NYU | TANDON SCHOOL OF ENGINEERING

SUBMITTED AS PER THE COURSE REQUIREMENT FOR ADVANCED HARDWARE DESIGN

32 Bit NYU-6463 Processor Design in VHDL

FINAL PROJECT REPORT

Kalpan Mehta	ksm469
Rhitvik Kumawat	rk3494
Yash Shah	Yks256
Himanshu Singh	Hs3773

Submission Date: December 15, 2018

DEMO LINK: https://youtu.be/qNwHtMy65f0

Block Diagram

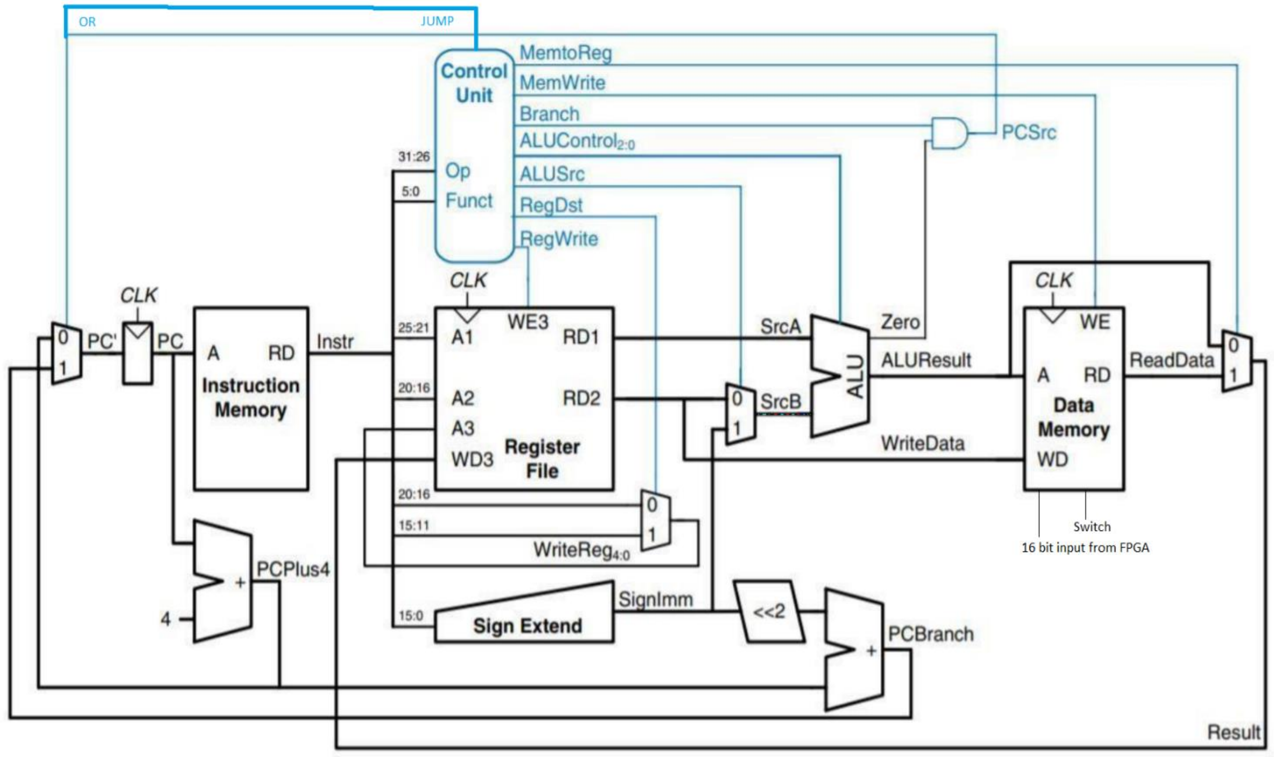


Figure 1 Processor Block Diagram

Processor Functioning:

Our processor functions in almost similar fashion as the original processor. We have added an additional line form the control unit to support jump instructions.

- At the beginning, when reset is set Program Counter will be initialized to 0. After each clock cycle (on the rising edge) PC Value increases by 4.
- That PC value will go into the Instruction Memory Module where, it will fetch the corresponding instruction and that instruction will be sent in the control module and register file to decode.
- Control unit will generate control signals like AlUControl, write enable signals for Register file and Data memory etc.
- The register file will fetch the corresponding values for Rs and Rt. If it is an I type instruction, then Rt value will be taken from signextend immediate. Both the values corresponding to Rs and Rt (or Sign extended immediate) will be the source A and Source B for the ALU. The ALU will do proper operation after receiving the signal ALU Control. The result will be sent into Data memory where the value to be written back to the register will be decided by a mux.

The working of each module and what the module includes is explained below:

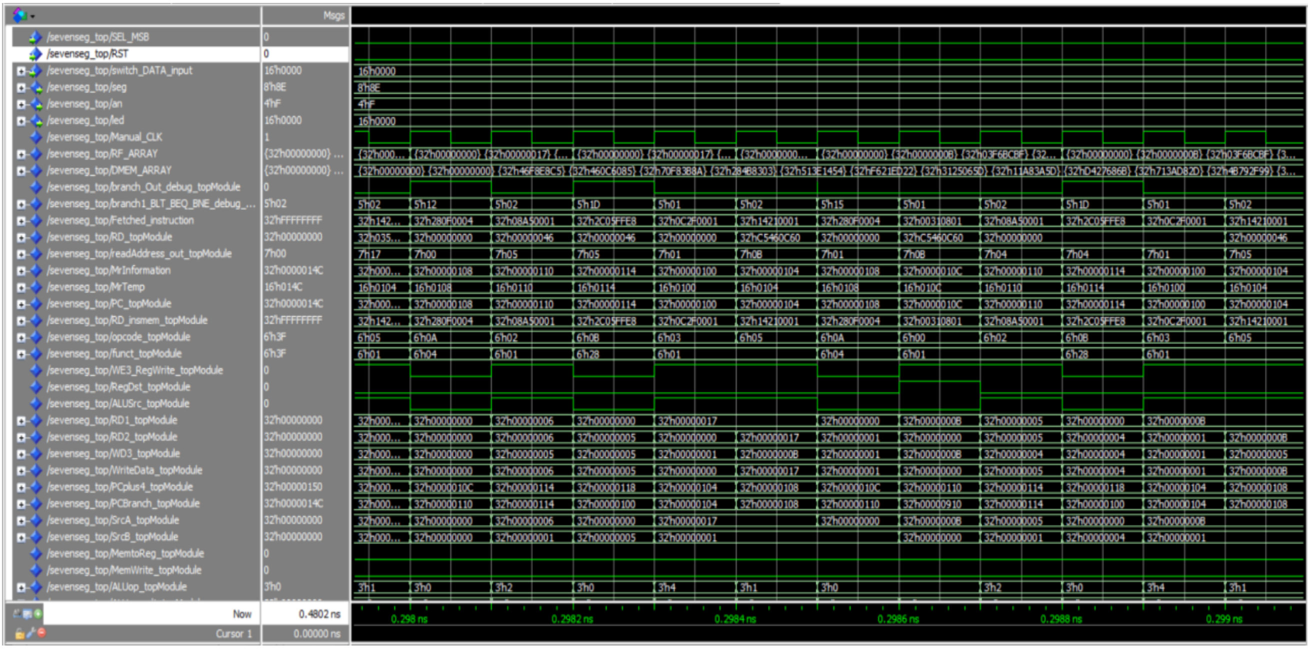
1) **Program Counter:**
    - It will decide the program counter value.
    - It will first increase the value of the program counter by 4 and sent that value to the next modules
    - There is a mux which will decide whether our instruction is a jump instruction or not. If we detect that it is a jump instruction, then our program counter value will be the jump address. Else it will be PC +4. We will call the output of the mux PC1
    - There is another mux which will decide whether we have a branch instruction or not. If Branch = 1 (From the control unit) then we will take the branch address as the new PC else, we will take PC1. We will call the result PC2
    - We have also added another multiplexer for Halt instruction as well.
    - If we detect that we have received halt instruction, then our PC value will not change at all. It will keep on fetching the same instruction until we hit reset.
    - Hence, if halt = 1 then we will take the old PC ({PC +4} -4) as our current PC else we will take PC2 as the output.
    - The resulting value will be our new Program Counter value.
2) **Instruction Memory:**
    - Instruction Memory is straightforward.
    - Our size of the instruction memory is “natural range”. Hence, it will depend on the number of instructions we have.
    - Our instruction memory is byte addressable so, we will concatenate. The instruction available on PC, PC+1, PC+ 2 and PC+3 to make a whole 32 bit instruction.
3) **Control Unit:**
    - Control Unit only cares about the opcode and function bits. Which are first 6 and last 6 bits from the instruction fetched.
    - Once we receive that we will determine various control signals and sent them accordingly to the other modules.
4) **Register File:**
    - Register File is the critical module in our code. Even more critical than control unit.
    - There are many small modules included in our register file.
    - For instance, register file will do the sign extended immediate. It will decide what should be the destination register. For I type, and R type instructions Destination Register is different.
    - Hence, we have used a mux to determine that.
    - Register File will also, determine what should be the input to the ALU (The corresponding value of Rt or Sign extended Immediate)
    - It will fetch Source A and Source B for the ALU.

- Another important thing Register File does is to determine that the branch is correct or not.
- We will compare the Source A and Source B in the register itself and determine whether we should jump to the branch or not.
- The resulting value will be sent to the program counter module.
- We will also calculate the branch address in this module as well.

5) **ALU:**
   - ALU will take the inputs from Register file and Control Unit and calculate the result and send it out

6) **Data memory:**
   - The data memory will determine what to do with the ALU Result. Whether to send it right away or fetch the instruction from the data memory.
   - If it's a load or store type instruction then we will fetch or store the corresponding instruction to or from the data memory.
   - There is a multiplexer to decide whether to take ALU Result as the output or the data memory result as the output.

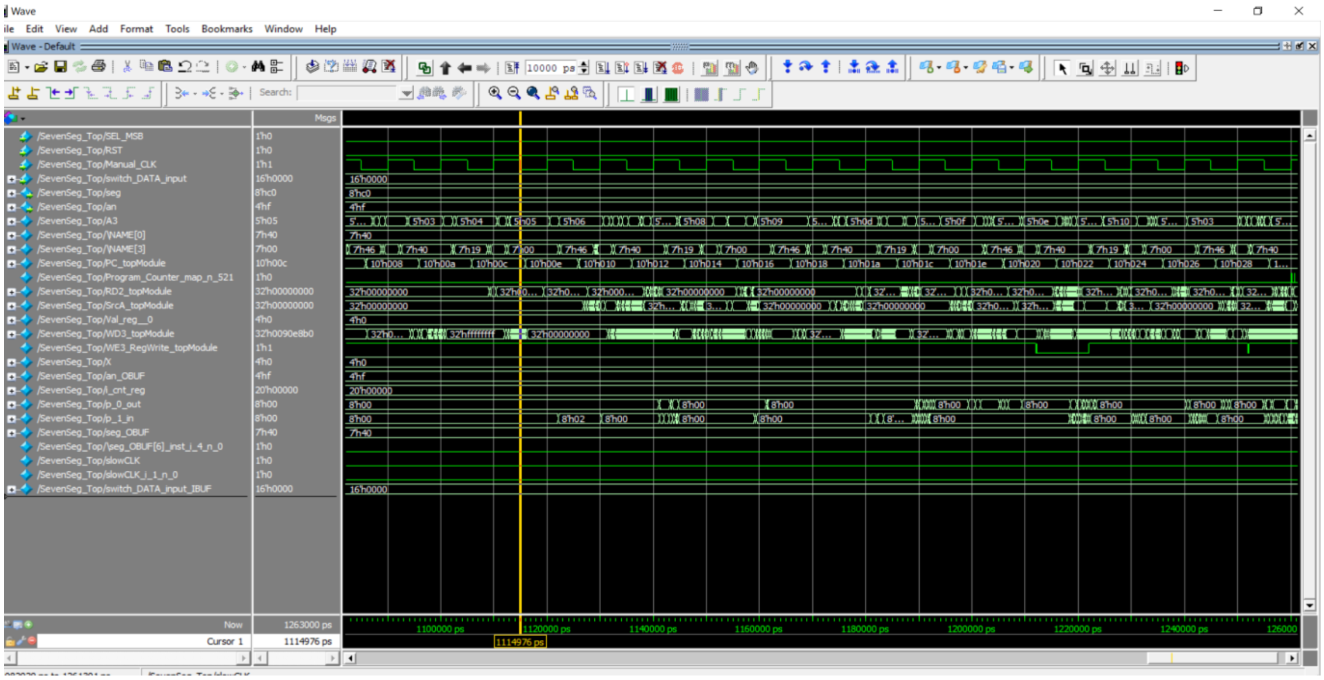
Functional Simulation



```
                                                     Msgs
/sevenseg_top/SEL_MSB                                0
/sevenseg_top/RST                                    0
/sevenseg_top/switch_DATA_input                      16'h0000
/sevenseg_top/seg                                    8'h8E
/sevenseg_top/an                                     4'hF
/sevenseg_top/led                                    16'h0000
/sevenseg_top/Manual_CLK                             1
/sevenseg_top/RF_ARRAY                               {32'h00000000} ...
/sevenseg_top/DMEM_ARRAY                             {32'h00000000} ...
/sevenseg_top/branch_Out_debug_topModule             0
/sevenseg_top/branch1_BLT_BEQ_BNE_debug_...          5'h02
/sevenseg_top/Fetched_instruction                    32'hFFFFFFFF
/sevenseg_top/RD_topModule                           32'h00000000
/sevenseg_top/readAddress_out_topModule              7'h00
/sevenseg_top/MrInformation                          32'h0000014C
/sevenseg_top/MrTemp                                 16'h014C
/sevenseg_top/PC_topModule                           32'h0000014C
/sevenseg_top/RD_insmem_topModule                    32'hFFFFFFFF
/sevenseg_top/opcode_topModule                       6'h3F
/sevenseg_top/funct_topModule                        6'h3F
/sevenseg_top/WE3_RegWrite_topModule                 0
/sevenseg_top/RegDst_topModule                       0
/sevenseg_top/ALUSrc_topModule                       0
/sevenseg_top/RD1_topModule                          32'h00000000
/sevenseg_top/RD2_topModule                          32'h00000000
/sevenseg_top/WD3_topModule                          32'h00000000
/sevenseg_top/WriteData_topModule                    32'h00000000
/sevenseg_top/PCplus4_topModule                      32'h00000150
/sevenseg_top/PCBranch_topModule                     32'h0000014C
/sevenseg_top/SrcA_topModule                         32'h00000000
/sevenseg_top/SrcB_topModule                         32'h00000000
/sevenseg_top/MemtoReg_topModule                     0
/sevenseg_top/MemWrite_topModule                     0
/sevenseg_top/ALUop_topModule                        3'h0
                                              Now    0.4802 ns
                                         Cursor 1    0.00000 ns
```

Figure 2 Functional Simulation

Timing Simulation

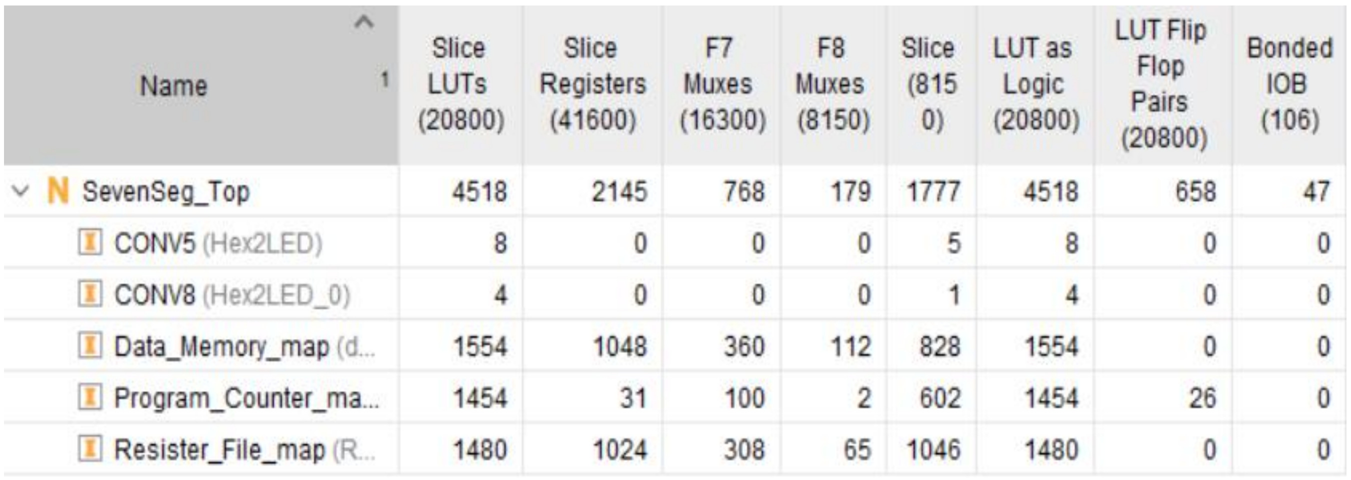


```
/SevenSeg_Top/SEL_MSB                          1'h0
/SevenSeg_Top/RST                              1'h0
/SevenSeg_Top/Manual_CLK                       1'h1
/SevenSeg_Top/switch_DATA_input                16'h0000
/SevenSeg_Top/seg                              8'hc0
/SevenSeg_Top/an                               4'hf
/SevenSeg_Top/A3                               5'h05
/SevenSeg_Top/\NAME[0]                         7'h40
/SevenSeg_Top/\NAME[3]                         7'h00
/SevenSeg_Top/PC_topModule                     10'h00c
/SevenSeg_Top/Program_Counter_map_n_521        1'h0
/SevenSeg_Top/RD2_topModule                    32'h00000000
/SevenSeg_Top/SrcA_topModule                   32'h00000000
/SevenSeg_Top/Val_reg__0                       4'h0
/SevenSeg_Top/WD3_topModule                    32'h0090e8b0
/SevenSeg_Top/WE3_RegWrite_topModule           1'h1
/SevenSeg_Top/X                                4'h0
/SevenSeg_Top/an_OBUF                          4'hf
/SevenSeg_Top/i_cnt_reg                        20'h00000
/SevenSeg_Top/p_0_out                          8'h00
/SevenSeg_Top/p_1_in                           8'h00
/SevenSeg_Top/seg_OBUF                         7'h40
/SevenSeg_Top/\seg_OBUF[6]_inst_i_4_n_0        1'h0
/SevenSeg_Top/slowCLK                          1'h0
/SevenSeg_Top/slowCLK_i_1_n_0                  1'h0
/SevenSeg_Top/switch_DATA_input_IBUF           16'h0000

Now        1263000 ps
Cursor 1   1114976 ps
```

Figure 3 Timing Simulation

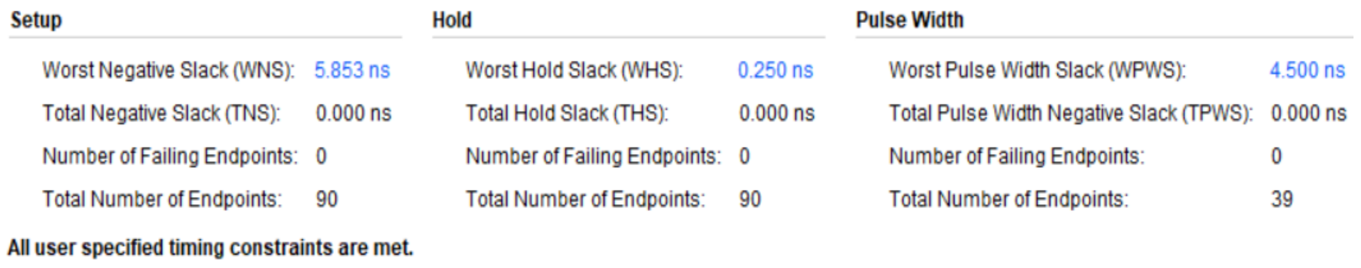
Resource Utilization



Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Bonded IOB (106)
SevenSeg_Top	4518	2145	768	179	1777	4518	658	47
CONV5 (Hex2LED)	8	0	0	0	5	8	0	0
CONV8 (Hex2LED_0)	4	0	0	0	1	4	0	0
Data_Memory_map (d...	1554	1048	360	112	828	1554	0	0
Program_Counter_ma...	1454	31	100	2	602	1454	26	0
Resister_File_map (R...	1480	1024	308	65	1046	1480	0	0

Figure 4 Resource Utilization

Speed of Design



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.853 ns	Worst Hold Slack (WHS):	0.250 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	90	Total Number of Endpoints:	90	Total Number of Endpoints:	39

All user specified timing constraints are met.

Figure 5 Worst Negative Slack

Critical Path Delay is 5.853 ns

Hence,

The Speed of Design will be: 24.1138 MHz

Description of RC5 Implementation

The most critical thing to implement was doing left rotate or right rotate using right shift.

We have prepared a diagram to give a general idea of how we have achieved it.

In both the cases (Left rotate or right rotate), we have done it one step at a time.

For instance, if we wanted to perform left rotate by 5, then we did left rotate by 1, 5 times.

- **The basic concept used here is that if we add one binary number with the same value then we will get a zero at the end. Which is also equivalent to multiplying the number by 2.**

So to achieve left shift by 1 we simply added the same value again. This will add a zero at the end and we will get a left shift by 1. E.G. If we want to left shift “1010” by 1 will simply do “1010” + “1010”

Which will give us”10100”. We will discard the MSB to get “0100” which is the correct result for left shift by 1.

LEFT ROTATE:

- We have first done right shift by 31 bits.
- This will give us the MSB in the position of LSB.
- Then we have added the number which we need to rotate by the same number.
- As proved earlier, we will get a left shift by 1 bit. I.e. there will be a zero in the position of LSB.
- We will add both the left shifted and right shifted value.
- This will give us left rotate by 1.
- We will keep on rotating until we get the actual rotate which we want by using a branch instruction.

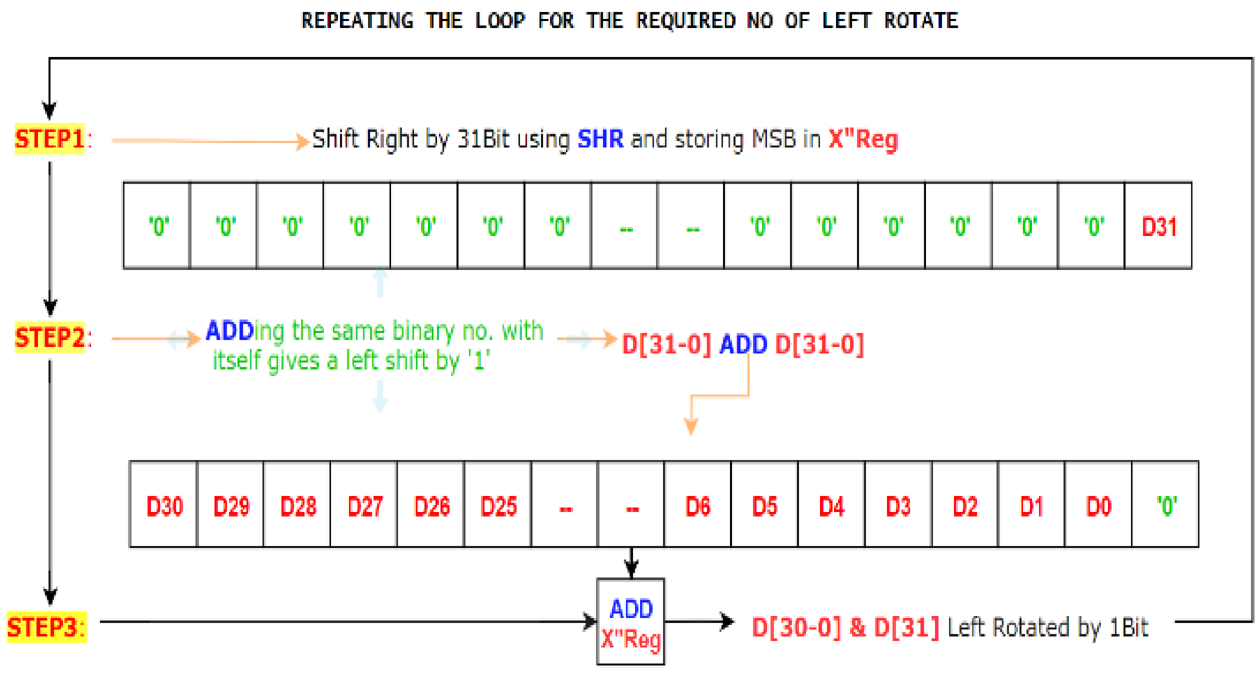
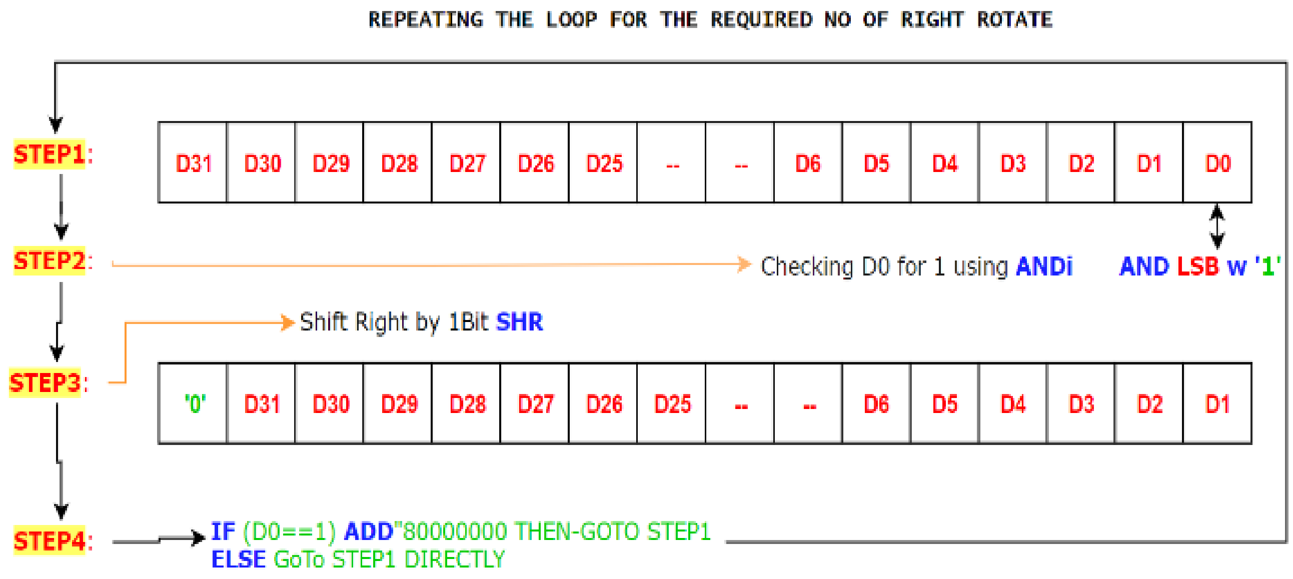

REPEATING THE LOOP FOR THE REQUIRED NO OF LEFT ROTATE
STEP1:
Shift Right by 31Bit using SHR and storing MSB in X"Reg
'0' '0' '0' '0' '0' '0' '0' -- -- '0' '0' '0' '0' '0' '0' D31
STEP2:
ADDing the same binary no. with itself gives a left shift by '1'
D[31-0] ADD D[31-0]
D30 D29 D28 D27 D26 D25 -- -- D6 D5 D4 D3 D2 D1 D0 '0'
STEP3:
ADD X"Reg
D[30-0] & D[31] Left Rotated by 1Bit

Figure 6 Left Rotate using right shift

RIGHT ROTATE:

- Right rotate is a little trickier then left rotate.
- We will first check whether the LSB is 1 or 0.
- Then we will right shift the number by 1 bit.
- If the LSB is 0 then we will do nothing else we will add 1 in the MSB position. (By adding x"80000000")
- This will give us right rotate by 1.
- After this we will use branch condition to run the loop.

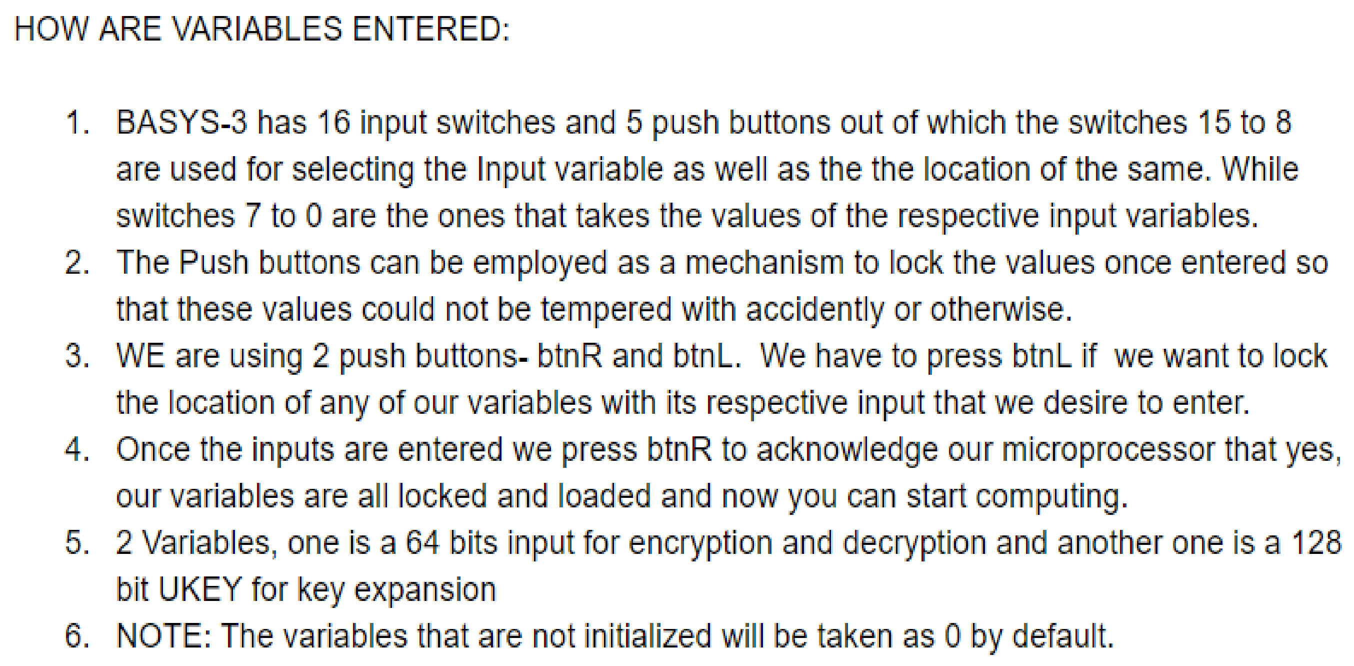


REPEATING THE LOOP FOR THE REQUIRED NO OF RIGHT ROTATE
STEP1:
D31 D30 D29 D28 D27 D26 D25 -- -- D6 D5 D4 D3 D2 D1 D0
STEP2:
Checking D0 for 1 using ANDi
AND LSB w '1'
Shift Right by 1Bit SHR
STEP3:
'0' D31 D30 D29 D28 D27 D26 D25 -- -- D6 D5 D4 D3 D2 D1
STEP4:
IF (D0==1) ADD"80000000 THEN-GOTO STEP1
ELSE GoTo STEP1 DIRECTLY

Figure 7 Right rotate using right shift

Processor Interfaces

Input:



HOW ARE VARIABLES ENTERED:

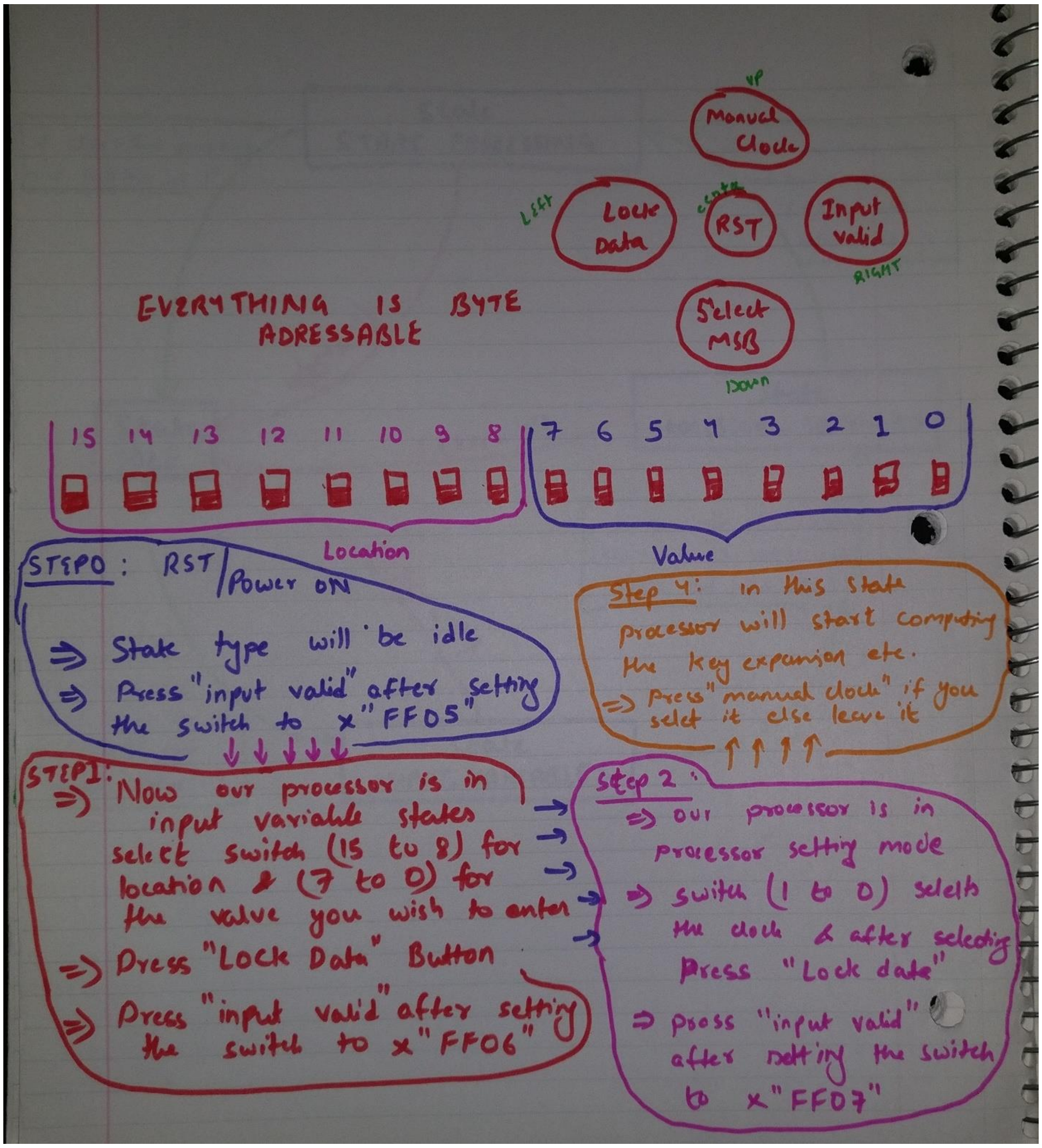
1. BASYS-3 has 16 input switches and 5 push buttons out of which the switches 15 to 8 are used for selecting the Input variable as well as the the location of the same. While switches 7 to 0 are the ones that takes the values of the respective input variables.
2. The Push buttons can be employed as a mechanism to lock the values once entered so that these values could not be tempered with accidently or otherwise.
3. WE are using 2 push buttons- btnR and btnL. We have to press btnL if we want to lock the location of any of our variables with its respective input that we desire to enter.
4. Once the inputs are entered we press btnR to acknowledge our microprocessor that yes, our variables are all locked and loaded and now you can start computing.
5. 2 Variables, one is a 64 bits input for encryption and decryption and another one is a 128 bit UKEY for key expansion
6. NOTE: The variables that are not initialized will be taken as 0 by default.

- **We have also added a switch for encryption and decryption.**
- If encryption is set to 1 then we will first generate keys and perform encryption.
- If encryption is set to 0 then we will perform decryption after key generation.
- The keys will not be generated again even if we change from encryption to decryption.
- If we wish to generate keys with another ukey then we will need to press reset.

Outputs:

The output is the **x-factor** of our processor.

- We can basically see everything on the LCD in hex.
- We can see Program counter, ALL the register file value and data memory value, ALU Result, every control signal from control unit etc. Almost everything which is interfaced through the top module can be observed on the FPGA Board.



EVERYTHING IS BYTE ADRESSABLE

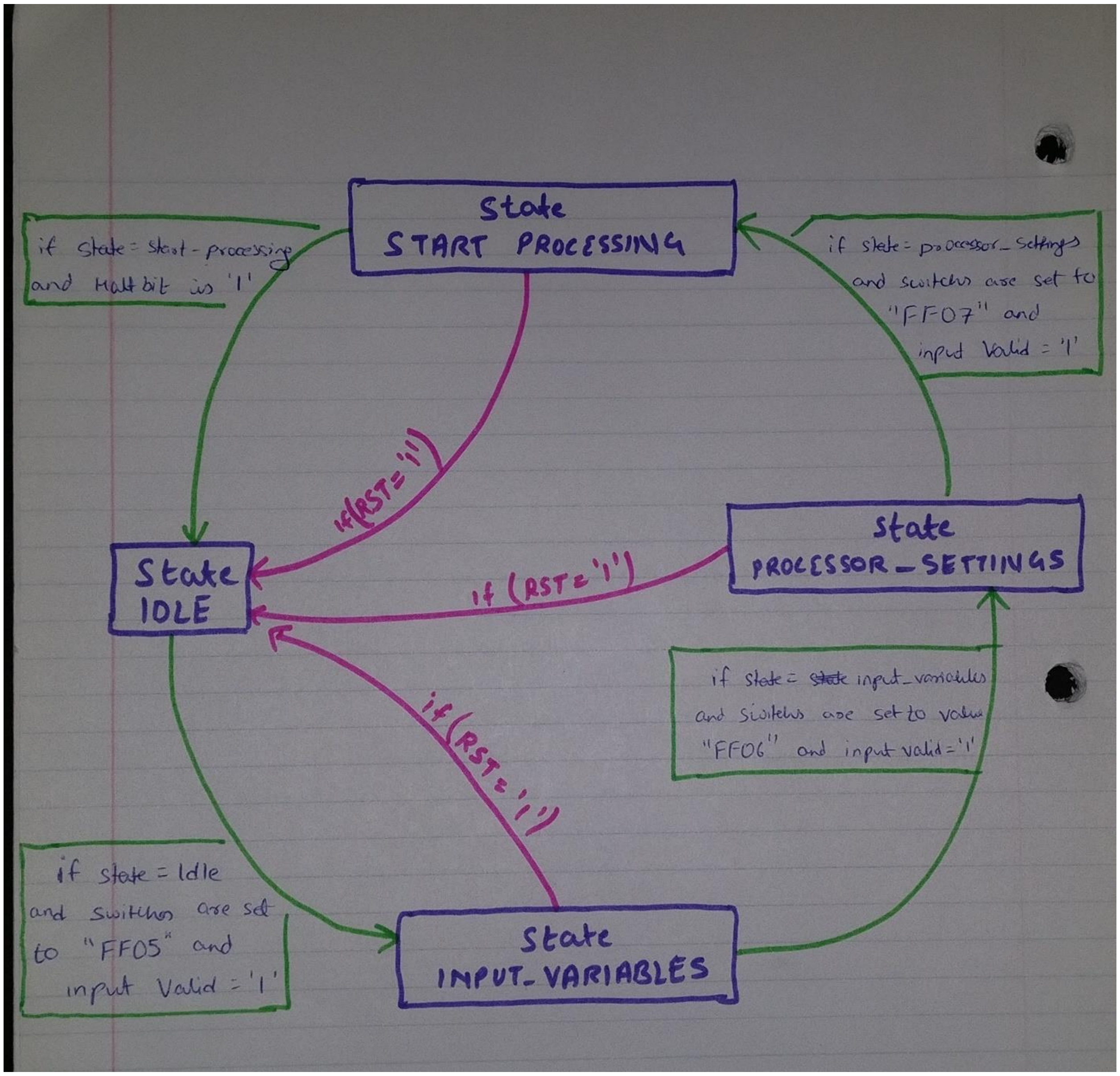
STEP0 : RST / Power ON
⇒ State type will be idle
⇒ Press "input valid" after setting the switch to x"FF05"

STEP1: ⇒ Now our processor is in input variable states select switch (15 to 8) for location & (7 to 0) for the valve you wish to enter
⇒ Press "Lock Data" Button
⇒ Press "input valid" after setting the switch to x"FF06"

Step 2: ⇒ our processor is in processor setting mode
⇒ switch (1 to 0) selects the clock & after selecting Press "Lock data"
⇒ press "input valid" after setting the switch to x"FF07"

Step 4: In this state processor will start computing the key expansion etc.
⇒ Press "manual clock" if you select it else leave it

Interfacing our processor with the outside world


State START PROCESSING
State IDLE
State PROCESSOR_SETTINGS
State INPUT_VARIABLES
if State = Start-processing and Haltbit is '1'
if state = processor_settings and switches are set to "FF07" and input Valid = '1'
if(RST='1')
if (RST='1')
if(RST='1')
if state = ~~state~~ input_variables and switches are set to value "FF06" and input valid='1'
if state = Idle and switches are set to "FF05" and input Valid = '1'

State Cycle of our Processor

Verification

- We have made a testbench which will take the inputs from a text file and store the output in another text file.
- If the outputs to the text file doesn't match with the expected result then the code will generate and error.
- The loop will run till the end of file. So, if there are 3 inputs in the file then loop will run 3 times, if there are 400 inputs then the loop will run 400 times.

Optimization in the future

- One obvious optimization to this single cycle processor is implementation of pipeline. We actually tried to implement the pipeline in this processor itself but could not figure out what things should be on the hardware level and what should be on the software level.
- The pipelining is the main reason why we are calculating the branch results in the RF stage. This will save us one cycle if we mis predict the branch. (As we have studied in the course Computer System Architecture).
- Another optimization can be done in the testbenches to make the entire process more fast and efficient. We will need time to figure that out though.
- Also, we could really improve the efficiency in writing the instructions if we just add left rotate and right rotate in the ALU itself. Which will drastically improve the performance as the code size will become small.

****