PROJECT DESIGN REPORT IITB-RISC

GROUP -9
Harshvardhan Tibrewal 140020023
Agrim Gupta 140020003
Ajinkya Gorad 140110033
Amritesh Sharma 14D260012

INDEX

I. DATAPATH DESCRIPTION AND DESIGN

1. Register File : The register file used by us has 5 inputs (A1,A2,A3,D3,D4) and 3 outputs (D1,D2,D5). It is controlled by two control signals : PC_WR and RF_WR.
If PC_WR is stated high, data from D4 is given to R7 (i.e. PC). If PC_WR is low, data from R7 is read and given to D5.
If RF_WR is stated high, data from D3 is given to register with address A3. If stated low, data from registers haviNg addresses A1 and A2 are given to D1 and D2 respectively.

2. Priority Encoder cum Bit clearer : The priority encoder created by us combines functionality of giving the position of least significant 1 present as it's output 1, and modifying the input 8 bit string, by making the least significant 1 as 0.
eg. if 010010**1**0 is given to priority encoder, it's output will be **010** and 010010**0**0
In adidition, it also gives a "Last" bit, to signify that all 1's have been cleared, so that we can exit from the LM/SM instruction if Last=1
eg. if 0**1**000000 is given to it, outputs will be 110,**0000000**, and **Last=1.**
-> Used in LM and SM instructions

3. SEN_6 and SEN_9 : Converts the 6 bit and 9 bit numbers to proper 16 bit numbers for usage in 16 bit ALU.
-> Used in ADI, BEQ, LW, SW and JAL

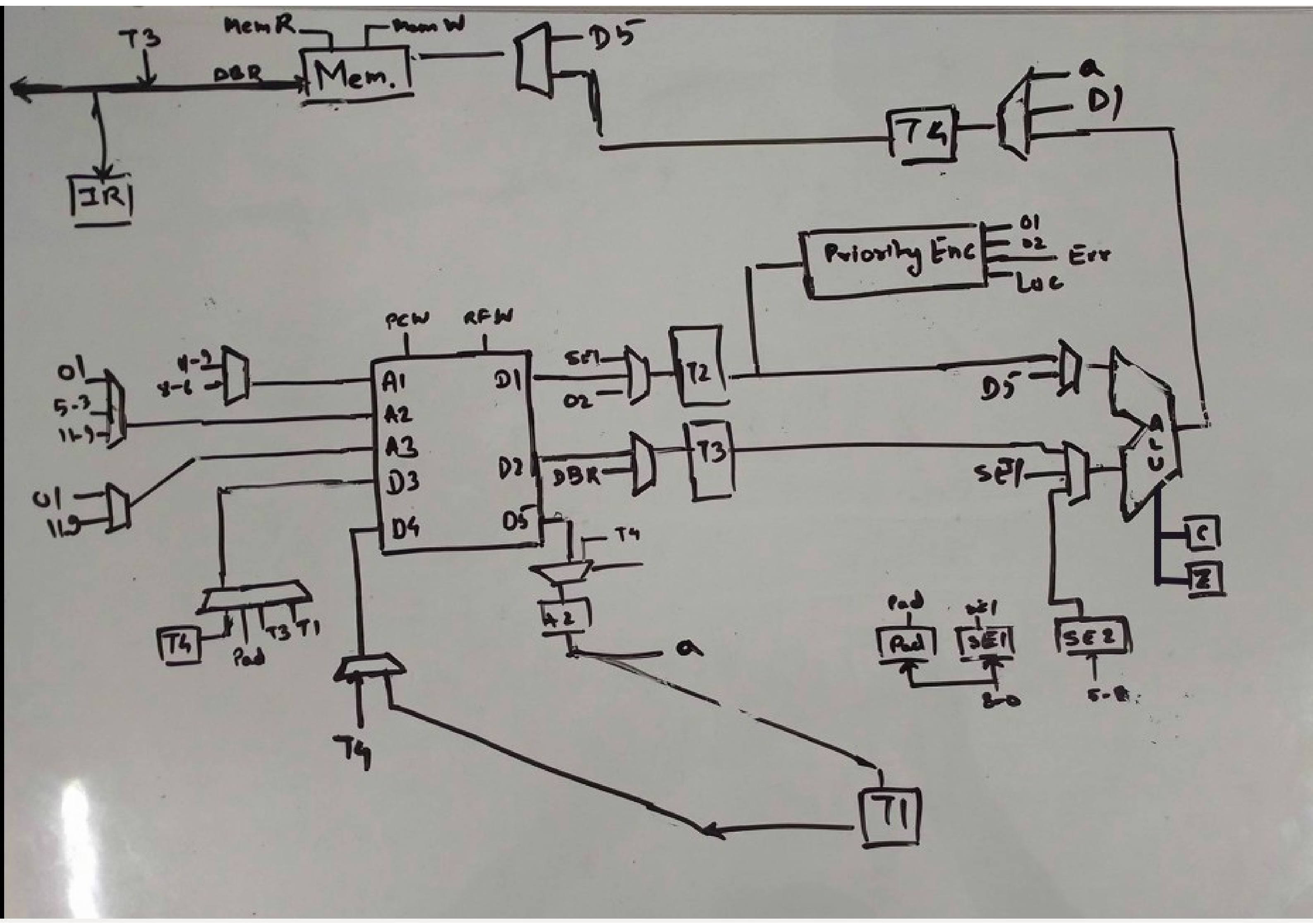
4. POST_PAD9: Pads 7 zeroes at end of 9 bit binary string, to make it a 16 bit string, with Most significant 9 bits as the 9 bits at input, and Least significant 7 bits as zeroes.
Eg. 111000111 is converted to 1110001110000000
-> Used in LHI

5. Memory: We have 16 bit addresses in memory, and each address stores 16 bit data it. Due to limitations of quartus, memory is limited to 1MB (by a proper masking string for 16 byte generic string making sure segmentation fault doesn't occur), with 512 indices for 16 bit data. The memory can be hardcoded with instructions and can also be initialised directly from testbench when mem_bit is set to 1 in the top level entity. (Ref IITB_RISC.vhd). That is, there are 2 pathways for memory initialisation, helping in debugging.

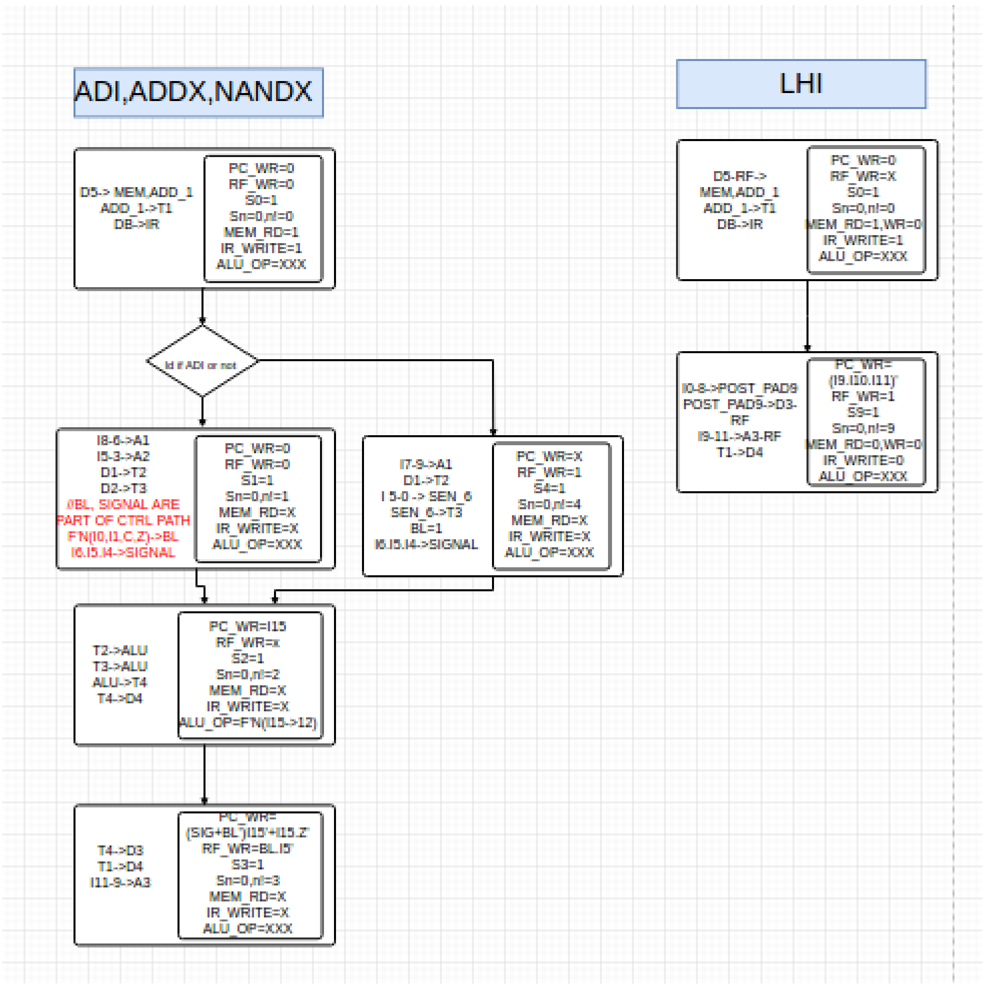
6. 16 bit ALU : In ALU, we have supported 3 operations, ADD, NAND, XOR. It has 3 inputs (first operand, second operand and op_code), and 3 outputs (C,Z and Result). It sets C and Z at every instruction, and these values are taken by the C and Z 1 bit registers at appropriate states by stating the C_Enable and Z_Enable appropriately.

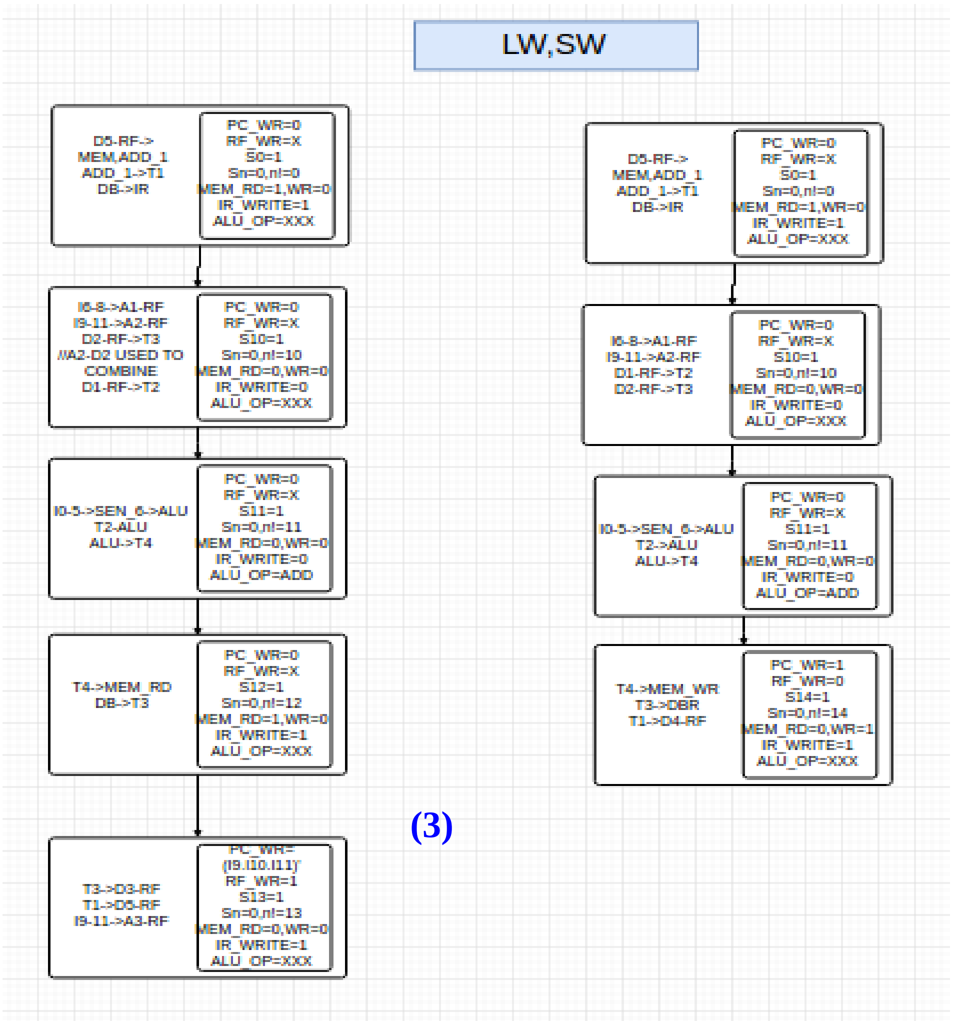
7. 16 bit ADD_1 : The Combinational circuit adds 1 to PC(16 bit) for updation

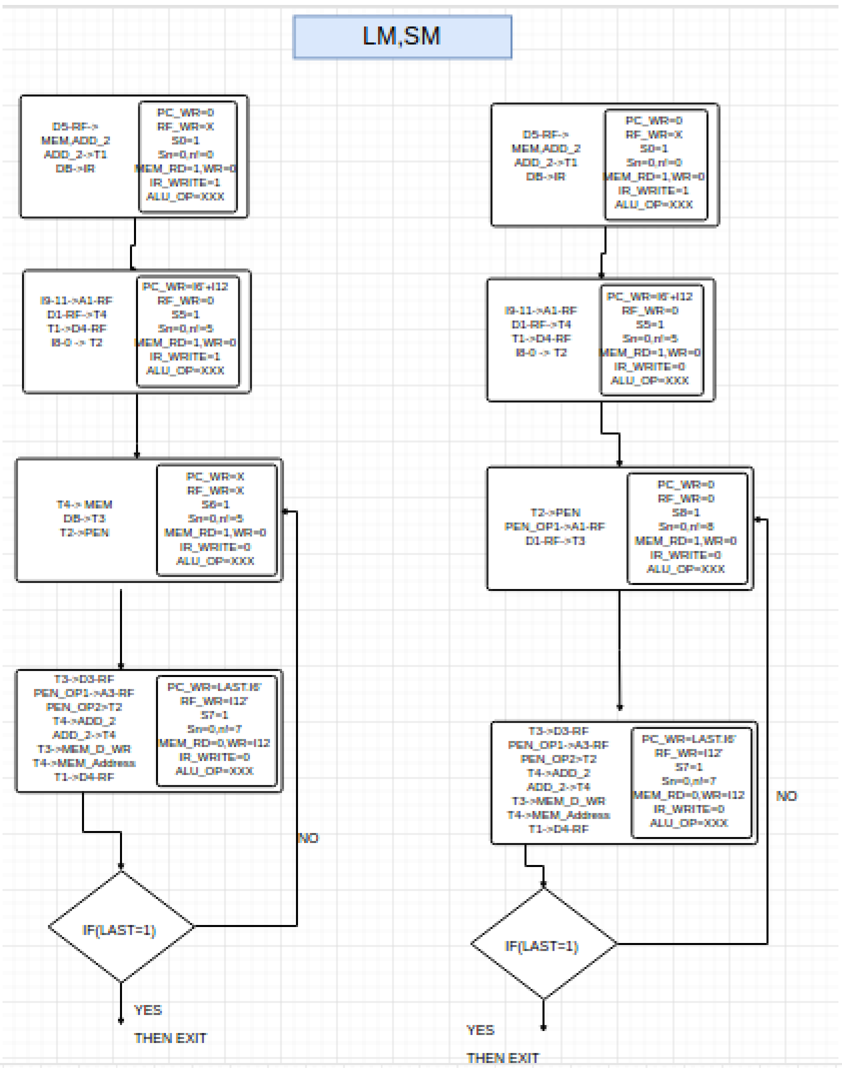
DATAPATH DESIGN :-

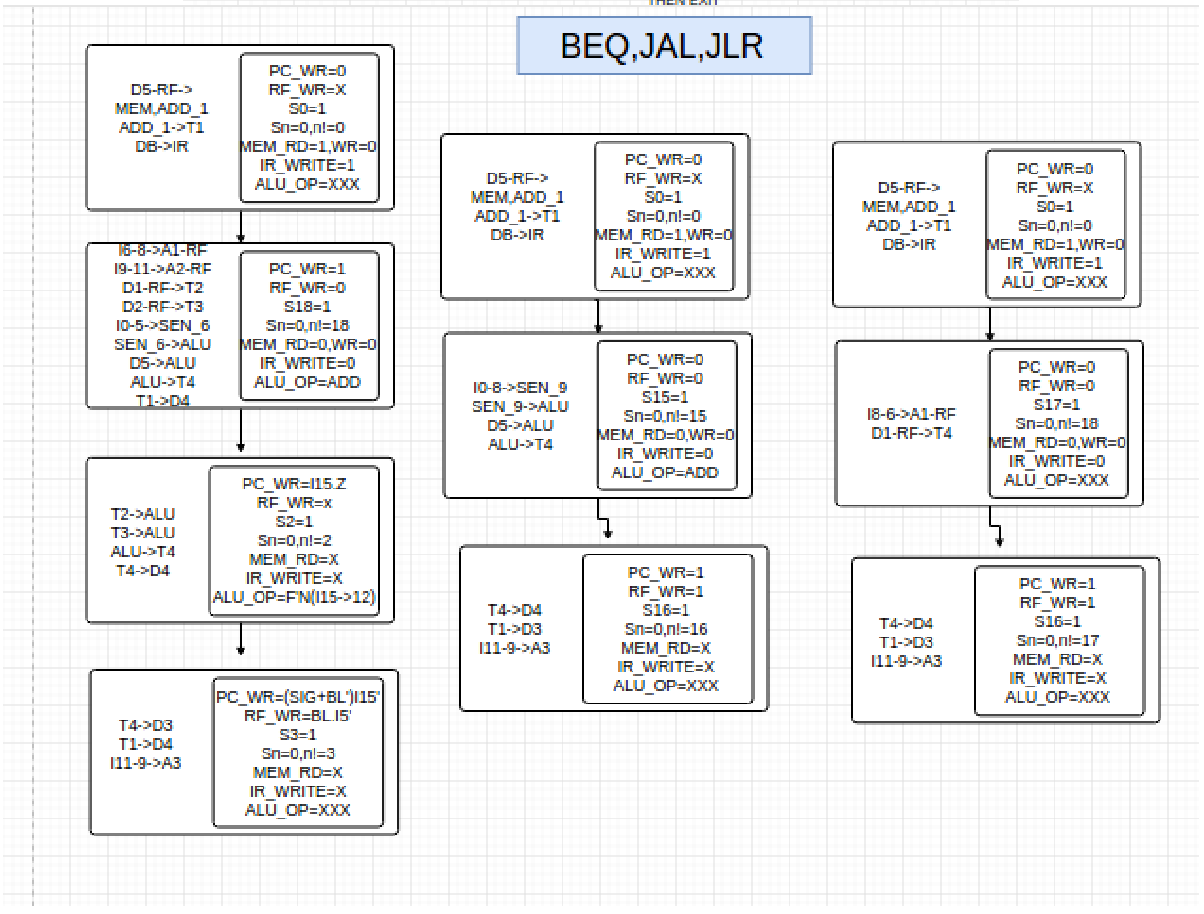


II. FLOWCHART DESCRIPTION



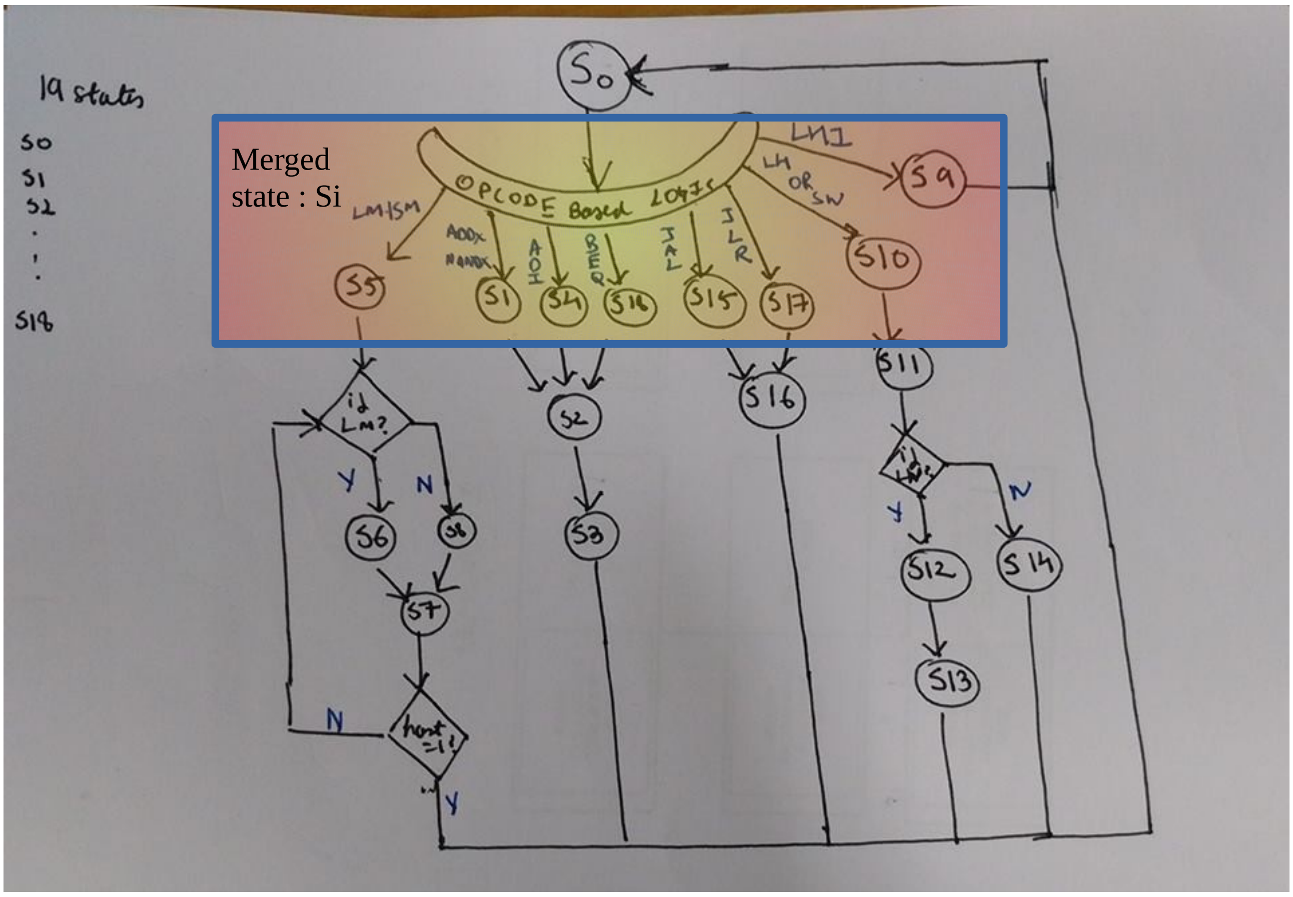



LM,SM
D5-RF->
MEM,ADD_2
ADD_2->T1
DB->IR
PC_WR=0
RF_WR=X
S0=1
Sn=0,n!=0
MEM_RD=1,WR=0
IR_WRITE=1
ALU_OP=XXX
I9-11->A1-RF
D1-RF->T4
T1->D4-RF
I8-0 -> T2
PC_WR=I6'+I12
RF_WR=0
S5=1
Sn=0,n!=5
MEM_RD=1,WR=0
IR_WRITE=1
ALU_OP=XXX
T4-> MEM
DB->T3
T2->PEN
PC_WR=X
RF_WR=X
S6=1
Sn=0,n!=5
MEM_RD=1,WR=0
IR_WRITE=0
ALU_OP=XXX
T3->D3-RF
PEN_OP1->A3-RF
PEN_OP2>T2
T4->ADD_2
ADD_2->T4
T3->MEM_D_WR
T4->MEM_Address
T1->D4-RF
PC_WR=LAST.I6'
RF_WR=I12'
S7=1
Sn=0,n!=7
MEM_RD=0,WR=I12
IR_WRITE=0
ALU_OP=XXX
NO
IF(LAST=1)
YES
THEN EXIT
D5-RF->
MEM,ADD_2
ADD_2->T1
DB->IR
PC_WR=0
RF_WR=X
S0=1
Sn=0,n!=0
MEM_RD=1,WR=0
IR_WRITE=1
ALU_OP=XXX
I9-11->A1-RF
D1-RF->T4
T1->D4-RF
I8-0 -> T2
PC_WR=I6'+I12
RF_WR=0
S5=1
Sn=0,n!=5
MEM_RD=1,WR=0
IR_WRITE=0
ALU_OP=XXX
T2->PEN
PEN_OP1->A1-RF
D1-RF->T3
PC_WR=0
RF_WR=0
S8=1
Sn=0,n!=8
MEM_RD=1,WR=0
IR_WRITE=0
ALU_OP=XXX
T3->D3-RF
PEN_OP1->A3-RF
PEN_OP2>T2
T4->ADD_2
ADD_2->T4
T3->MEM_D_WR
T4->MEM_Address
T1->D4-RF
PC_WR=LAST.I6'
RF_WR=I12'
S7=1
Sn=0,n!=7
MEM_RD=0,WR=I12
IR_WRITE=0
ALU_OP=XXX
NO
IF(LAST=1)
YES
THEN EXIT


BEQ,JAL,JLR
D5-RF->
MEM,ADD_1
ADD_1->T1
DB->IR
PC_WR=0
RF_WR=X
S0=1
Sn=0,n!=0
MEM_RD=1,WR=0
IR_WRITE=1
ALU_OP=XXX
I6-8->A1-RF
I9-11->A2-RF
D1-RF->T2
D2-RF->T3
I0-5->SEN_6
SEN_6->ALU
D5->ALU
ALU->T4
T1->D4
PC_WR=1
RF_WR=0
S18=1
Sn=0,n!=18
MEM_RD=0,WR=0
IR_WRITE=0
ALU_OP=ADD
T2->ALU
T3->ALU
ALU->T4
T4->D4
PC_WR=I15.Z
RF_WR=x
S2=1
Sn=0,n!=2
MEM_RD=X
IR_WRITE=X
ALU_OP=F'N(I15->12)
T4->D3
T1->D4
I11-9->A3
PC_WR=(SIG+BL')I15'
RF_WR=BL.I5'
S3=1
Sn=0,n!=3
MEM_RD=X
IR_WRITE=X
ALU_OP=XXX
D5-RF->
MEM,ADD_1
ADD_1->T1
DB->IR
PC_WR=0
RF_WR=X
S0=1
Sn=0,n!=0
MEM_RD=1,WR=0
IR_WRITE=1
ALU_OP=XXX
I0-8->SEN_9
SEN_9->ALU
D5->ALU
ALU->T4
PC_WR=0
RF_WR=0
S15=1
Sn=0,n!=15
MEM_RD=0,WR=0
IR_WRITE=0
ALU_OP=ADD
T4->D4
T1->D3
I11-9->A3
PC_WR=1
RF_WR=1
S16=1
Sn=0,n!=16
MEM_RD=X
IR_WRITE=X
ALU_OP=XXX
D5-RF->
MEM,ADD_1
ADD_1->T1
DB->IR
PC_WR=0
RF_WR=X
S0=1
Sn=0,n!=0
MEM_RD=1,WR=0
IR_WRITE=1
ALU_OP=XXX
I8-6->A1-RF
D1-RF->T4
PC_WR=0
RF_WR=0
S17=1
Sn=0,n!=18
MEM_RD=0,WR=0
IR_WRITE=0
ALU_OP=XXX
T4->D4
T1->D3
I11-9->A3
PC_WR=1
RF_WR=1
S16=1
Sn=0,n!=17
MEM_RD=X
IR_WRITE=X
ALU_OP=XXX

(4)

III. FSM DESCRIPTION OF THE DESIGN



-> While coding, we merged S1, S4, S5, S18, S15, S17, S10, S9 in a single merged Si state, for ease of usage. Rest all were kept intact

-> The control path had a 32 word "Control Word" which described which sections of the datapath were to be used and how (for more details refer Control_Signals.ods).This was the prime output of control path to datapath.

-> The control path used op-codes and 4 predicates (from priority encoder and other carry/zero bits) for state transition decisions. The predicates and op code were fed as inputs to control path from datapath

IV. CHALLENGING AND INTERESTING PARTS

There were a lot of challenging and interesting parts encountered during the project:

1. **Minimising # of states and Cycles per instruction**
Strategies used involved developing a modified Register File which had separate access for PC, tweaking priority encoder to give a finish bit, and the next address too, at the same time and doing various things in one single state, so that it can be merged, and deciding on basis of op-code which part of that "merged state" shall be put to use (eg States S2, S3, S7).

2. **Describing the Control Signals for Datapath**
Having a 32 bit control word to signal which elements in datapath must be activated, we had a daunting task to describe these for all 19 states accurately. By systematically using a tabular format, we were able to complete it correctly. (Ref. Control_Signals.ods)

3. **Debugging state transitions**
Initially, we had a lot of unwanted transitions due to sequential codes, which caused some extra cycles, which we debugged using GTKwave (more on this in next sections)

4. **Hardware running**
Initially, we had kept memory of 2^16-1 Bytes, which was not synthesisable in Quartus, and there were a lot of wild "FFFF" address calls, causing "segmentation faults", which we effectively solved using a masking string. Creation of a pilot file, which when dumped on FGPA will do the required job was also interesting. We didn't face quite a challenge when we moved to hardware testing, maybe because of stronger debugging in the previous states, but it was very interesting to see it run on FGPA, nevertheless.

V. MAKING OF A SYNTHESISABLE CODE

During the debugging on GTKwave stage, we made a lot of changes in datapath to make sure that state transitions occur when they are supposed to, like :-

1. **Making RegisterFile read combinational** :- Initially we had made RF read clocked, which had caused us some problems in state transitions, but making it combinational, solved most of it, alotugh we had to now control the enable bits of registers connected to it more carefully, but that tradeoff had to be made, and it did pay of for us.

2. **Limiting Memory Space** :- We had a (2^16-1)*2 Bytes of memory, which was not synthesisable by Quartus, which was limited to 1024 bytes, with help of a masking vector.

3. **Initialising Registers :-** We had kept the registers unintialised initially (-u-), which caused problems, but initialisation of all of them to zero vector solved the problem

4. **Proper placement of "Process:" blocks in RAM, Reg File and other relevant places :**
We had to do some changes in where the process started, what was done in it etc to get the code running.

VI. DEBUGGING PROCEDURES USED

Precursor to debugging, we developed a C++ code to convert assembly equivalent code to 16 bit instructions, which made our debugging scope wider and easier. (Ref. Assembly.cpp) Also, instead of blindly hardcoding the memory, we made 2 pathways for memory initialisation, one via a testbench(Ref. Testbench_app2.vhd), and other being the traditional hardcoded approach. We could easily switch between the 2 approaches, using the mem-bit (which decided which pathway to use). Initialising memory from a testbench gave us flexibility to test for more codes in a lesser time.

-> GTKwave Simulations:
We first debugged all instructions one by one in GTKwave, correcting their transition diagrams to what was expected along with it. As we couldn't see the registers directlt in GTKwave, we inspected the inputs to register files (a1,a2,a3,d3,d4) at required clock instances, alongwith other relevant inputs to memory (Data_out, Data_in etc) depending on the instruction being checked.

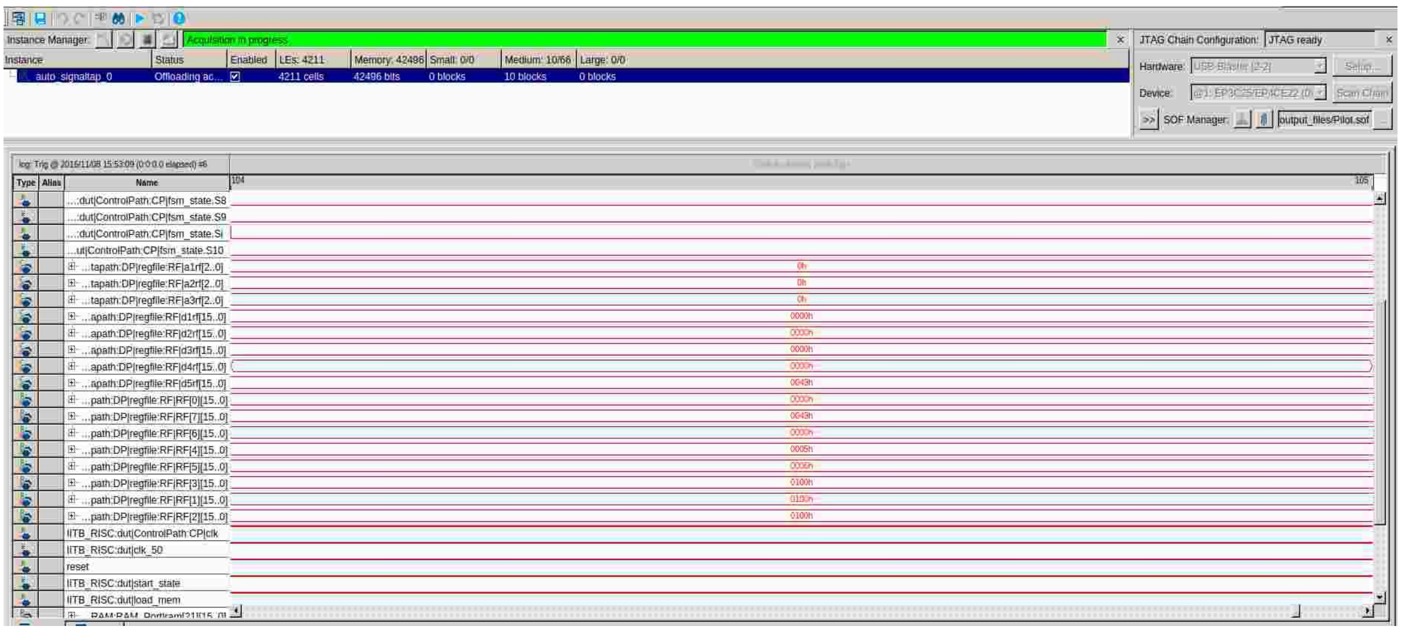
After the individual testing was over, we got a program from our TA (which multiplied 2 16 bit numbers, and stored the resultant 32 bit number in 2 places in memory, and also counted # on zeroes in a 16 bit string). The program had all the instrcutions in it, along with good branch logics, which if ran succesfully could almost make sure that the logic was correct. We initially faced problems running this code due to errogeneous entries in datapath, but we were able to correct them with help of our TA very effectively.

->Modelsim simulations:
After GTKwave ran, we proceeded to modelsim, which didn't notify us of any additional problems regarding the design.

-> Timing Analyzer
After this we checked for setup/hold violations in the TimingQuest Analyzer, which also didn't report anything of much interest to us.

-> SignalTap II Logic Analyzer
Finally, we burned the code on FGPA, and tapped the relevant signals (relevant registers and memory locations), to observe whether it was working on FGPA or not. We had problems with the Analyzer on Quartus 12 [It showed “**Invalid JTAG Configuration”**] but it run flawlessly on Quartus 15. We checked for the TA's multiplication program and another program given by our TA, both of them running perfectly.

FINAL SIGNAL TAP OUTPUT



```
Instance Manager:   Acquisition in progress.
Instance          Status          Enabled  LEs: 4211   Memory: 42496  Small: 0/0  Medium: 10/66  Large: 0/0
auto_signaltap_0  Offloading ac...  ☑      4211 cells  42496 bits     0 blocks    10 blocks      0 blocks

JTAG Chain Configuration:  JTAG ready
Hardware:  USB-Blaster [2-2]      Setup...
Device:    @1: EP3C25/EP4CE22 (0)  Scan Chain
>>  SOF Manager:  output_files/Pilot.sof

log: Trig @ 2016/11/08 15:53:09 (0:0:0.0 elapsed) #6
Type  Alias  Name                                     104                     105
             ...:dut|ControlPath:CP|fsm_state.S8
             ...:dut|ControlPath:CP|fsm_state.S9
             ...:dut|ControlPath:CP|fsm_state.Si
             ...ut|ControlPath:CP|fsm_state.S10
             ...tapath:DP|regfile:RF|a1rf[2..0]        0h
             ...tapath:DP|regfile:RF|a2rf[2..0]        0h
             ...tapath:DP|regfile:RF|a3rf[2..0]        0h
             ...apath:DP|regfile:RF|d1rf[15..0]        0000h
             ...apath:DP|regfile:RF|d2rf[15..0]        0000h
             ...apath:DP|regfile:RF|d3rf[15..0]        0000h
             ...apath:DP|regfile:RF|d4rf[15..0]        0000h
             ...apath:DP|regfile:RF|d5rf[15..0]        0049h
             ...path:DP|regfile:RF|RF[0][15..0]        0000h
             ...path:DP|regfile:RF|RF[7][15..0]        0049h
             ...path:DP|regfile:RF|RF[6][15..0]        0000h
             ...path:DP|regfile:RF|RF[4][15..0]        0005h
             ...path:DP|regfile:RF|RF[5][15..0]        0006h
             ...path:DP|regfile:RF|RF[3][15..0]        0100h
             ...path:DP|regfile:RF|RF[1][15..0]        0100h
             ...path:DP|regfile:RF|RF[2][15..0]        0100h
             IITB_RISC:dut|ControlPath:CP|clk
             IITB_RISC:dut|clk_50
             reset
             IITB_RISC:dut|start_state
             IITB_RISC:dut|load_mem
             ...RAM:RAM_Port|ram[21][15..0]
```

VII. FILE DICTIONARY

1. **RAM_2.vhd** : Combines the hardcoded and non hardcoded approaches for mem initialisation
2. **Pilot.vhd** : Pilot Program for hardware testing which can be directly dumped on FGPA to run the complete design which has been hardcoded into memory
3. **IITB_RISC** : Grand Entity to combine all the other elements of design
4. **TestBench_app2.vhd** : Testbench for simulation of non hardcoded memory
5. **TestBench_hw.vhd** : Testbench for simulation of hardcoded memory
6. **Assembler.cpp** : Converts Equivalent Assembly Code (for an eg. Ref ASSEMBLY_EQ.txt) to Binary 16 bit instructions
7. **Controlpath.vhd** and **Datapath.vhd** : Self Explanatory
8. **TA_FILEh** : File for hardcoded memory
9. **run_TAhf.vcd** : Final VCD for Program given by TA
10. **TA_FILE** : File for non harcoded memory
11. **ASSEMBLY_EQ.txt** : Equivalent Assembly program

-> Refer README.md for further description
-> Refer Test_Program_with_LM_and_SM.txt for further description of the test program loaded into memory