|  |
| --- |
|  |
| Computer Architecture |
| Project Two |
|  |
| Sipeng Wang  Jiafang Liu  Christopher Marc Licata |
| **2016/10/14** |
|  |

|  |
| --- |
|  |

**1. Introduction**

**1.1. Purpose**

The purpose of this document is to include design notes and architecture for all phases of the computer simulator. The purpose of this project is to design and implement a computer simulator in order to gain broader understanding of the computer architecture including ISA, memory structure, operations and I/O capabilities.

**1.2. Component**

The class project is structured into four segments of increasing difficulty that build towards a detailed understanding of the internal design of computer systems and a fairly complex simulation of a computer system. Each segment is due to the grader at about 3 week intervals (see Syllabus).

The four components are:

|  |  |
| --- | --- |
| **Component** | **Description** |
| I: Basic Machine | Design and implement the basic machine architecture.  Implement a simple memory  Execute Load and Store instructions  Build initial user interface to simulator |
| II: Memory and Cache  Design | Design and implement the modules for enhanced  memory and cache operations  Implement all instructions except for MSR, CHK, and  Floating Point/Vector operations  Extend the user interface.  Demonstrate 1st program running on your simulator. |
| III: Execute All Instructions | *Phase3 here* |
| IV: DO 1 OF:  A. Floating Point and Vector Operations  B: Enhanced Scheduling | *Phase4 here* |

**2. System Overview**

*[General system Overview]*

**3. Architectural Design and Design Notes**

**3.1. Program Control and Flow**

programLoader

Operater Unit

(instructions)

.SingleCircle()

Control Unit

mGUI

Memory

Cache

The program control flow is primarily controlled by the GUI and the computer’s clock cycle (in control Unit) in Phase 1. The Control Unit loads the special instruction from operator unit. In phase 2, we use programLoader to read assembly program from txt file, transfer to machine code and write into memory directly. We also use programLoader to initialize the registers for programs. OperatorUnit uses the memory by using cache. The mechanism of cache will be explained in next part.

**Memory and Cache Design:**

In Phase2, we implement a simple cache, which sits between memory and the rest of the processor. The cache only fetches and cleans on demand from the operator unit. A least recently used replacement policy is used to determine which cache blocks should remain in memory. Main memory is updated via a write though mechanism by tracking the dirty status of each cache block. The cache is fully implemented and functional.

yes

Load from memory

Get value

getValuefromCache()

In?

no

Add cache block

getValuefromCache()

Write to memory

In?

Set dirty bit to 1

Change cache

no

yes

**3.2 Instruction Decomposition:**

**3.2.1 Phase 1 – Load and Store Design**

* **LDR r, x, address[,I]**
* *MAR <- PC*
* *MBR <- C(MAR)*
* *IR <- MBR*
* *Decode IR*
* *IR0-5 <- OPCODE*
* *IR6-7 <- R(0-3)*
* *IR8-9 <- IX(1-3)*
* *IR10 <- Indirect Addressing Bit*
* *IR11-15 <- Address*
* *Compute EA*
* *R<-C( EA) r <- c(c(EA)), if I bit set*
* *PC <- PC + 1*
* **STR r, x, address[,I]**
* *MAR <- PC*
* *MBR <- C(MAR)*
* *IR <- MBR*
* *Decode IR*
* *IR0-5 <- OPCODE*
* *IR6-7 <- R(0-3)*
* *IR8-9 <- IX(1-3)*
* *IR10 <- Indirect Addressing Bit*
* *IR11-15 <- Address*
* *Compute EA*
* *C(EA)<-R c(c(EA)) <- c(r) if I-bit set //Load the contents of the Register into the memory*
* *PC <- PC + 1 // Increment PC*
* **LDA r, x, address[,I]**
* *MAR <- PC*
* *MBR <- C(MAR)*
* *IR <- MBR*
* *Decode IR*
* *IR0-5 <- OPCODE*
* *IR6-7 <- R(0-3)*
* *IR8-9 <- IX(1-3)*
* *IR10 <- Indirect Addressing Bit*
* *IR11-15 <- Address*
* *Compute EA*
* *R <- EA r  c(EA), if I bit set*
* *PC <- PC + 1*
* **LDX x, address[,I]**
* *MAR <- PC*
* *MBR <- C(MAR)*
* *IR <- MBR*
* *Decode IR*
* *IR0-5 <- OPCODE*
* *IR6-7 <- R(0-3)*
* *IR8-9 <- IX(1-3)*
* *IR10 <- Indirect Addressing Bit*
* *IR11-15 <- Address*
* *Compute EA*
* *IX <- C(EA)*
* *PC <- PC + 1*
* **STX x, address[,I]**
* *MAR <- PC*
* *MBR <- C(MAR)*
* *IR <- MBR*
* *Decode IR*
* *IR0-5 <- OPCODE*
* *IR6-7 <- R(0-3)*
* *IR8-9 <- IX(1-3)*
* *IR10 <- Indirect Addressing Bit*
* *IR11-15 <- Address*
* *Compute EA*
* *EA <- c(X0) C(EA) <- c(Xx), if I-bit set*
* *PC <- PC + 1 //Increment PC*
* **AMR r, x, address[,I]**
* *MAR <- PC*
* *MBR <- C(MAR)*
* *IR <- MBR*
* *Decode IR*
* *IR0-5 <- OPCODE*
* *IR6-7 <- R(0-3)*
* *IR8-9 <- IX(1-3)*
* *IR10 <- Indirect Addressing Bit*
* *IR11-15 <- Address*
* *Compute EA*
* *R <- C(R) + C(EA)*
* *PC <- PC + 1 ;Increment PC*
* **SMR r, x, address[,I]**
* *MAR <- PC*
* *MBR <- M(MAR)*
* *IR <- MBR*
* *Decode IR*
* *IR0-5 <- OPCODE*
* *IR6-7 <- R(0-3)*
* *IR8-9 <- IX(1-3)*
* *IR10 <- Indirect Addressing Bit*
* *IR11-15 <- Address*
* *Compute EA*
* *R <- C(R)-C(EA)*
* *PC <- PC + 1 ;Increment PC*
* **AIR r, immed**
* *MAR <- PC*
* *MBR <- M(MAR)*
* *IR <- MBR*
* *Decode IR*
* *IR0-5 <- OPCODE*
* *IR6-7 <- R(0-3)*
* *IR8-9 <- IX(1-3)*
* *IR10 <- Indirect Addressing Bit*
* *IR11-15 <- Immed*
* *R <-C(R)+Immed*
* *PC <- PC + 1*
* **SIR r, immed**
* *MAR <- PC*
* *MBR <- M(MAR)*
* *IR <- MBR*
* *Decode IR*
* *IR0-5 <- OPCODE*
* *IR6-7 <- R(0-3)*
* *IR8-9 <- IX(1-3)*
* *IR10 <- Indirect Addressing Bit*
* *IR11-15 <- Immed*
* *R <- C(R) - Immed*
* *PC <- PC + 1*

**Effective Address (EA) Computation:**

EA =

if I = 0:

if IX = 00, Address

if IX = 1..3, c(Xj) + Address, where j = c(IX)

if I = 1:

if IX = 00, c(Address) // indirect addressing

if IX = 1..3, c(c(Xj) + Address), where j = c(IX)

**3.2.2 Phase 2 - Memory and Cache Design**

* **JZ r, x address[, I]**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- R(0-3)
* IR8-9 <- IX(1-3)
* IR10 <- Indirect Addressing Bit
* IR11-15 <- Address
* IF C(r)==0 PC<- C(EA) else PC<PC+1
* **JNE r, x address[, I]**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- R(0-3)
* IR8-9 <- IX(1-3)
* IR10 <- Indirect Addressing Bit
* IR11-15 <- Address
* IF(CR!=0) PC<- C(EA) else PC<PC+1
* **JCC cc, x address[, I]**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- CC(0-3)
* IR8-9 <- IX(1-3)
* IR10 <- Indirect Addressing Bit
* IR11-15 <- Address
* IF CC(0-3)==1 PC<- C(EA) else PC<PC+1
* **JMA x, address[, I]**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- R(0-3) (Ignored)
* IR8-9 <- IX(1-3)
* IR10 <- Indirect Addressing Bit
* IR11-15 <- Address
* PC<- C(EA)
* **JSP r, x address[, I]**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- R(0-3)
* IR8-9 <- IX(1-3)
* IR10 <- Indirect Addressing Bit
* IR11-15 <- Address
* R3<-PC+1
* IF I bit set PC<- C(EA) else PC<-EA
* **RFS immed**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- R(0-3)
* IR8-9 <- IX(1-3)
* IR10 <- Indirect Addressing Bit
* IR11-15 <- Immed
* R0 <- Immed
* PC <- C(R3) IX, I fields are ignored
* **SOB r, x address[, I]**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- R(0-3)
* IR8-9 <- IX(1-3)
* IR10 <- Indirect Addressing Bit
* IR11-15 <- Address
* R <- C(R)-1
* If C(R)>0 PC <- EA else PC<-PC+1
* **JGE r, x address[, I]**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- R(0-3)
* IR8-9 <- IX(1-3)
* IR10 <- Indirect Addressing Bit
* IR11-15 <- Address
* If C(R)>=0 Then PC <- EA or C(EA), If I bit set;

Else PC <-PC + 1

* **MLT rx, ry**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- RX
* IR8-9 <- RY
* RX <- C(RX)\*C(RY) RX must be 0 or 2
* RX+1 <- C(RX)\*C(RY)
* **DVD rx, ry**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- RX
* IR8-9 <- RY
* RX <- C(RX)/C(RY)
* RX+1 <- C(RX)/C(RY)
* **TRR rx, ry**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- RX
* IR8-9 <- RY
* If C(RX)==C(RY) CC(4) <- 1 else CC(4) <-0
* **AND rx, ry**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- RX
* IR8-9 <- RY
* C(RX) <- C(RY) AND C(RY)
* **ORR rx, ry**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- RX
* IR8-9 <- RY
* C(RX) <- C(RY) OR C(RY)
* **ORR rx, ry**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- RX
* IR8-9 <- RY
* C(RX) <- NOT C(RY)
* **SRC r, count, L/R, A/L**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- R(0-3)
* IR8 <- AL
* IR9 <- LR
* IR10 -15<- Count
* R < - Shift Register
* **RRC r, count, L/R, A/L**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- R(0-3)
* IR8 <- AL
* IR9 <- LR
* IR10 -15<- Count
* R < - Rotator Register
* **IN r, devid**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- R(0-3)
* IR8-9 <- NULL
* IR10 -15<- devid
* R < - input number
* **out r, devid**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- R(0-3)
* IR8-9 <- NULL
* IR10 -15<- devid
* R < - out put number
* **CHK r, devid**
* MAR <- PC
* MBR <- M(MAR)
* IR <- MBR
* Decode IR
* IR0-5 <- OPCODE
* IR6-7 <- R(0-3)
* IR8-9 <- NULL
* IR10 -15<- devid
* R < - device status

**3. 3.3 Phase 3 - Execute All Instructions**

**Load/Store Instructions:**

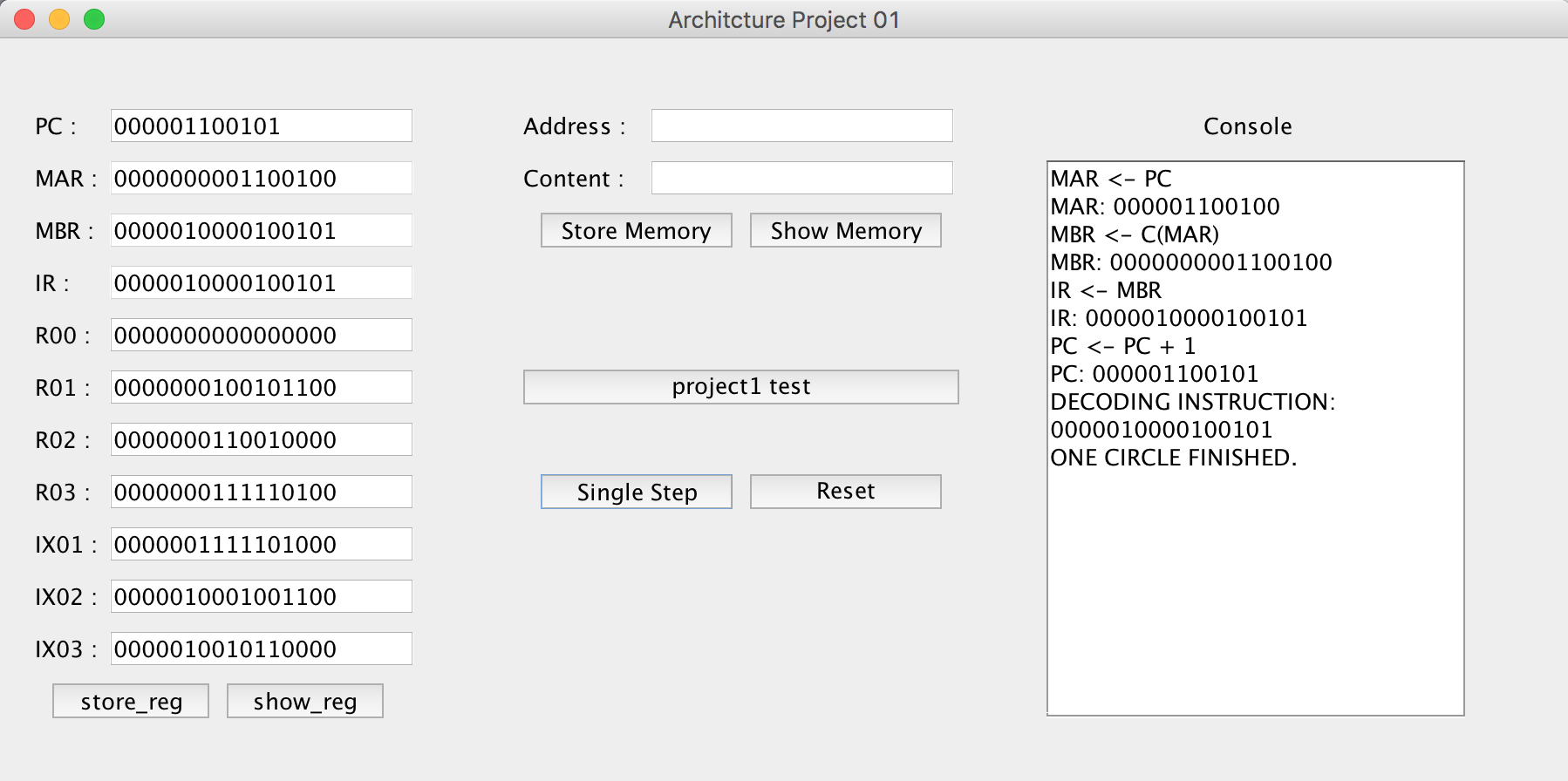
LDR:

Instruction: 0000010000000101

Opcode:000001 R:0 IX:0 I:0 ADDR：101

Excepted：R0 = 1 (0000000000000001)

Result: Passed



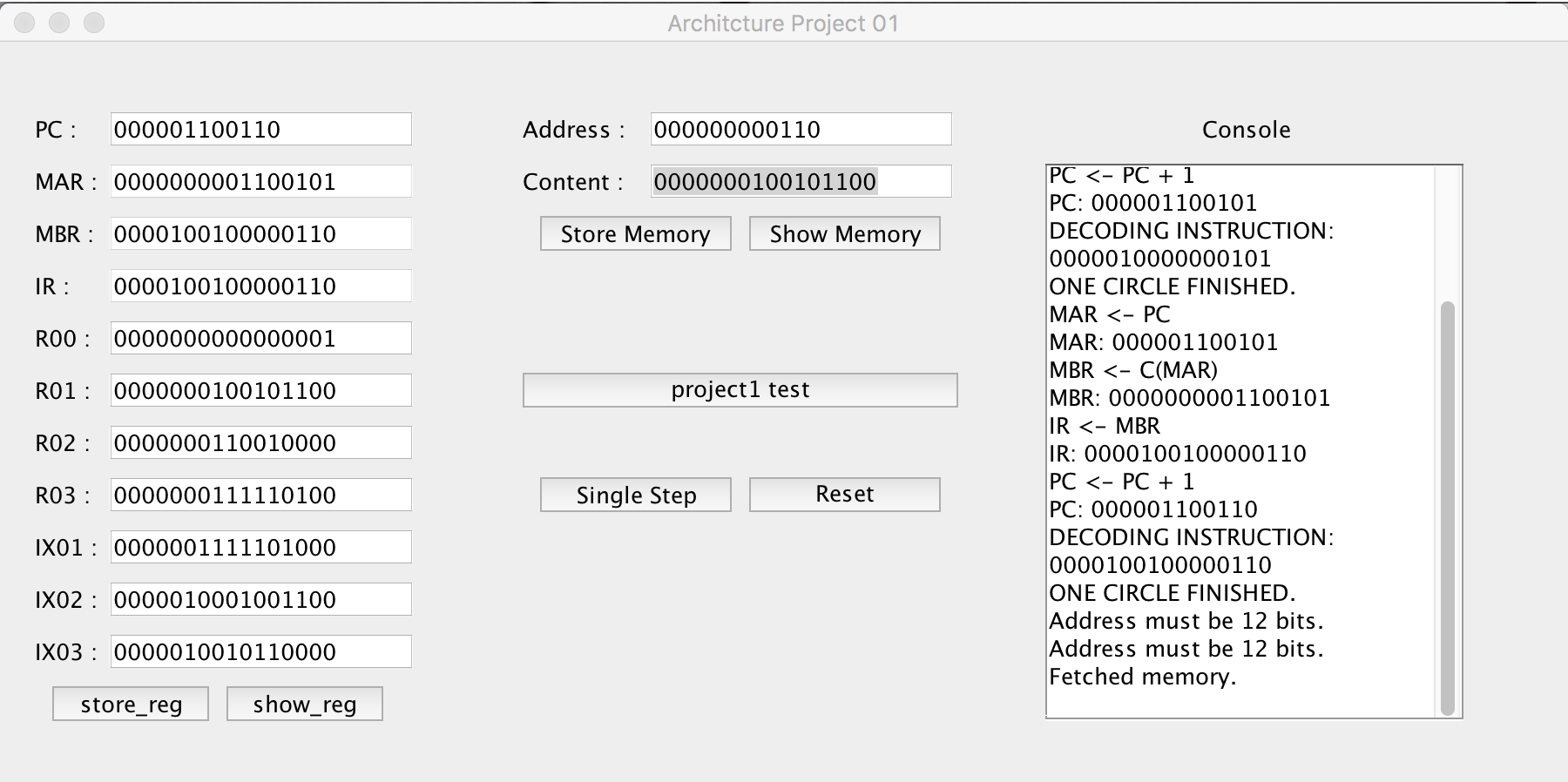
STR:

Instruction: 0000100100000110

Opcode:000010 R:01 IX:0 I:0 ADDR：110

Excepted：address(0000000000000110) = 0000000100101100

Result: Passed



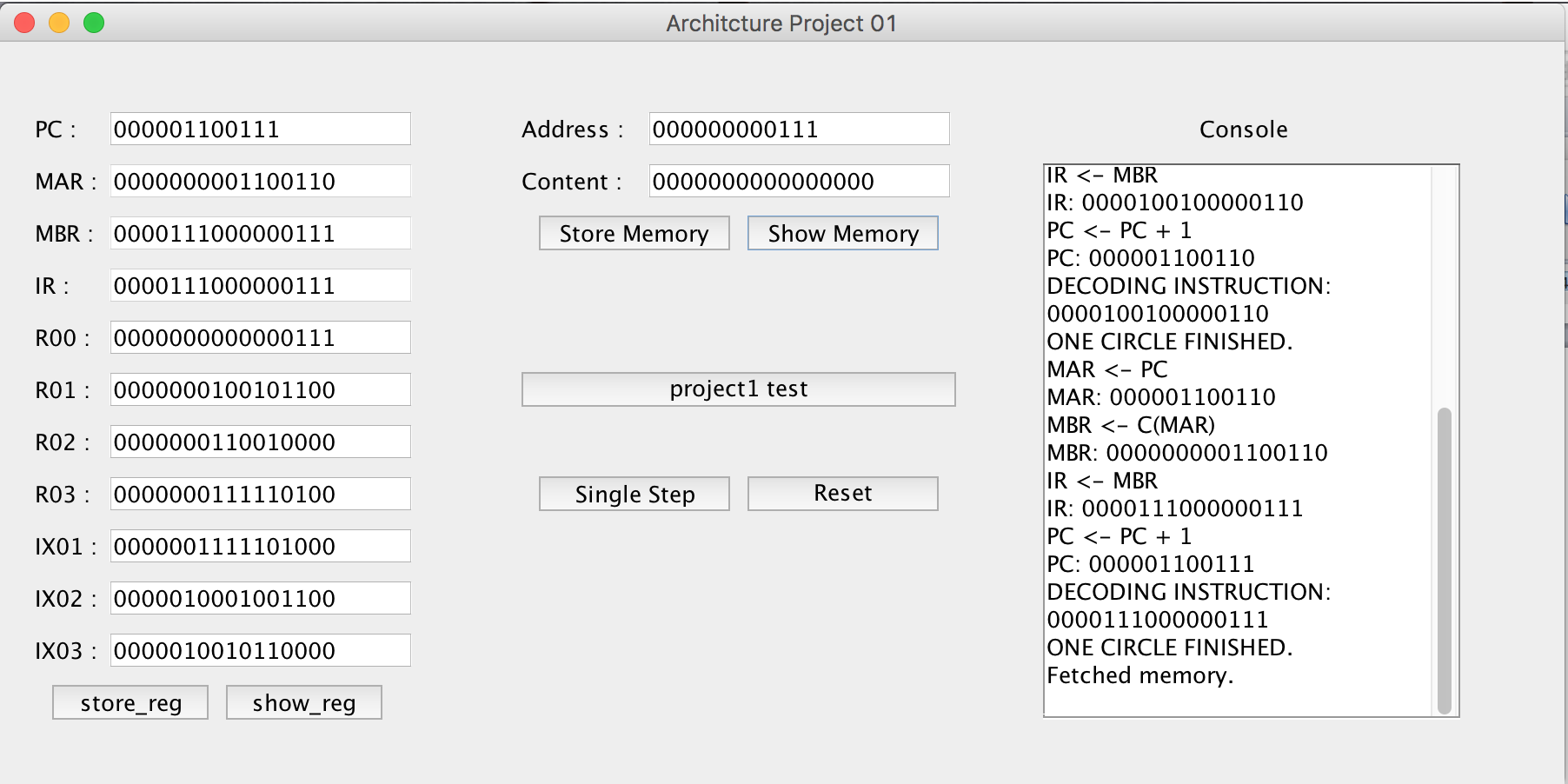
LDA:

Instruction: 0000111000000111

Opcode:000011 R:10 IX:0 I:0 ADDR：111

Excepted: R00 = 0000000000000111

Result: Passed



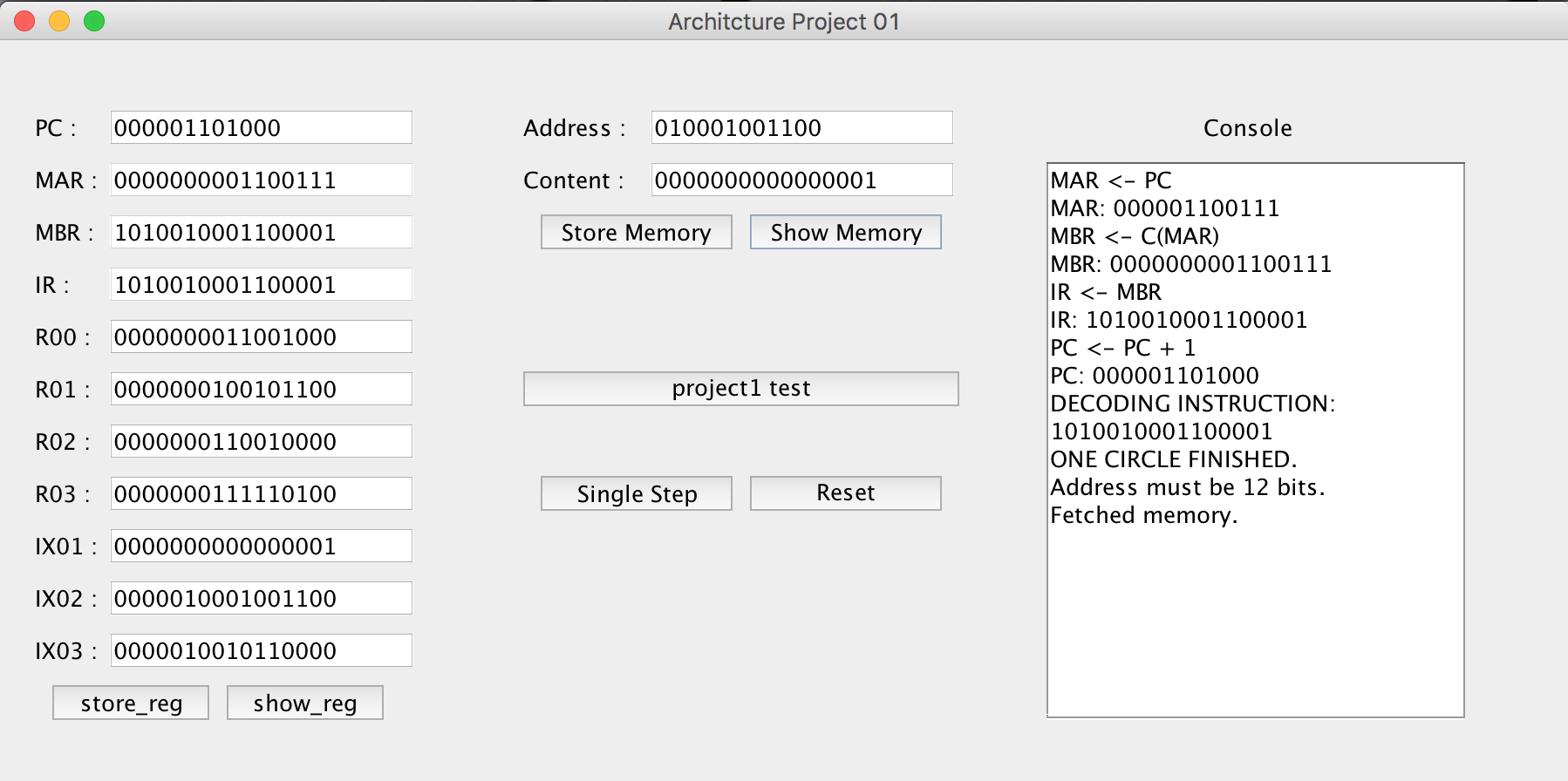
LDX:

Instruction: 1010010001100001

Opcode: 101001R:00 IX:01 I:1 ADDR：1

Excepted: IX01 = 0000000000000001

Result: Passed



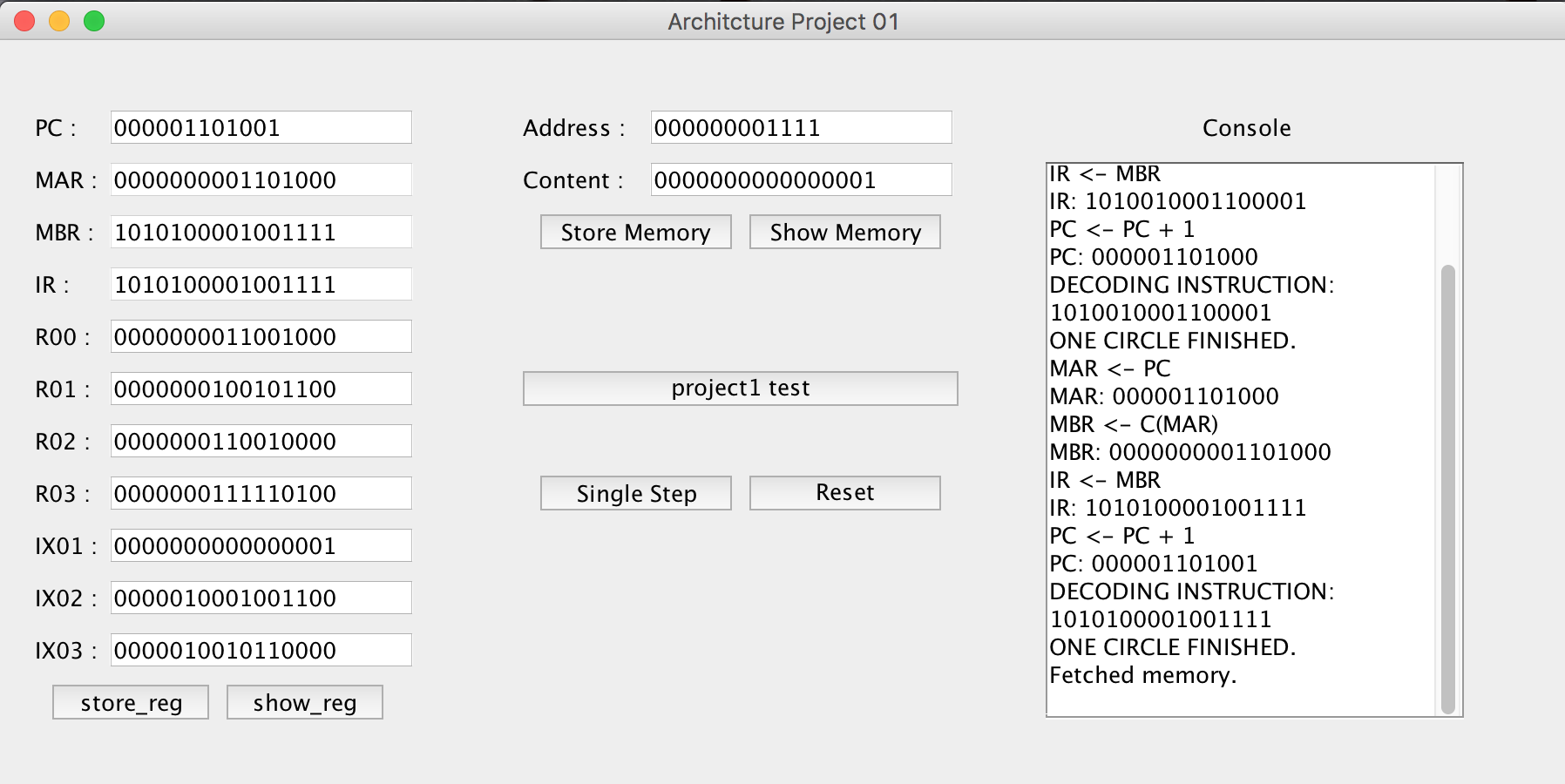
STX:

Instruction: 1010100001001111

Opcode: 101010 R:00 IX:01 I:1 ADDR：1111

Excepted: address(0000000000001111) = 0000000000000001

Result: Passed



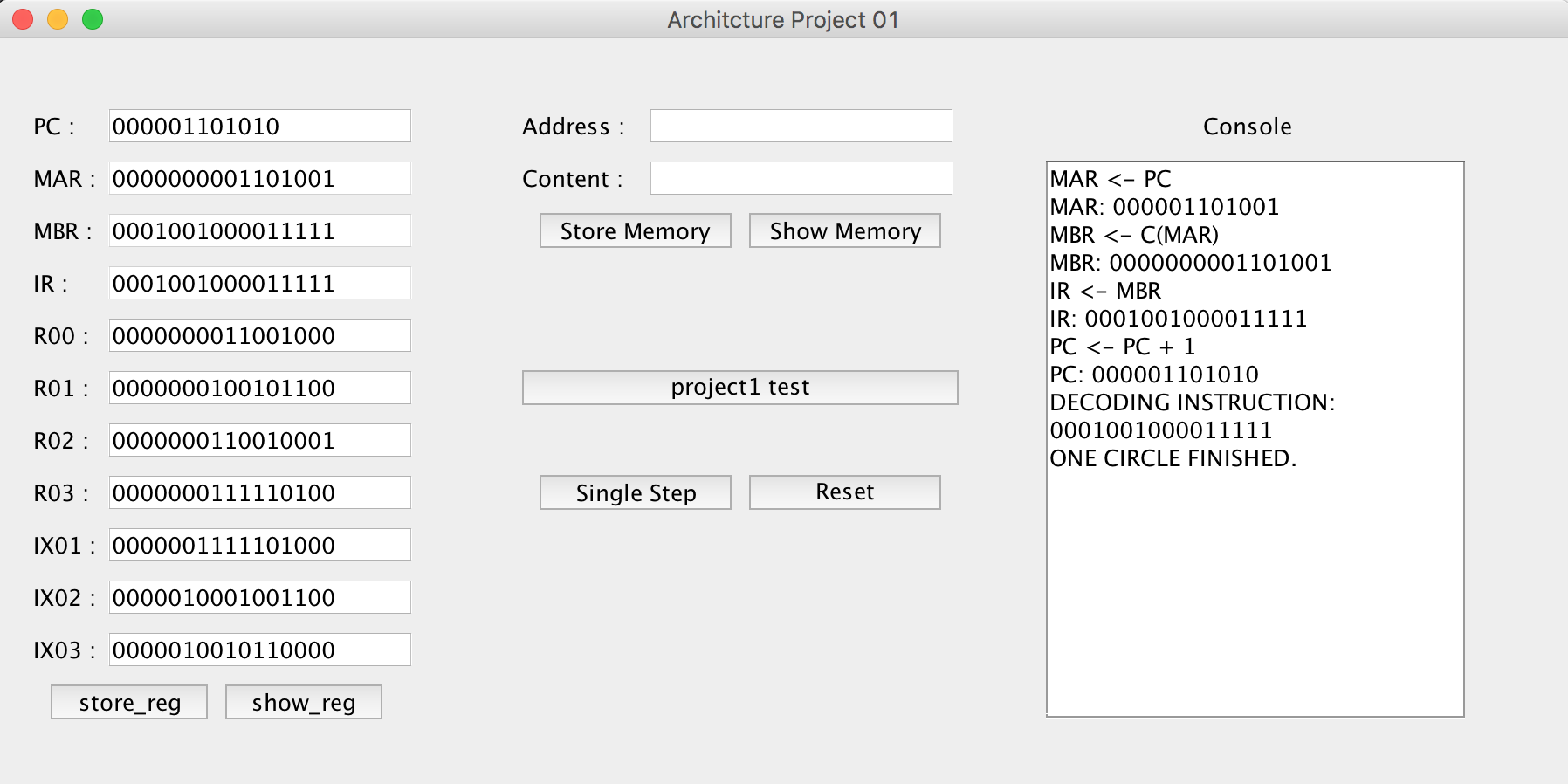
AMR:

Instruction: 0000101000011111

Opcode: 000010 R:10 IX:00 I:0 ADDR：11111

Excepted: R2 = 0000000110010001

Result: Passed



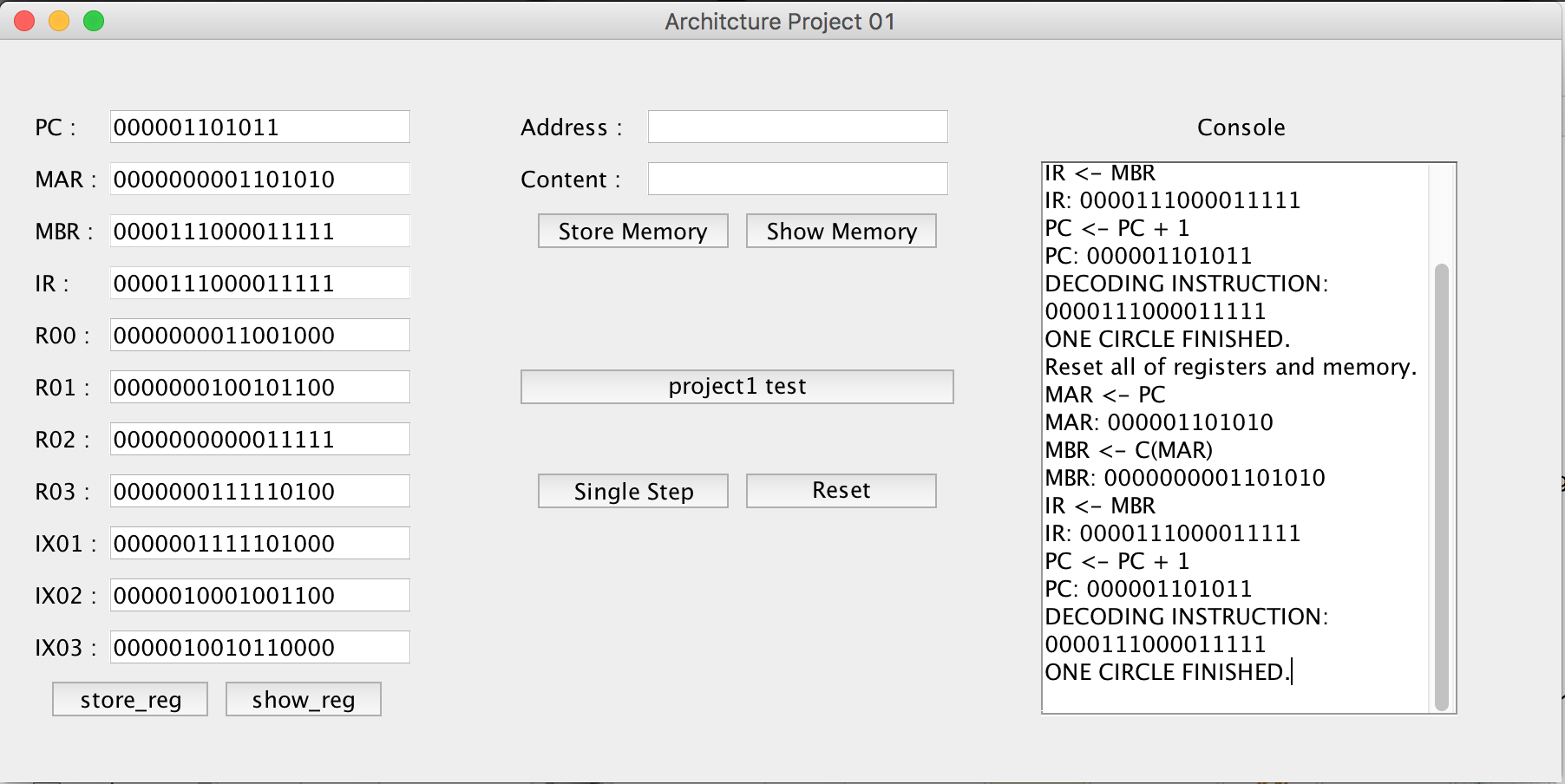
SMR:

Instruction: 0000111000011111

Opcode: 000011 R:10 IX:00 I:0 ADDR：11111

Excepted: R2 = 0000000000011111

Result: Passed



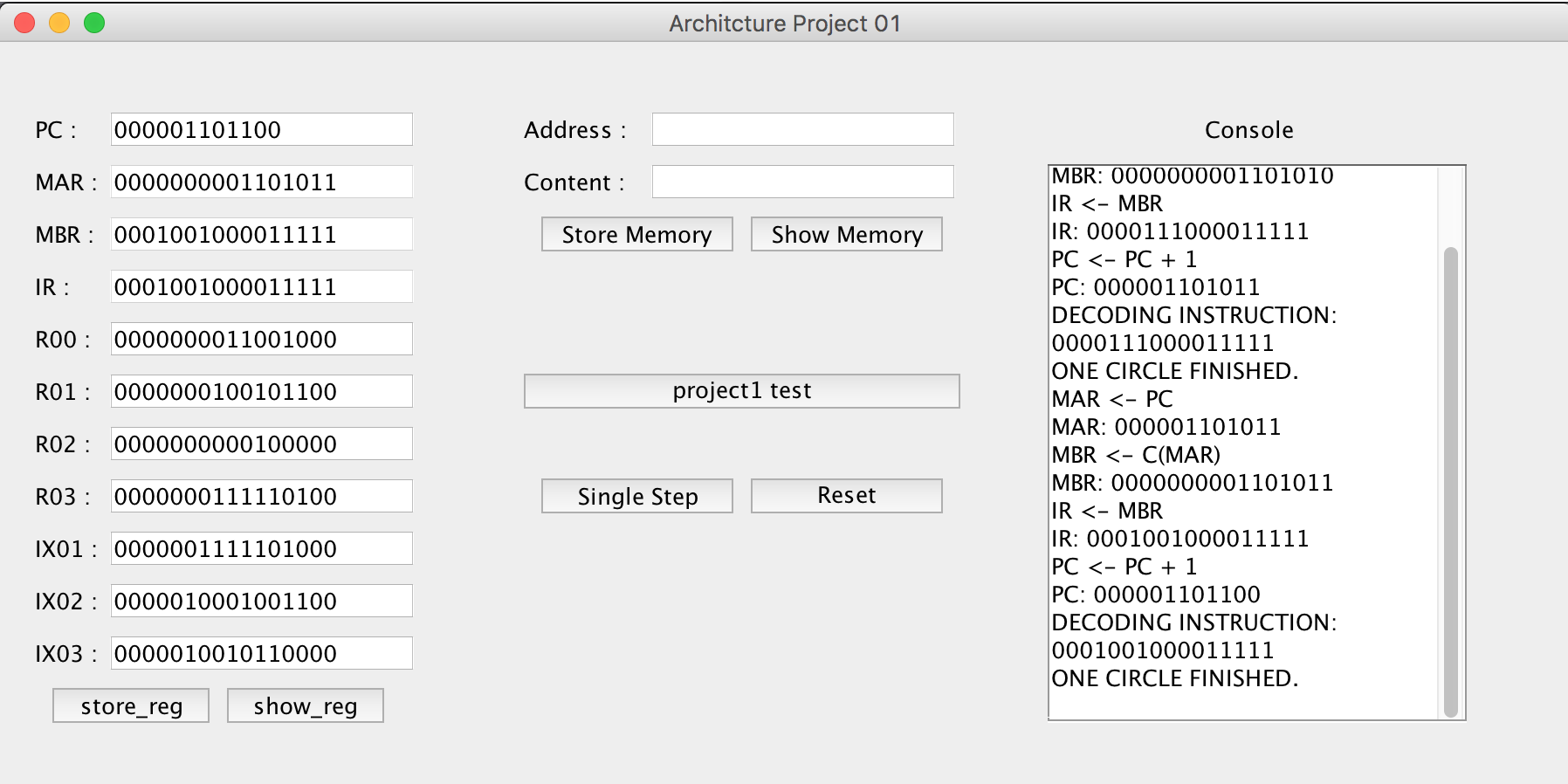
AIR:

Instruction: 0001001000011111

Opcode: 000100 R:10 IX:00 I:0 immed：11111

Excepted: R2 =0000000000100000

Result: Passed



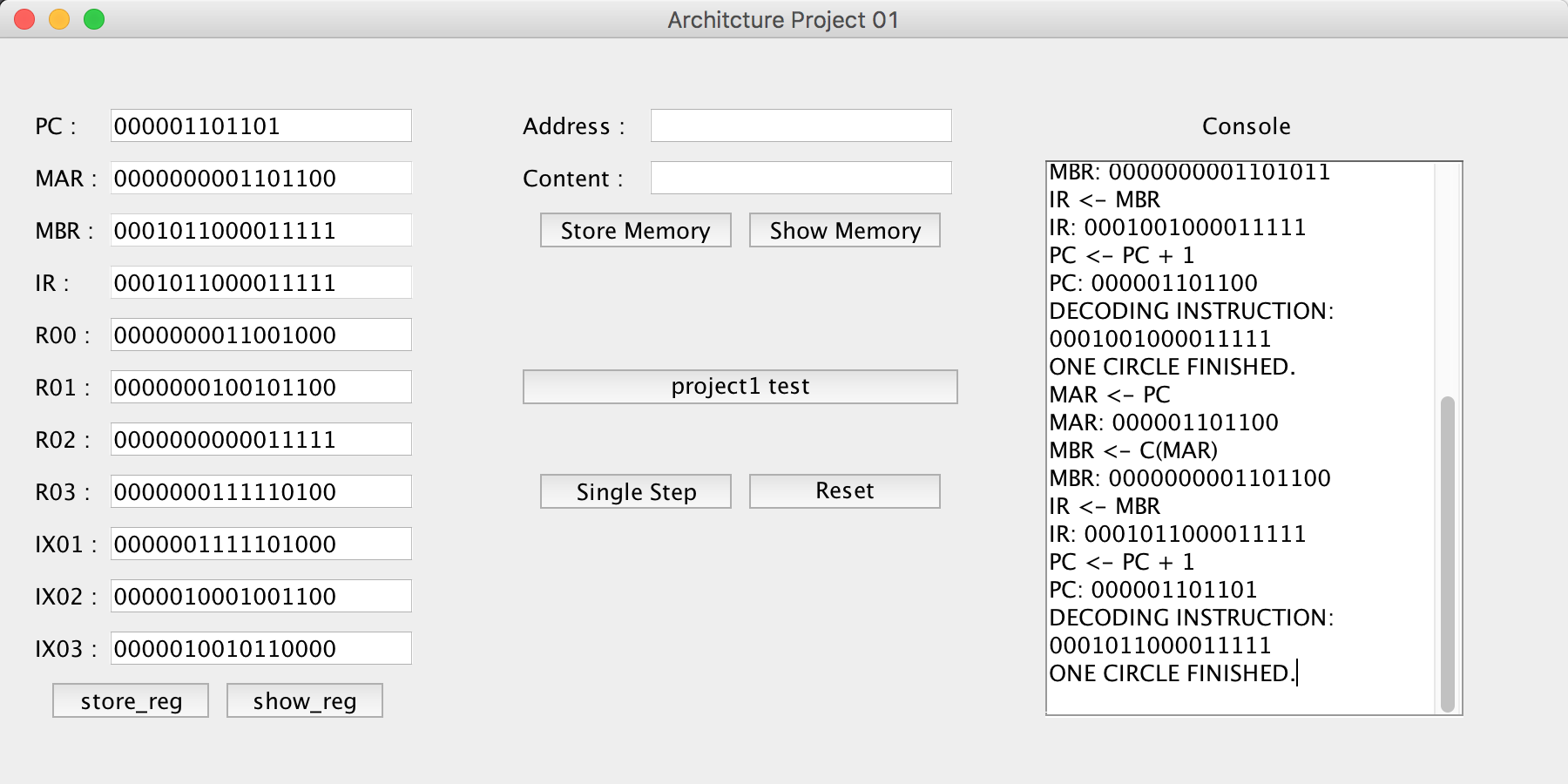
SIR:

Instruction: 1001011001011111

Opcode: 000101R:10 IX:00 I:0 immed：1111

Excepted: R2 = 0000000000011111

Result: Passed



**Transfer Instructions:**

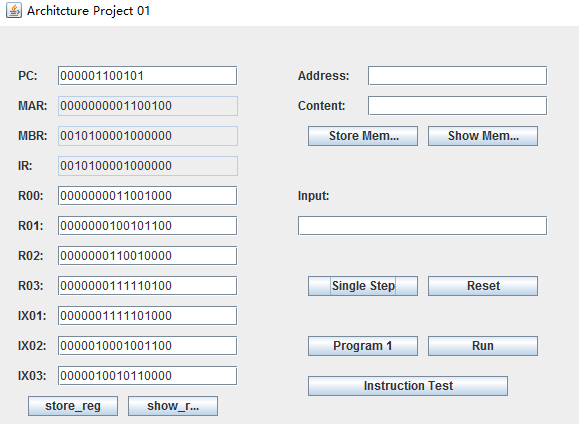
JZ:

Instruction: 001010 00 01 000000

Opcode: 001010 R:00 IX: 01 I: 0 address：0000

Excepted: R0 = 0000000011001000

Result: Passed



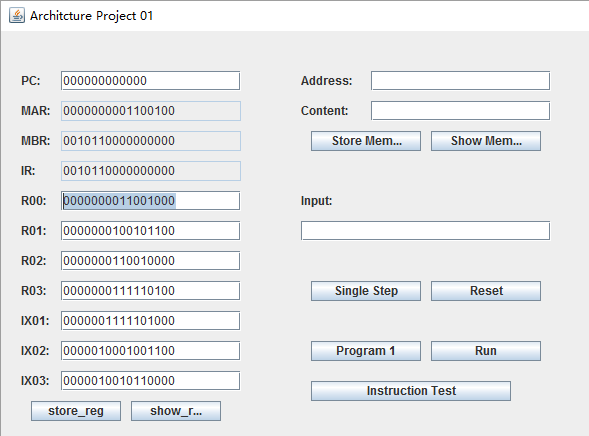
JNE:

Instruction: 001011 00 01 000000

Opcode: 001011 R:00 IX: 01 I: 0 address：0000

Excepted: R0 = 0000000011001000

Result: Passed



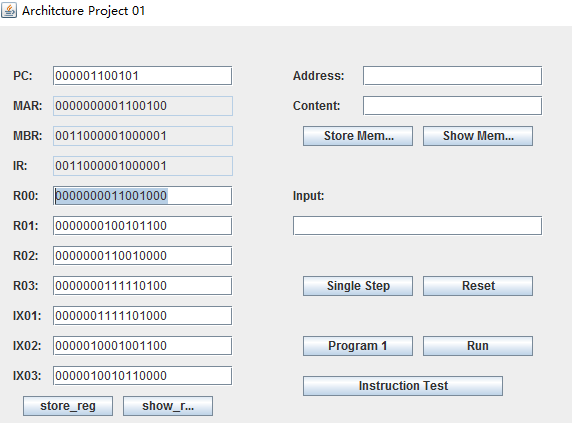
JCC:

Instruction: 001100 00 01 000001

Opcode: 001011 R:00 IX: 01 I: 0 address：0000

Excepted: R0 = 0000000011001000

Result: Passed



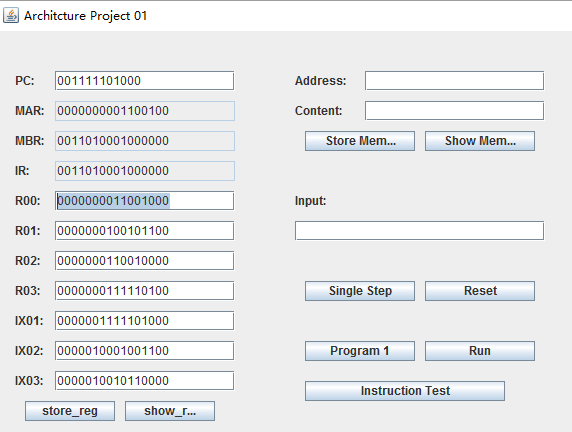
JMA:

Instruction: 001101 00 01 000001

Opcode: 001011 R:00 IX: 01 I: 0 address：0000

Excepted: R0 = 0000000011001000

Result: Passed



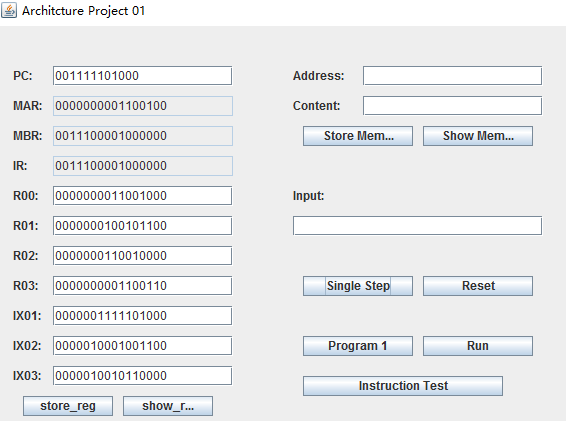
JSR:

Instruction: 001110 00 01 000001

Opcode: 001011 R:00 IX: 01 I: 0 address：0000

Excepted: R0 = 0000000011001000

Result: Passed



RFS:

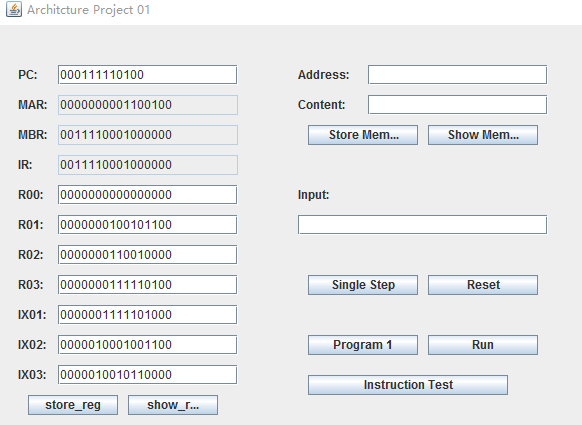
Instruction: 001111 00 01 000001

Opcode: 001011 R:00 IX: 01 I: 0 address：0000

Excepted: R0 = 0000000000000000

PC= 000111110100

Result: Passed



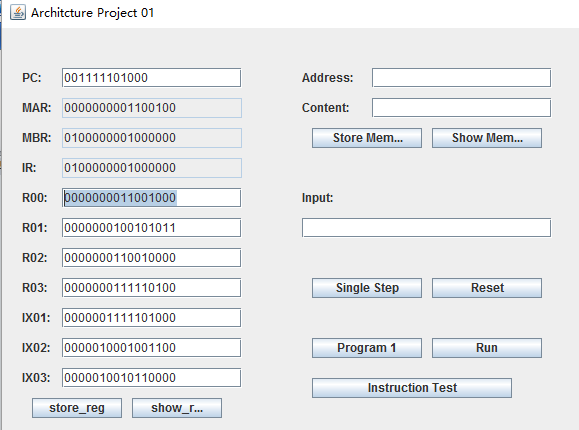
RFS:

Instruction: 010000 00 01 000001

Opcode: 001011 R:00 IX: 01 I: 0 address：0000

Excepted: R0 = 001111101000

Result: Passed



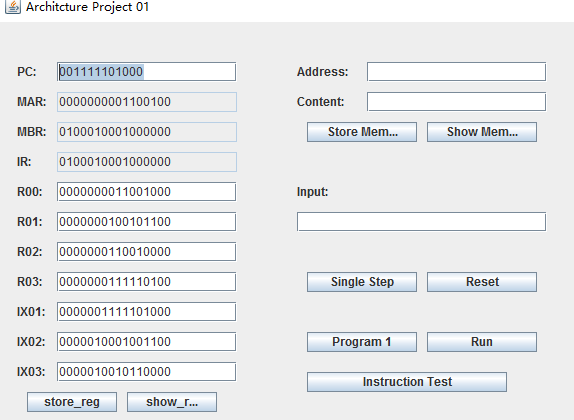
RFS:

Instruction: 010001 00 01 000001

Opcode: 001011 R:00 IX: 01 I: 0 address：0000

Excepted: PC = 001111101000

Result: Passed



MLT:

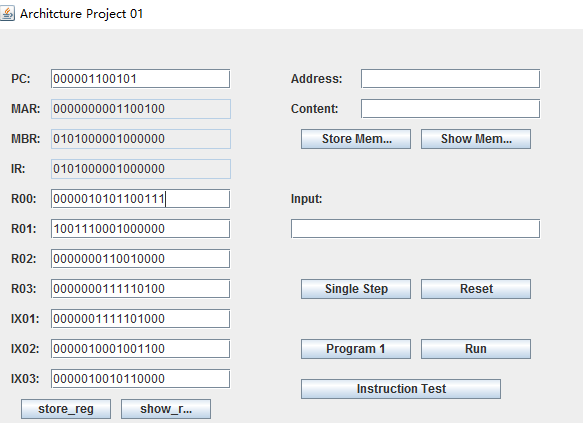
Instruction: 010100 00 01 000001

Opcode: 001011 RX:00 RY: 01 I: 0 address：0000

Excepted: RX-R0: 0000010101100111

RY-R1: 1001110001000000

Result: Passed



DVD:

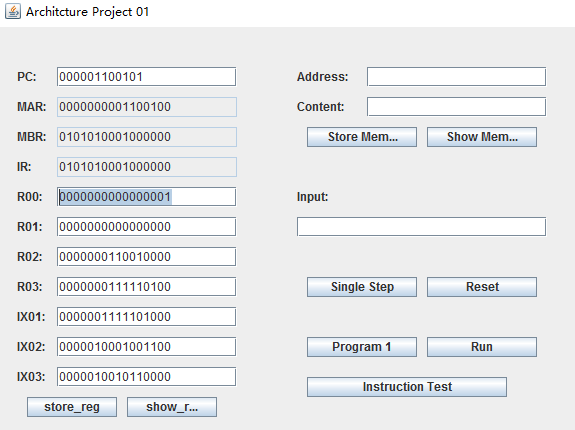
Instruction: 111101 00 01 000000

Opcode: 001011 RX:00 RY: 01 I: 0 address：0000

Excepted: RX-R0: 0000000000000001

RY-R1: 0000000000000000

Result: Passed



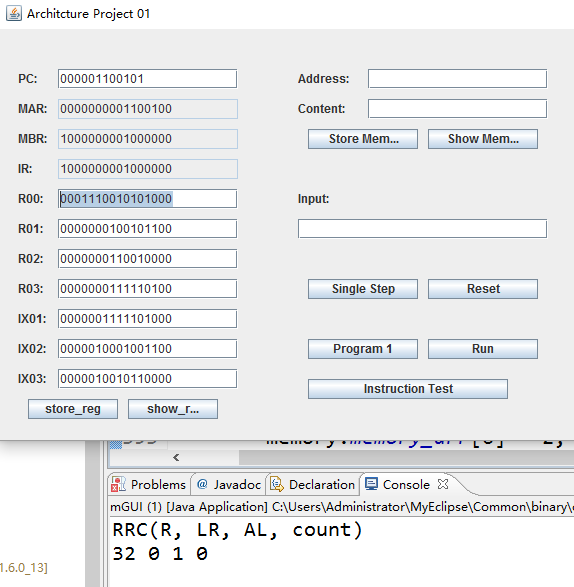
RRC:

Instruction: 100000 0 0 01 00 0000

Opcode: 100000 R:00 AL:0 BL:1 count: 0000

Excepted: R0: 0001110010101000

Result: Passed



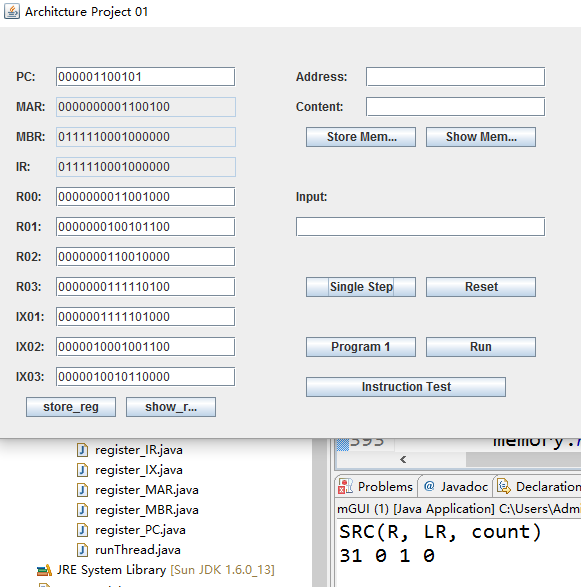
SRC:

Instruction: 011111 0 0 01 00 0000

Opcode: 011111 R:00 AL:0 BL:1 count: 0000

Excepted: R0: 0000000011001000

Result: Passed

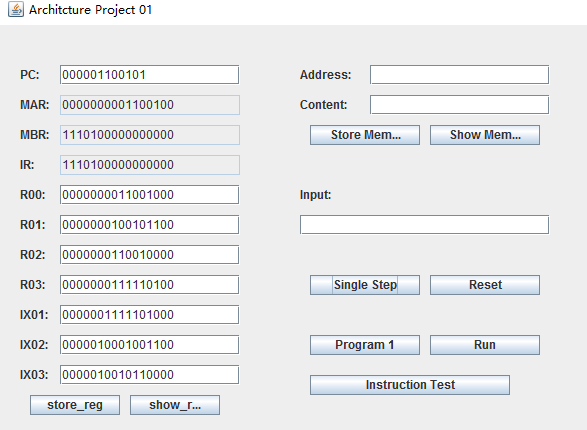


IN:

Instruction: 111101 00 01 000000

Opcode: 111101 R:00 DevID：000000

Result: Passed

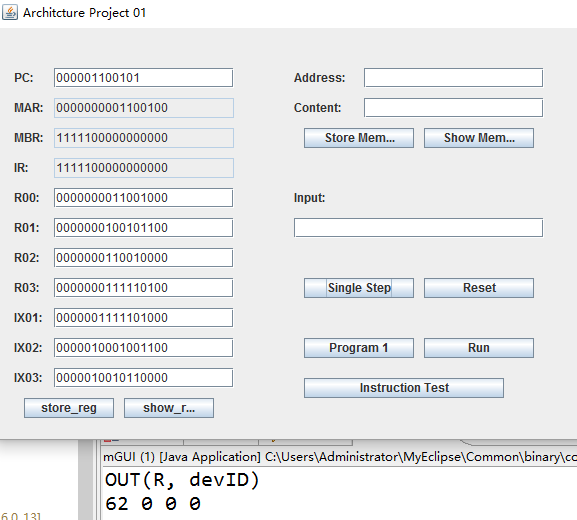


OUT:

Instruction: 111110 00 01 000000

Opcode: 111110 R:00 DevID：000000

Result: Passed



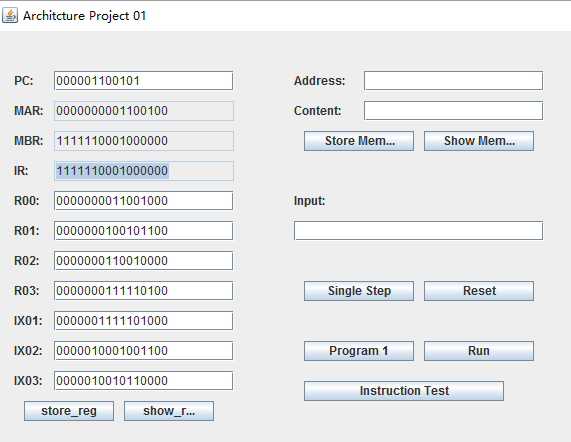
CHK:

Instruction: 111111 00 01 000000

Opcode: 111111 R:00 Devid：000000

Excepted: R0: 0000000011001000

Result: Passed



**3.4 Program1 :**

**Program1 Logical explanation:**

1. **input 20 numbers one by one in temp[20]**
2. **put in memory 500—519 and input the good number in find put in memory 520**
3. **C version Pseudocode of program1 in my simulator all the instructions are store from memory[100]**
4. **all the data are start from 500 memory before 100 are used to store temporary data**
5. **address from 11 to 31 are especially important int differ = find - temp[0]**

**int m, n, a, b;**

**int s**

**if (differ < 0)**

**m = -differ;**

**else**

**m = differ;**

**n = temp[0];**

**int i = 0**

**for( i = 1; i < 20; i++)**

**{**

**s = find - temp[i];**

**if(s < 0)**

**a = -s;**

**else**

**a = s;**

**if(a - m < 0)**

**{**

**m = a;**

**n = temp[i];**

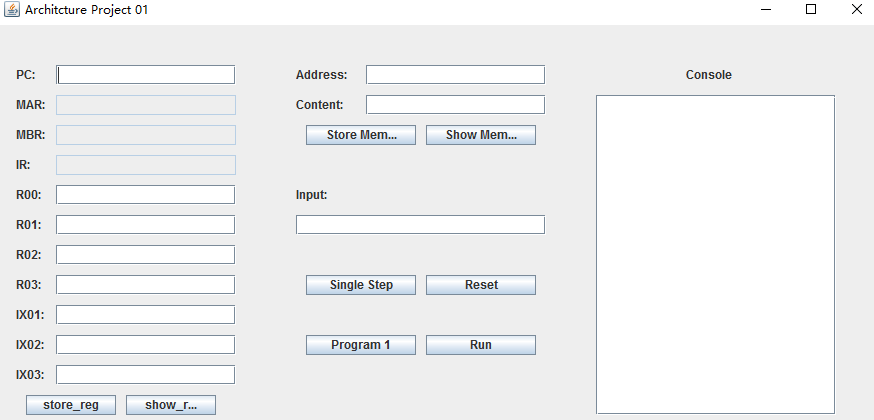
**}**

**}**

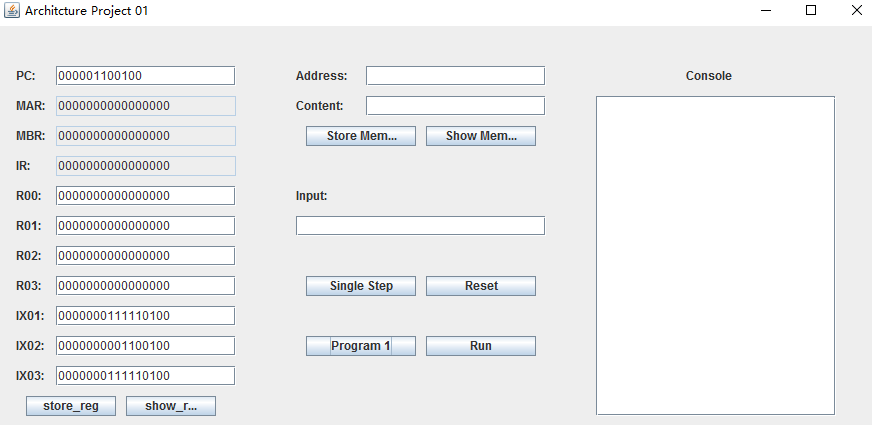
**printf("%d",n);**

**Program 1 Test:**

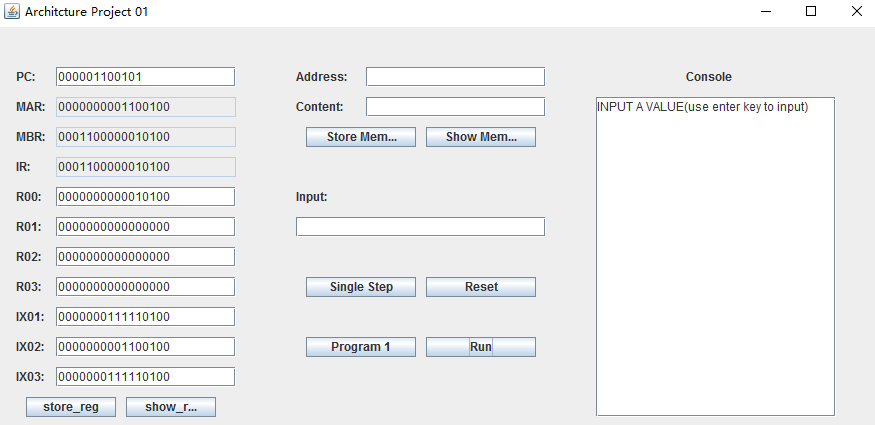
**Beginning Blank**



**First step: Click Program1 to load program1**



**Second Step: Click Run Button to run program1**



**Third Step : Input 20 number in the input textbox. Use enter key after putting a number in the input textbox every time.**

The testing results:

The red words is the 20 numbers that could be inputted by users.

The yellow background is the final result of program 1.

INPUT A VALUE(use enter key to input)

OUTPUT:

10100 (20)

OUTPUT:

1111111111111110 (65534)

INPUT A VALUE(use enter key to input)

OUTPUT:

10011 (19)

OUTPUT:

1000101011100 (4444)

INPUT A VALUE(use enter key to input)

OUTPUT:

10010 (18)

OUTPUT:

1101100100000011 (55555)

INPUT A VALUE(use enter key to input)

OUTPUT:

10001 (17)

OUTPUT:

101001101 (333)

INPUT A VALUE(use enter key to input)

OUTPUT:

10000 (16)

OUTPUT:

11011110 (222)

INPUT A VALUE(use enter key to input)

OUTPUT:

1111 (15)

OUTPUT:

1101111 (111)

INPUT A VALUE(use enter key to input)

OUTPUT:

1110 (14)

OUTPUT:

0 (0)

INPUT A VALUE(use enter key to input)

OUTPUT:

1101 (13)

OUTPUT:

1101000001010 (6666)

INPUT A VALUE(use enter key to input)

OUTPUT:

1100 (12)

OUTPUT:

101001101 (333)

INPUT A VALUE(use enter key to input)

OUTPUT:

1011 (11)

OUTPUT:

100010101110 (2222)

INPUT A VALUE(use enter key to input)

OUTPUT:

1010 (10)

OUTPUT:

10011100001111 (9999)

INPUT A VALUE(use enter key to input)

OUTPUT:

1001 (9)

OUTPUT:

1111111111111010 (65530)

INPUT A VALUE(use enter key to input)

OUTPUT:

1000 (8)

OUTPUT:

101001101 (333)

INPUT A VALUE(use enter key to input)

OUTPUT:

111 (7)

OUTPUT:

101001101 (333)

INPUT A VALUE(use enter key to input)

OUTPUT:

110 (6)

OUTPUT:

1000101101000 (4456)

INPUT A VALUE(use enter key to input)

OUTPUT:

101 (5)

OUTPUT:

10001010111 (1111)

INPUT A VALUE(use enter key to input)

OUTPUT:

100 (4)

OUTPUT:

11011110 (222)

INPUT A VALUE(use enter key to input)

OUTPUT:

11 (3)

OUTPUT:

1111100111 (999)

INPUT A VALUE(use enter key to input)

OUTPUT:

10 (2)

OUTPUT:

10001010111000 (8888)

INPUT A VALUE(use enter key to input)

OUTPUT:

1 (1)

OUTPUT:

1100001001 (777)

INPUT A VALUE(use enter key to input)

OUTPUT:

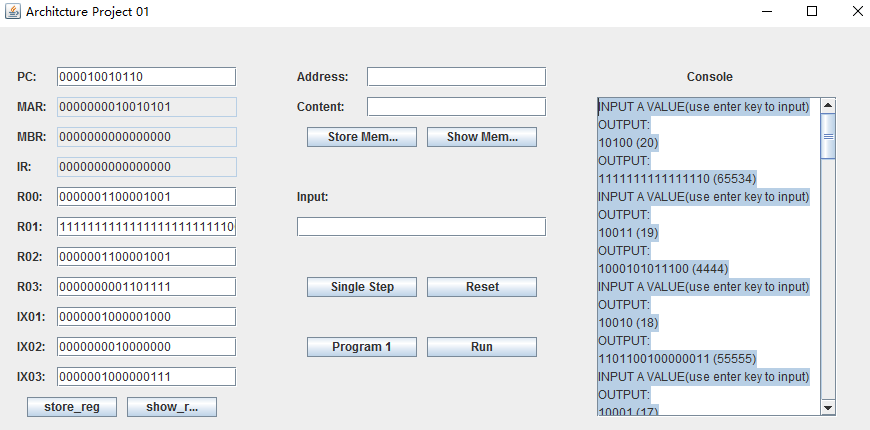
0 (0)

OUTPUT:

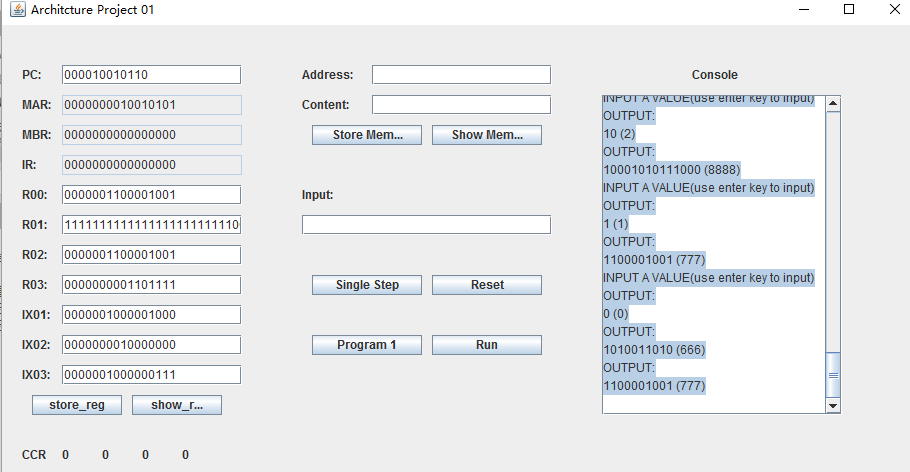
1010011010 (666)( this is the number, which enter by user to find the most closest number )

OUTPUT:

1100001001 (777)( this the result that is the most closest number to 666)



**The result:**



**3. Components**

**3.1 Interface Design**

**Phase1:**

In our project, we separate the whole project into two parts, UI design and implement background stage. The initial stage is that we try our best to create a UI that could put all the Components. In the left, we use JLabel to tag the name of the components and JTextBox to input contents and show results. We have Seven JButtons which have the name that tell users the functions on it. The STORE\_REGISTER button and SHOW\_REGISTER button, user could use it the Store the register information in any one Register and show the information about the all register. On the top of the screen is The SHOWMEMORY JButton, user could input the memory address and contents, then clicking STORE MEMORY store what users input. Users also could click the SHOW MEMORY to see what contents in address. In the middle of the screen we design Two JButtons that required to design in Phase one Project1 test, Single step. The reset button use to the reset all the information in the whole program. In the right of the screen, we have a JTextfield to display the all the information and state about this program.

Phase2: We add Input textbox using to input 20 numbers. Program1 button and Run button are necessary to implement Program1 test. User needs to click program1 then click Run button before inputting numbers. The right console is used to display information program 1 test Results.

**The final UI:**

