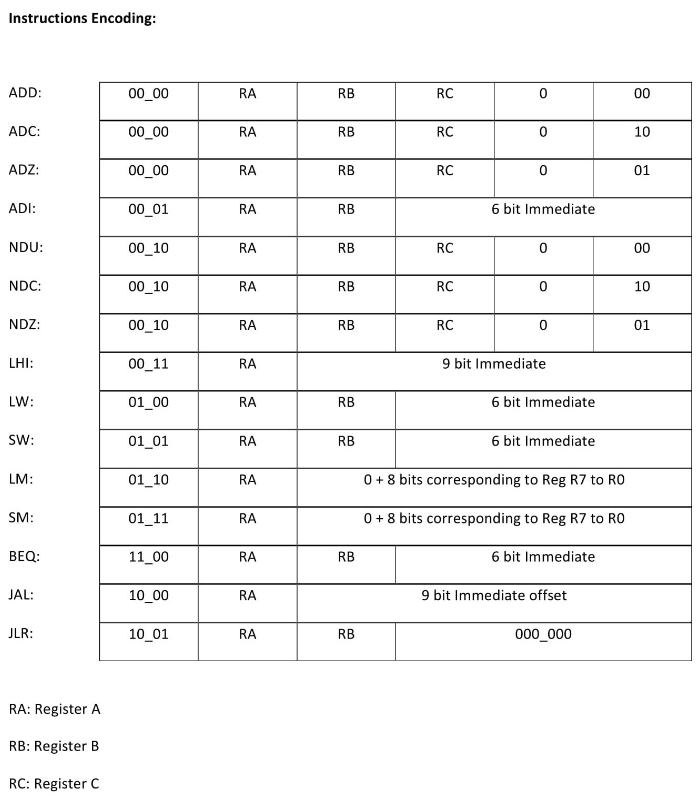
EE309 Project
RISC design Microprocessor

Shubham Birange, 160070014
Pravesh Trivedi, 160070041
Parshva Shah, 160070009
Bhuyashi Deka, 16D070015

November 28, 2018

1 Overview

The aim of this project was to implement a fully functional RISC based computer design, IITB RISC whose ISA is provided below. IITB RISC is a 8 - register, 16-bit architecture using multi-cycle implementation.



Instructions Encoding:

ADD:	00_00	RA	RB	RC	0	00
ADC:	00_00	RA	RB	RC	0	10
ADZ:	00_00	RA	RB	RC	0	01
ADI:	00_01	RA	RB	6 bit Immediate		
NDU:	00_10	RA	RB	RC	0	00
NDC:	00_10	RA	RB	RC	0	10
NDZ:	00_10	RA	RB	RC	0	01
LHI:	00_11	RA	9 bit Immediate			
LW:	01_00	RA	RB	6 bit Immediate		
SW:	01_01	RA	RB	6 bit Immediate		
LM:	01_10	RA	0 + 8 bits corresponding to Reg R7 to R0			
SM:	01_11	RA	0 + 8 bits corresponding to Reg R7 to R0			
BEQ:	11_00	RA	RB	6 bit Immediate		
JAL:	10_00	RA	9 bit Immediate offset			
JLR:	10_01	RA	RB	000_000		

RA: Register A

RB: Register B

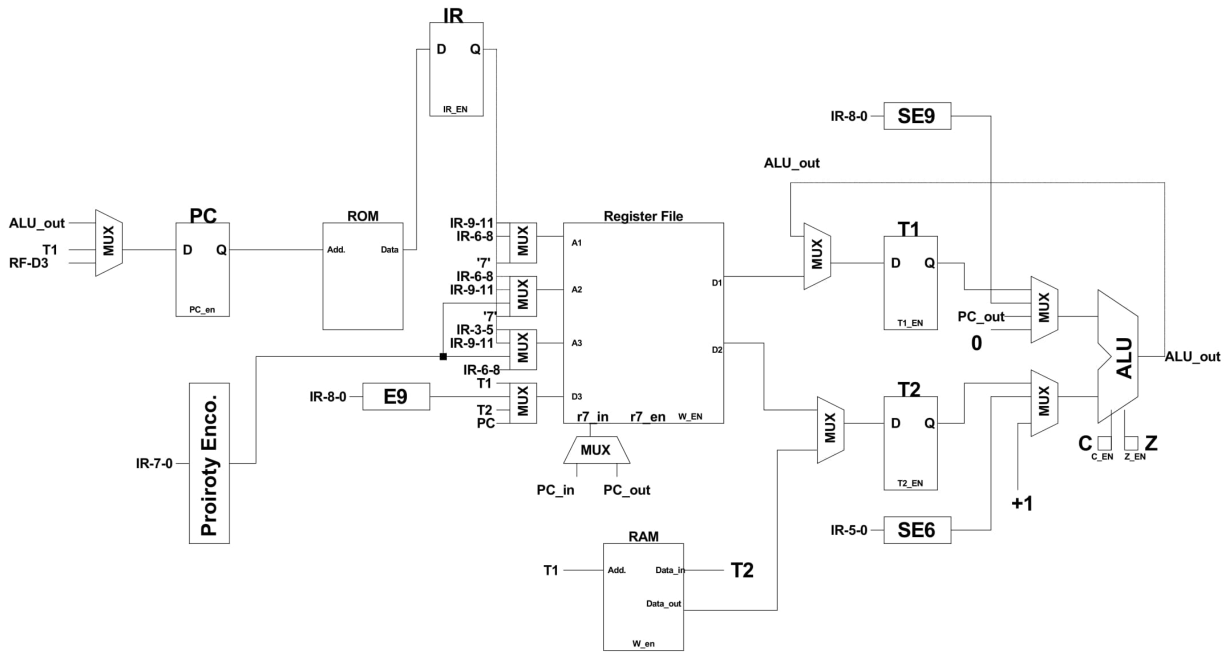
RC: Register C

2 Introduction

In a multi-cycle implementation the instruction fetched is completely executed before the next instruction is fetched. The design implementation mainly consists of two parts a data-path and a controller. For this purpose a Level - 1 Hardware flow chart was designed which was later optimized in the Level - 2 Hardware flow chart.

The data-path consists of the PC and IR registers, a ROM, a Register file with 8 registers, two temporary registers T1 and T2 and an ALU with adder, subtractor and nand operation.

While the controller is a Mealy FSM optimized for 16 states.



--- C:\Users\Vaibhav Birange\Desktop\Quartus\RISC\Spice\Draft6.asc ---

Figure 1: Data-path for IITB RISC

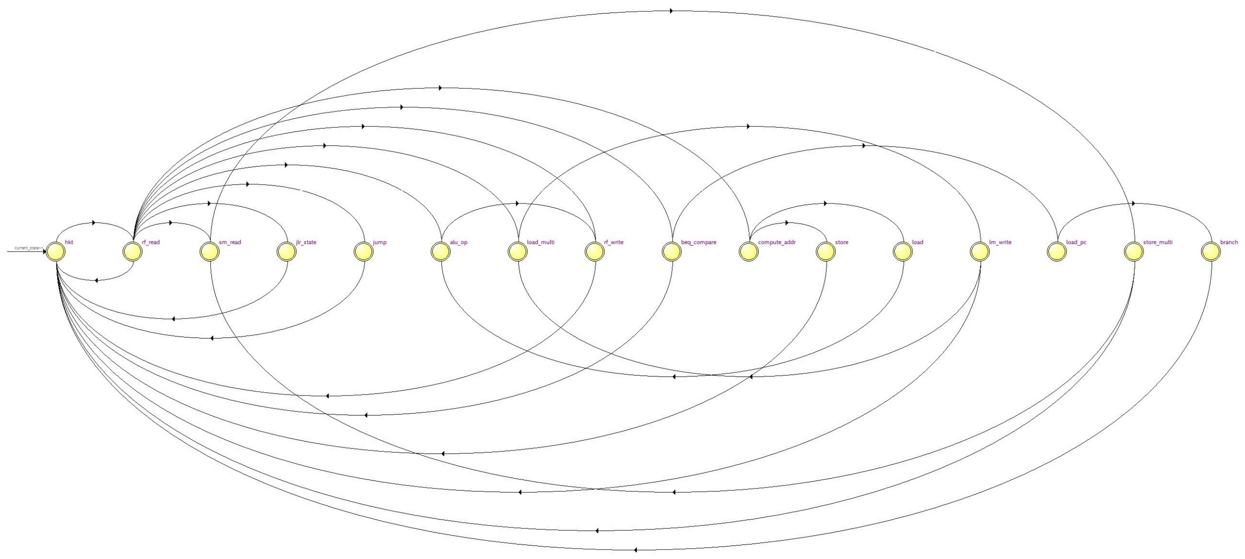


Figure 2: State Transition graph of the controller

3 Instructions

The following are the hardware flowcharts for the ISA:

3.1 Arithmetic and logical Instructions

ADD/ ADC/ ADZ/ NDU/ NDC/ NDZ

PC → Mem-a, ALU_a Mem-d → IR +1 → ALU_b ALU_out → PC	CURRENT STATE HKT	NEXT STATE rf_read

IR 9-11 → rf_a1
IR 6-8 → rf_a2
rf_d1 → T1
rf_d2 → T2
 	
if **(IR is ADC/NDC and !C)** or **(IR is ADZ/NDZ and !Z)** then
PC → **R7**
	rf_read	if IR is ADD/NDU then alu_op
		elsif IR is ADC/NDC then
		if C then alu_op else HKT
		elsif IR is ADZ/NDZ then
		if Z then alu_op else HKT
		else HKT

T1 → ALU_a T2 → ALU_b ALU_out → T1	alu_op	rf_write
IR 3-5 → rf_a3 T1 → rf_d3 **if rf_a3 == '7' then** **rf_d3 → PC** **else** **PC → r7_in**	rf_write	HKT

ADI

PC → Mem-a, ALU_a Mem-d → IR +1 → ALU_b ALU_out → PC	CURRENT STATE HKT	NEXT STATE rf_read

IR 9-11 → rf_a1
IR 6-8 → rf_a2
rf_d1 → T1
rf_d2 → T2
	rf_read	if IR is ADD/NDU/ADI then
		alu_op
		elsif IR is ADC/NDC then
		if C then alu_op else HKT
		elsif IR is ADZ/NDZ then
		if Z then alu_op else HKT
		else HKT

T1 → ALU_a **IR 0-5 → SE6** → ALU_b ALU_out → T1	alu_op	rf_write
IR 3-5 → rf_a3 T1 → rf_d3 **if rf_a3 == '7' then** **rf_d3 → PC** **else** **PC → r7_in**	rf_write	HKT

3.2 LOAD LW

LW

PC → Mem-a, ALU_a Mem-d → IR +1 → ALU_b ALU_out → PC	CURRENT STATE HKT
	NEXT STATE rf_read

IR 8-6 → rf_a1 XX → rf_a2 rf_d1 → T1 rf_d2 → T2	rf_read
	if IR is **ADD/NDU/ADI** then **alu_op** elsif IR is **ADC/NDC** then if **C** then **alu_op** else **HKT** elsif IR is **ADZ/NDZ** then if **Z** then **alu_op** else **HKT** elsif IR is **LHI** then **rf_write** elsif IR is **LW** then **compute_add**. else **HKT**

T1 → ALU_a **IR 0-5 → SE6** → ALU_b ALU_out → T1	compute_add.
	LOAD

T1 → Mem-a Mem-d → **T2**	LOAD
	alu_op

'0' → ALU_a **T2** → ALU_b ALU_out → T1	alu_op.
	rf_write

IR 9-11 → rf_a3 T1 → rf_d3 **if rf_a3 == '7' then** **rf_d3 → PC** **else** **PC → r7_in**	rf_write
	HKT

3.3 STORE SW and LOAD LHI

LHI

PC → Mem-a, ALU_a Mem-d → IR +1 → ALU_b ALU_out → PC	CURRENT STATE HKT
	NEXT STATE rf_read
IR 9-11 → rf_a1 IR 6-8 → rf_a2 rf_d1 → T1 rf_d2 → T2	rf_read

	if IR is **ADD/NDU/ADI** then
	alu_op
	elsif IR is **ADC/NDC** then
	if **C** then **alu_op** else **HKT**
	elsif IR is **ADZ/NDZ** then
	if **Z** then **alu_op** else **HKT**
	elsif IR is **LHI** then **rf_write**
	else **HKT**

IR 11-9 → rf_a3 IR 8-0 → rf_d3 **if rf_a3 == '7' then** **rf_d3 → PC** **else** **PC → r7_in**	rf_write
	HKT

SW

PC → Mem-a, ALU_a Mem-d → IR +1 → ALU_b ALU_out → PC	CURRENT STATE HKT
	NEXT STATE rf_read

IR 8-6 → rf_a1 **IR 9-11→** rf_a2 rf_d1 → T1 rf_d2 → T2	rf_read
	if IR is **ADD/NDU/ADI** then **alu_op** elsif IR is **ADC/NDC** then if **C** then **alu_op** else **HKT** elsif IR is **ADZ/NDZ** then if **Z** then **alu_op** else **HKT** elsif IR is **LHI** then **rf_write** elsif IR is **LW/SW** then **compute_add**. else **HKT**

T1 → ALU_a **IR 0-5 → SE6** → ALU_b ALU_out → T1	compute_add.
	if IR is **LW** then **LOAD** elsif IR is **SW** then **STORE**

T1 → Mem-a T2 → Mem-d **PC → r7_in**	STORE
	HKT

3.4 LOAD Multiple LM

LM

PC → Mem-a, ALU_a Mem-d → IR +1 → ALU_b ALU_out → PC	CURRENT STATE HKT
	NEXT STATE rf_read
IR 11-9 → rf_a1 XX → rf_a2 rf_d1 → T1 if IR is **LM** then IR 0-7 → **enco_register**	rf_read
	if IR is **ADD/NDU/ADI** then **alu_op** elsif IR is **ADC/NDC** then if **C** then **alu_op** else **HKT** elsif IR is **ADZ/NDZ** then if **Z** then **alu_op** else **HKT** elsif IR is **LHI** then **rf_write** elsif IR is **LW/SW** then **compute_add**. elsif IR is **LM** then LOAD_mutli else **HKT**
T1 → Mem-a Mem-d → T2 T1 → ALU_a +1 → ALU_b ALU_out → T1	LOAD_mutli
	LM_write

enco_address → rf_a3 T2 → rf_d3 **if rf_a3 == ‘7’ then** **rf_d3 → PC** **update enco_register** by enoc_sel = ‘0’ x <= Enco_register AND Decoder_o/p **if X == ‘0’ and IR(7) == ‘0’ then** **PC → r7_in**	LM_write

	if **x is ‘1’** then **LOAD_multi**
	else HKT

3.5 STORE Multiple SM

SM

PC → Mem-a, ALU_a Mem-d → IR +1 → ALU_b ALU_out → PC	CURRENT STATE HKT
	NEXT STATE rf_read
IR 11-9 → rf_a1 XX → rf_a2 rf_d1 → T1 if IR is **LM/SM** then IR 0-7 → **enco_register**	rf_read
	if IR is **ADD/NDU/ADI** then **alu_op** elsif IR is **ADC/NDC** then if **C** then **alu_op** else **HKT** elsif IR is **ADZ/NDZ** then if **Z** then **alu_op** else **HKT** elsif IR is **LHI** then **rf_write** elsif IR is **LW/SW** then **compute_add**. elsif IR is **LM** then **Load_multi** **elsif** IR is **SM** then **SM_read** else **HKT**
Encoder Address → rf_a2 rf_d2→ T2	**SM_read**
	STORE_multi

T1 → Mem-a T2 → Mem-d T1 → ALU_a +1 → ALU_b ALU_out → T1 **update enco_register** by enoc_sel = ‘0’ x <= Enco_register AND Decoder_o/p **if X == ‘0’ then** **PC → r7_in**	**STORE_multi**

	if **x is ‘1’** then **SM_read**
	else HKT

3.6 Branch JAL and JLR

JAL

PC → Mem-a, ALU_a	CURRENT STATE HKT
Mem-d → IR	
+1 → ALU_b	NEXT STATE rf_read
ALU_out → PC	

	rf_read
'XX' → rf_a1	if IR is **ADD/NDU/ADI** then **alu_op**
'7' → rf_a2	elsif IR is **ADC/NDC** then if **C** then **alu_op** else **HKT**
rf_d2 → T2 (PC -> T2)	elsif IR is **ADZ/NDZ** then if **Z** then **alu_op** else **HKT**
if IR is **LM/SM** then	elsif IR is **LHI** then **rf_write**
IR 0-7 → **enco_register**	elsif IR is **LW/SW** then **compute_add**.
	elsif IR is **LM/SM** then **compare**
	elsif IR is **BEQ** then **BEQ_compare**
	elsif IR is **JAL** then **Jump**
	else **HKT**

IR 8-0 → **SE9** → ALU_a	JUMP
T2 → ALU_b	
ALU_out → PC	
IR 9-11 → rf_a3	HKT
T2 → rd_d3	
if rf_a3 == '7' then rf_d3 → PC_in	
else PC_in → r7_in	

JLR

PC → Mem-a, ALU_a	CURRENT STATE HKT
Mem-d → IR	
+1 → ALU_b	NEXT STATE rf_read
ALU_out → PC	

	rf_read
IR 8-6 → rf_a1	if IR is **ADD/NDU/ADI** then **alu_op**
'7' → rf_a2	elsif IR is **ADC/NDC** then if **C** then **alu_op** else **HKT**
rf_d1 → T1	elsif IR is **ADZ/NDZ** then if **Z** then **alu_op** else **HKT**
rf_d2 → T2 (PC -> T2)	elsif IR is **LHI** then **rf_write**
if IR is **LM/SM** then	elsif IR is **LW/SW** then **compute_add**.
IR 0-7 → **enco_register**	elsif IR is **LM/SM** then **compare**
	elsif IR is **BEQ** then **BEQ_compare**
	elsif IR is **JAL** then **Jump**
	elsif IR is **JLR** then **JLR**
	else **HKT**

T1 → PC	JLR
IR 9-11 → rf_a3	
T2 → rd_d3	HKT
if rf_a3 == '7' then rf_d3 → PC_in	
else PC_in → r7_in	

3.7 Branch BEQ

BEQ

PC → Mem-a, ALU_a	CURRENT STATE HKT
Mem-d → IR	
+1 → ALU_b	NEXT STATE rf_read
ALU_out → PC	

	rf_read
IR 11-9 → rf_a1	if IR is **ADD/NDU/ADI** then **alu_op**
IR 8-6 → rf_a2	elsif IR is **ADC/NDC** then if **C** then **alu_op** else **HKT**
rf_d1 → T1	elsif IR is **ADZ/NDZ** then if **Z** then **alu_op** else **HKT**
rf_d2 → T2	elsif IR is **LHI** then **rf_write**
if IR is **LM/SM** then	elsif IR is **LW/SW** then **compute_add**.
IR 0-7 → **enco_register**	elsif IR is **LM/SM** then **compare**
	elsif IR is **BEQ** then BEQ_compare
	else **HKT**

T1 → ALU_a	BEQ_compare
T2 → ALU_b	if **Z** then **Load_PC**
ALU_out → T1 (do not modify Z flag)	else **HKT**
if Z = '0' then **PC_out → r7_in**	

'7' → rf_a1	Load_PC
rf_d1 → T1	**Branch**

T1 → ALU_a	**Branch**
IR 0-5 → SE6 → ALU_b	
ALU_out → PC	HKT
PC_in → r7_in	

4 Testing

The testing and verification of the design was done using the following code:

```
    0 => "0001100100000111",--ADI R4 R4 111
1 => "0001101101000010",--ADI R5 R5 10
2 => "0111110000100000",--SM R6 00100000
3 => "0101100110000001",--SW R4 R6 01
4 => "0011100000000000",--LHI R4 0000
5 => "0011101000000000",--LHI R5 0000
6 => "0001011011000111",--ADI R3 R3 7
7 => "0110101000000110",--LM R5 00000110
8 => "1100001101000011",--BEQ R1 R5 3
9 => "0001001001111111",--ADI R1 R1 -1
10 => "1000110111111110",--JAL R6 -2
11 => "1100010101000100",--BEQ R2 R5 4
12=> "0001010010111111",--ADI R2 R2 -1
13=> "0100001101000001",--LW R1 R5 1
14=> "1000110111111010",--JAL R6 -6
15=> "0001000000000001",--ADI R0 R0 1
16=> "0011110000000000",--LHI R6 0
17=> "0001110110000011",--ADI R6 R6 3
18=> "1100000110000010",--BEQ R6 R0 +2
19=> "0000101011111000",--ADD R5 R3 R7
20=> "0011000000000000",--LHI R0 0
21=> "0000101011111000",--ADD R5 R3 R7
```

This is equivalent to:

```
    R0 = 0
    for(R1 = 7, R1 < 0, R1--)
        for(R2 j = 2, R2 < 0, R2--)
            if(R0 = 3)
                R0 = 0
            else
                R0++
```

4.1 Simulation Results

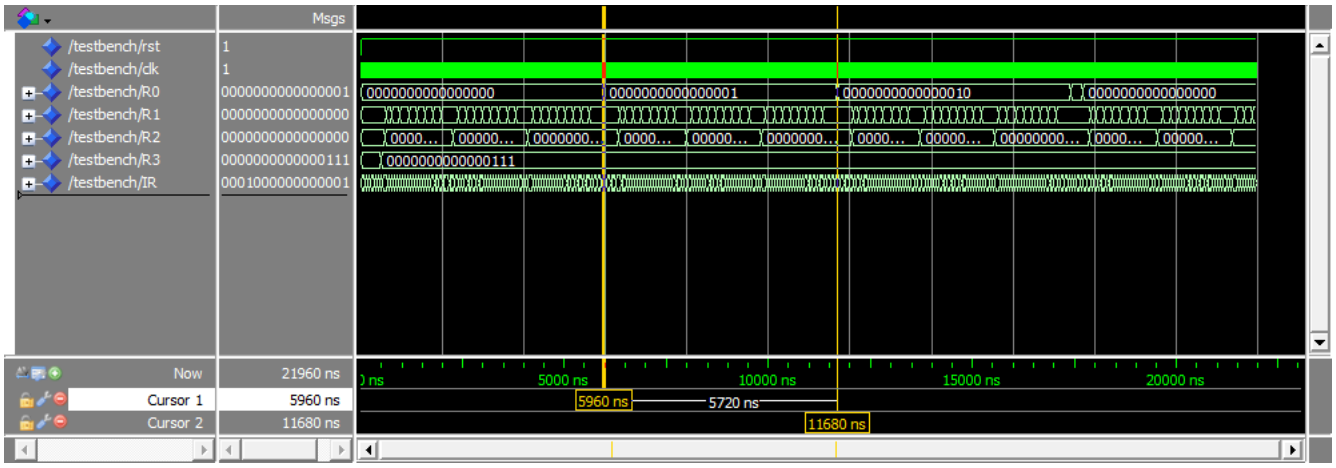


Figure 3: RTL simulation for the above code. Note the time for R0 to change is 5.72us

5 Code VHDL Files

1. **alu.vhd -** contains the code for the alu used in the design. It can perform addition, subtraction and nand operation on 16-bit numbers with the respective carry and zero flags.
2. **busmultiplexer.vhd -** code for a generic input 2^n multiplexer.
3. **controller.vhd -** contains the code for the FSM which decodes the instruction and gives the control signals as output.
4. **encoder.vhd -** contains the priority encoder and decoder used for decoding the address in LM and SM instructions.
5. **processor.vhd -** this is the top-level entity that combines all the components of the datapath and controller.
6. **RAM.vhd -** useed as the RAM in the processor containing 16-bit data-in, data-out and an address bus.
7. **reg-file.vhd -** contains the code for the 8 registers and the multiplexers for selecting them.
8. **ROM.vhd -** memory containing the instructions.
9. **register.vhd -** generic n-bit register