CISC Simulator Design Document

Rev3.0

Group 2
Jonathan Pritchett
Ziwei Li
Haorong Xiao
Siyuan Zhang

Table of Contents

Description

The CISC Simulator is a program that creates a simple simulator of a rudimentary computer. In its current iteration, this simulated computer consists of a set of registers, a simple memory, a small set of instructions, and a graphical user interface to serve as the computer front panel.

Data Representation

Data in the computer shall be stored in 16-bit words. To represent this, a class "Word" will be created which stores a 16-bit word as a 16 char array. This class will have accessor methods allowing for easy translation of this data into ints and strings and will allow for accessing of individual and subsets of bits.

Registers

In this first iteration of the CISC simulator, the following 11 16-bit registers are used.

R0...R3	4 General-purpose registers
PC	Program counter
IR	Instruction Register
MAR	Memory Address Register
MBR	Memory Buffer Register
X1...X3	3 Index Registers

Each register consists of a single Word object. All of the registers are collectively stored within a single class, "Registers", which may be instantiated once and shared among the components of the simulator.

Memory

The Memory system for this version consists of a single class that contains an array of 2048 "Word" objects. The memory is instantiated once and shared throughout the simulator.

Instructions

Thrity-five instructions are currently implemented within the simulator.

Instruction	OpCode	Description

LDR r, x, address[,I]	1	Load Register From Memory, r = 0..3
		r<-c(EA)
		r <- c(c(EA)), if I bit set

STR r, x, address[,I]	2	Store Register To Memory, r = 0..3 Memory(EA)<-c(r)
LDA r, x, address[,I]	3	Load Register with Address, r = 0..3 r<-EA
LDX x, address[,I]	41	Load Index Register from Memory, x = 1..3 Xx <- c(EA)
STX x, address[,I]	42	Store Index Register to Memory. X = 1..3 Memory(EA) <- c(Xx)
JZ r, x, address[,I]	10	Jump If Zero: If c(r) = 0, then PC<-EA Else PC <- PC+1
JNE r, x, address[,I]	11	Jump If Not Equal: If c(r) != 0, then PC<-- EA Else PC <- PC + 1
JCC cc, x, address[,I]	12	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
JMA x, address[,I]	13	Unconditional Jump To Address PC <- EA, Note: r is ignored in this instruction
JSR x, address[,I]	14	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

RFS Immed	15	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.

SOB r, x, address[,I]	16	Subtract One and Branch. R = 0..3
		r<-c(r) – 1
		If c(r) > 0,PC <- EA;
		Else PC <- PC + 1

JGE r,x, address[,I]	17	Jump Greater Than or Equal To: If c(r) >= 0, then PC <- EA Else PC <- PC + 1
AMR r, x, address[,I]	4	Add Memory To Register, r = 0..3 r<-c(r) + c(EA)
SMR r, x, address[,I]	5	Subtract Memory From Register, r = 0..3 r<-c(r) – c(EA)
AIR r, immed	6	AddImmediate 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
SIR r, immed	7	SubtractImmediatefrom 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
MLT rx,ry	20	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
DVD rx,ry	21	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)

TRR rx, ry	22	Test the Equality of Register and Register If c(rx) = c(ry), set cc(4)<-1; else, cc(4)<-0
AND rx, ry	23	Logical And of Register and Register c(rx)<-c(rx) AND c(ry)
ORR rx, ry	24	Logical Or of Register and Register c(rx)<-c(rx) OR c(ry)
NOT rx	25	Logical Not of Register To Register C(rx)<-NOT c(rx)
SRC r, count, L/R, A/L	31	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
RRC r, count, L/R, A/L	32	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
IN r, devid	61	Input Character To Register from Device, r = 0..3
OUT r, devid	62	Output Character to Device from Register, r = 0..3
CHK r, devid	63	Check Device Status to Register, r = 0..3 c(r) <- device status
FADD fr, x, address[,I]	33	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
FSUB fr, x, address[,I]	34	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
VADD fr, x, address[,I]	35	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 V1 be vector at address; Let V2 be vector at address+1

		Then, V1[i] = V1[i]+ V2[i], i = 1, c(fr).
VSUB fr, x, address[,I]	36	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 V1 be vector at address; Let V2 be vector at address+1 Then, V1[i] = V1[i] - V2[i], i = 1, c(fr).
CNVRT r, x, address[,I]	37	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.

LDFR fr, x, address [,i]	50	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

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

Pipelining

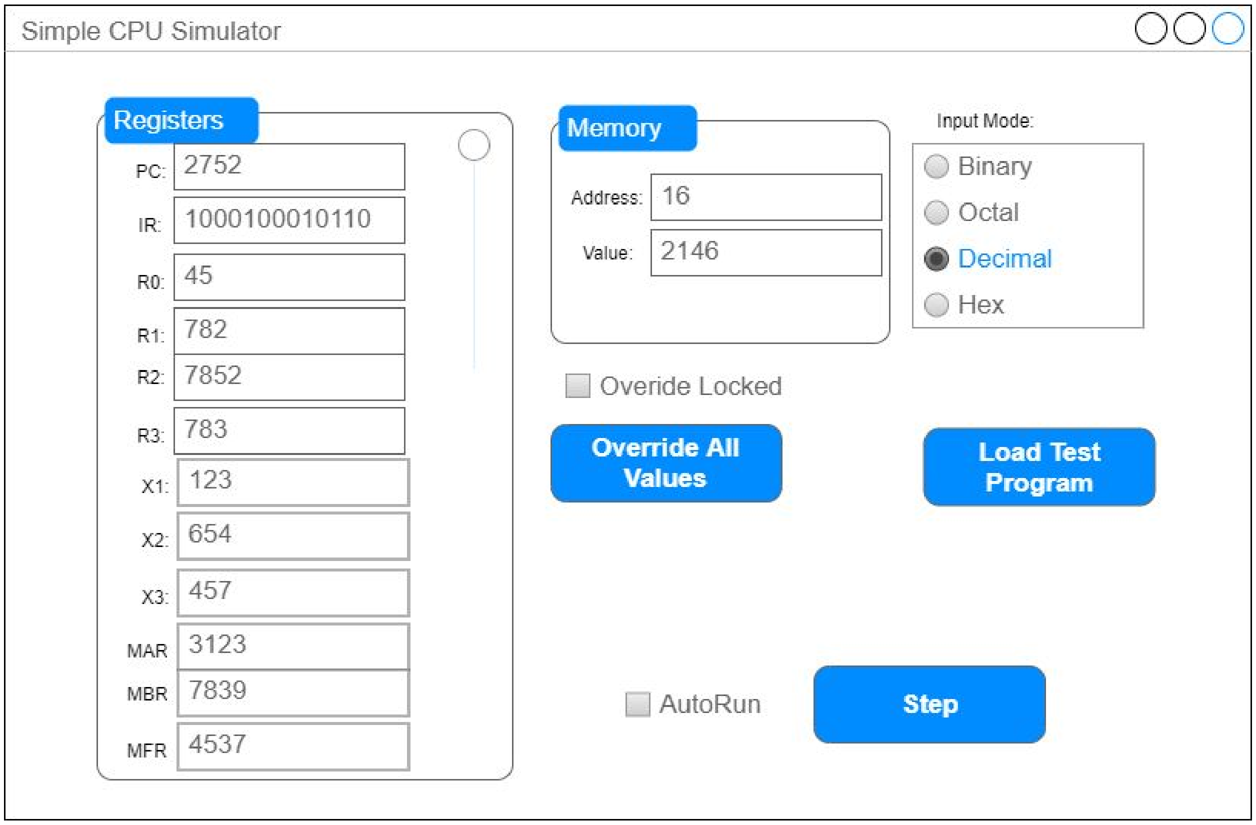
To simulate a pipelined architecture, four pipeline stages are implemented, “Instruction Fetch (IF)”, “Instruction Decode(ID)”, “Execute Instructing (EX)”, and “Write to Memory (MEM)”. If a “branch” instruction reaches the MEM stage, the other stages are flushed, and the program resumes from wherever the instruction sets the PC to. Similarly, if the instruction in the MEM stage accessing the same register or memory location as the instruction in the EX stage the other stages will be flushed and the pc set to the MEM instructions address + 1.

Front Panel GUI

The front panel GUI provides the user with a way to view the contents of all of the registers and memory locations used by the simulator. It also allows the user to override any of these values manually and to run the simulator from its current state.

Figures

UI Mockup



PC: 2752
IR: 1000100010110
R0: 45
R1: 782
R2: 7852
R3: 783
X1: 123
X2: 654
X3: 457
MAR 3123
MBR 7839
MFR 4537

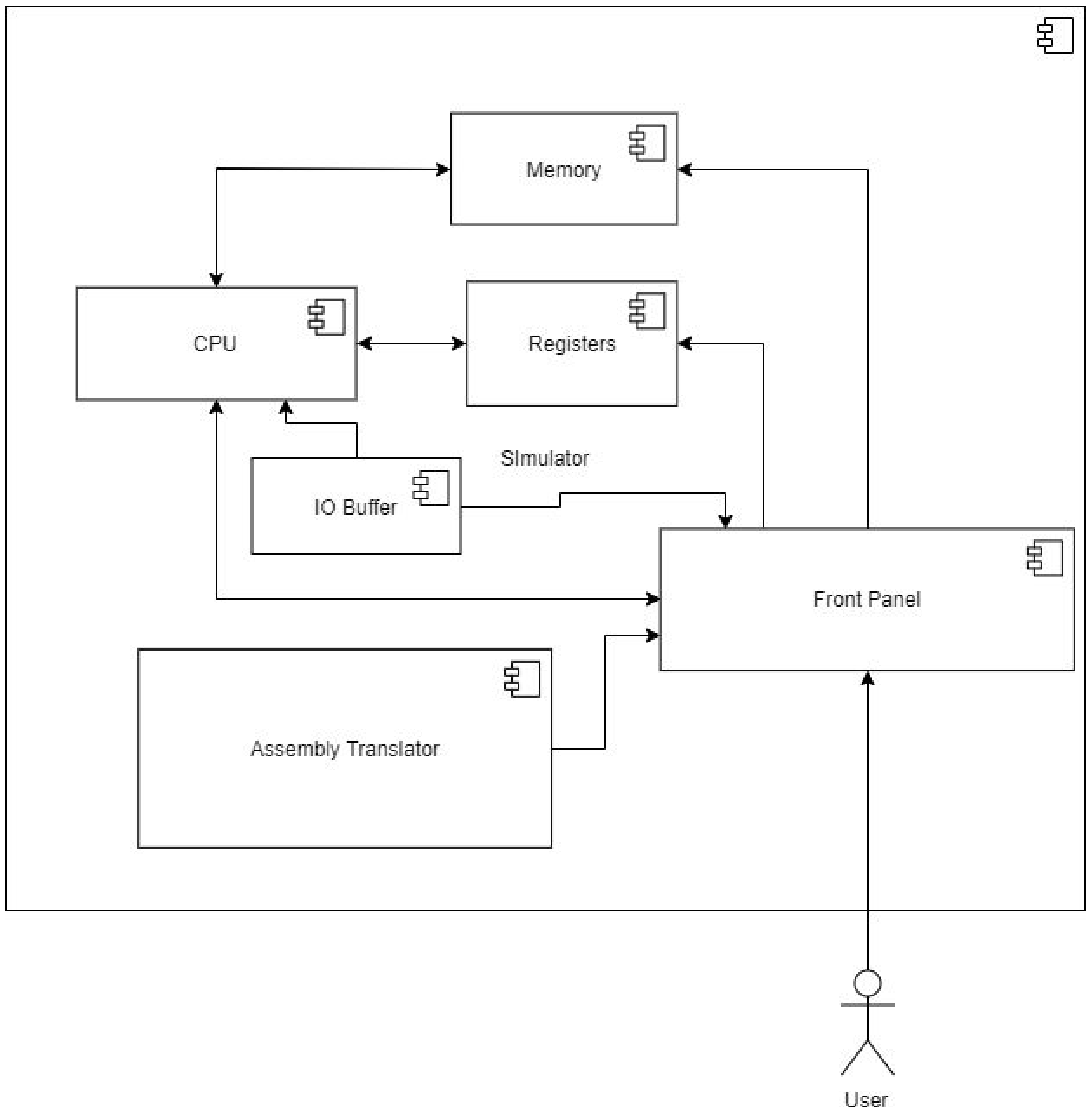
Address: 16
Value: 2146

Input Mode: Binary Octal [✓ Decimal] Hex

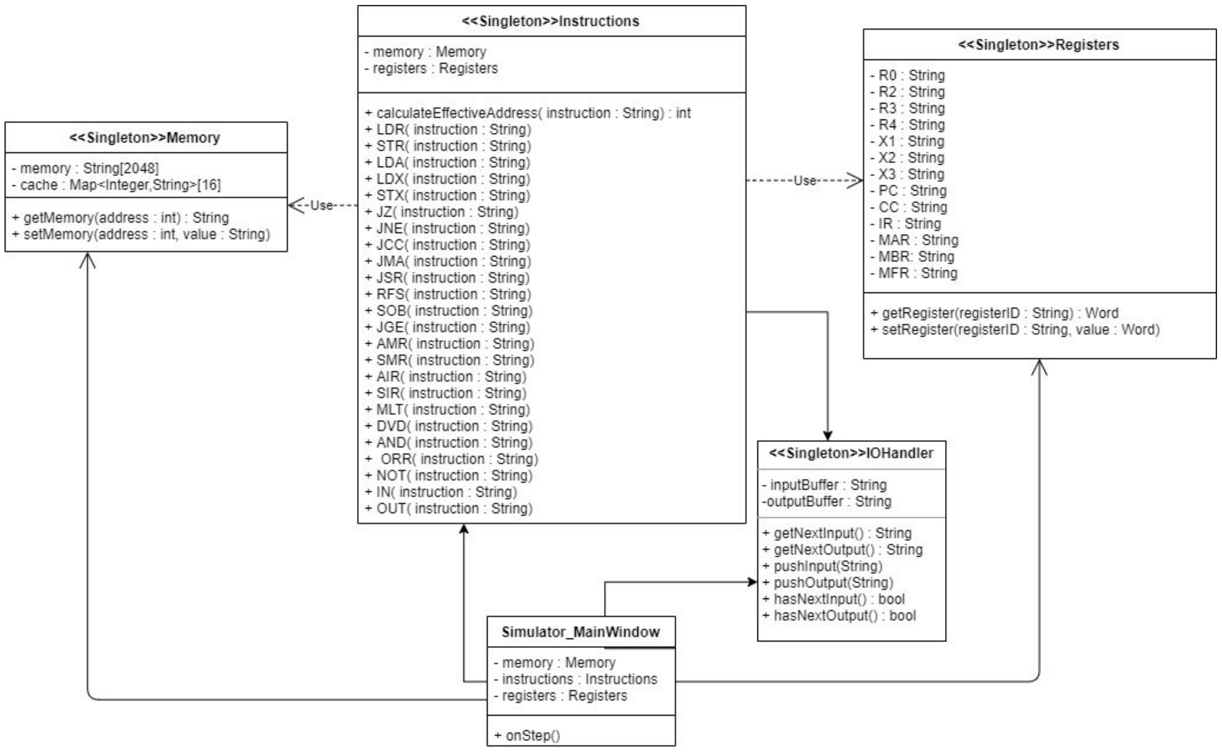
☐ Overide Locked

☐ AutoRun

Component Diagram



Class Diagram


<<Singleton>>Instructions
- memory : Memory
- registers : Registers
+ calculateEffectiveAddress(instruction : String) : int
+ LDR(instruction : String)
+ STR(instruction : String)
+ LDA(instruction : String)
+ LDX(instruction : String)
+ STX(instruction : String)
+ JZ(instruction : String)
+ JNE(instruction : String)
+ JCC(instruction : String)
+ JMA(instruction : String)
+ JSR(instruction : String)
+ RFS(instruction : String)
+ SOB(instruction : String)
+ JGE(instruction : String)
+ AMR(instruction : String)
+ SMR(instruction : String)
+ AIR(instruction : String)
+ SIR(instruction : String)
+ MLT(instruction : String)
+ DVD(instruction : String)
+ AND(instruction : String)
+ ORR(instruction : String)
+ NOT(instruction : String)
+ IN(instruction : String)
+ OUT(instruction : String)
<<Singleton>>Memory
- memory : String[2048]
- cache : Map<Integer,String>[16]
+ getMemory(address : int) : String
+ setMemory(address : int, value : String)
Use
<<Singleton>>Registers
- R0 : String
- R2 : String
- R3 : String
- R4 : String
- X1 : String
- X2 : String
- X3 : String
- PC : String
- CC : String
- IR : String
- MAR : String
- MBR: String
- MFR : String
+ getRegister(registerID : String) : Word
+ setRegister(registerID : String, value : Word)
Use
<<Singleton>>IOHandler
- inputBuffer : String
-outputBuffer : String
+ getNextInput() : String
+ getNextOutput() : String
+ pushInput(String)
+ pushOutput(String)
+ hasNextInput() : bool
+ hasNextOutput() : bool
Simulator_MainWindow
- memory : Memory
- instructions : Instructions
- registers : Registers
+ onStep()

Test Program 1

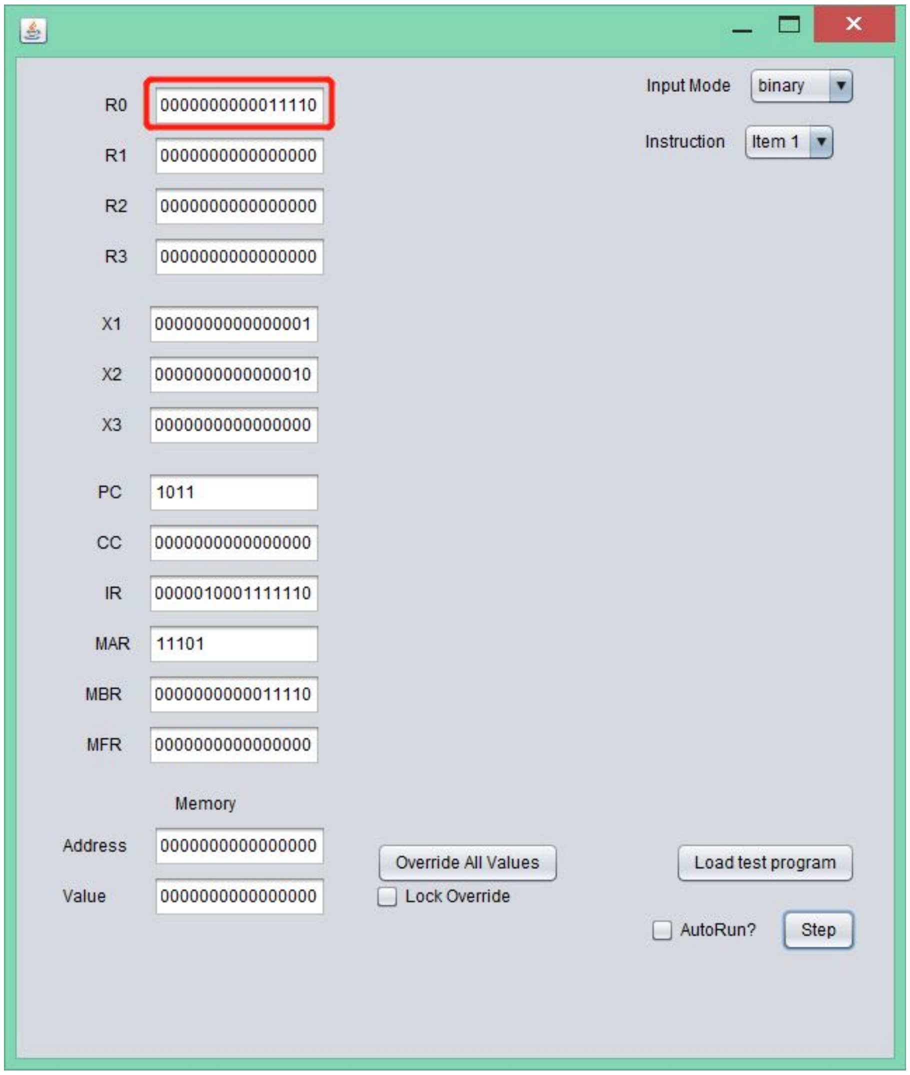
We initialize the test program by pressing the "load test program" button in the lower right corner.

We set up 4 memory addresses and 2 index registers:

- memory address 31 stores integer value 29
- memory address 30 stores integer value 31
- memory address 29 stores integer value 30
- index register X1 stores integer value 1 and index register X2 stores integer value 2.

Then, we have stored 5 instructions in memory:

(1) Load register from memory: 000001 00 01 1 11110. (EA = memory[31] = 29)
This instruction loads effective address 29 by indexing and indirecting to register R0, which stores 30 representing by '11110'.'

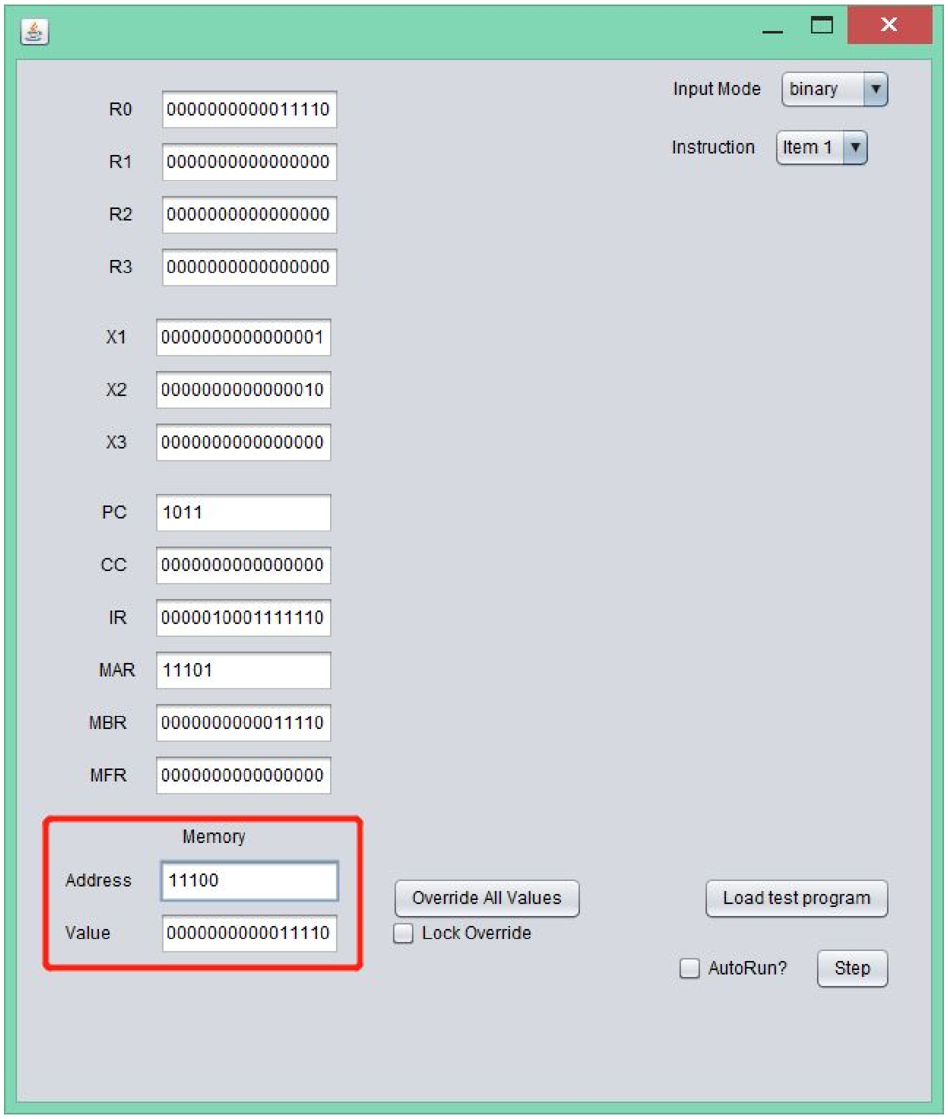


R0 0000000000011110
R1 0000000000000000
R2 0000000000000000
R3 0000000000000000
X1 0000000000000001
X2 0000000000000010
X3 0000000000000000
PC 1011
CC 0000000000000000
IR 0000010001111110
MAR 11101
MBR 0000000000011110
MFR 0000000000000000

Memory
Address 0000000000000000
Value 0000000000000000

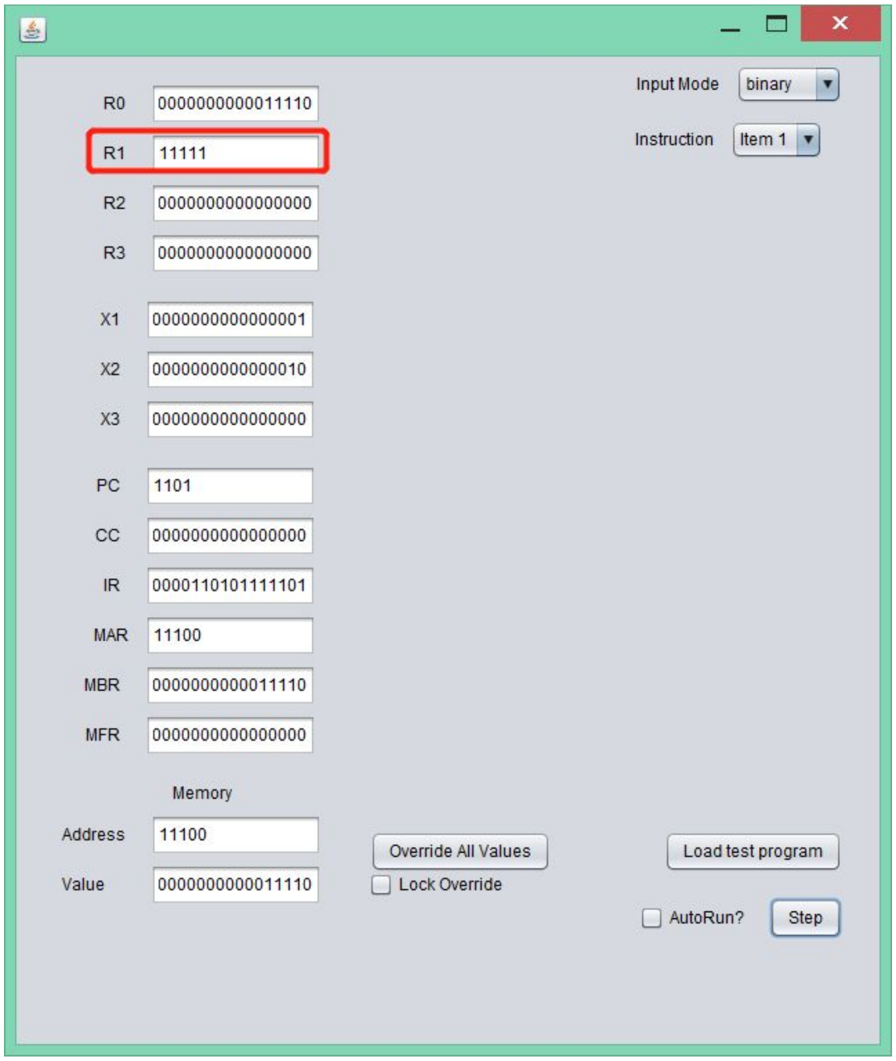
Input Mode binary
Instruction Item 1

(2) Store register to memory: 000010 00 00 0 11100. (EA = 28)
This instruction stores the value of R0 to the effective address 28. Then, the content of memory address 28 become 30.



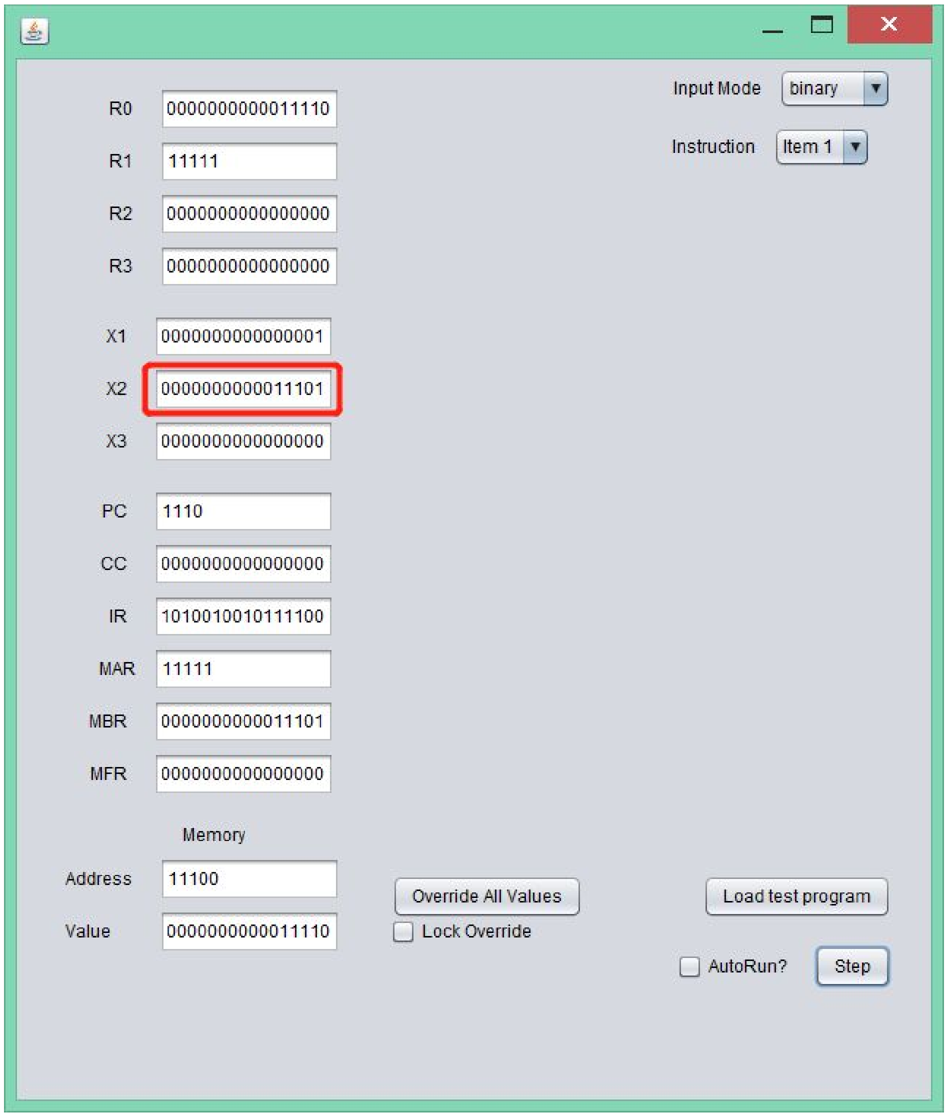
R0 0000000000011110
R1 0000000000000000
R2 0000000000000000
R3 0000000000000000
X1 0000000000000001
X2 0000000000000010
X3 0000000000000000
PC 1011
CC 0000000000000000
IR 0000010001111110
MAR 11101
MBR 0000000000011110
MFR 0000000000000000
Memory
Address 11100
Value 0000000000011110
Input Mode binary
Instruction Item 1

(3) Load register with address : 000011 01 01 1 11101. (EA = memory[30] = 31)
This instruction loads the content in effective address to register R1 by indexing and indirecting. The effective address here is 31. Therefore, 31 is stored in R1.



R0	0000000000011110
R1	11111
R2	0000000000000000
R3	0000000000000000
X1	0000000000000001
X2	0000000000000010
X3	0000000000000000
PC	1101
CC	0000000000000000
IR	0000110101111101
MAR	11100
MBR	0000000000011110
MFR	0000000000000000
Address	11100
Value	0000000000011110

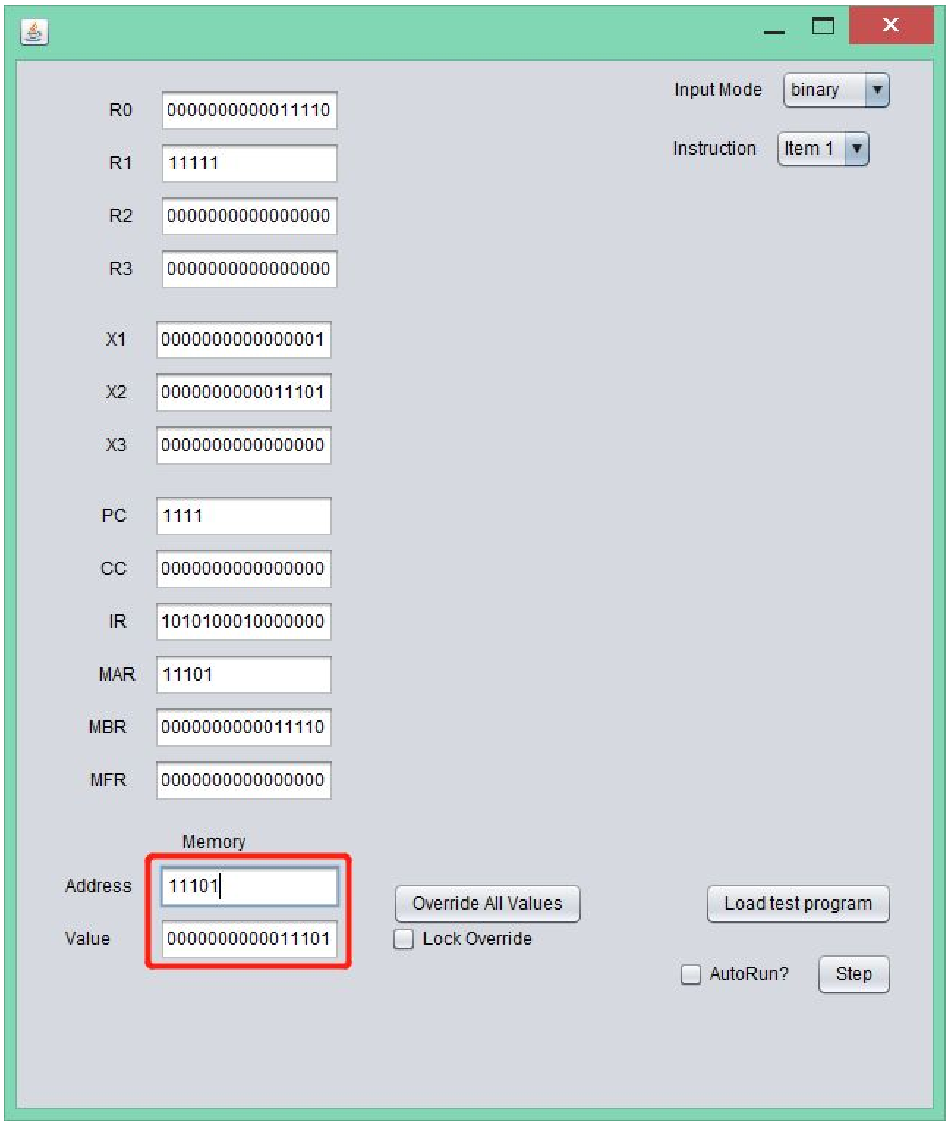
(4) Load index register from memory: 101001 00 10 1 11100. (EA = memory[30] = 31)
This instruction loads the content in effective address to index register X2 by indexing and indirecting. Consequently, the content of Index Register 2 becomes 29.


R0 0000000000011110
R1 11111
R2 0000000000000000
R3 0000000000000000
X1 0000000000000001
X2 0000000000011101
X3 0000000000000000
PC 1110
CC 0000000000000000
IR 1010010010111100
MAR 11111
MBR 0000000000011101
MFR 0000000000000000

Memory
Address 11100
Value 0000000000011110

Input Mode binary
Instruction Item 1

(5) Store index register to memory: 101010 00 10 0 00000. (EA = 29)
This instruction stores the value of index register 2 to memory[29].



R0	0000000000011110
R1	11111
R2	0000000000000000
R3	0000000000000000
X1	0000000000000001
X2	0000000000011101
X3	0000000000000000
PC	1111
CC	0000000000000000
IR	1010100010000000
MAR	11101
MBR	0000000000011110
MFR	0000000000000000

Input Mode: binary
Instruction: Item 1

Memory
Address: 11101
Value: 0000000000011101

Test Program 2

Here is the general idea of our code. We store the paragraph and the word to find in the memory. Every character is stored in an address. We keep recording the character until we meet a space, a period or a question mark. Then, we record the word number and the sentence number. After that, we compare the word we record for now and the word supposed to find. If it is the same, the program will output the word number and the sentence number. Otherwise, it will keep recording the next word and compare until it finds the answer.