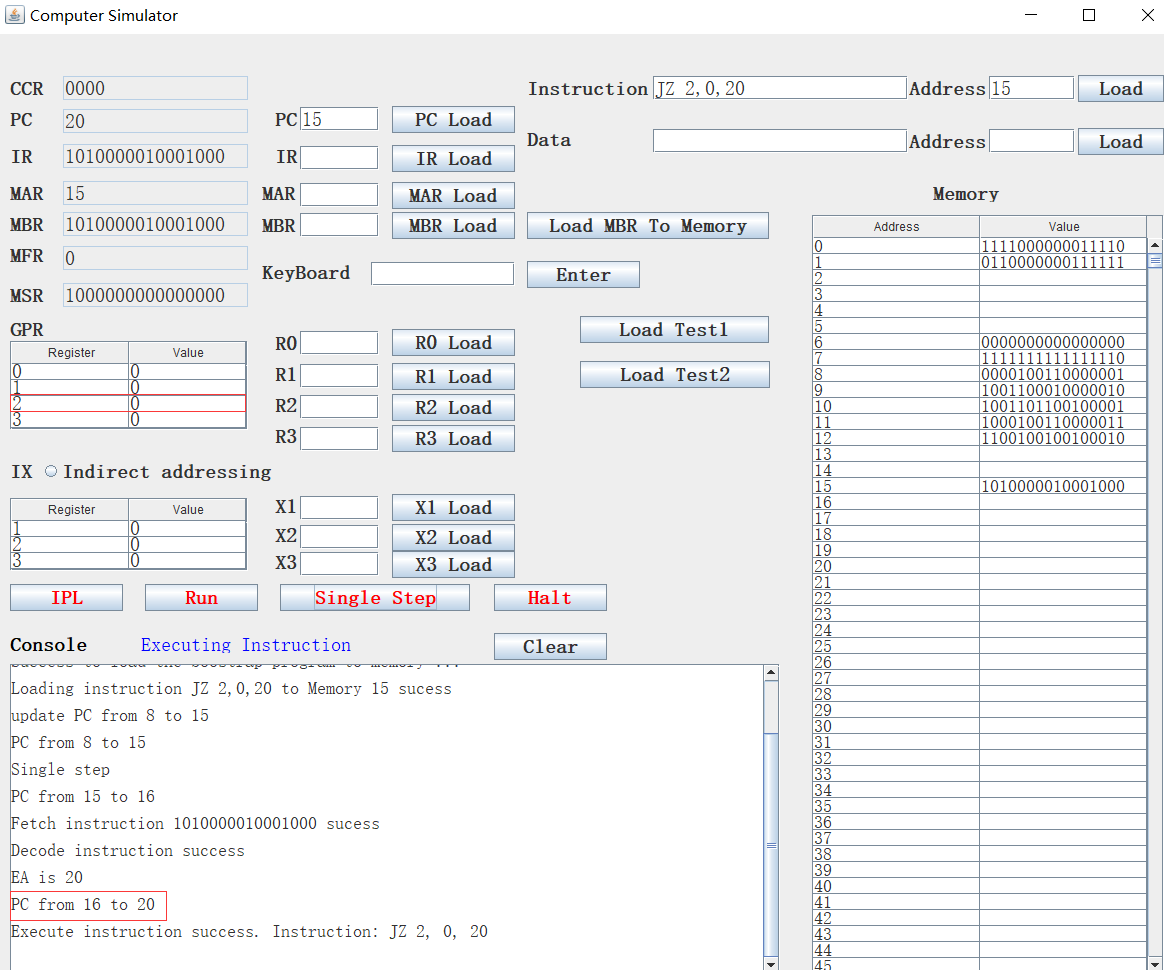
061(49) IN r, devid

062(50) OUT r, devid

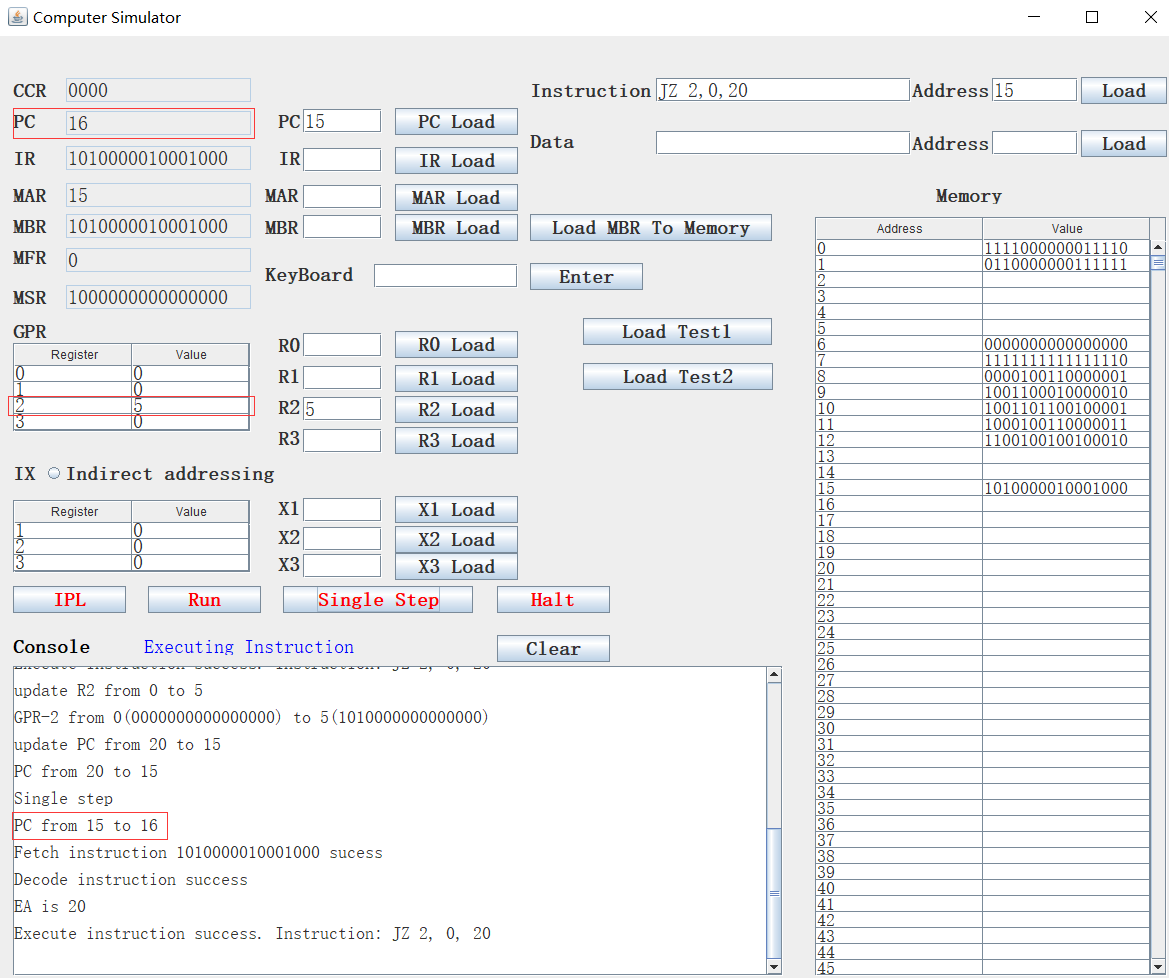
063(51) CHK, r, devid

1. JZ

JZ instruction’s format is JZ r, x, address[,I]. It will jump to EA if c(r) is equal to 0, otherwise, it will go to the next instruction. For example, we execute the instruction JZ 2,0, 20 in our simulator, R2’s content is 0. The EA is 20, so PC jumps from 16 to 20.

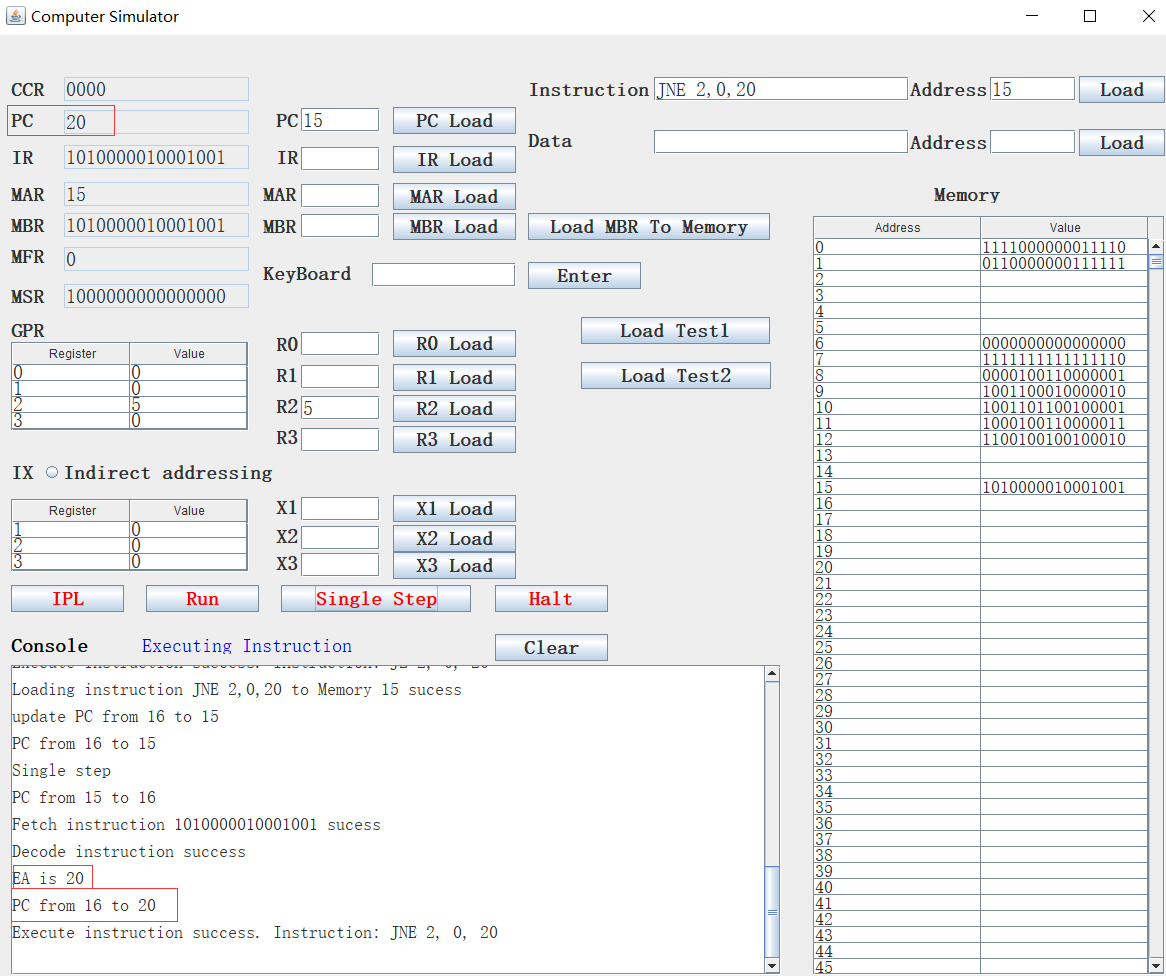


We input 5 to R2 and re-execute it again, PC from 15 to 16.

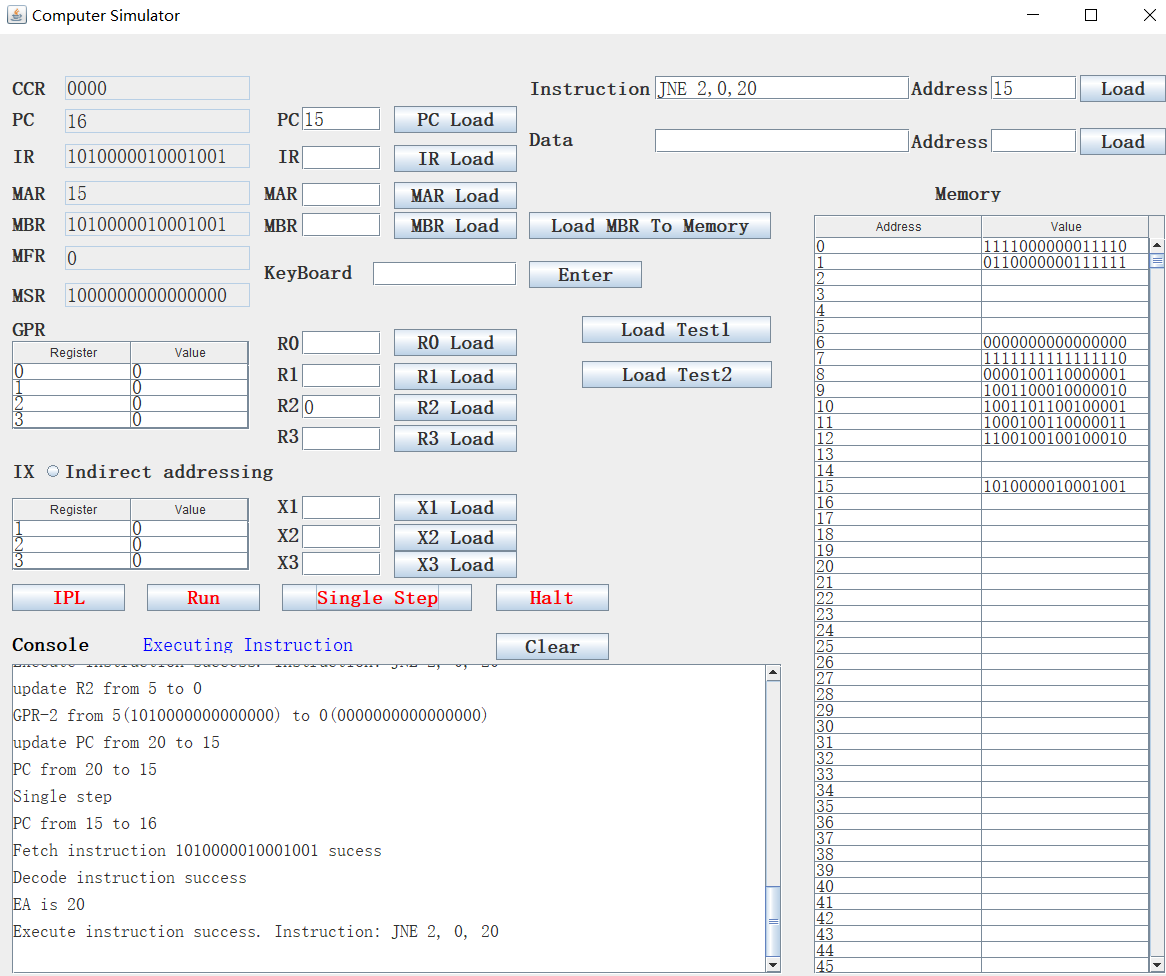


1. JNE

JNE instruction’s format is JNE r, x, address[,I]. it is opposite to JZ which means it will jump to EA if c(r) is not equal to 0, otherwise, it will go the next instruction. For example, we execute the instruction JNE 2, 0, 20 in our simulator, and we set R2 to 5, then PC jumps to 20 because R2 is not 0:



We set R2 to 0 and re-execute this instruction, this time PC jumps to 16:



1. DVD

DVD instruction’s format is DVD rx,ry.

Divide Register by Register

rx, rx+1 <- c(rx)/ c(ry)

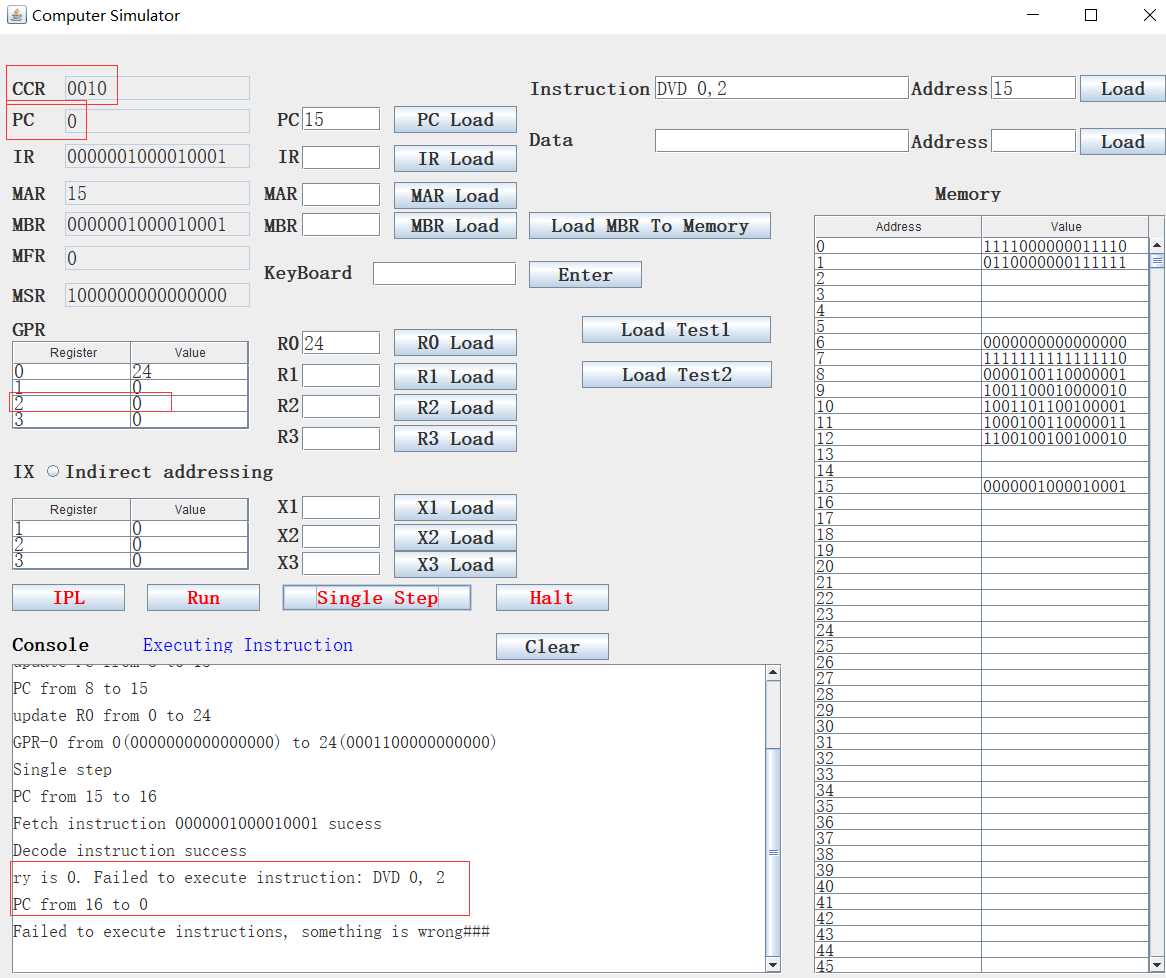
rx must be 0 or 2

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)

For example, we set R0 to 24, R2 to 0 and execute instruction DVD 0, 2. The console printer shows the DIVZERO error and the cc(3) was set to 1, meanwhile, PC was set to 0, where the TRAP instruction is located:



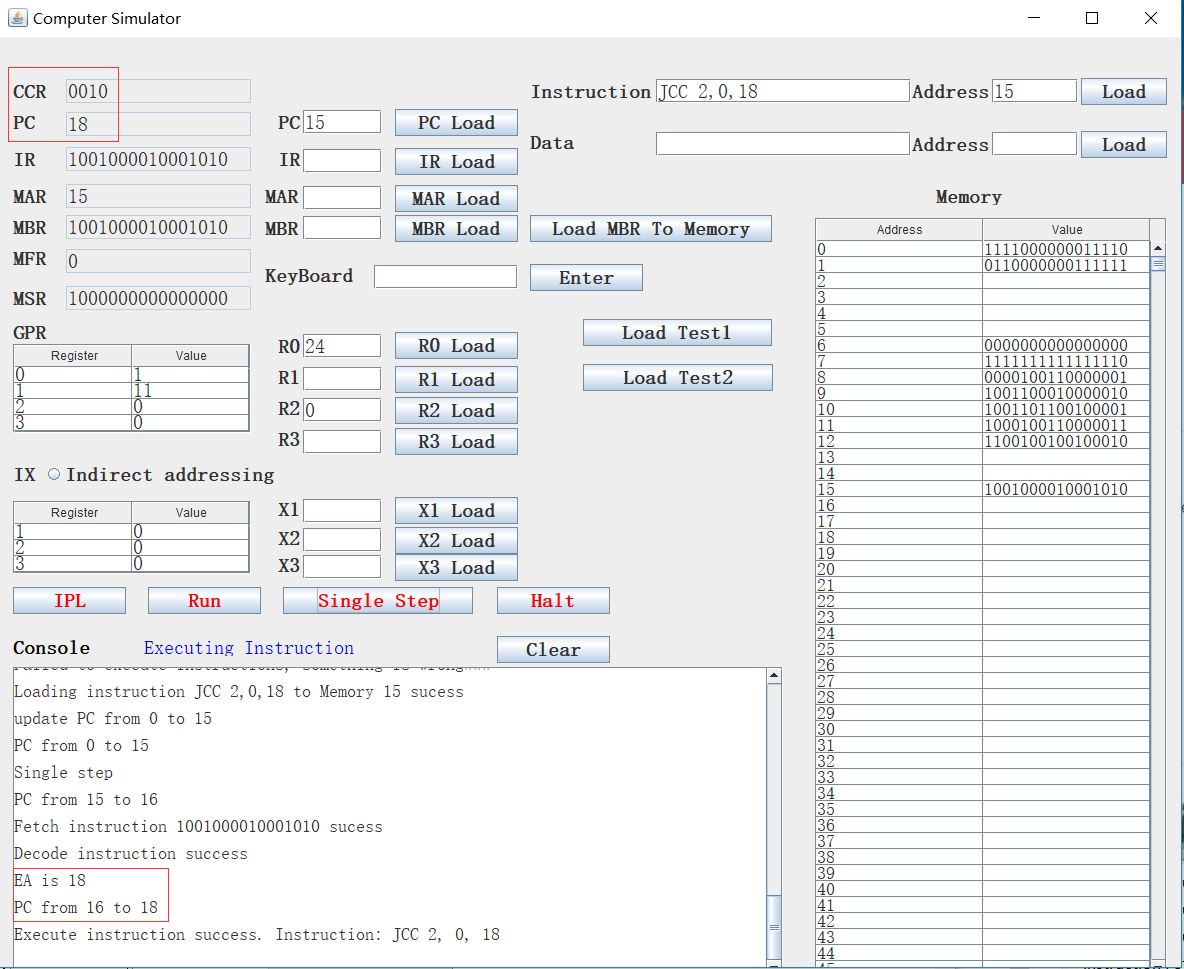
We set R2 to 13 and re-execute this instruction, then the remainder was kept to R1, and quotient was kept to R0.



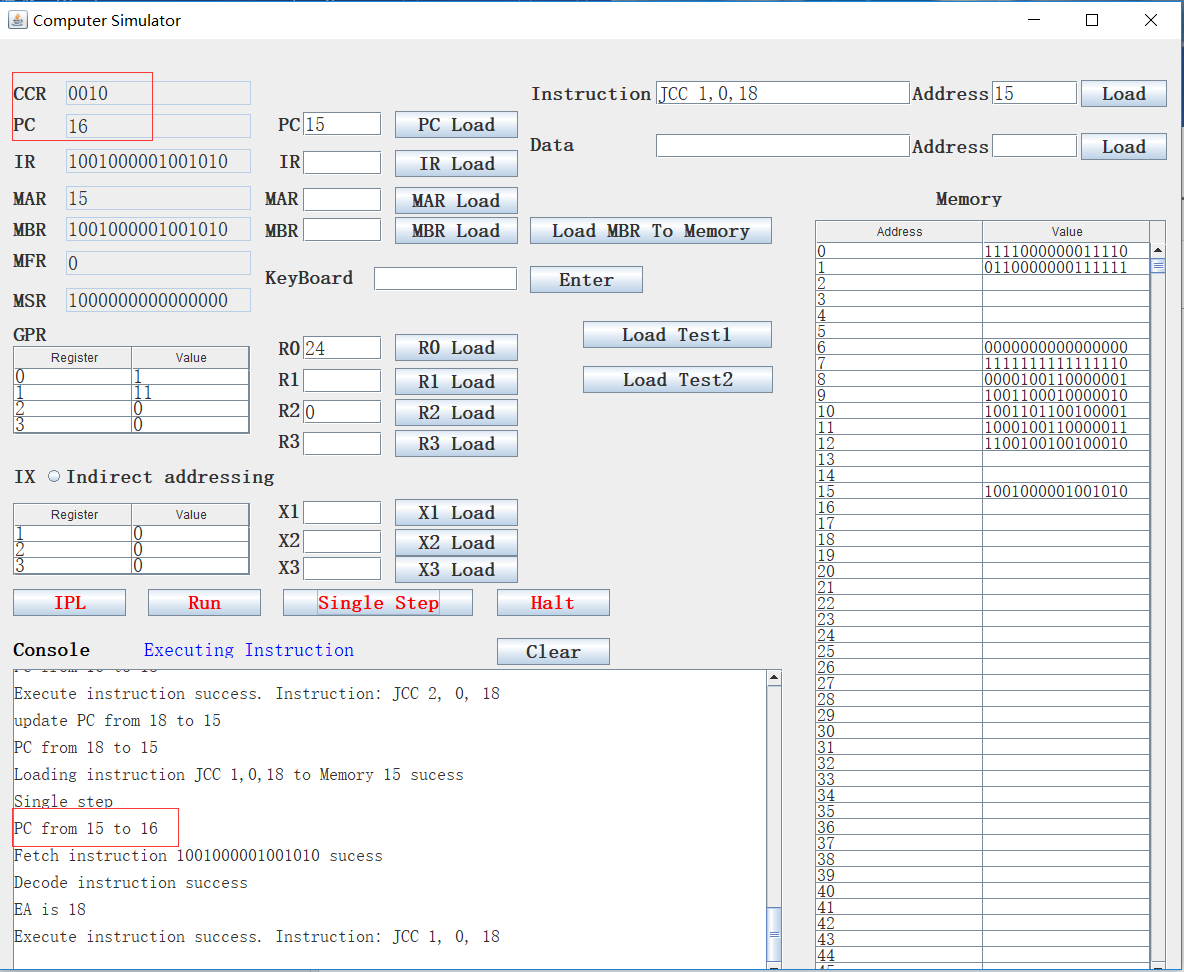
1. JCC

JCC instruction’s format is JCC cc, x, address[,I]. It will check the specified bit in the condition code register then decide to jump where. If cc bit = 1, PC  EA

Else PC <- PC + 1. For example, we execute the DVD 0,2 which cause the CC(3) to 1, then we execute the instruction JCC 2,0,18, PC jumps to 18.



We execute JCC 1,0,18, then PC jumps to 15:



1. JMA

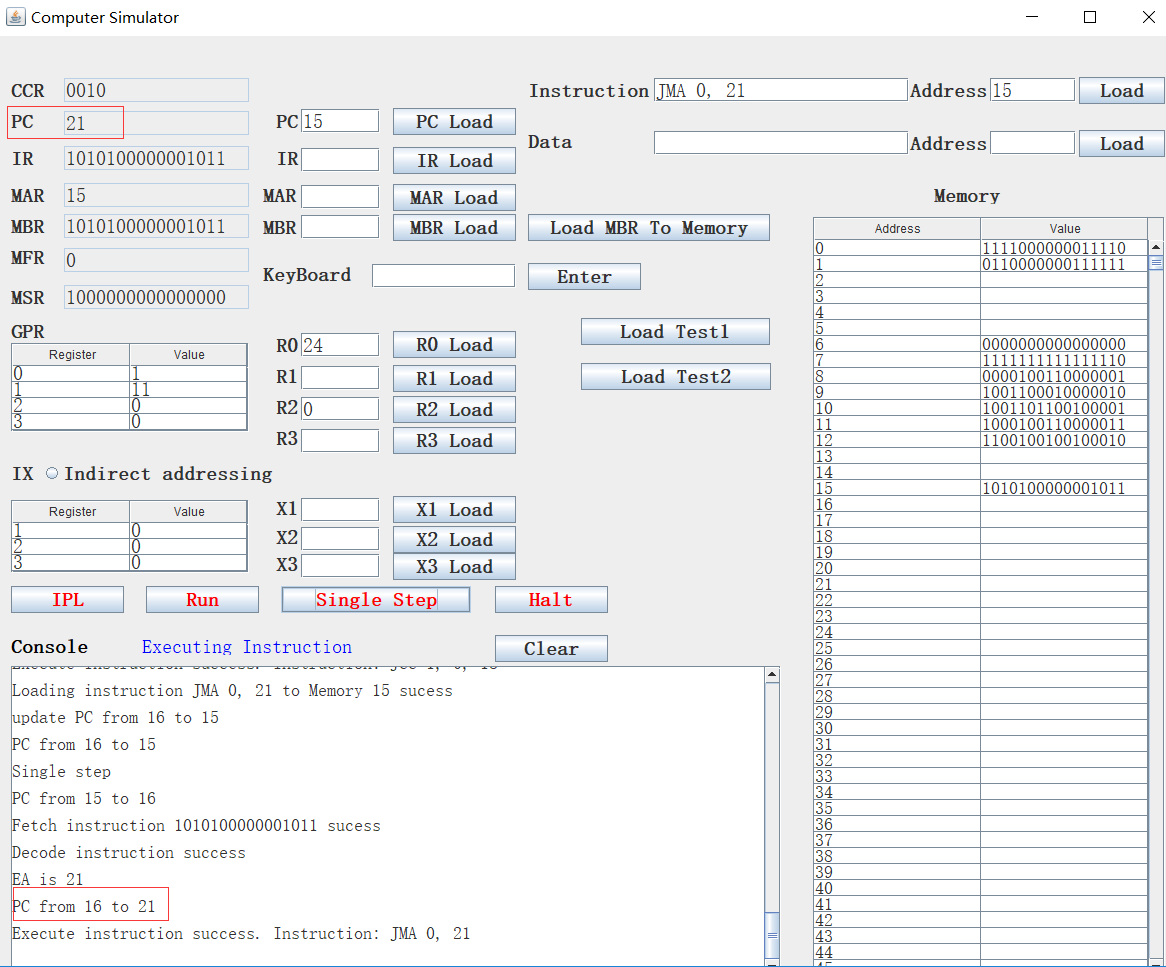
JMA instruction format is JMA x, address[,I]

Unconditional Jump To Address

PC <- EA,

Note: r is ignored in this instruction

For example, we execute JMA 0, 21 in our simulator, then PC jumps to 21 directly.



1. JSR

JSR instruction format is JSR x, address[,I]

Jump and Save Return Address:

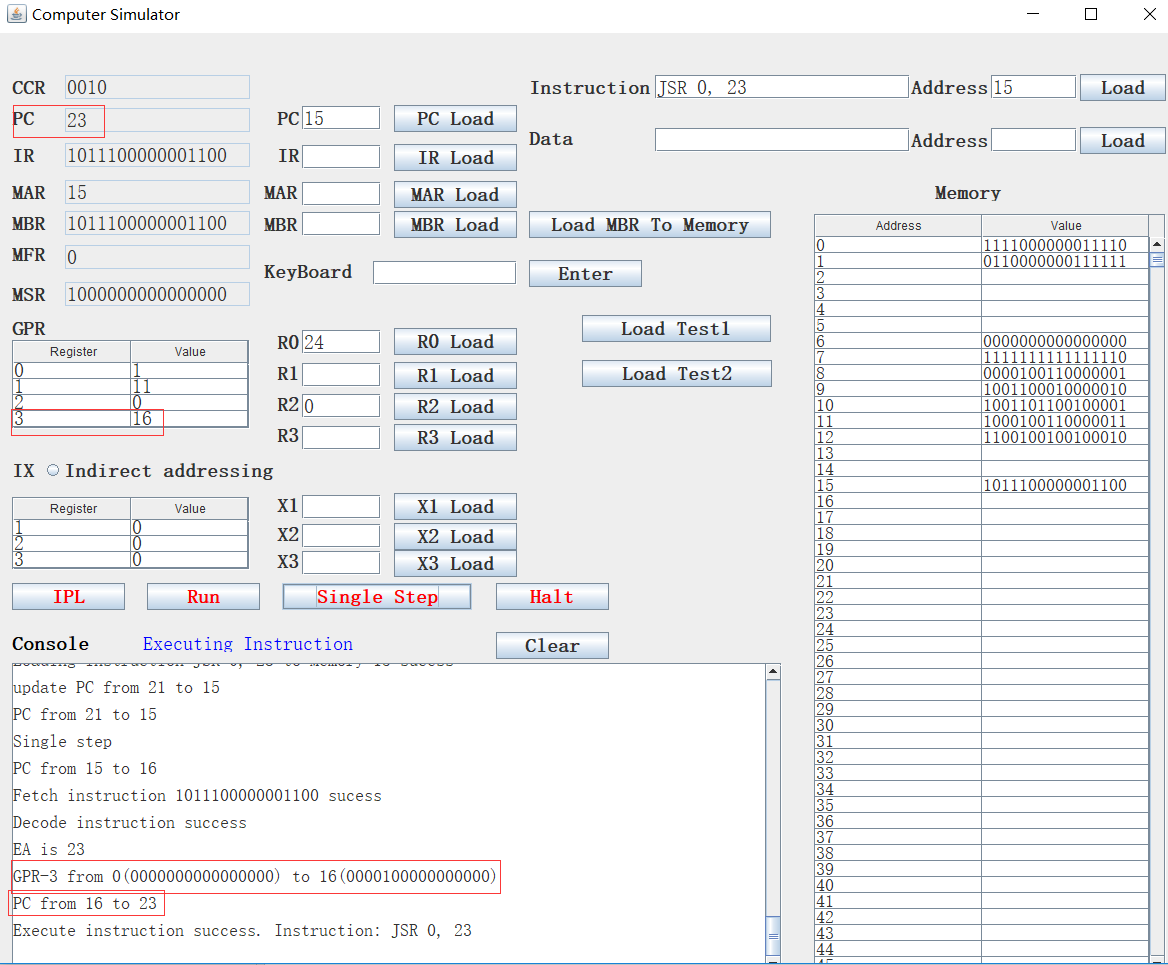
R3  PC+1;

PC  EA

R0 should contain pointer to arguments

Argument list should end with –1 (all 1s) value

For example, we execute JSR 0, 24, then PC +1, which is 16, is kept to R3, and PC jumps to 24.



1. RFS

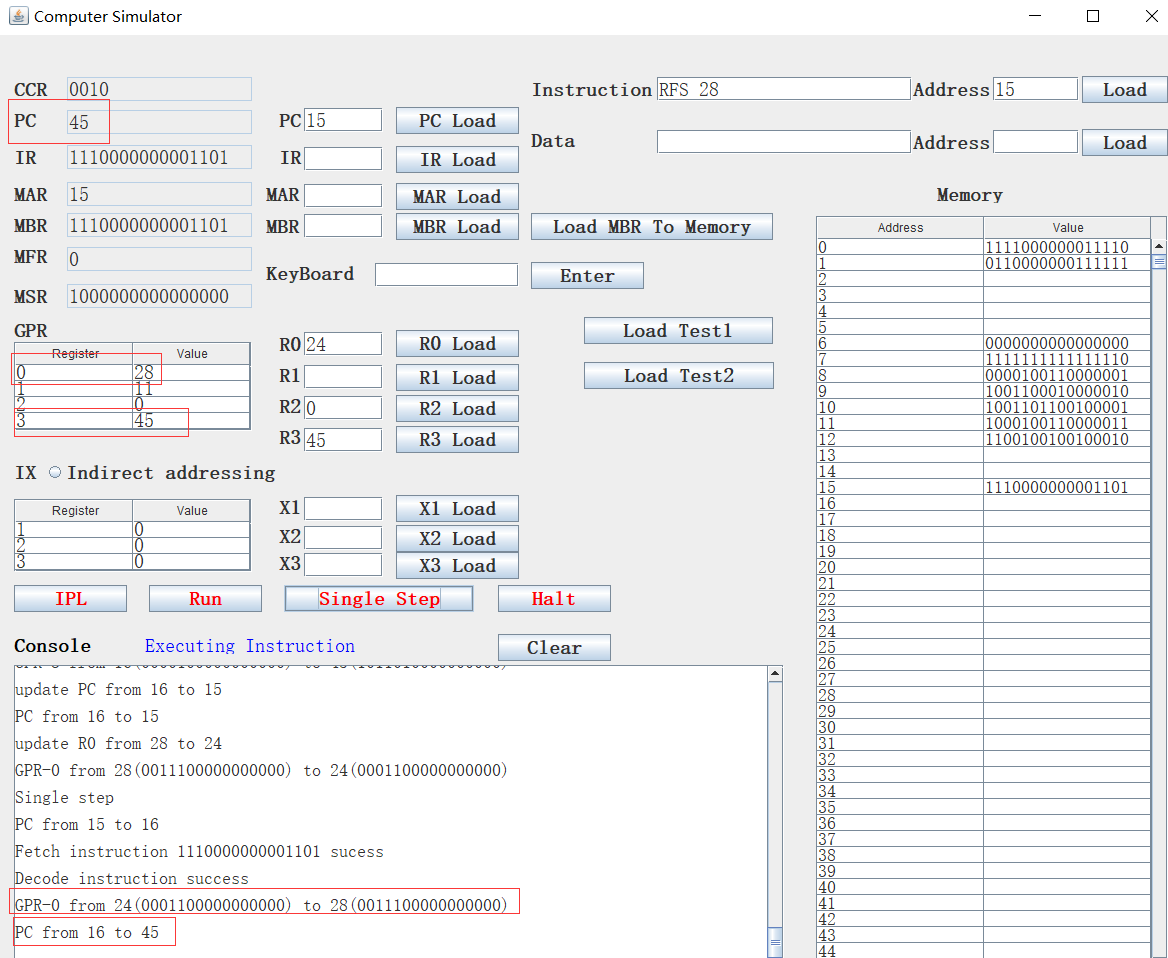
RFS instruction’s format is RFS Immed.

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.

For example, c(R3) is 45, we execute RFS 28, then the immediate number 28 is kept to R0, 45 is set to PC:



1. SOB

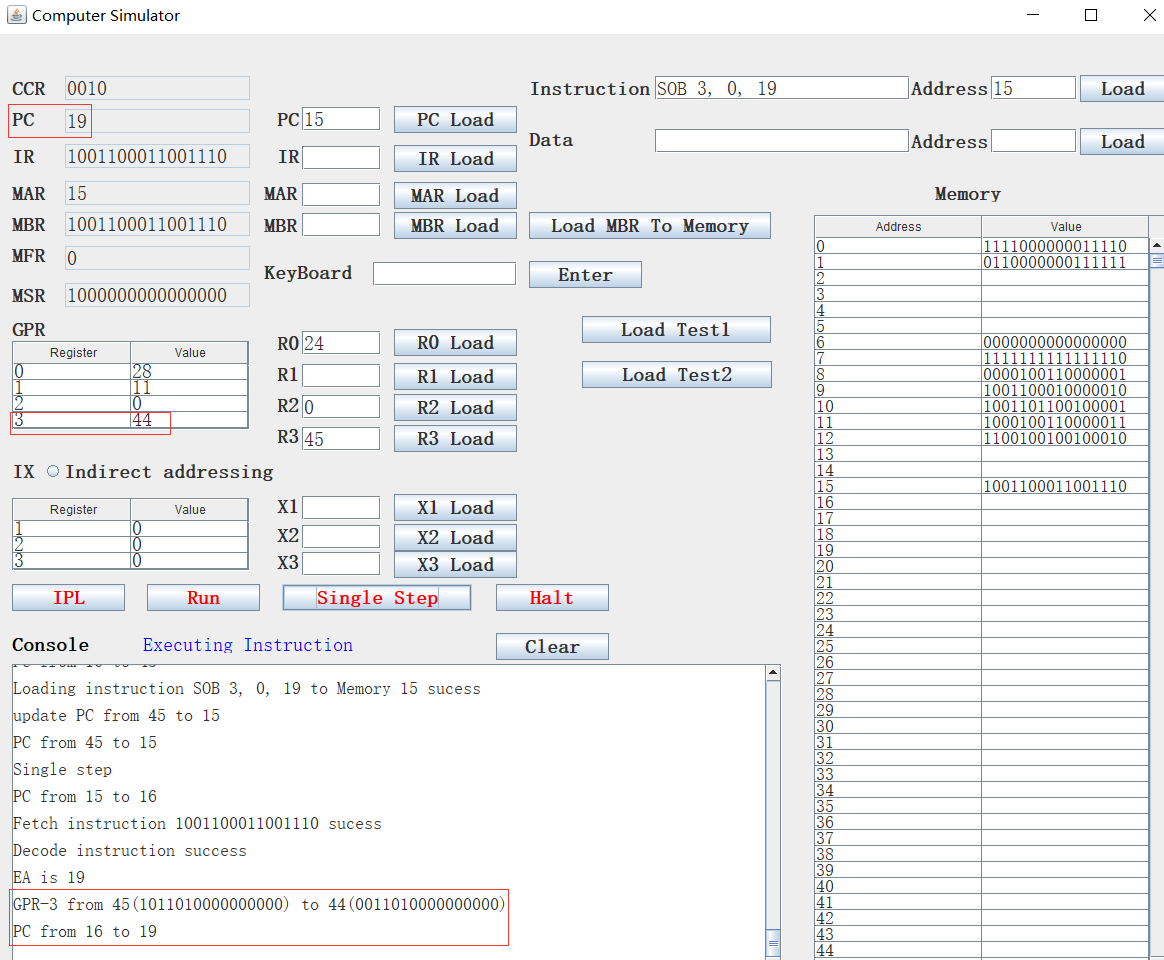
SOB instruction format is SOB r, x, address[,I].

Subtract One and Branch. R = 0..3

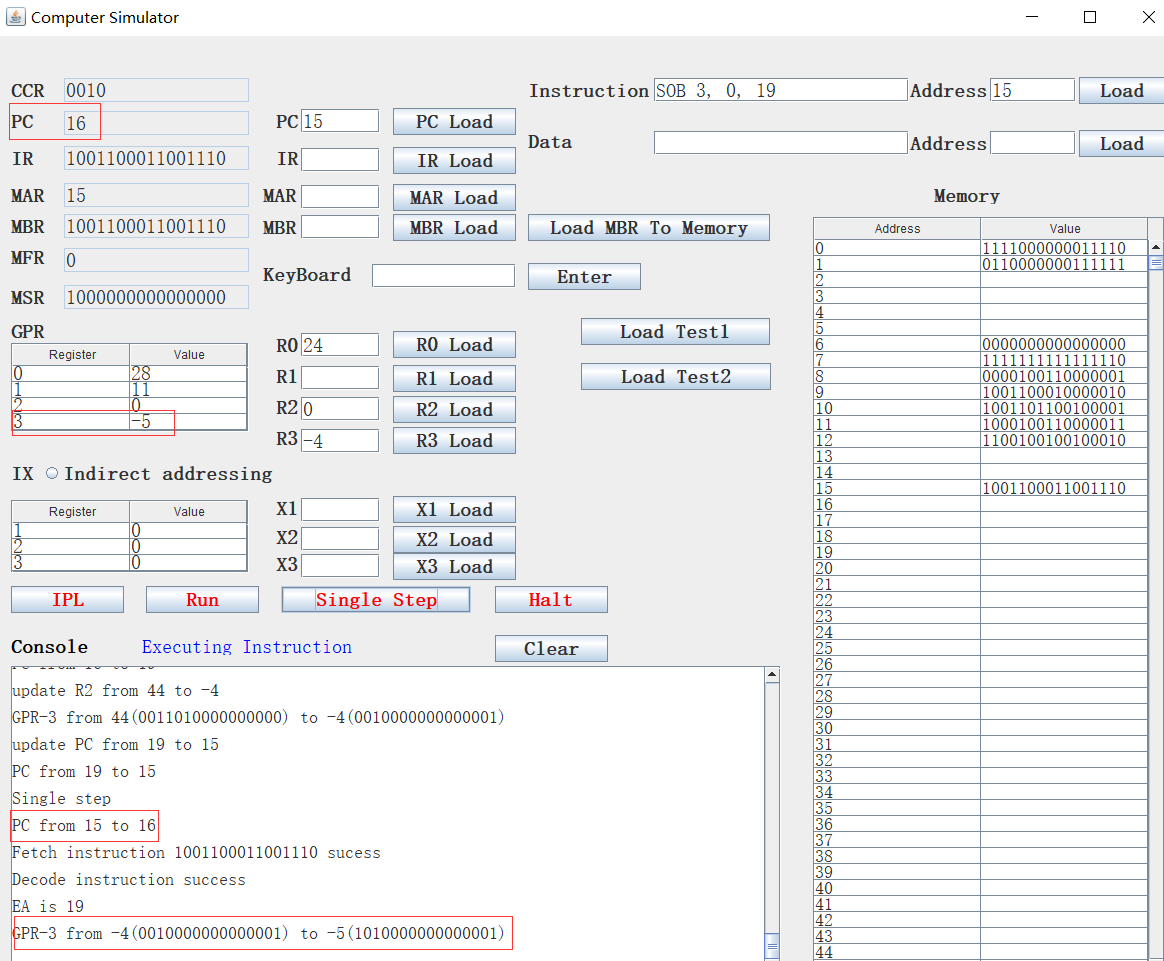
r  c(r) – 1. If c(r) > 0, PC <- EA;

Else PC <- PC + 1

For example, we set R3 to 45, which is larger than 0, then we execute SOB 3, 0, 19. We see R3 is set to c(R3) -1, which is 44. PC is set to EA, which is 19:



If we set R3 to -4, then re-execute SOB 3, 0, 19, we see R3 is set to c(R3) -1, which is -5, and PC is set to PC+1, which is 16.



1. JGE

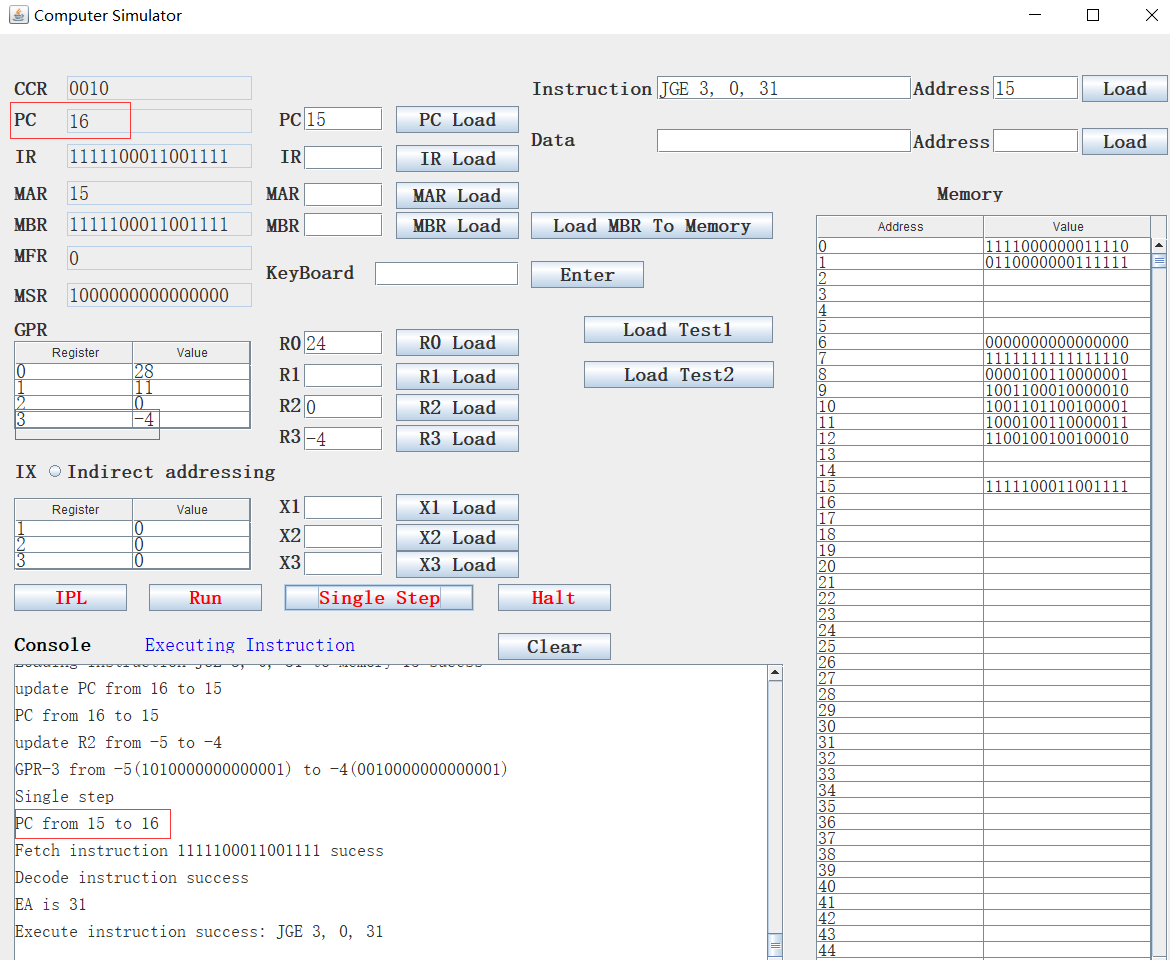
JGE instruction format is JGE r, x, address[,I].

Jump Greater Than or Equal To:

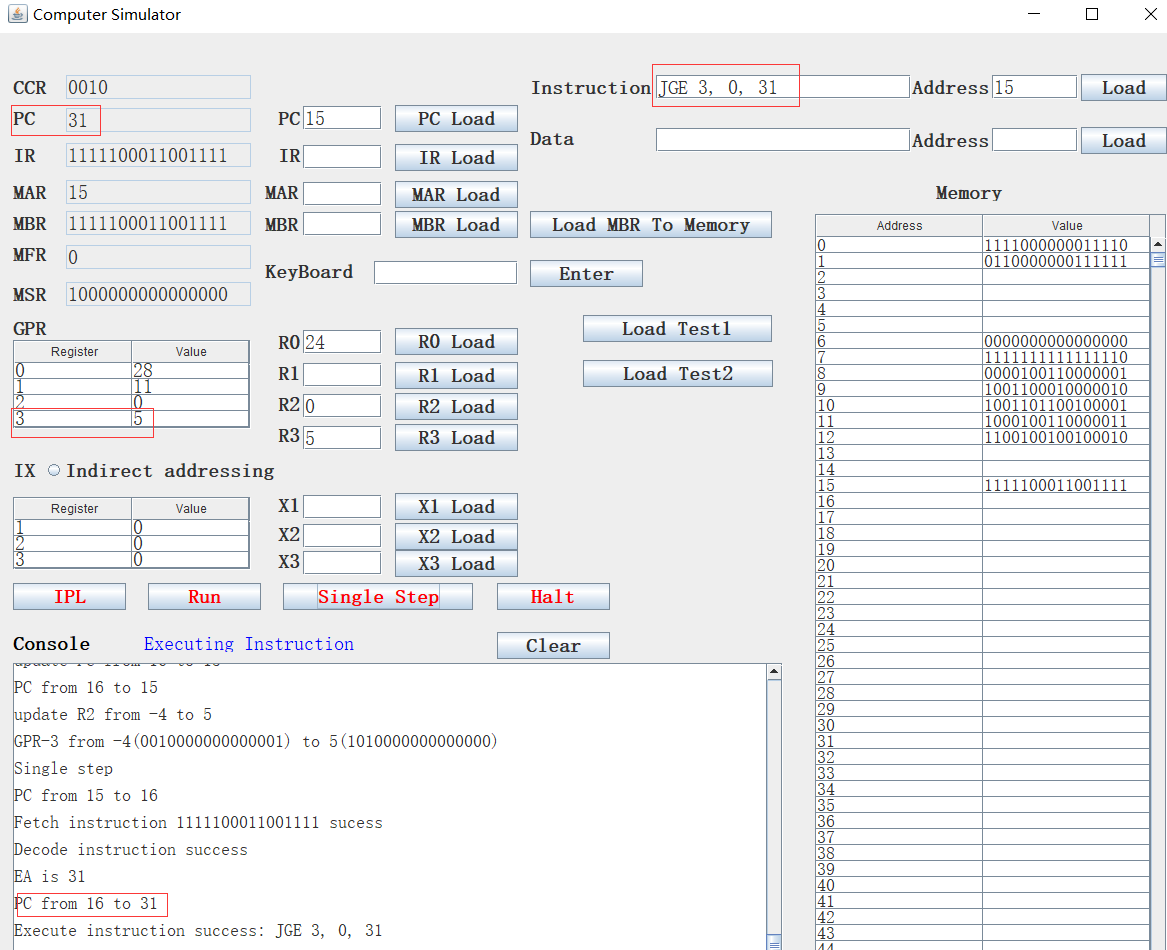
If c(r) >= 0, then PC <- EA

Else PC <- PC + 1

For example, we set R3 to -4, and execute JGE 3, 0, 31, since R3 < 0, so PC is set to PC + 1, which is 16:



If set R3 to 5 and re-execute JGE 3, 0, 31, PC is set to EA, which is 31, because R3 >= 0:



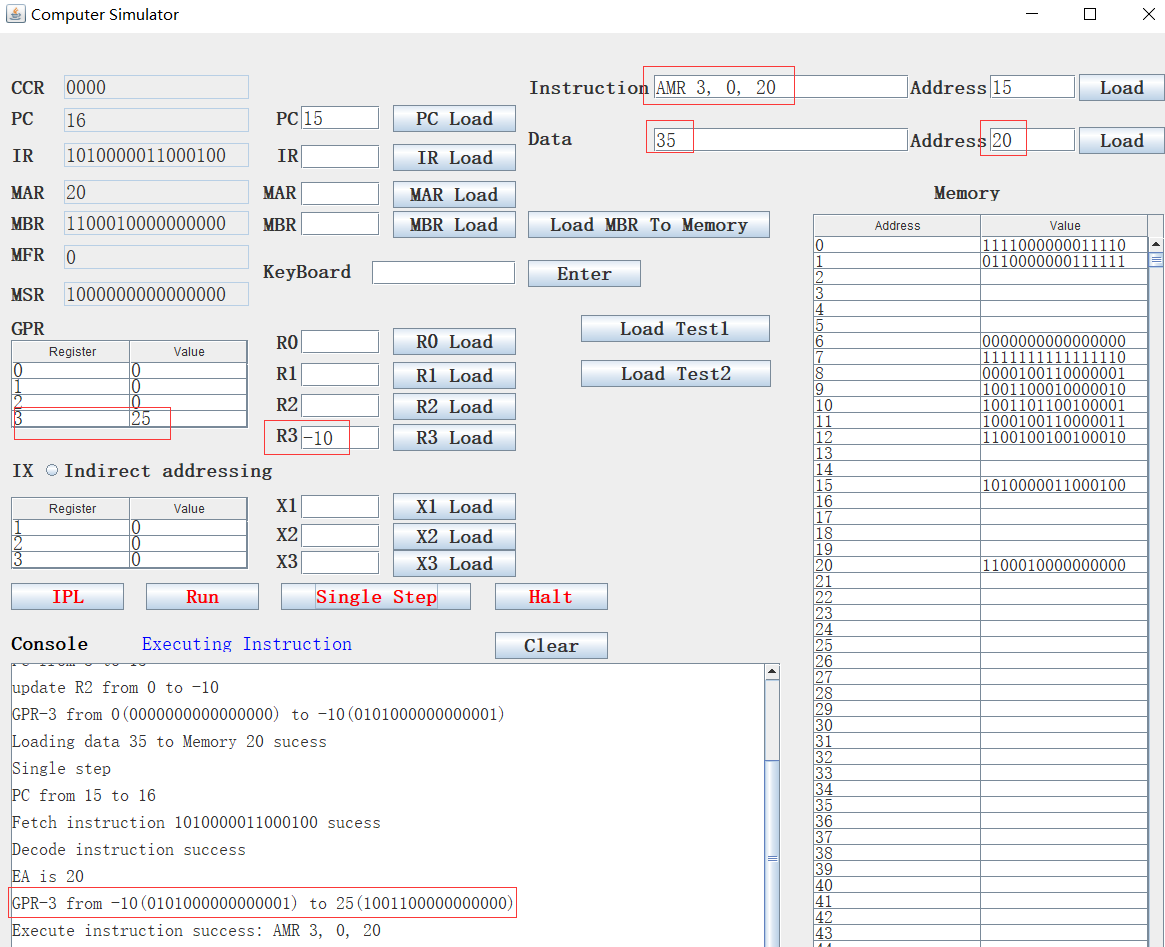
1. AMR

AMR instruction format is AMR r, x, address[,I].

Add Memory To Register, r = 0..3

r c(r) + c(EA)

we set M[20] = 35, c(R3) = -10, then we execute AMR 3, 0, 20, the R3 should be c(R3) + M[20], which is 25:



1. SMR

SMR instruction format is SMR r, x, address[,I].

Subtract Memory From Register, r = 0..3

r c(r) – c(EA)

For example, we set M[20] = 35, c(R3) = -10, then we execute SMR 3, 0, 20, the R3 should be c(R3) - M[20], which is -45:



1. AIR

AIR instruction format is AIR 3, immed.

Add Immediate to Register, r = 0..3

r  c(r) + Immed

Note:

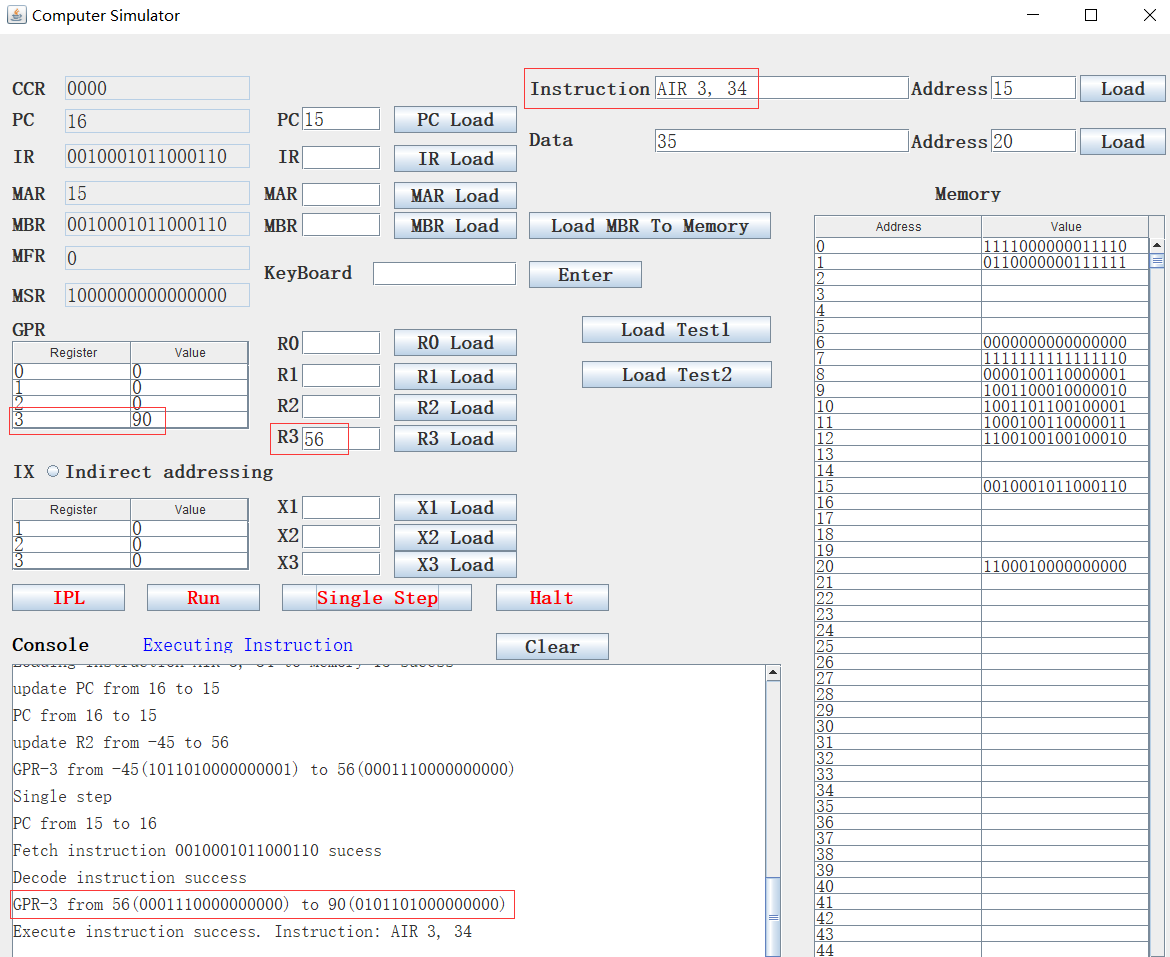
1. if Immed = 0, does nothing

2. if c(r) = 0, loads r with Immed

IX and I are ignored in this instruction

Since IX and I are ignored, we extend immediate to 8 bits, the highest bit is the sign bit, so the immediate range is [-127, 127].

For example, we set R3 to 56, then execute AIR 3, 34, the R3 is set to c(R3) + 34, which is 90:



1. SIR

SIR instruction format is SIR, r, immed

Subtract Immediate from Register, r = 0..3

r  c(r) - Immed

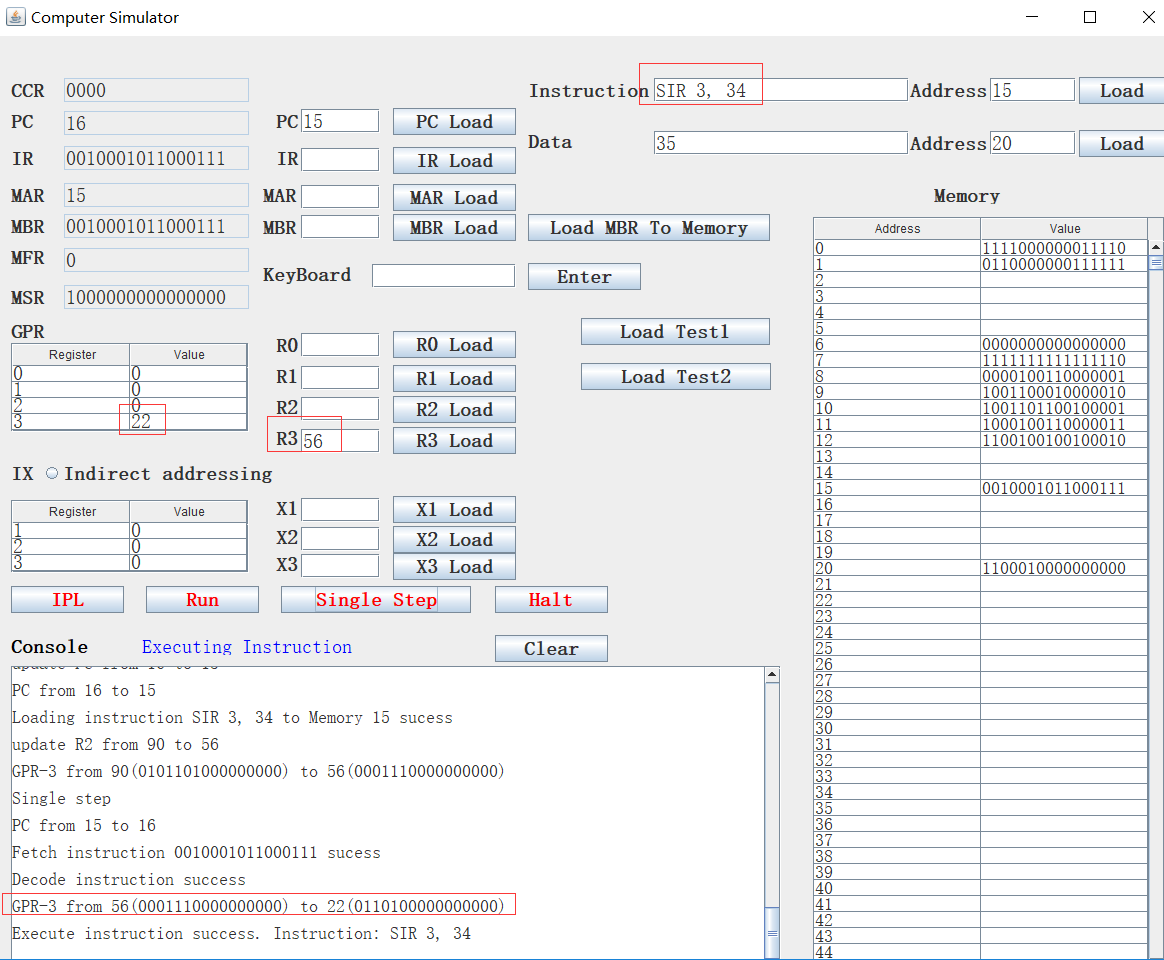
Note:

1. if Immed = 0, does nothing

2. if c(r) = 0, loads r1 with –(Immed)

IX and I are ignored in this instruction

For example, we set R3 to 56, then execute SIR 3, 34, the R3 is set to c(R3) - 34, which is 22



1. MLT

MLT instruction format is MLT rx, ry.

Multiply Register by Register

rx, rx+1 <- c(rx) \* c(ry)

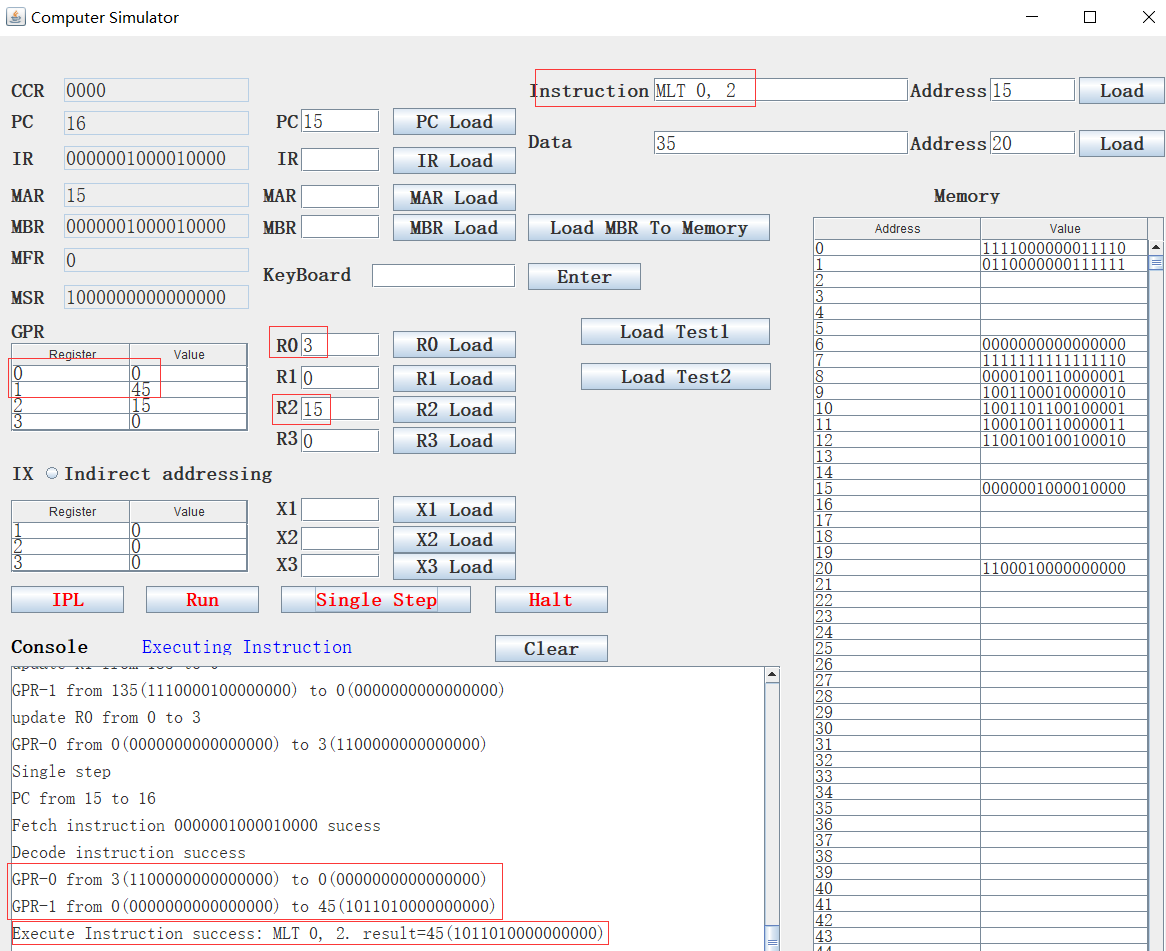
rx must be 0 or 2

ry must be 0 or 2

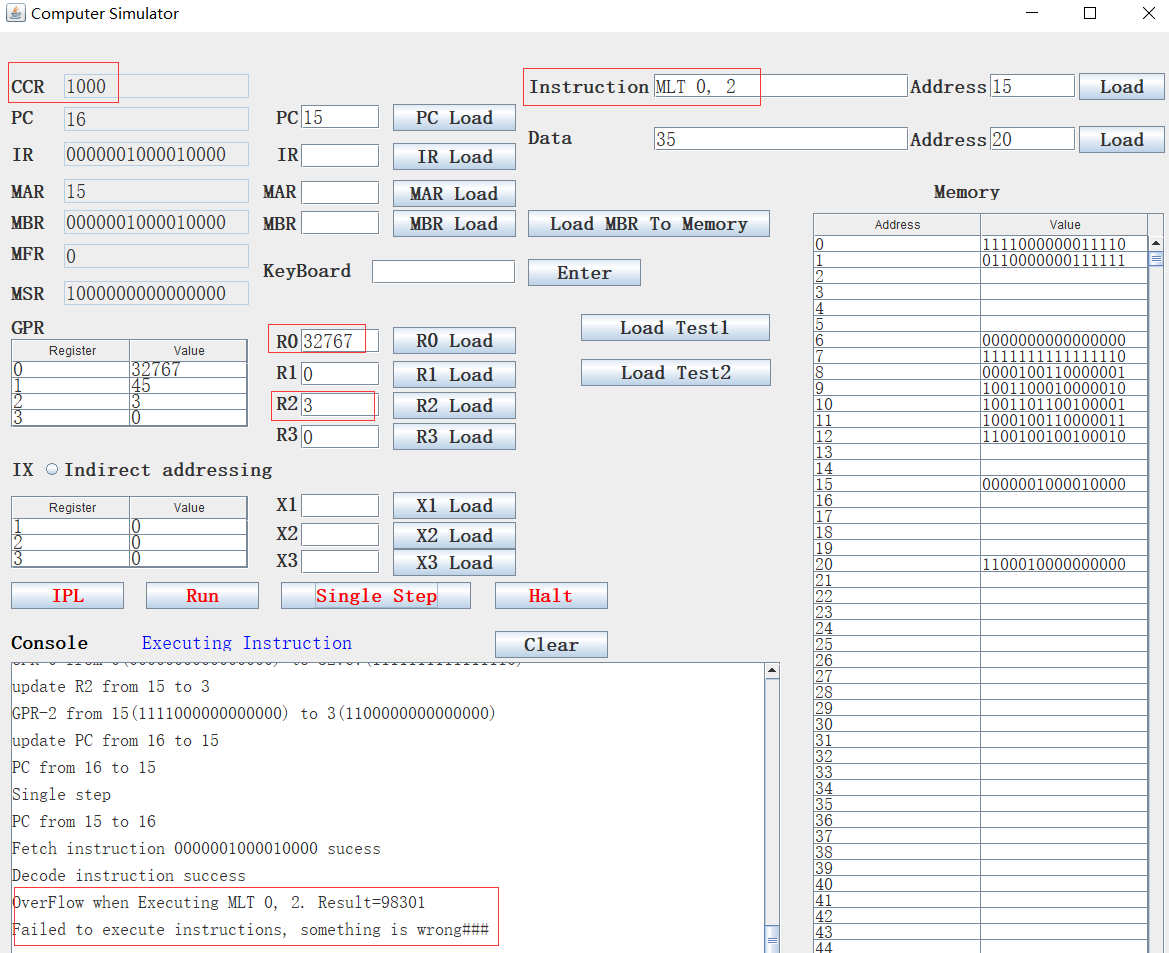
rx contains the high order bits, rx+1 contains the low order bits of the result

Set OVERFLOW flag, if overflow

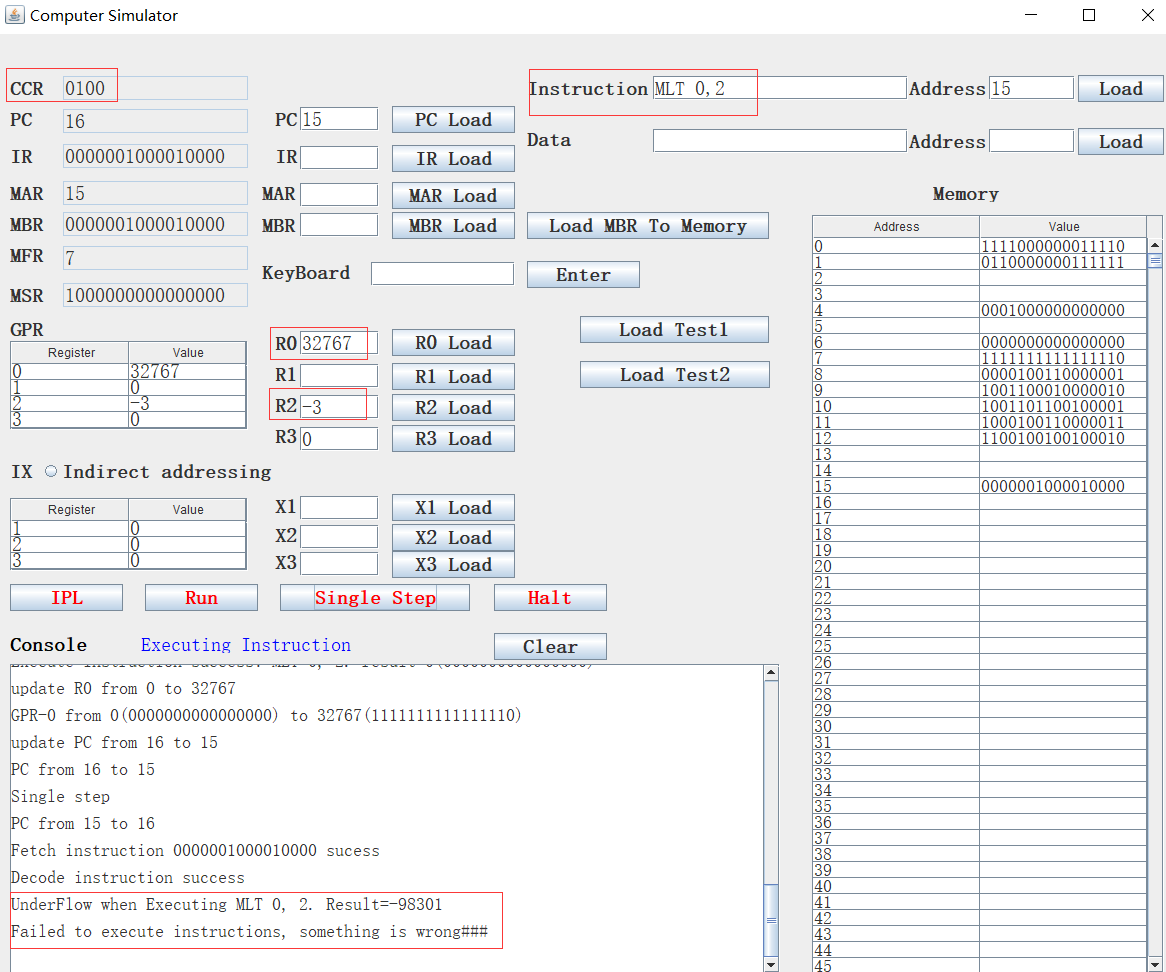
For example, we set R0 to 3, R2 to 15, then execute MLT 0, 2, R0 should be R0\*R2 = 45. R0 contains the high order bits, which is 00000000 , R2 contains the low order bits, which is 10110100:



If we set R0 to 32767, R2 to 3, the multiplication result will be overflow, then the CC(1) should be set to 1:



If we set R0 to 32767, R2 to -3, the multiplication result will be underflow, then the CC(2) should be set to 1:



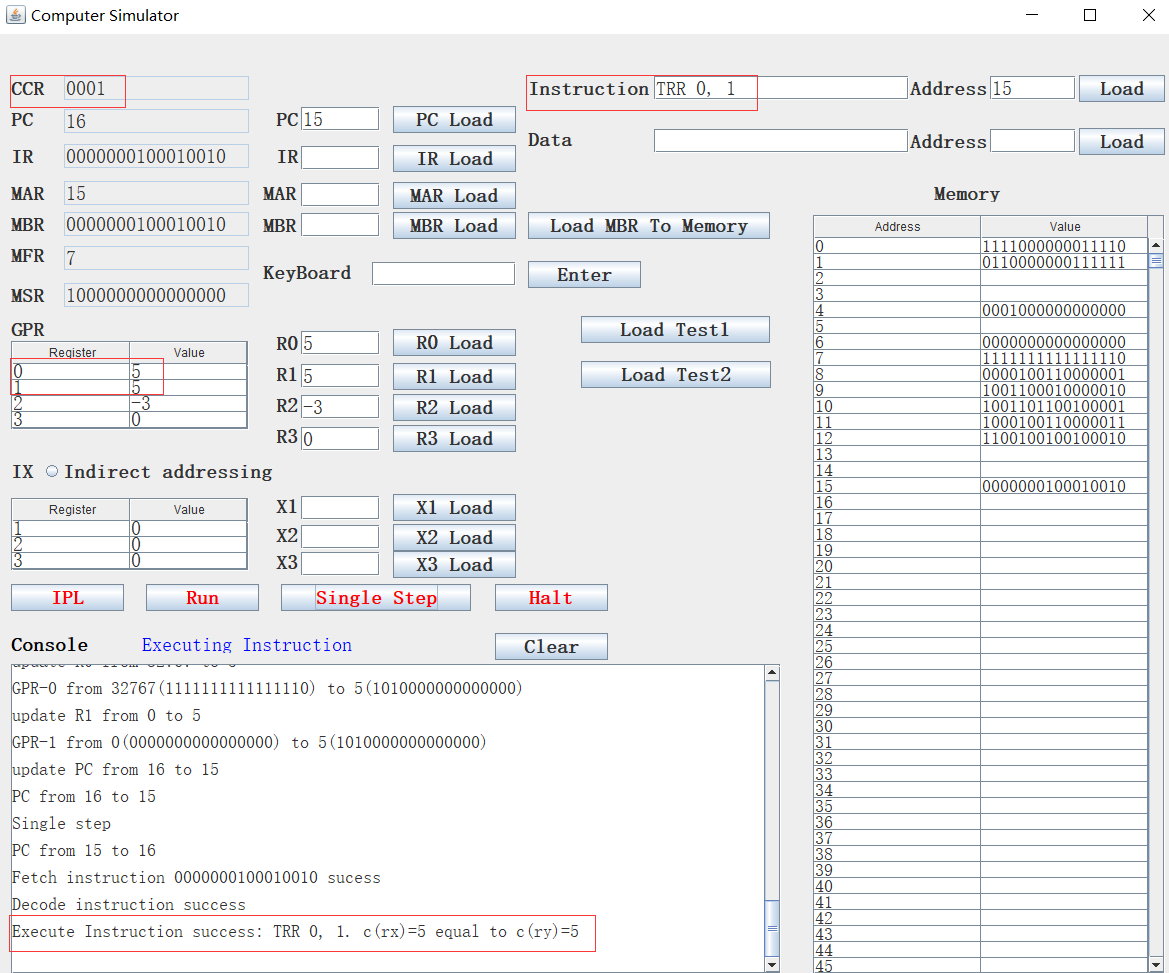
1. TRR

TRR instruction format is TRR rx, ry.

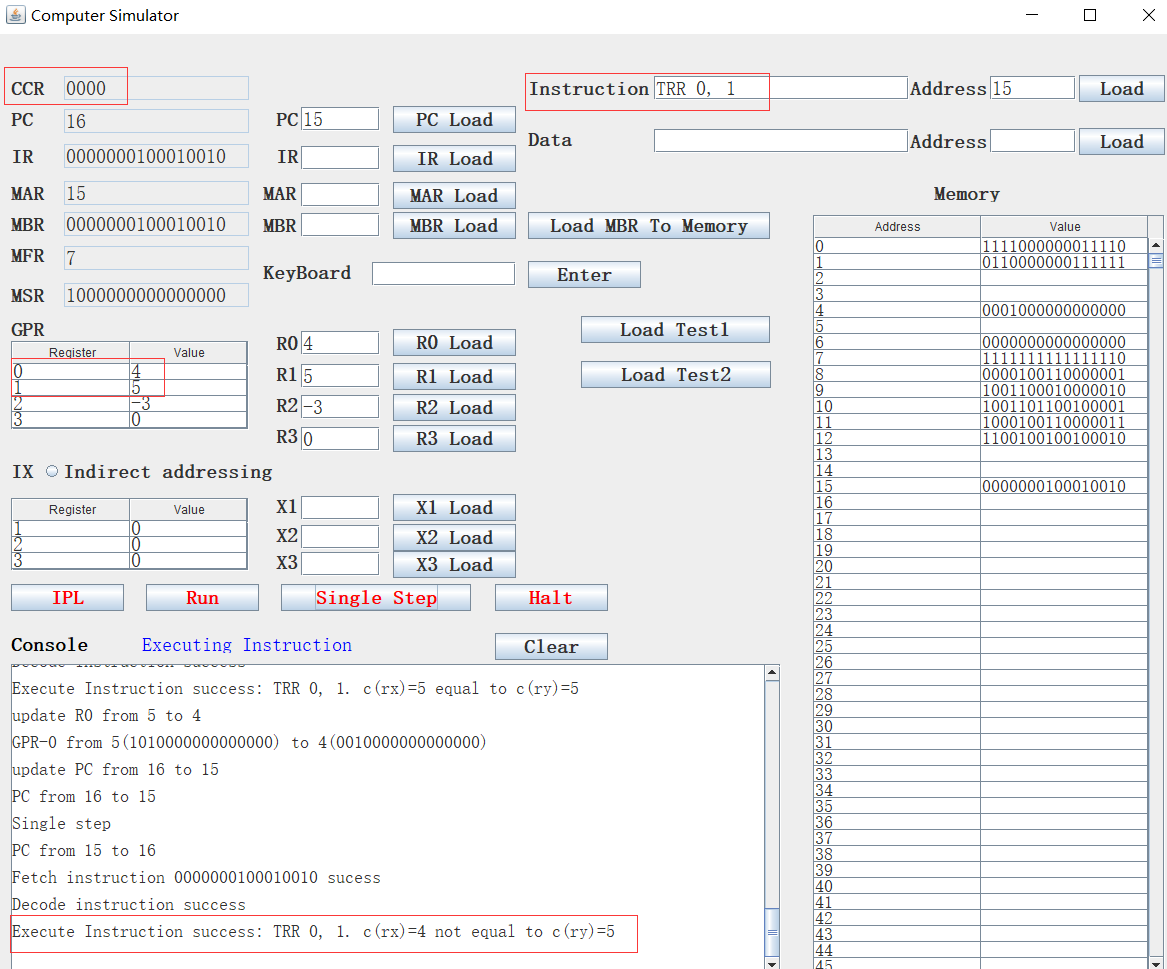
Test the Equality of Register and Register

If c(rx) = c(ry), set cc(4)  1; else, cc(4)  0

For example, we set R0 = R1 = 5, then execute TRR 0,1, CC(4) should be 1:



If we set R0 to 4, R1 to 5, and re-execute TRR 0, 1, then CC(4) should be 0:



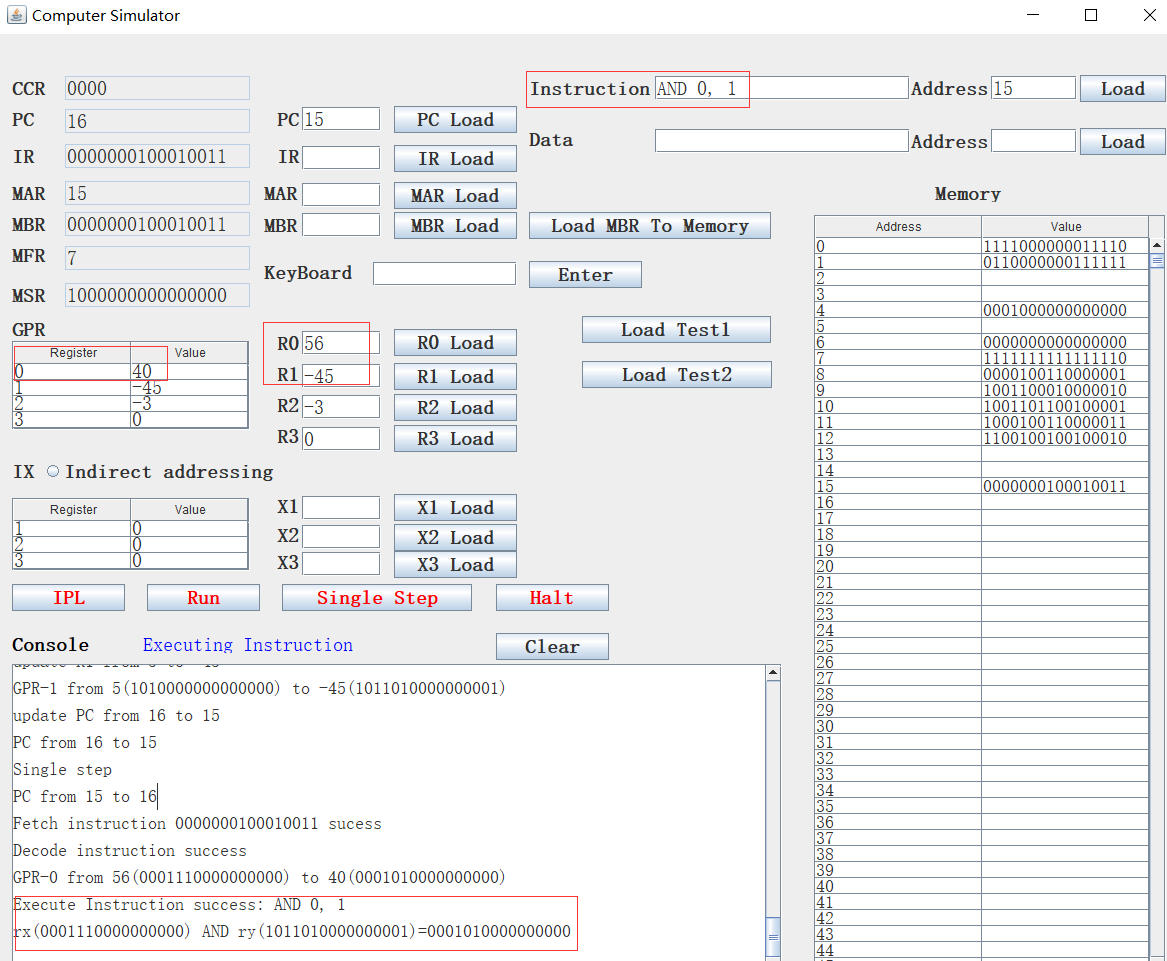
1. AND

AND instruction format is AND rx, ry.

Logical And of Register and Register

c(rx)  c(rx) AND c(ry)

For example, we set R0 to 56, which binary is 0001110000000000, R1 to -45, which binary is 1011010000000001, then execute AND 0,1, the result is 0001010000000000, which is 40.



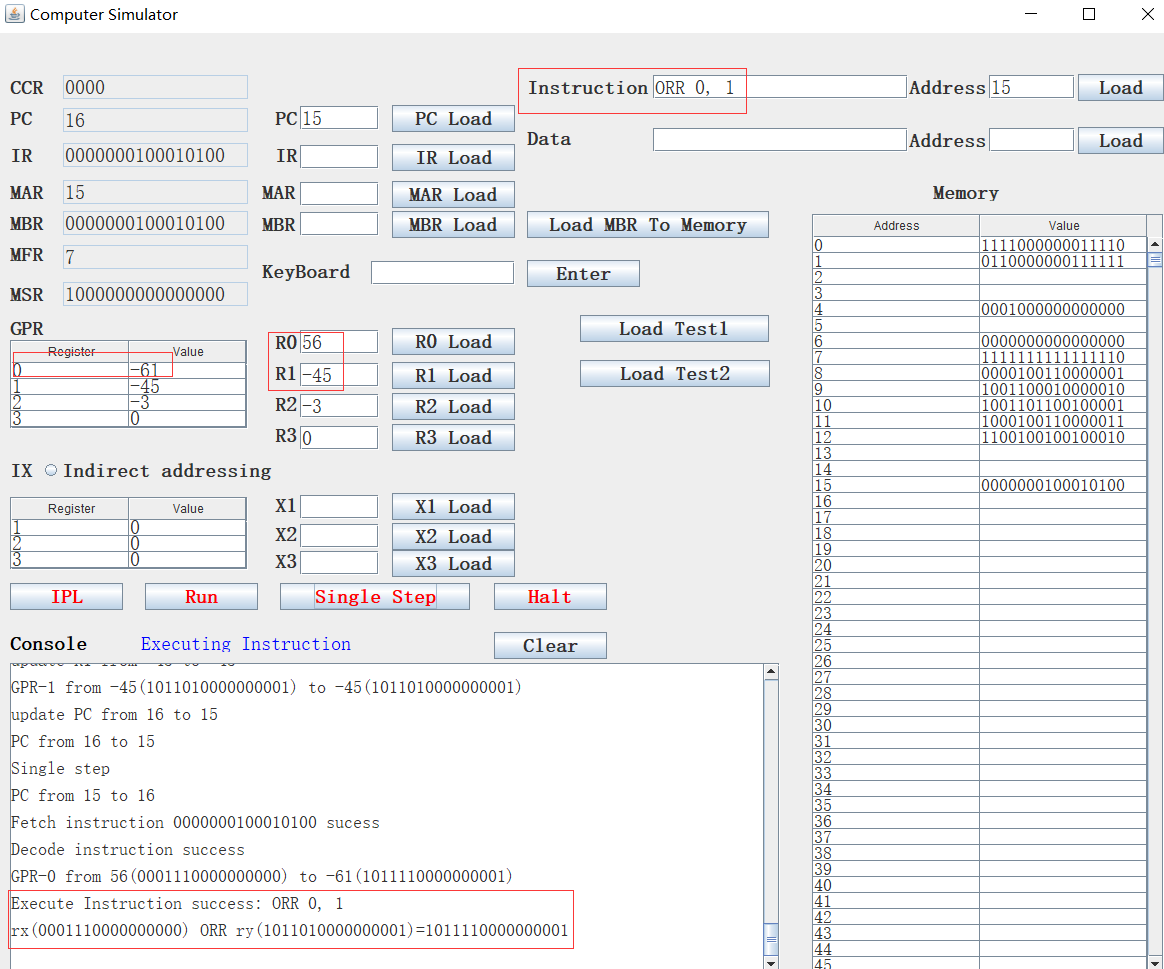
1. ORR

ORR instruction format is ORR rx, ry.

Logical Or of Register and Register

c(rx)  c(rx) OR c(ry)

For example, we set R0 to 56, which binary is 0001110000000000, R1 to -45, which binary is 1011010000000001, then execute ORR 0,1, the result is 1011110000000001, which is -61.



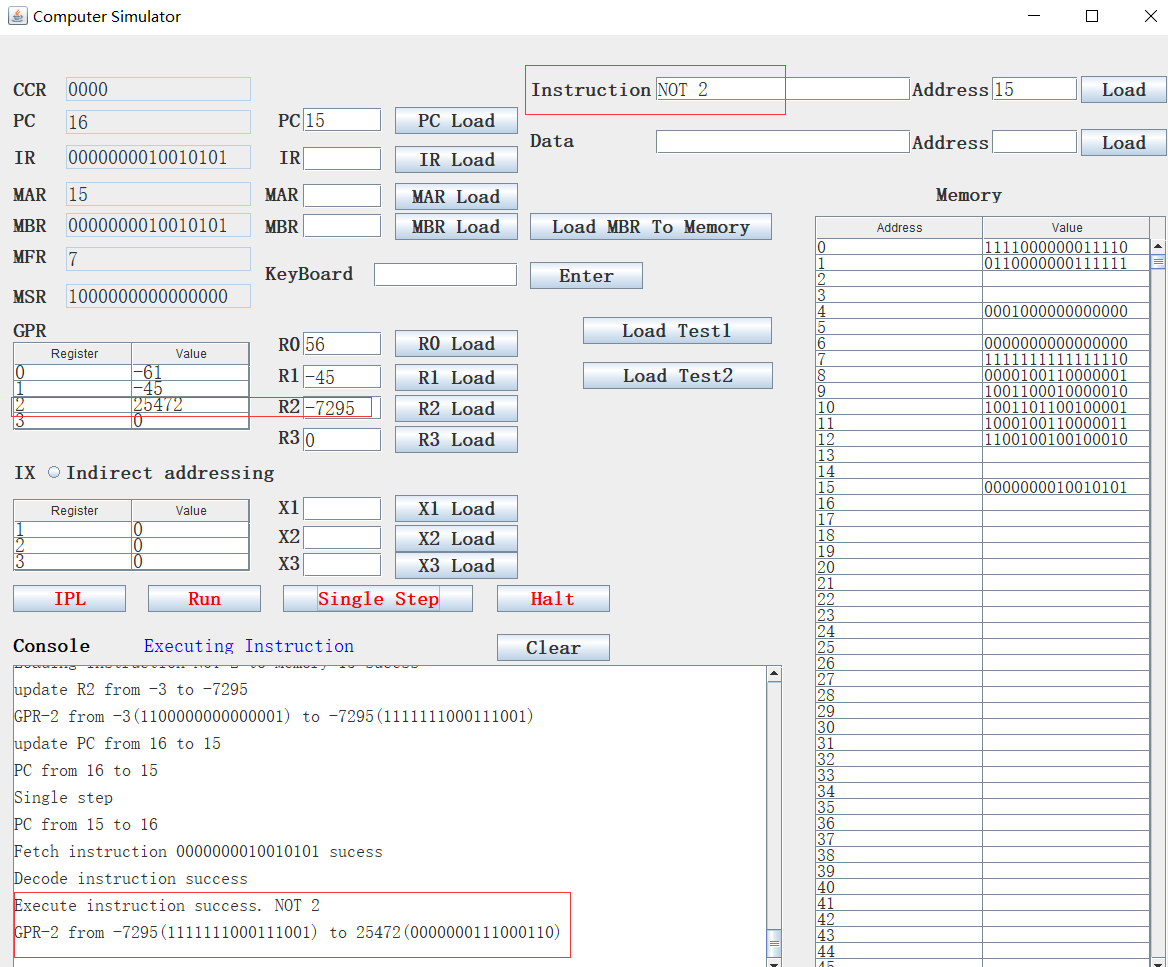
1. NOT

NOT instruction format is NOT rx.

Logical Not of Register To Register

C(rx)  NOT c(rx)

For example, we set R2 to -7295, then execute NOT 2, the result is 25472:



1. SRC

SRC instruction format is 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)

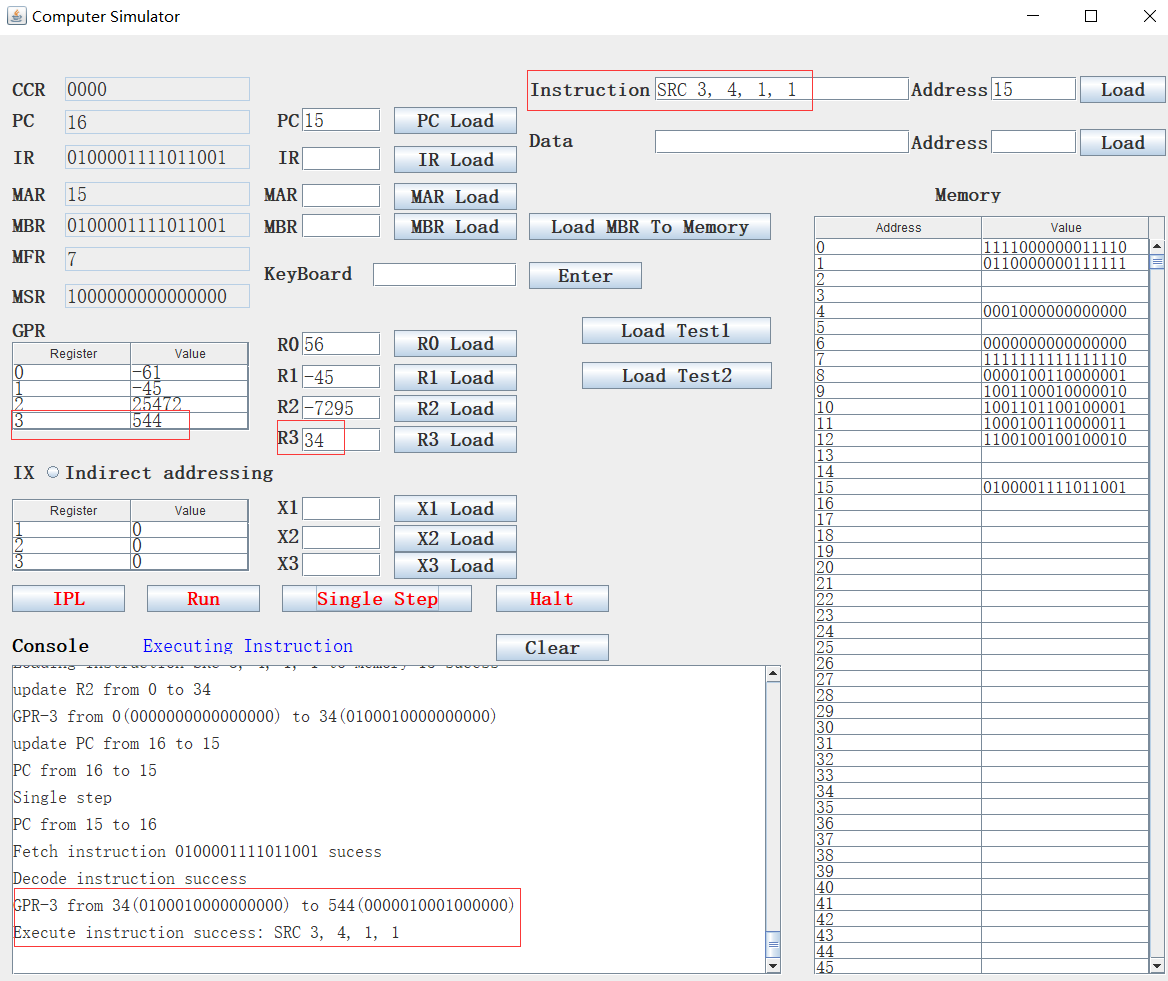
XX, XXX are ignored

Count = 0…15

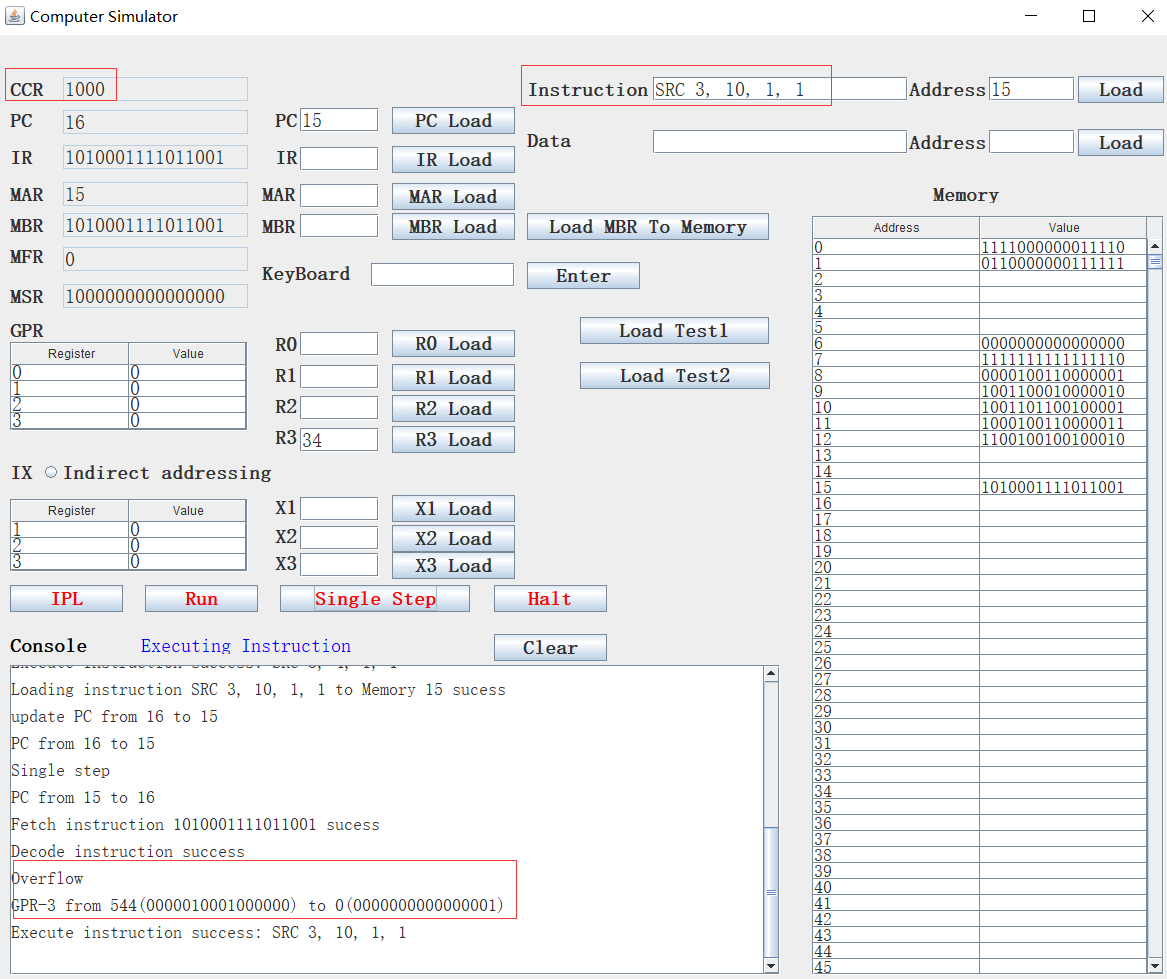
If Count = 0, no shift occurs

Logical left shifting

For example, we set R3 to 34, then execuate SRC 3, 4, 1, 1. This instruction is logical left shifting, left shifts 4bits. R3 should be c(R3) \* 2^4= 34\*16=544:

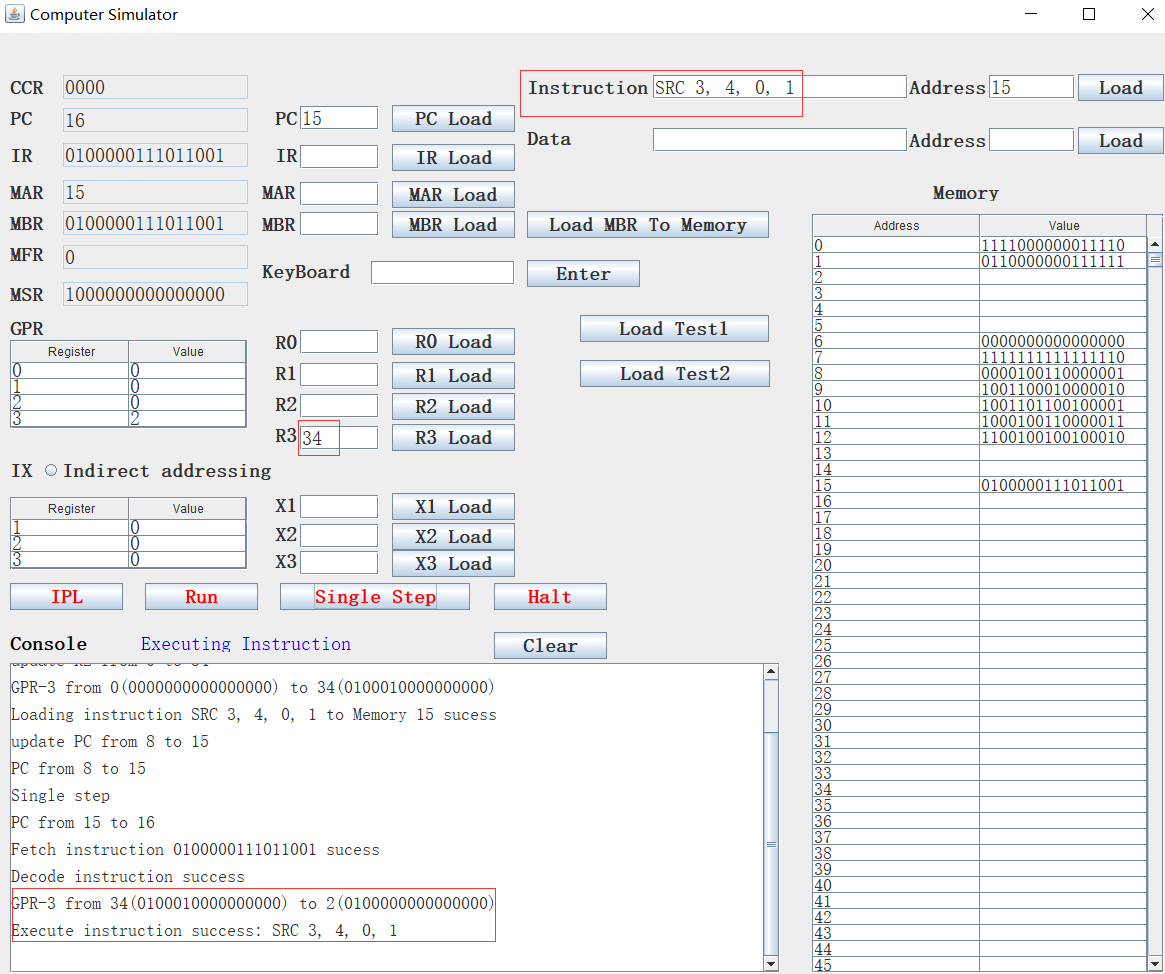


If we change count from 4 to 10, then the left shifting will cause overflow:



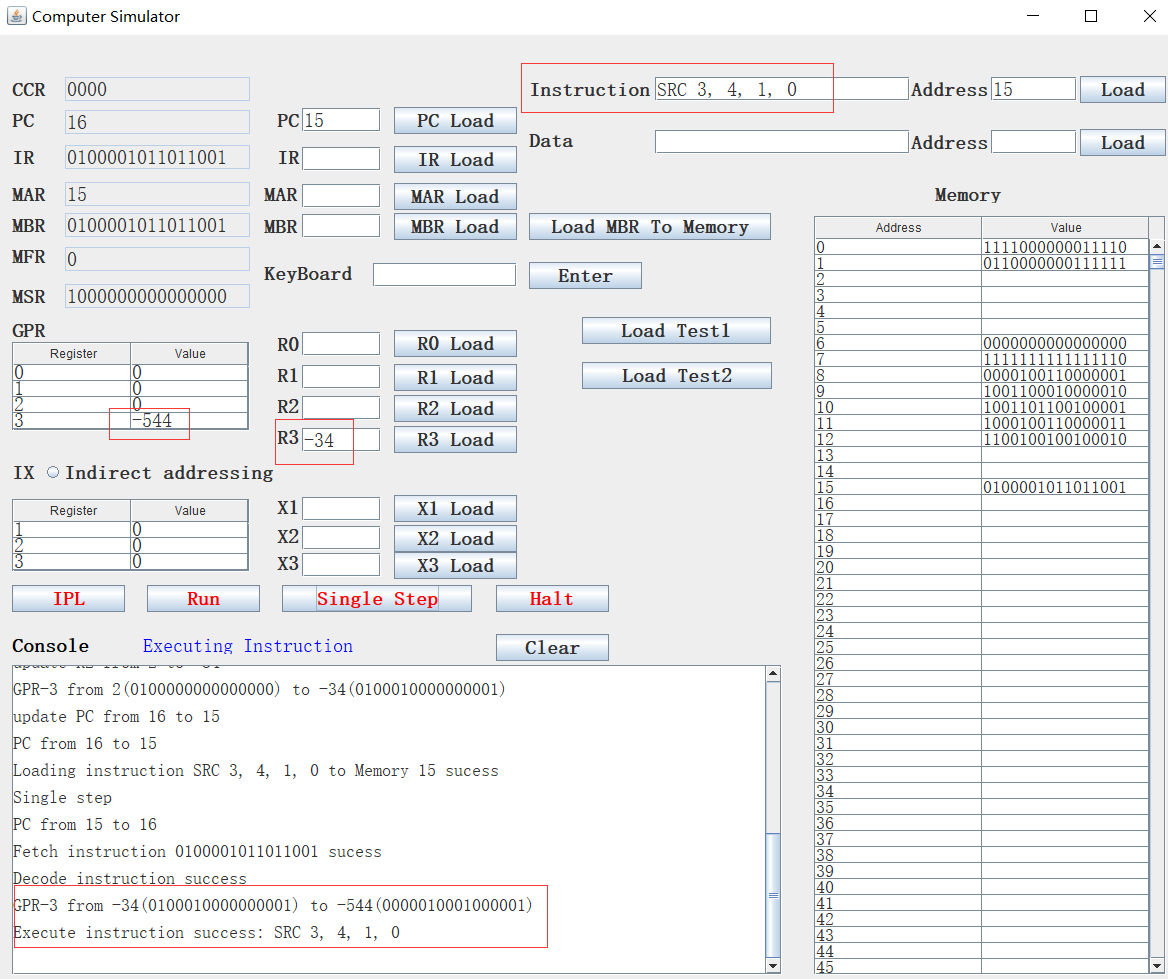
Logical right shifting:

We set R3 to 34, then execuate SRC 3, 4, 0, 1. This instruction is logical right shifting, right shifts 4bits. R3 should be c(R3) / 2^4= 34/16=2:



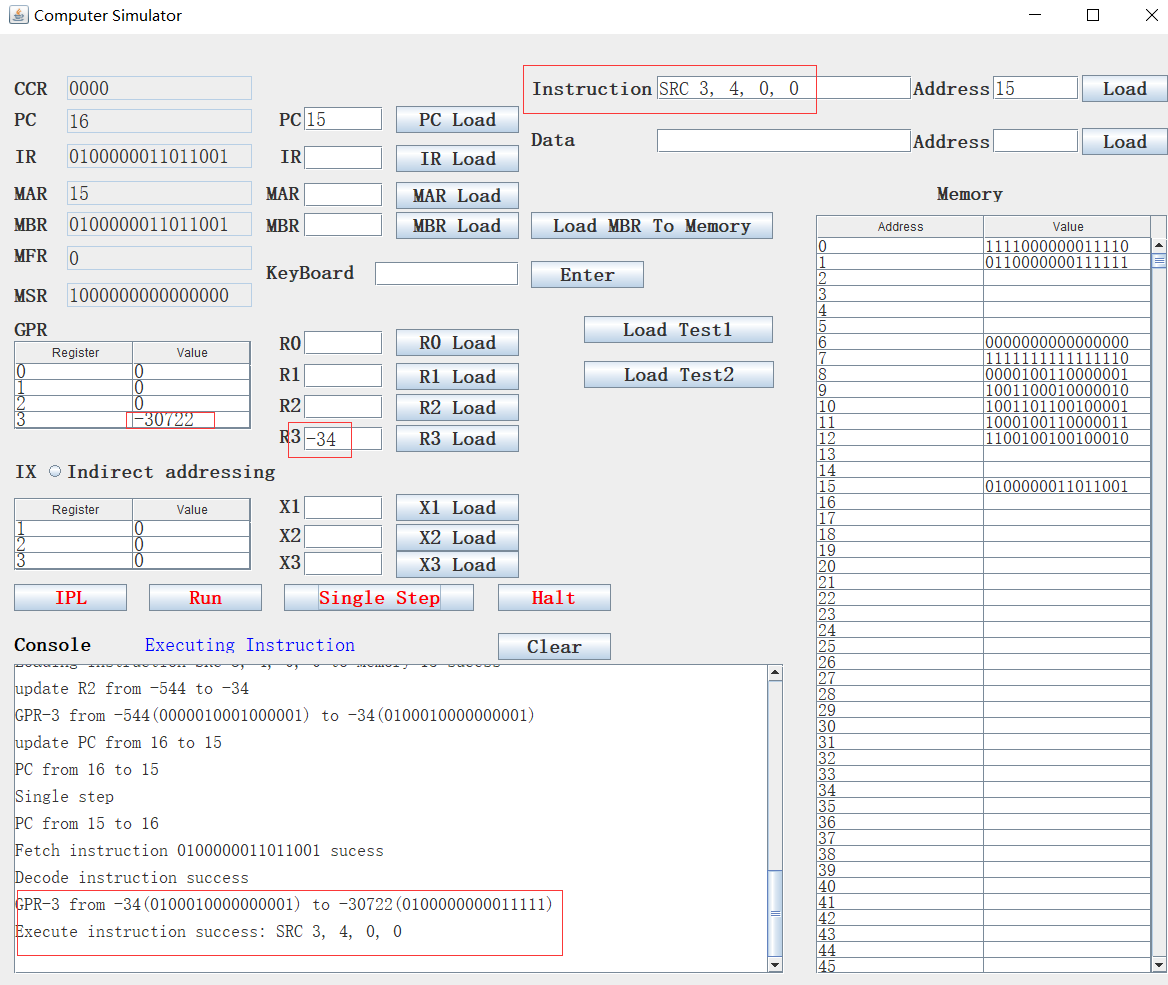
Arithmetic left shiting:

we set R3 to -34, then execuate SRC 3, 4, 1, 0. This instruction is arithmetic left shifting, left shifts 4bits. Arithmetic shifting will keep the sign bit, so R3 should be c(R3) \* 2^4= -34\*16=-544:



Arithmetic right shiting:

we set R3 to -34, then execuate SRC 3, 4, 0, 0. This instruction is arithmetic right shifting, right shifts 4bits. Arithmetic shifting will keep the sign bit. For right shifting, the missing bits will be compensated by sign bits, so 0100010000000001 right shift 4 bits should be 0100000000011111:



1. RRC

RRC instruction fomat is 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)

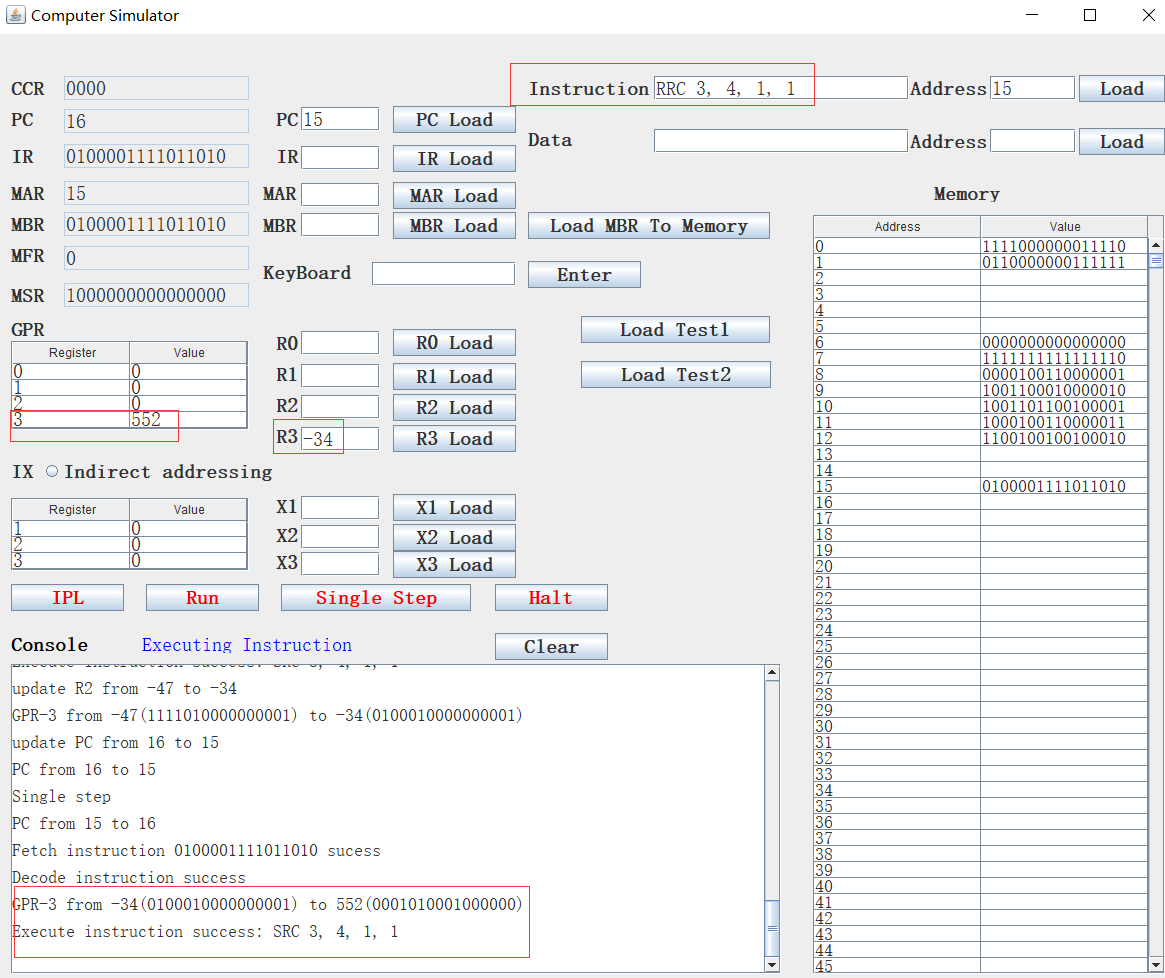
XX, XXX is ignored

Count = 0…15

If Count = 0, no rotate occurs.

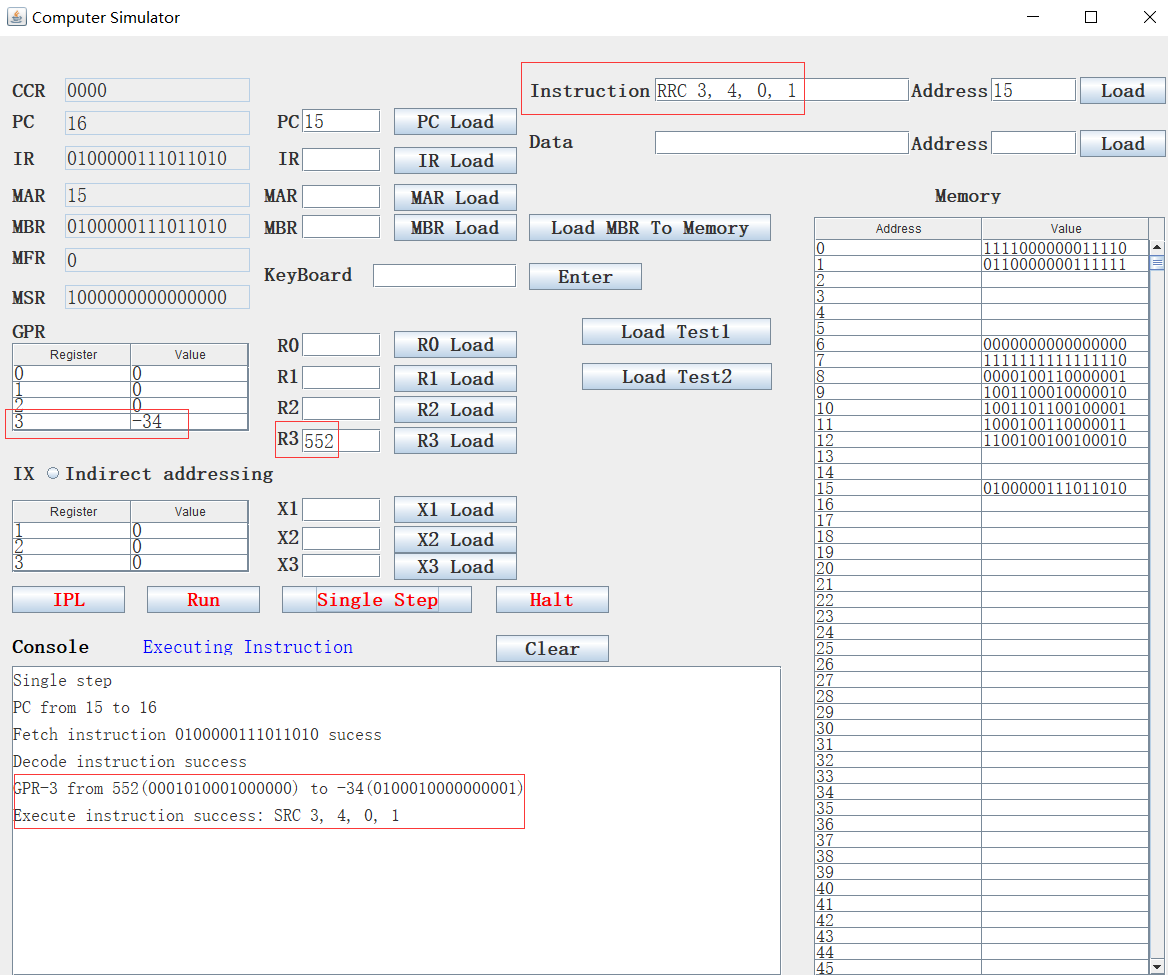
Left rotate

For example, we set R3 to -34, then execuate RRC 3, 4, 1, 1. This instruction will rotate left 4bits. so 0100010000000001 rotate left 4 bits should be 0001010001000000, which is 552.



Right rotate

We set R3 to 522, then execuate RRC 3, 4, 0, 1. This instruction will rotate right 4bits. so 0001010001000000 rotate right 4 bits should be 0100010000000001, which is -34.



1. IN

IN instruction’s format is IN r, devid.

Input Character To Register from Device, r = 0..3

The device id is defined as the following:

DEVID Device

0 Console Keyboard

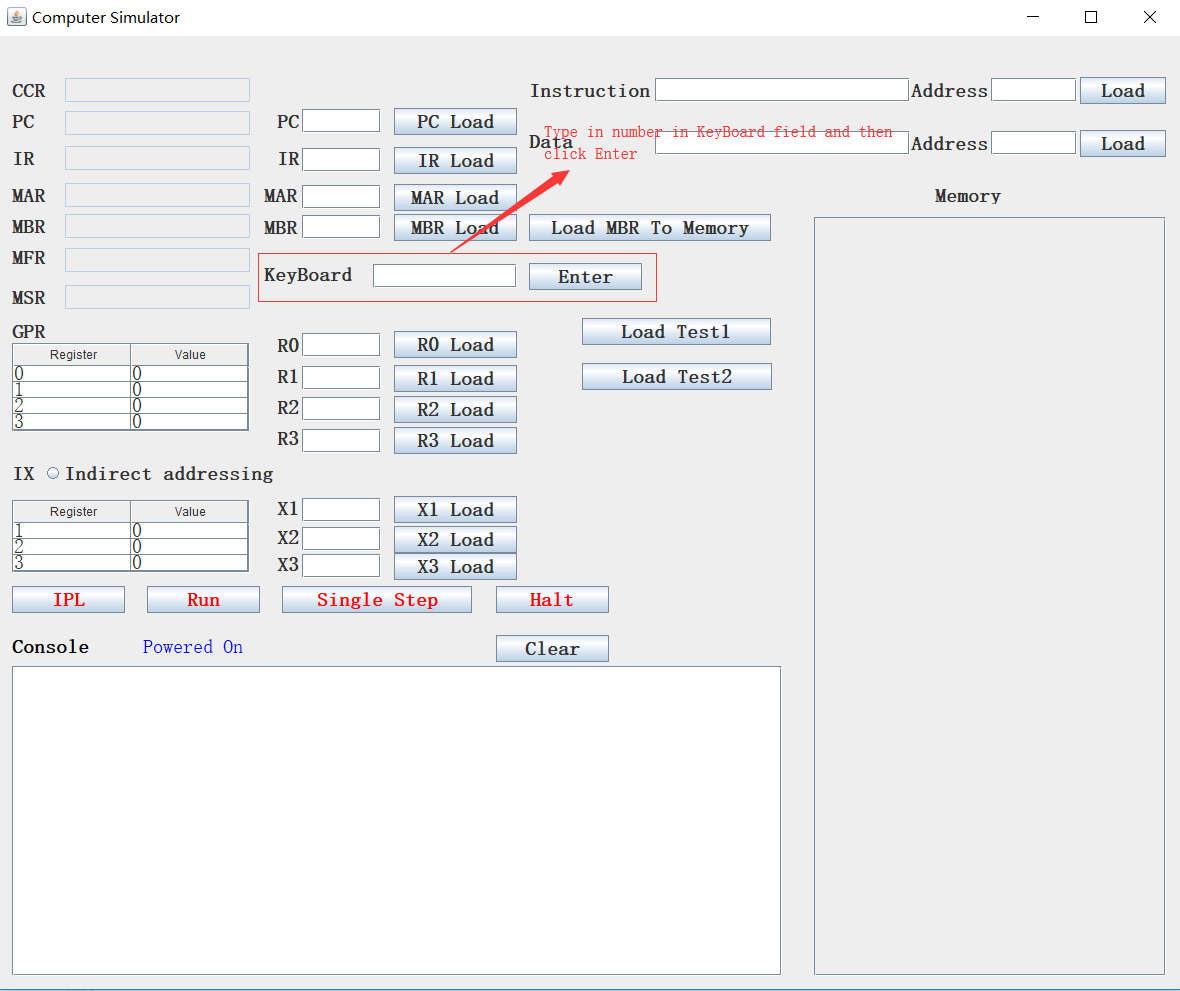
1 Console Printer

2 Card Reader

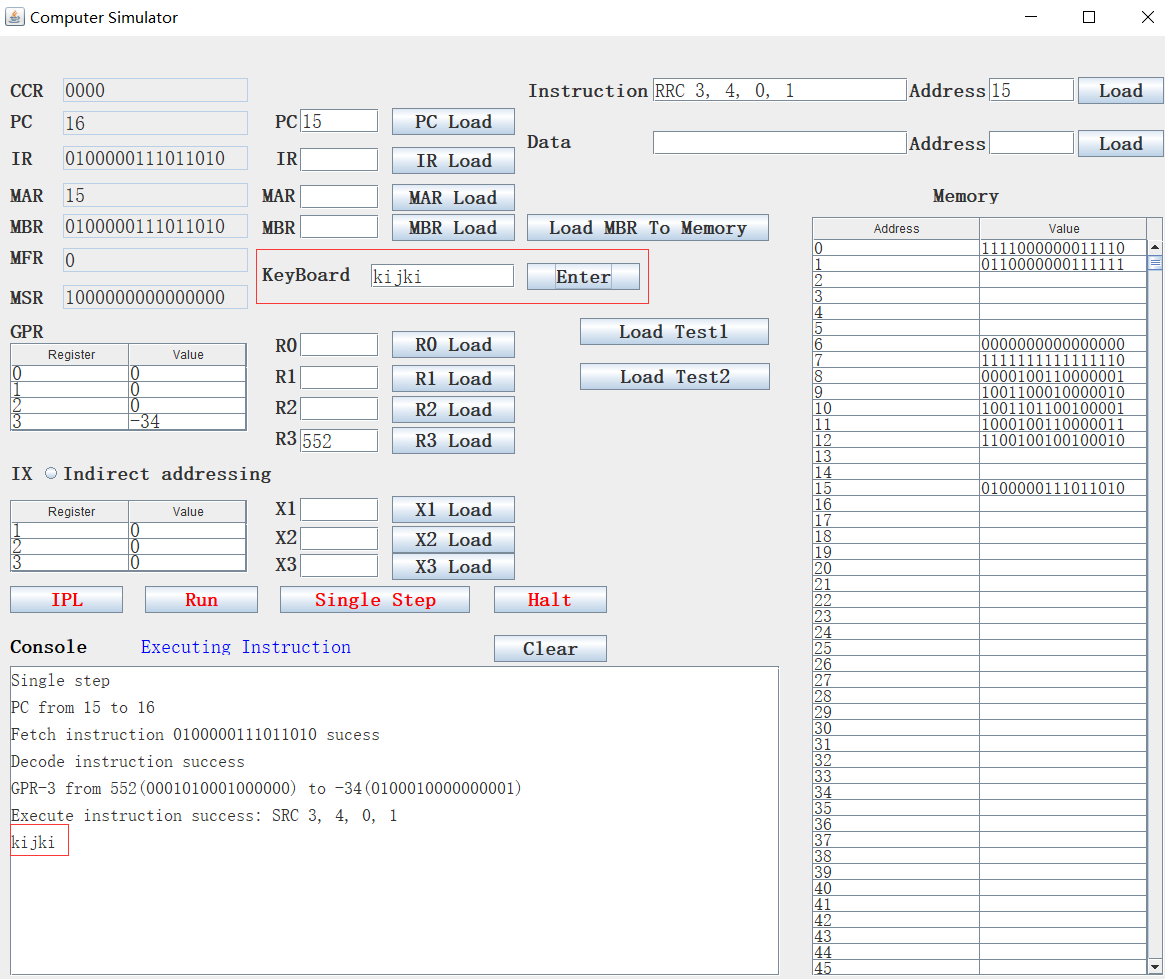
3-31 Console Registers, switches, etc

We only simulate console keyboard and console printer.

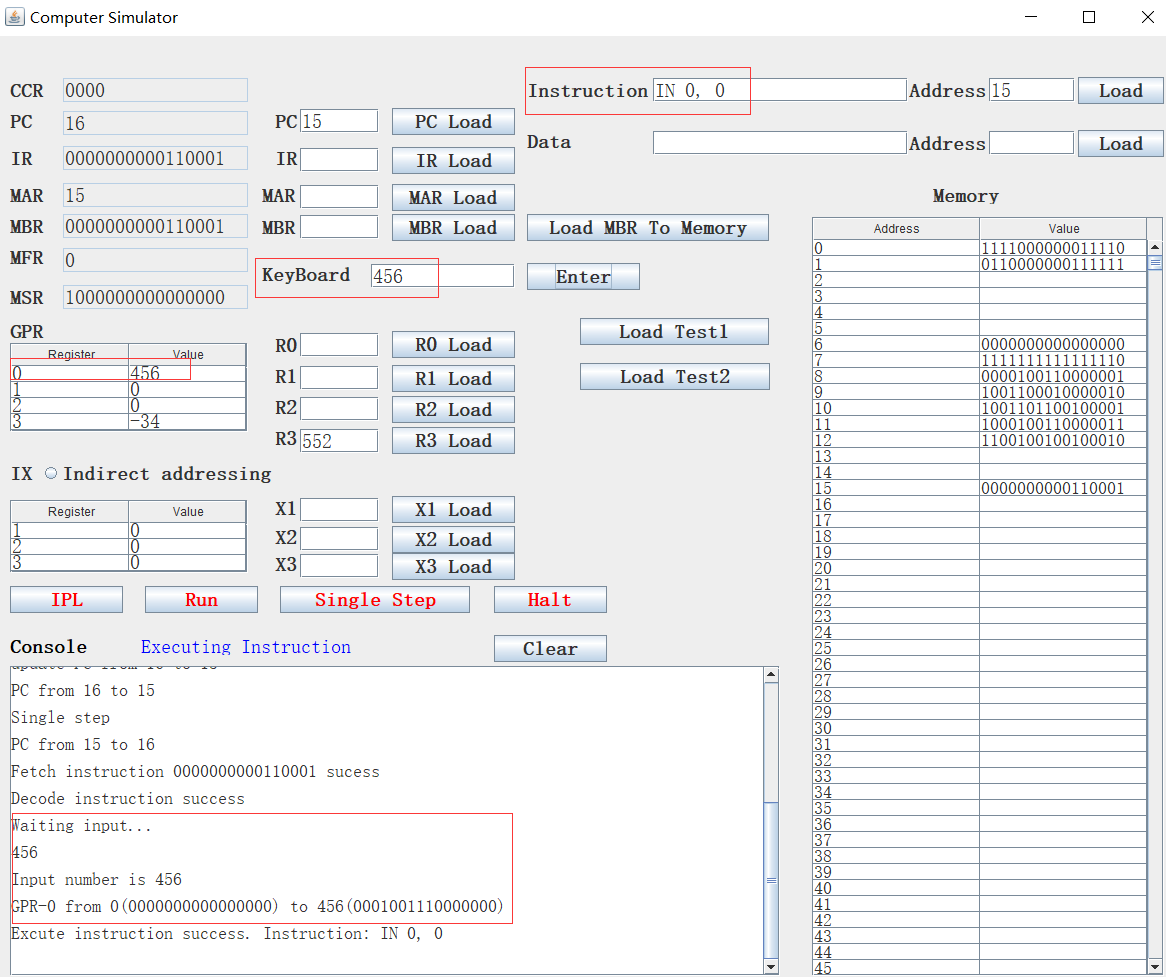
We add a textfield to simulate the keyboard, and a button “Enter” to simulate the “Enter” key. See the following:



We can type any character or number in the keyboard field and then click “Enter”, the corresponding input will be displayed on the console printer:



If we execute IN 0,0, it will waiting for users to input a number and then move this number into R0:

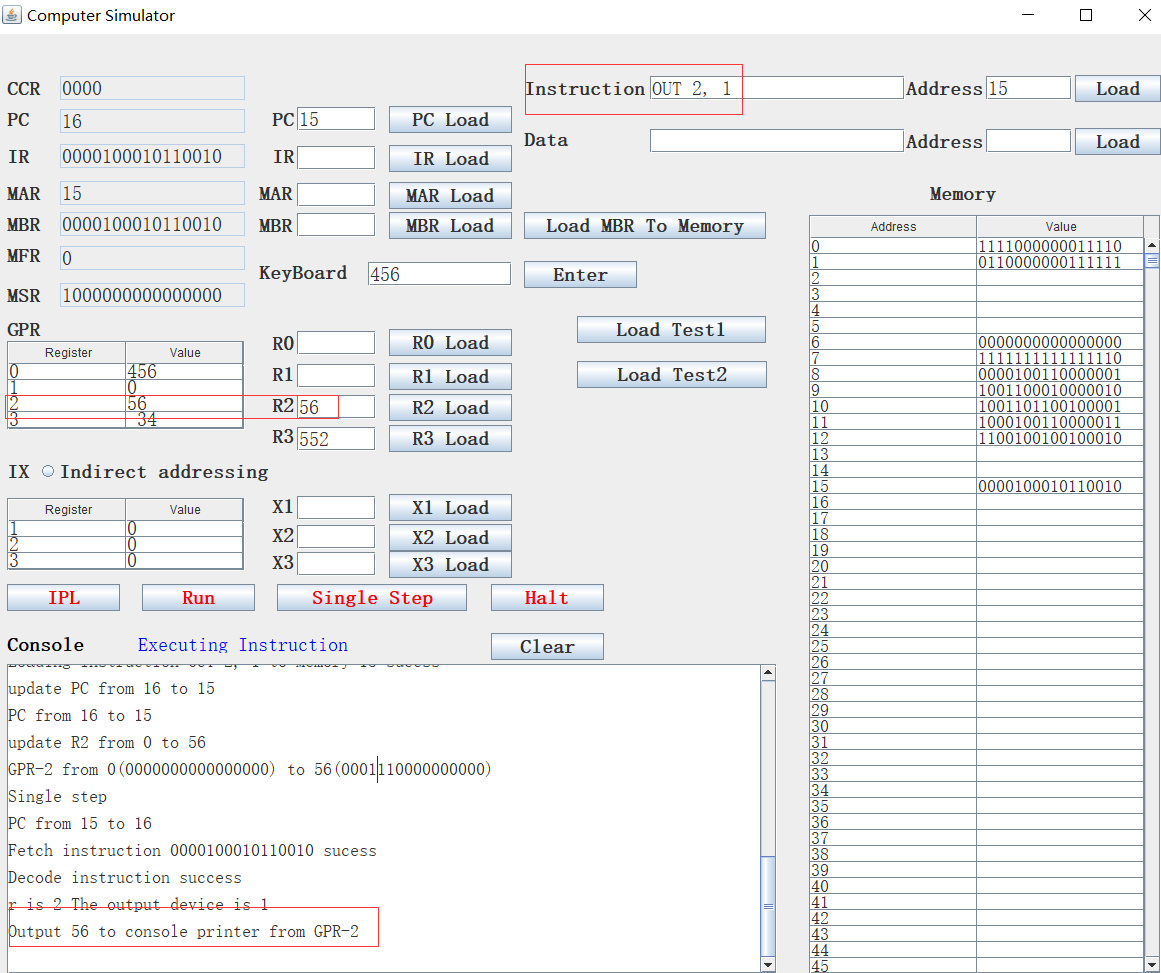


1. OUT

OUT instruction’s format is OUT r, devid.

Output Character to Device from Register, r = 0..3

For example, we set R2 to 56 and then execute OUT 2, 1, the 56 will be output on the console printer:



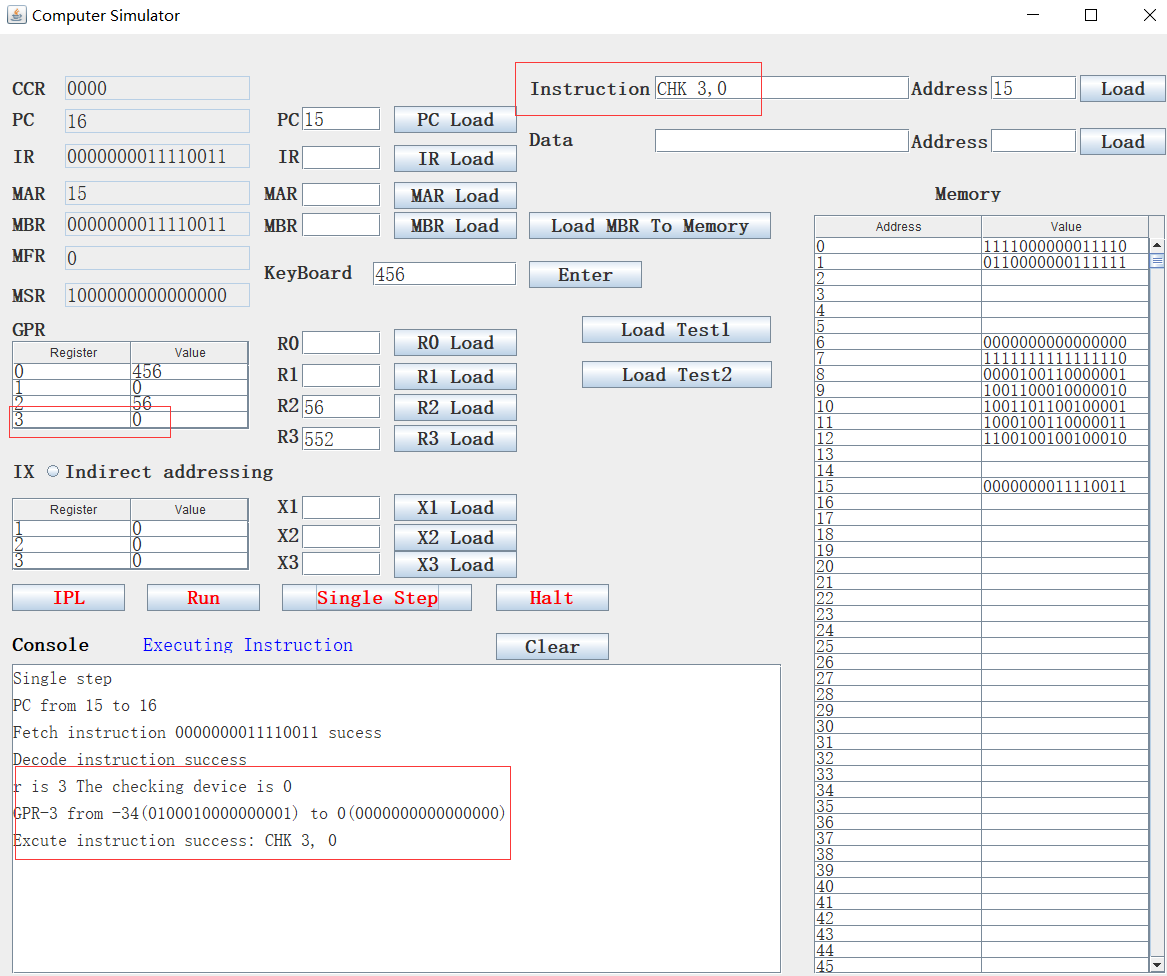
1. CHK

CHK instruction’s format is CHK r, devid.

Check Device Status to Register, r = 0..3

c(r) <- device status

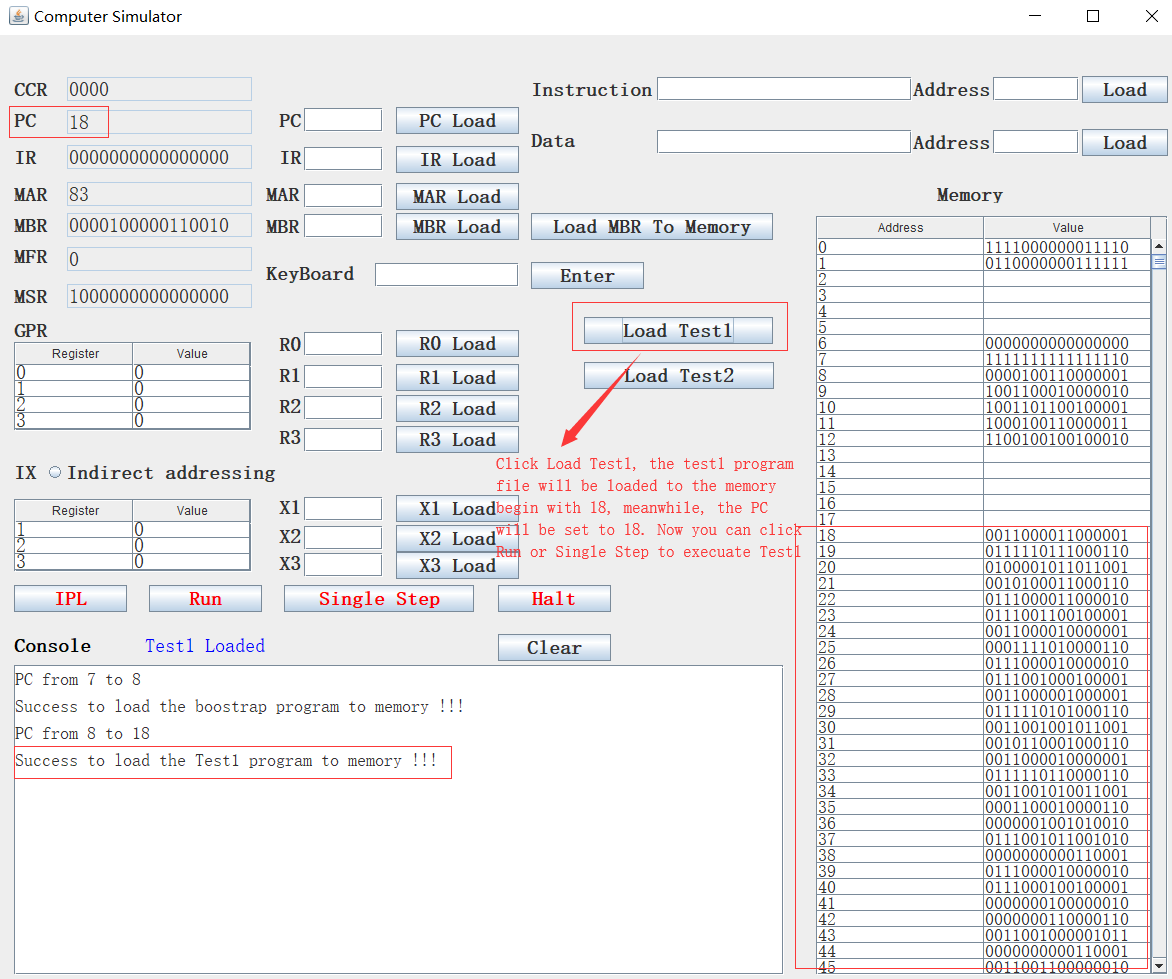
We just simply set any device’s status to 0, so when execute CHK 3, 0, the R3 will be set to 0:



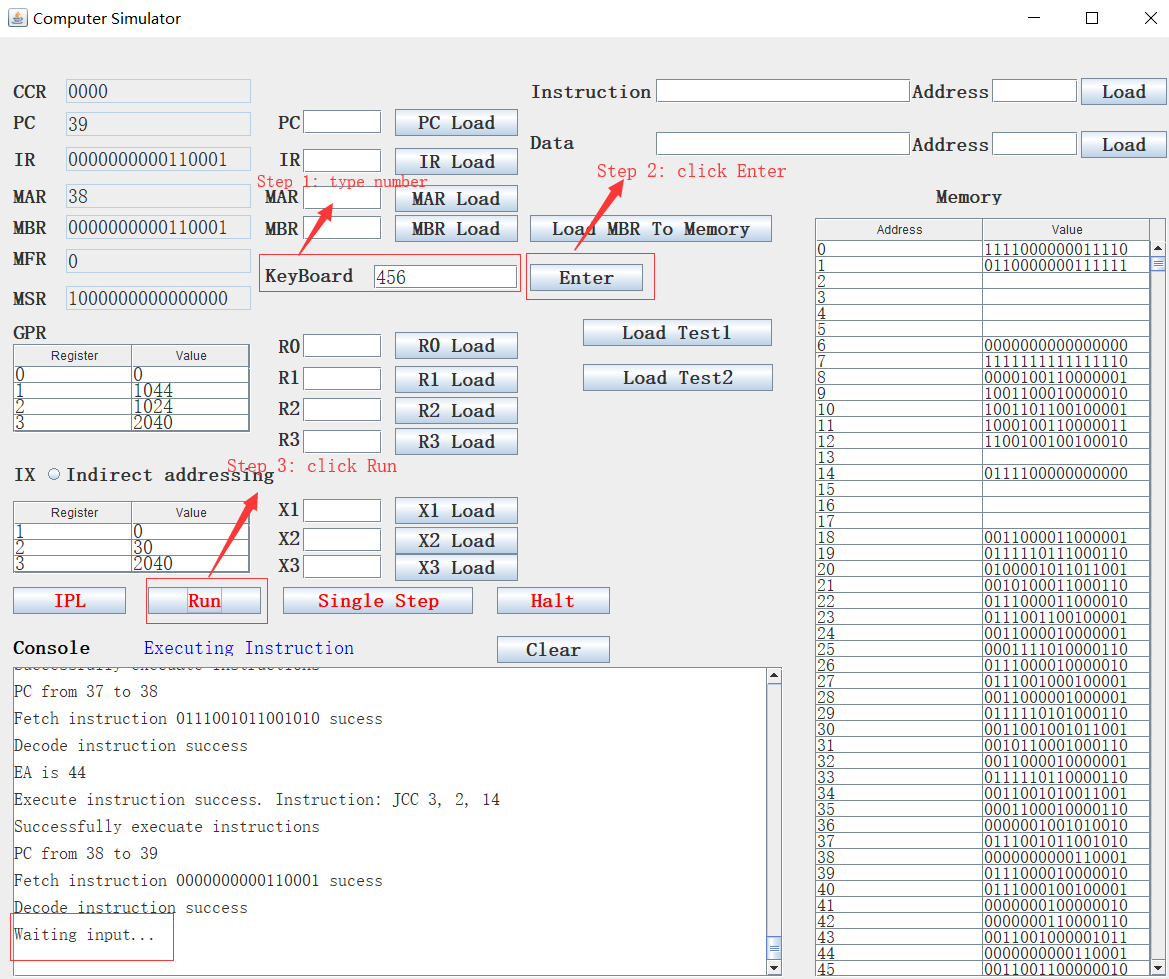
1. **Test 1 program**

We wrote the test 1 program which asks users to input 20 numbers and one searched number, then the program will output the number closest to the searched number. The number range is [-32767, 32767].

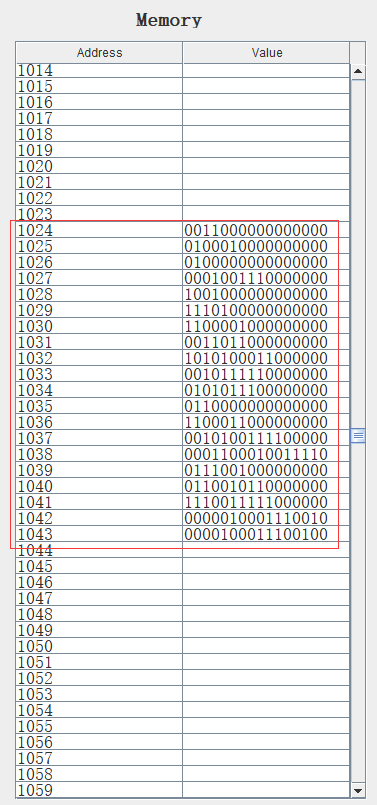
Users can click “Load Test1” button to load the test 1 program into memory begin with memory location 18, meanwhile, the PC will be set to 18. see the following screen shot:



After loading test1 to memory, users can click “Run” or “Single Step” to run this program. During the running of this program, it will stop and prompt users to input number. Users should type any number in the keyboard field then click “Enter”. After inputting numbers, users should click “Run” or “Single Step” to continue this program. See the following screen short:



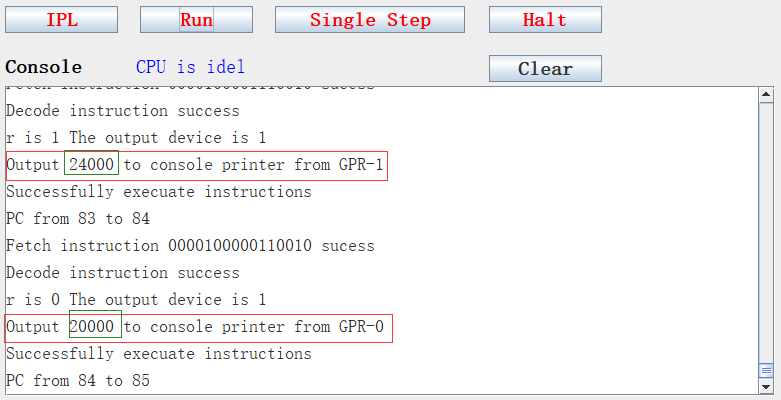
Our program will move the input numbers to memory beginning with memory location 1024, see the following screen short:



Memory location 1024 to 1043 will holds the 20 input numbers. Memory location 2046 will holds the searched number.

We give two groups of input number, the first group of 20 numbers are: 12, 34, 2, 456, 9, 23 67, 108, 789, 500, 234, 6, 99, 1940, 31000, 78, 422, 999, 20000, 10000. The searched number is 24000.

After we input the 20 numbers and the searched number, the program will find the closest number to the searched number from these 20 number, the following is the final result:



We can see the program first output the searched number 24000, then output the number 20000 which is closest to it.

We give another group of test 20 numbers, they are: 12，34，2，456，9，23，67，108，789，-500，234，6，99，1940，31000，78，422， 999，-200，-3. The searched number is 5. The following is the result:



We can see the program first output the searched number 5, then output the number 6 which is closest to it.

1. **The program and its comments:**

//Set R3 to 0

18 LDR 3, 0, 6

//Add 125 to R3

19 AIR 3, 125

//R3 shift left 4 bits, which multiply 16, so R3 now is 2000

20 SRC 3, 4, 1, 0

// Add 40 to R3, now R3 is 2040

21 AIR 3, 40

//Move R3's content to memory 14, now M[14] = 2040

22 STR 3,0,14

//Load content of memory 14 to X3, now X3 is 2040

23 LDX 3, 14

//Set R2 to 0

24 LDR 2, 0, 6

//Add 30 to R2, now R2 is 30

25 AIR 2, 30

//move R2's content to memory location 14

26 STR 2,0,14

//load content of memory location 14 to X2, now X2 is 30

27 LDX 2, 14

//Set R1 to 0

28 LDR 1, 0, 6

//Add 125 to R1

29 AIR 1, 125

//Left shift R1 3 bits, which multiply 8, now R1 is 1000

30 SRC 1, 3, 1, 0

//Add 44 to R1, now R1 is 1044

31 AIR 1, 44

//Set R2 to 0

32 LDR 2, 0, 6

//Add 125 to R2, now R2 is 125

33 AIR 2, 125

//left shift R2 left 3 bits, which multiply 8, now R2 is 1000

34 SRC 2, 3, 1, 0

//Add 24 to R2, now R2 is 1024

35 AIR 2, 24

// Test if R1 is equal to R2

36 TRR 1,2

// if R1 equal to R2, then jump to memory location 30+14=44, that is "IN, 0, 0", the searched number input by user

37 JCC 3,2,14

//If R1 not equal to R2, then wait for input, accept the input to R0

38 IN 0, 0

//Move R2's content to memory location 14, now M[14]=1024

39 STR 2,0,14

// Move the content of memory location 14 to X1, now X1 is 1024

40 LDX 1,14

// Store R0's content to memory location 1024, R0's content is the input

41 STR 0,1,0

//increase R2's content, now R2 is 1025

42 AIR 2,1

//jump directly to 30+6=36, that is "TRR 1,2"

43 JMA 2,6

//wait for the searched number, accept it to R0

44 IN 0, 0

//Move R0's content to 2040+6=2046, now M[2046]= the searched number

45 STR 0,3,6

//load the content of memory location 7 to R3, M[7] has already been set to 32767, now R3 //is 32767

46 LDR 3,0,7

//Move R3's content to 2040+4=2044, now M[2044] = 32767

47 STR 3,3,4

// Set R2 to 0

48 LDR 2, 0, 6

// Add 60 to R2, now R2 is 60

49 AIR 2, 60

// move R2’s content to M[14]

50 STR 2,0,14

// move M[14] to X2

51 LDX 2, 14

// Set R1 to 0

52 LDR 1, 0, 6

// add 125 to R1, now R1 is 125

53 AIR 1, 125

// left shift R1 3 bits, which multiply 8, now R1 is 1000

54 SRC 1, 3, 1, 0

// add 44 to R1, now R1 is 1044

55 AIR 1, 44

// Set R2 to 0

56 LDR 2, 0, 6

// Add 125 to R2, now R2 is 125

57 AIR 2, 125

// left shift R1 3 bits, which multiply 8, now R1 is 1000

58 SRC 2, 3, 1, 0

// add 23 to R1, now R1 is 1023

59 AIR 2, 23

// add 1 to R1, now R1 is 1024

60 AIR 2,1

// Test if R1 is equal to R2

61 TRR 1,2

// Jump to 80 if R1 equal to R2, otherwise, go to the next instruction

62 JCC 3,2,20

// Move the content of M[2046] to R0, M[2046] contains the searched number

63 LDR 0, 3, 6

//Move R2’s content to M[14], R2 now is 1024

64 STR 2,0,14

// Move M[14] to X1, now X1 is 1024

65 LDX 1,14

//R0 = R0 – M[1024], which means searched number – M[1024]

66 SMR 0, 1, 0

//Move the result to M[2042]

67 STR 0, 3, 2

//if result greater than 0, then jump to 73, else go to the next instruction

68 JGE 0,2,13

// Move the result to M[2045]

69 STR 0, 3, 5

// Set R0 to 0

70 LDR 0, 0, 6

// R0 = R0 – M[2045], now the negative number will become to positive number

71 SMR 0, 3, 5

//Move this positive number to M[2042]

72 STR 0, 3, 2

//R0 = R0 – M[2044], M[2044] contains the closest distance

73 SMR 0, 3, 4

//if the result larger than 0, which means the current number is far than the last number, drop //it and continue to calculate the next number, so jump to 60 to execute. If not, replace last //distance and number with this one

74 JGE 0,2,0

//load M[2042] to R0

75 LDR 0, 3, 2

//Move M[2042] to M[2044]

76 STR 0, 3, 4

//load M[1024] to R0

77 LDR 0, 1, 0

//move M[1024] to M[2043]

78 STR 0, 3, 3

//Jump directly to 60

79 JMA 2, 0

//Load M[2043] to R0, M[2043] keeps the closest number

80 LDR 0, 3, 3

//Load M[2046] to R1, M[2046] keeps the searched number

81 LDR 1, 3, 6

//Output R1, that is the searched number

82 OUT 1, 1

//Output R0, that is the closest number.

83 OUT 0, 1

1. **Design notes:**
2. Integer range

Since project 2 has many instructions which involves in negative numbers’ comparison and calculation, and each register and memory slot is 16 bits, so we narrowed the number range in our simulator to [-32767, 32767], and the highest bit is the sign bit.

1. Number display

All numbers in our simulator are displayed in a reverse order, so don’t be confused when see the number shifting or rotating operations in our examples.