IITB-RISC-22

Koustubh Rao
Moiz Shakruwala
Pinkesh Raghuvanshi
Sarthak Mittal

May 7, 2022

Contents

1 Abstract **1**

2 Intoduction **1**
2.1 Instruction Format 1
2.2 Instruction Set Encoding 2
2.3 Flag Settings 2
2.4 Addressing Modes 3

3 Implementation Idea **3**
3.1 Microprocessor Blocks 3
3.2 Derived Instructions 3

4 Microprocessor Design **5**
4.1 State Elaboration 5
4.2 Flow Diagrams 6

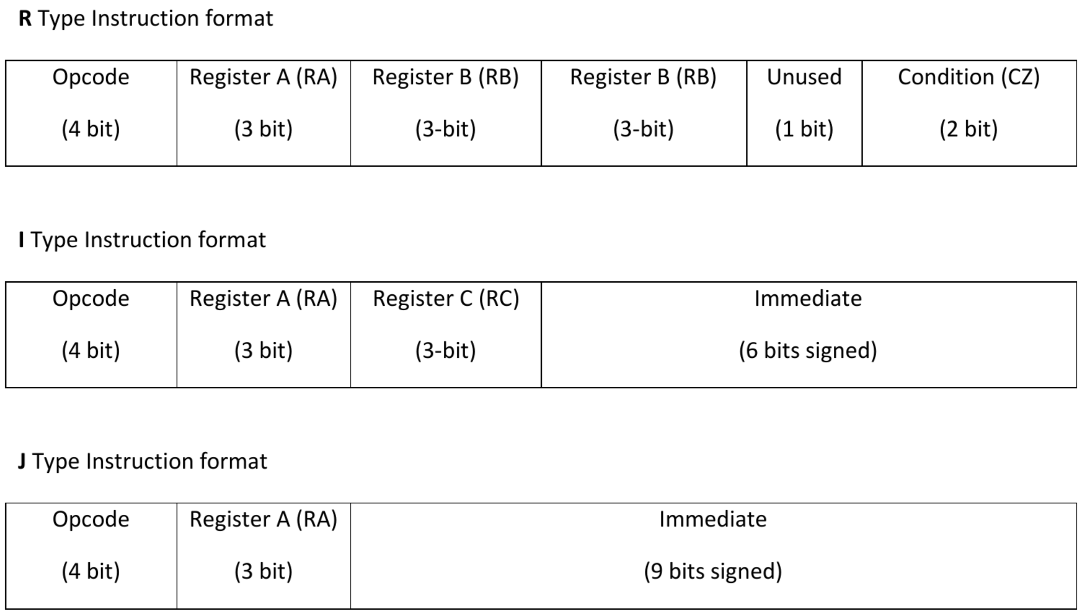
1 Abstract

Project submission for the course CS 230: Digital Logic Design and Computer Architecture by the collaborative efforts of Sarthak Mittal, Pinkesh Raghuvanshi, Moiz Shakruwala and Koustubh Rao of CSE 2024 under the guidance of Professor Virendra Singh.

2 Intoduction

IITB-RISC is a 16-bit very simple computer developed for the teaching that is based on the Little Computer Architecture. The IITB-RISC is an 8-register, 16-bit computer system. It has 8 general-purpose registers (R0 to R7). Register R7 is always stores Program Counter. All addresses are short word addresses (i.e., address 0 corresponds to the first two bytes of main memory, address 1 corresponds to the second two bytes of main memory, etc.). This architecture uses condition code register which has two flags Carry flag (C) and Zero flag (Z). The IITB-RISC is very simple, but it is general enough to solve complex problems. The architecture allows predicated instruction execution and multiple load and store execution. There are three machine-code instruction formats (R, I, and J type) and a total of 17 instructions.

2.1 Instruction Format



R Type Instruction format

Opcode (4 bit)	Register A (RA) (3 bit)	Register B (RB) (3-bit)	Register B (RB) (3-bit)	Unused (1 bit)	Condition (CZ) (2 bit)

I Type Instruction format

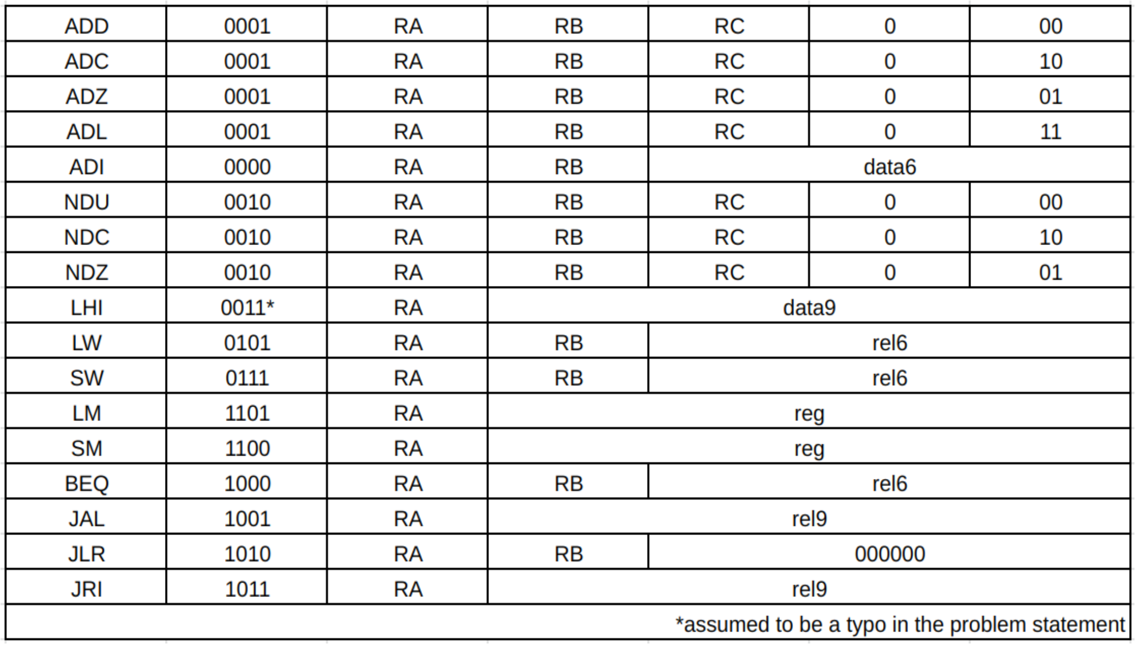
Opcode (4 bit)	Register A (RA) (3 bit)	Register C (RC) (3-bit)	Immediate (6 bits signed)

J Type Instruction format

Opcode (4 bit)	Register A (RA) (3 bit)	Immediate (9 bits signed)

Figure 1: Instruction Types

2.2 Instruction Set Encoding

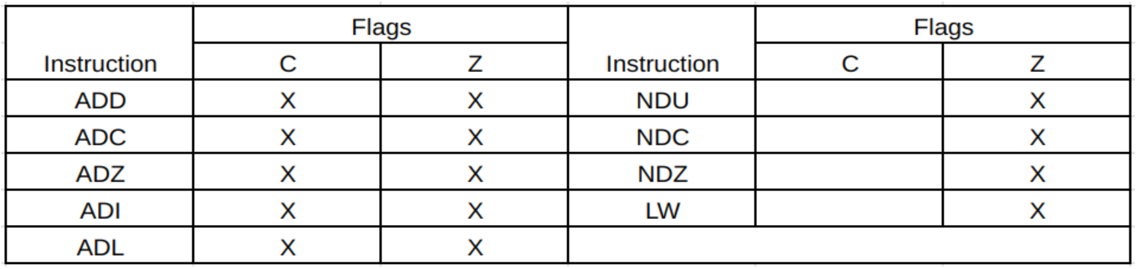


ADD	0001	RA	RB	RC	0	00
ADC	0001	RA	RB	RC	0	10
ADZ	0001	RA	RB	RC	0	01
ADL	0001	RA	RB	RC	0	11
ADI	0000	RA	RB	data6		
NDU	0010	RA	RB	RC	0	00
NDC	0010	RA	RB	RC	0	10
NDZ	0010	RA	RB	RC	0	01
LHI	0011*	RA	data9			
LW	0101	RA	RB	rel6		
SW	0111	RA	RB	rel6		
LM	1101	RA	reg			
SM	1100	RA	reg			
BEQ	1000	RA	RB	rel6		
JAL	1001	RA	rel9			
JLR	1010	RA	RB	000000		
JRI	1011	RA	rel9			

*assumed to be a typo in the problem statement

Figure 2: Instruction Set Encoding

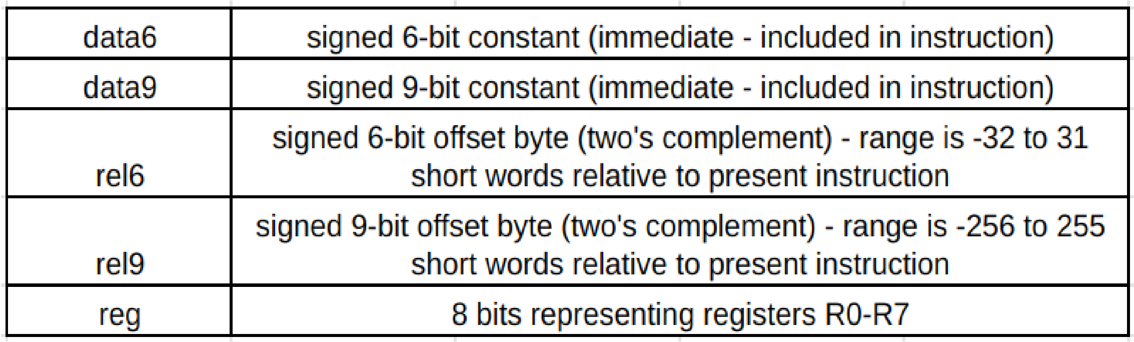
2.3 Flag Settings



Instruction	Flags		Instruction	Flags	
	C	Z		C	Z
ADD	X	X	NDU		X
ADC	X	X	NDC		X
ADZ	X	X	NDZ		X
ADI	X	X	LW		X
ADL	X	X			

Figure 3: Flag Settings

2.4 Addressing Modes



data6	signed 6-bit constant (immediate - included in instruction)
data9	signed 9-bit constant (immediate - included in instruction)
rel6	signed 6-bit offset byte (two's complement) - range is -32 to 31 short words relative to present instruction
rel9	signed 9-bit offset byte (two's complement) - range is -256 to 255 short words relative to present instruction
reg	8 bits representing registers R0-R7

Figure 4: Addressing Modes

3 Implementation Idea

3.1 Microprocessor Blocks

1. **Data Path:** Data path corresponding to RTL layout of microprocessor. The comments give details of transfer and predicate signal mapping.

2. **Control Path:** This has the Moore FSM. The first process controls the states (flip-flops), the second one has next state logic and third process has output logic (based on present state). The transfer signals have been accompanied by expected results in the data path.

3. **IITB_RISC:** This is the top-level entity that combines control path and data path. The output register is used to display processes of microprocessor outside hardware.

4. **test_final:** This is a test bench which can be used to simulate the entire microprocessor in Altera's modelsim. It basically functions to provide the clock and reset signals to the microprocessor.

5. **assembler:** This is essential for converting English like assembly to hex. It basically works by taking as its input a text file containing code written in an assembly like language, and creates an Intel hex file which can be directly provided to Quartus which uses it to preload memory. The microprocessor treats this hex file as the primary input.

6. **serial_boot:** This can be used in conjunction with the bootloader hardware to program the microprocessor memory to contain any set of instructions.

3.2 Derived Instructions

1. ADL(RC,RA,RB) = ADD(RC,RA,RB) + ADD(RC,RC,RB)

2. JRI(RA,Imm) = LLI(RB,Imm) + ADD(RB,RA,RB) + JLR(RA,RB)

We treat ADL and JRI as derived instructions.

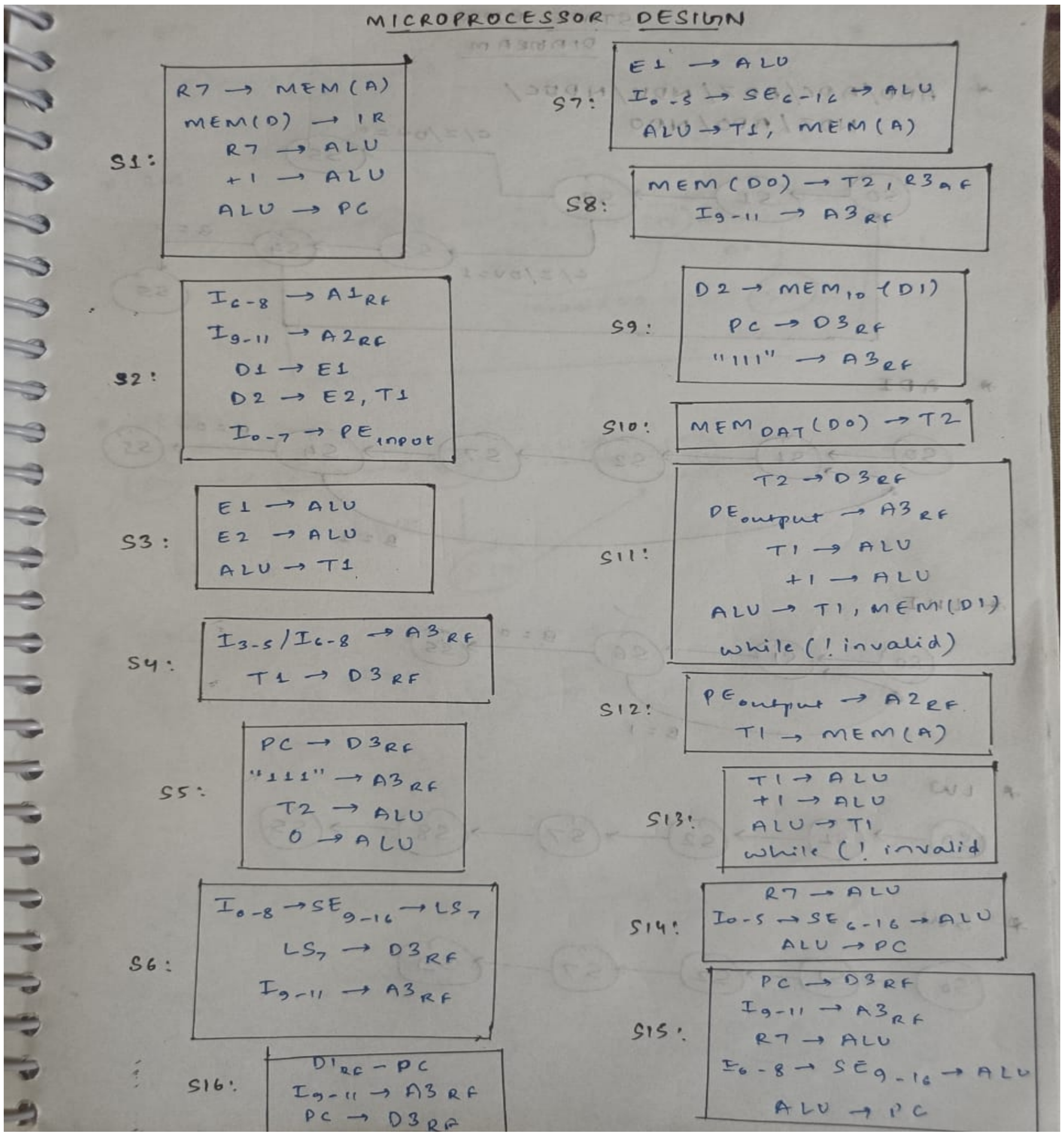
ADL: The ADL, i.e. ADL(RC,RA,RB) instruction adds content of regB (after one bit left shift) to regA and stores the result in regC. But observe that a left shift is equivalent to multiplying by two (since we are in binary) hence adding the contents of regB to regA twice and then store it in regC.

Hence in assembler.py we replace instructions of the form ADL(RC,RA,RB) by additional instructions, ADD(RC,RA,RB) and ADD(RC,RC,RB).

JRI: Similarly we treat JRI as a derived instruction. We first load the immediate into the lower 9 significant bits of a dummy register. Then we add the values in regA and the dummy register, and store into dummy register. Then we branch off to value stored in dummy register, and store updated PC in regA.
Therefore the assembler.py replaces instructions of the form JRI(RA,Imm) by additional instructions, LLI(RB,Imm), ADD(RB,RA,RB) and JLR(RA,RB).

4 Microprocessor Design

4.1 State Elaboration



MICROPROCESSOR DESIGN

S1:
$R7 \rightarrow MEM(A)$
$MEM(D) \rightarrow IR$
$R7 \rightarrow ALU$
$+1 \rightarrow ALU$
$ALU \rightarrow PC$

S2:
$I_{6-8} \rightarrow A1_{RF}$
$I_{9-11} \rightarrow A2_{RF}$
$D1 \rightarrow E1$
$D2 \rightarrow E2, T1$
$I_{0-7} \rightarrow PE_{input}$

S3:
$E1 \rightarrow ALU$
$E2 \rightarrow ALU$
$ALU \rightarrow T1$

S4:
$I_{3-5} / I_{6-8} \rightarrow A3_{RF}$
$T1 \rightarrow D3_{RF}$

S5:
$PC \rightarrow D3_{RF}$
"111" $\rightarrow A3_{RF}$
$T2 \rightarrow ALU$
$0 \rightarrow ALU$

S6:
$I_{0-8} \rightarrow SE_{9-16} \rightarrow LS_7$
$LS_7 \rightarrow D3_{RF}$
$I_{9-11} \rightarrow A3_{RF}$

S16:
$D1_{RF} - PC$
$I_{9-11} \rightarrow A3_{RF}$
$PC \rightarrow D3_{RF}$

S7:
$E1 \rightarrow ALU$
$I_{0-3} \rightarrow SE_{6-16} \rightarrow ALU$
$ALU \rightarrow T1, MEM(A)$

S8:
$MEM(D0) \rightarrow T2, R3_{RF}$
$I_{9-11} \rightarrow A3_{RF}$

S9:
$D2 \rightarrow MEM_{10}(D1)$
$PC \rightarrow D3_{RF}$
"111" $\rightarrow A3_{RF}$

S10:
$MEM_{DAT}(D0) \rightarrow T2$

S11:
$T2 \rightarrow D3_{RF}$
$PE_{output} \rightarrow A3_{RF}$
$T1 \rightarrow ALU$
$+1 \rightarrow ALU$
$ALU \rightarrow T1, MEM(D1)$
while (! invalid)

S12:
$PE_{output} \rightarrow A2_{RF}$
$T1 \rightarrow MEM(A)$

S13:
$T1 \rightarrow ALU$
$+1 \rightarrow ALU$
$ALU \rightarrow T1$
while (! invalid

S14:
$R7 \rightarrow ALU$
$I_{0-5} \rightarrow SE_{6-16} \rightarrow ALU$
$ALU \rightarrow PC$

S15:
$PC \rightarrow D3_{RF}$
$I_{9-11} \rightarrow A3_{RF}$
$R7 \rightarrow ALU$
$I_{6-8} \rightarrow SE_{9-16} \rightarrow ALU$
$ALU \rightarrow PC$

Figure 5: States

4.2 Flow Diagrams

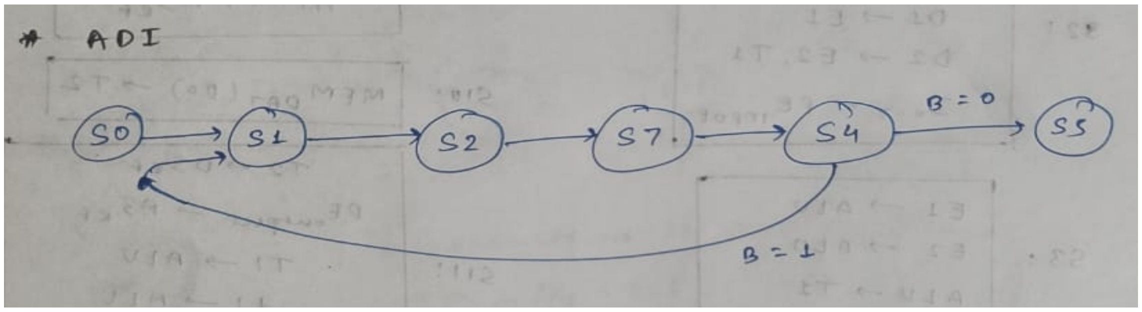


Figure 6: State Diagram for ADI instruction

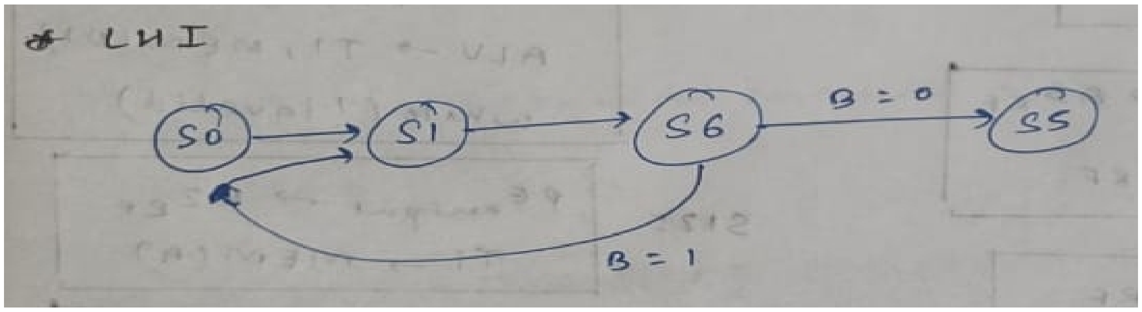


Figure 7: State Diagram for LHI instruction

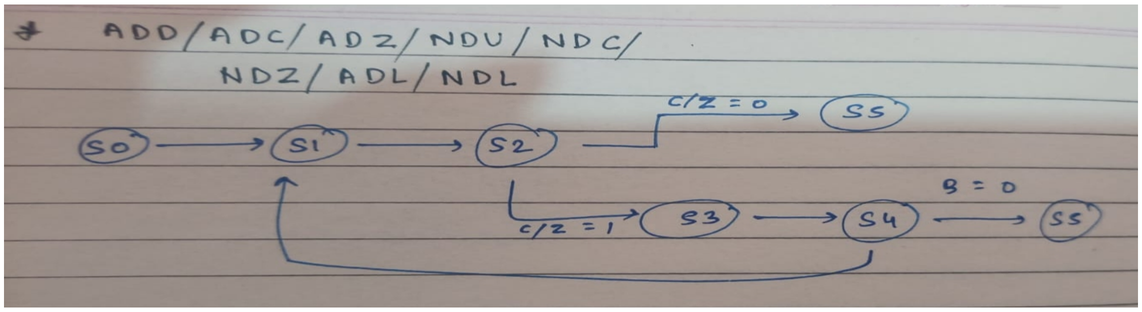


Figure 8: State Diagram for instructions

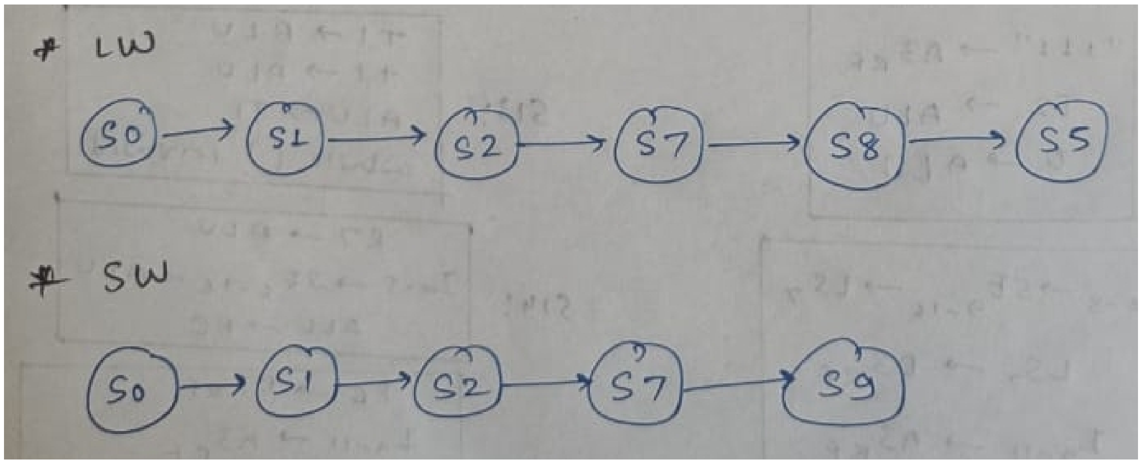


Figure 9: State Diagram for instructions

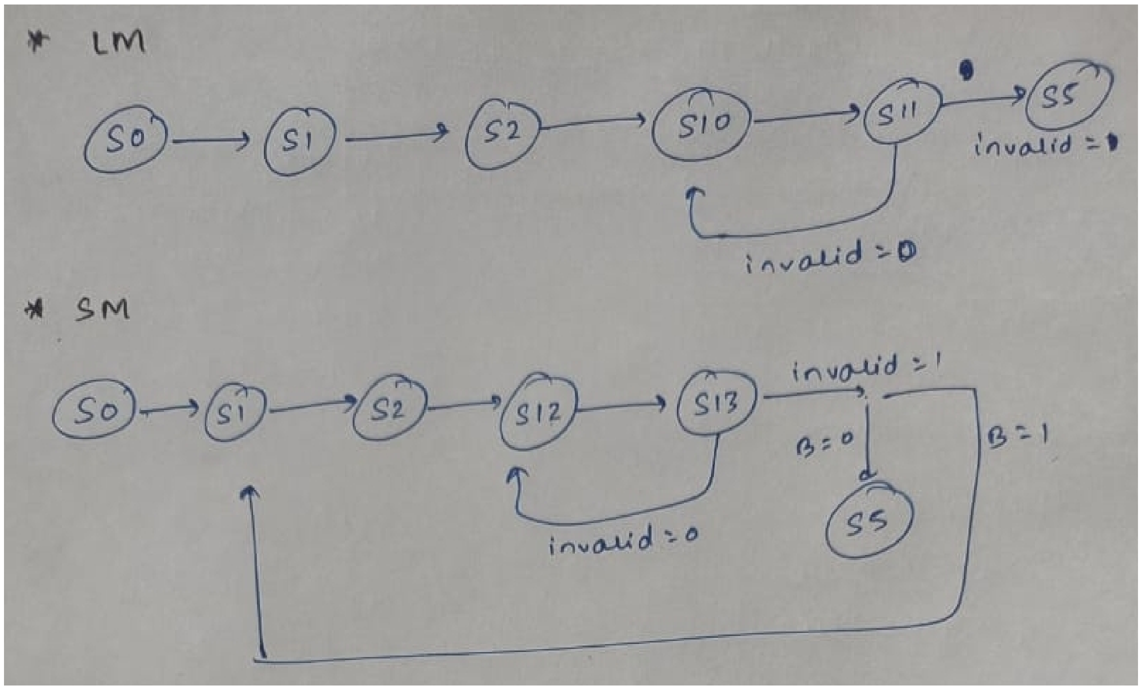


Figure 10: State Diagram for instructions

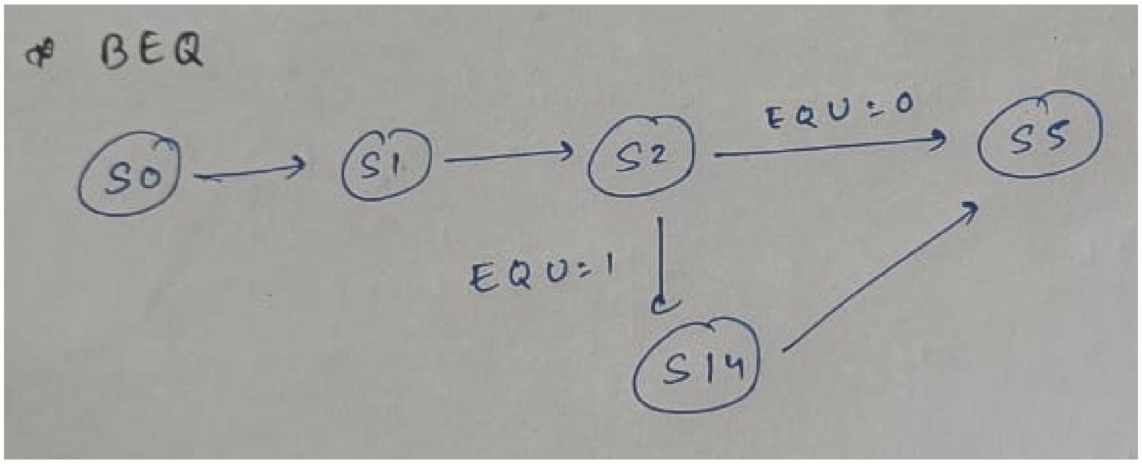


Figure 11: State Diagram for instructions

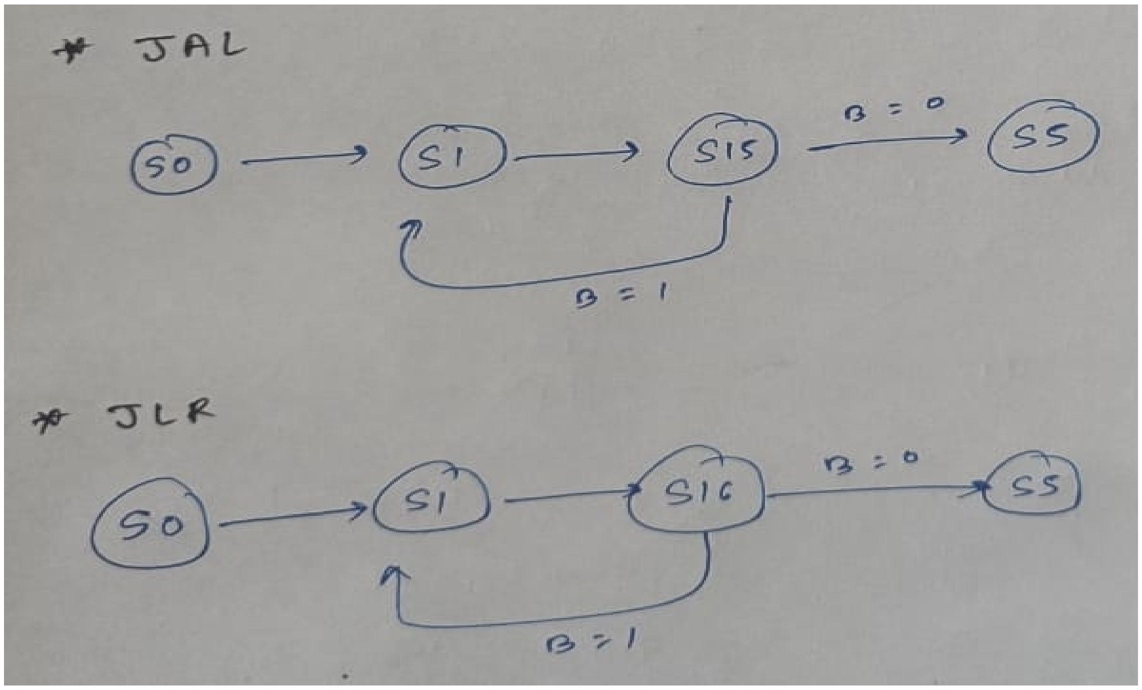


Figure 12: State Diagram for instructions