ELE 432 – ADVANCED DIGITAL DESIGN

PROJECT REPORT

HACETTEPE UNIVERSITY

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

PROJECT TITLE: Design of a 16-bit RISC Processor Using VHDL

PROJECT GROUP MEMBERS:

Özgürcan EDİŞ	Selçuk HOLEP	Burak Celal KAN	Mehmet Bahadır MAKTAV
21728223	21528191	21728385	21628607

SUBMISSION DATE: 27.05.2021

SPRING 2020-2021

TABLE OF CONTENTS

1. INTRODUCTION

A central processing unit (CPU) is the electronic circuitry that executes instructions such as basic arithmetic, logic, controlling and input/output operations specified by the instructions in the program. To perform these instructions, there are components inside the CPU such as CU, ALU, Registers, MAR, RAM, etc.

Processors are used in a wide range of applications such as computers, telephones, satellites, IoT systems and so on.

Modern processors employ intelligent processor architectures to exploit the parallelism of the computer programs. Von-Neumann architecture, Harvard architecture, CISC (Complex Instruction Set Computer), RISC (Reduced Instruction Set) can be given as examples for these processors.

Most complex instructions in CISC processor take many processor cycles to execute. In a pipelined processor, the overall speed of the processor depends on the slowest operation being performed. This means that the relatively complex instructions even slow down the execution of simpler instructions. That is why the RISC architecture is so popular in today's technology. First, simpler instructions could speed up the pipeline and thus provide a performance improvement. Second, simple instruction set implies less computer hardware and thus reduced cost. Therefore, the design goal was to provide simple instructions, which could execute faster. Compilers could use these instructions to construct complex operations.

In this report, we are going to present our design of a 16-bit RISC processor using VHDL. The fetch, decode and execute parts of the instruction cycle will be explained and the advantages of our design with respect to the base article of the project will be discussed. Environmental and economical aspects of a CPU design is an important issue for the future and will also be explained throughout the project report.

2. DESIGN AND SOLUTION

2.1 REFERENCE DESIGN

Schematic of the article given on the project instructions page is as given below.

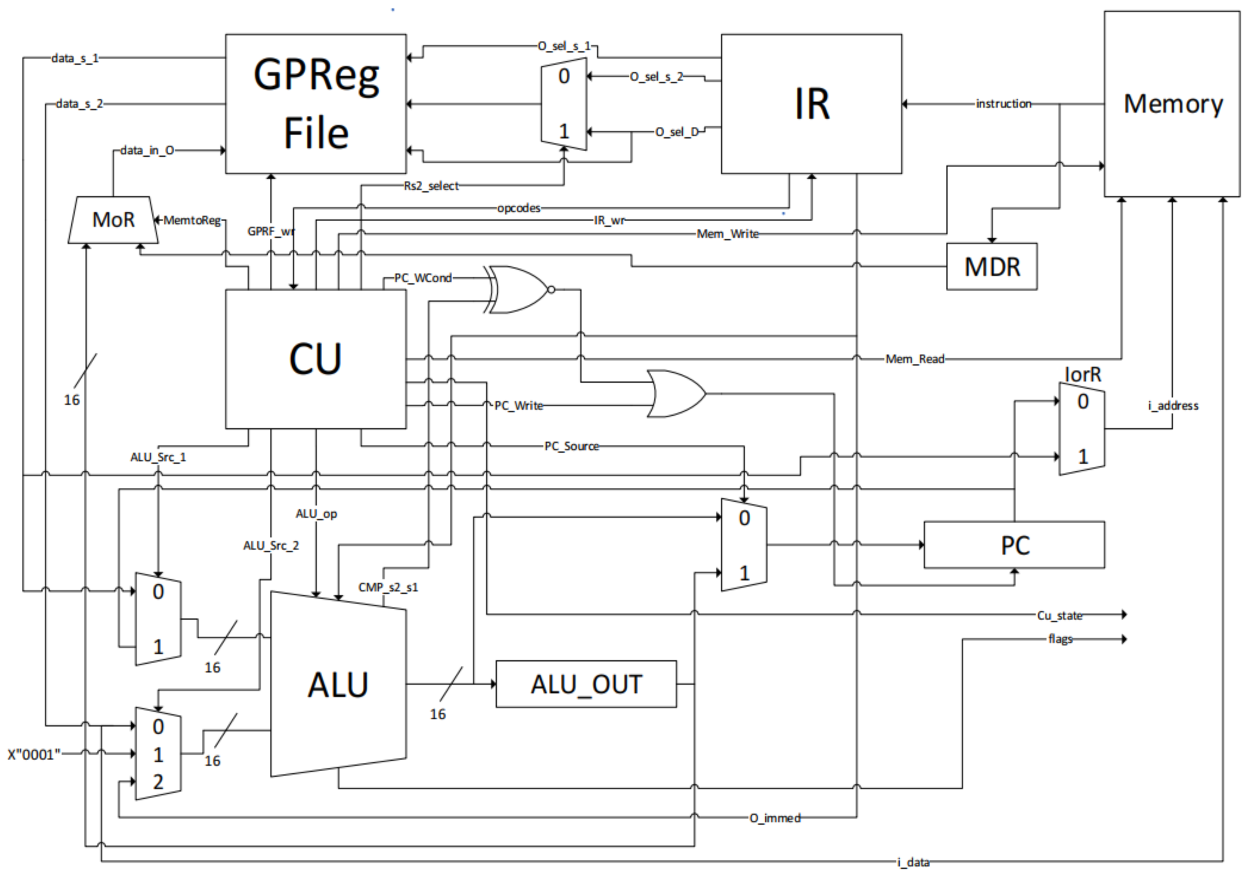


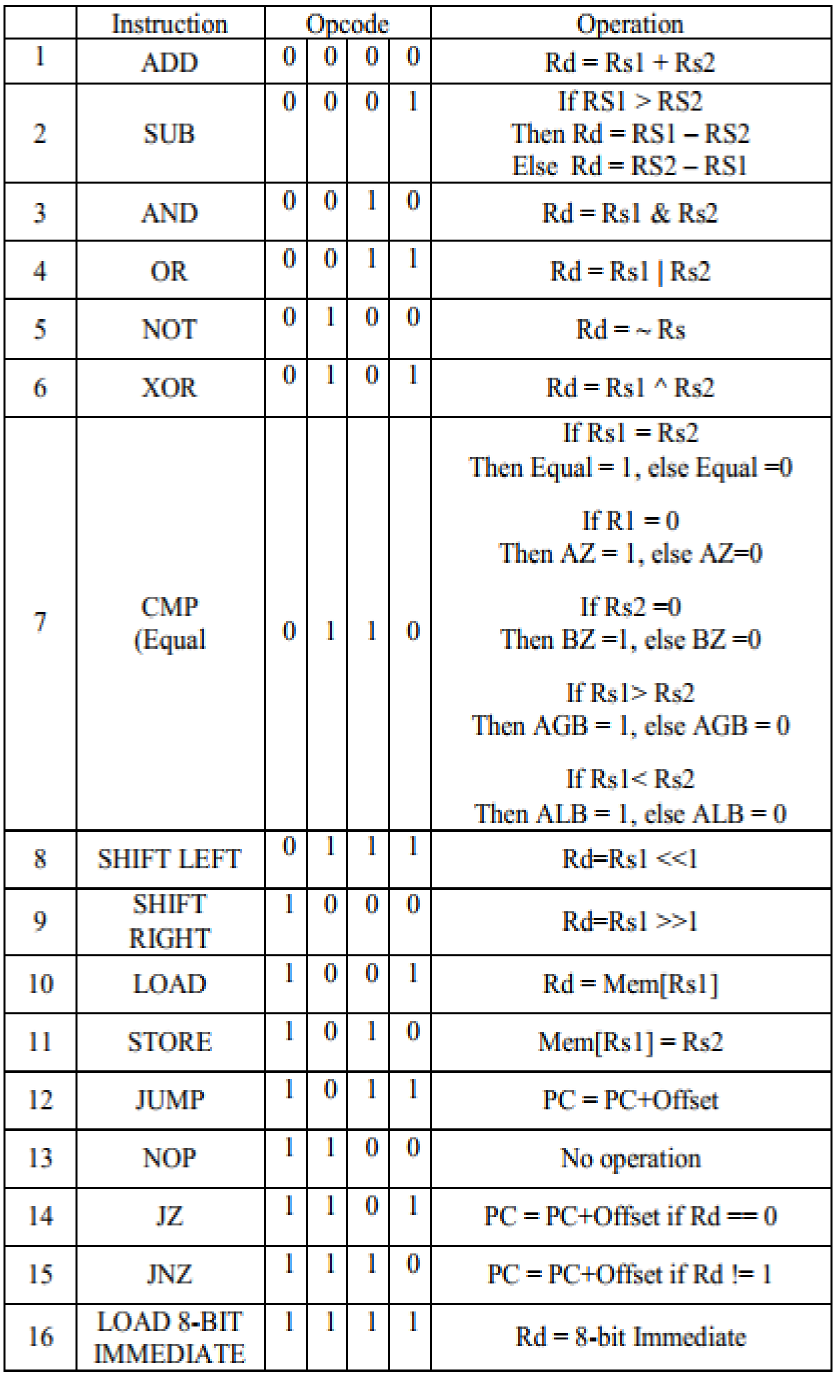
Figure 2.1: Reference Design

As can be seen from the schematic, design units are given as blocks. So, the interna design of these blocks and thus, the design of the general schematic is up to the readers' choices. Names of the registers and data lines are changed in our design to ensure the user friendliness of the written VHDL code.

Some multiplexers and the logic gates are implemented into the design units for better readability. Data busses for the design units are connected to the same general line and the enable signals are used to specify the direction of the data flow.

Keeping this in mind, our RTL schematic is as given below.

Instruction set of the base design is as given below. Some of these instructions are implemented using a single opcode. As an example, MOV, STORE and LOAD instructions are implemented using a single opcode. IR is modified to realize this operation. Nevertheless, outputs of the functions match with the theoretical ones for our design.



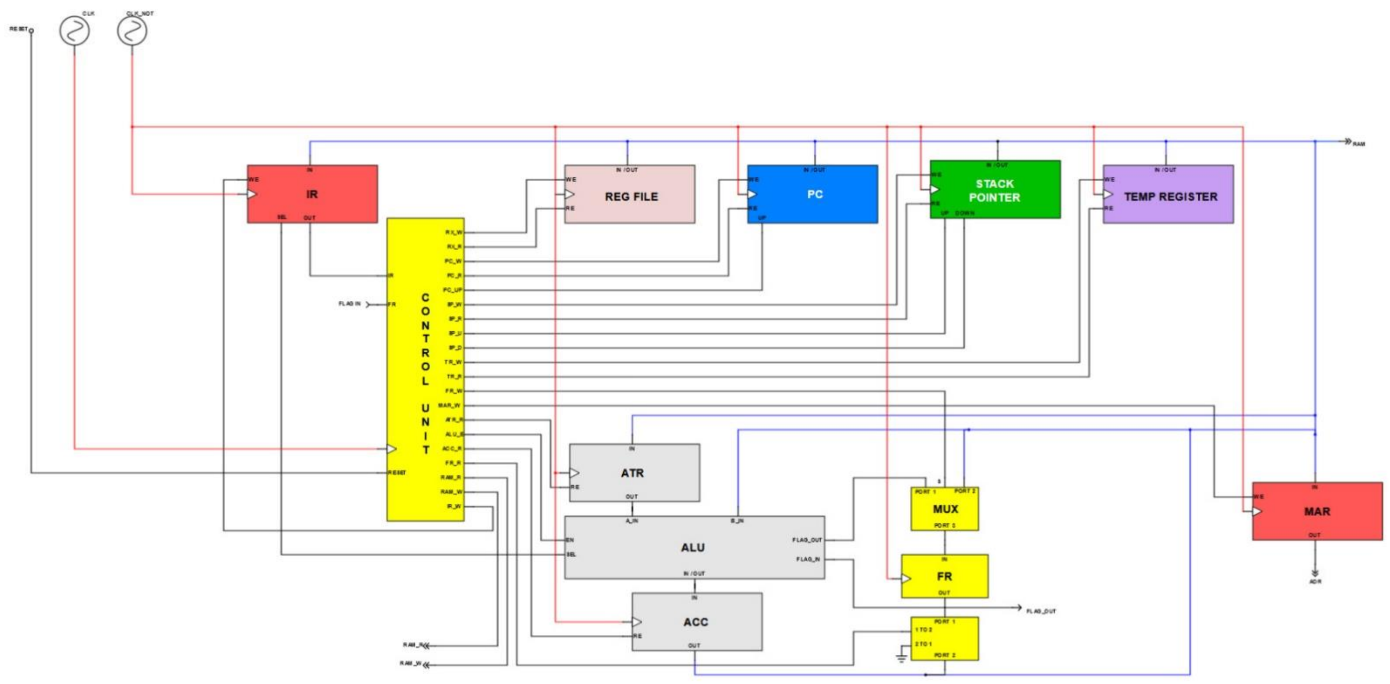
	Instruction	Opcode				Operation
1	ADD	0	0	0	0	Rd = Rs1 + Rs2
2	SUB	0	0	0	1	If RS1 > RS2 Then Rd = RS1 – RS2 Else Rd = RS2 – RS1
3	AND	0	0	1	0	Rd = Rs1 & Rs2
4	OR	0	0	1	1	Rd = Rs1 | Rs2
5	NOT	0	1	0	0	Rd = ~ Rs
6	XOR	0	1	0	1	Rd = Rs1 ^ Rs2
7	CMP (Equal	0	1	1	0	If Rs1 = Rs2 Then Equal = 1, else Equal =0 If R1 = 0 Then AZ = 1, else AZ=0 If Rs2 =0 Then BZ =1, else BZ =0 If Rs1> Rs2 Then AGB = 1, else AGB = 0 If Rs1< Rs2 Then ALB = 1, else ALB = 0
8	SHIFT LEFT	0	1	1	1	Rd=Rs1 <<1
9	SHIFT RIGHT	1	0	0	0	Rd=Rs1 >>1
10	LOAD	1	0	0	1	Rd = Mem[Rs1]
11	STORE	1	0	1	0	Mem[Rs1] = Rs2
12	JUMP	1	0	1	1	PC = PC+Offset
13	NOP	1	1	0	0	No operation
14	JZ	1	1	0	1	PC = PC+Offset if Rd == 0
15	JNZ	1	1	1	0	PC = PC+Offset if Rd != 1
16	LOAD 8-BIT IMMEDIATE	1	1	1	1	Rd = 8-bit Immediate

Figure 2.2: Reference Instruction Table

Now, we will be explaining our design and instruction set. First, here is our schematic for the inner design of the CPU.

2.2 SOLUTION

Figure 2.3: Schematic of the Processor



As can be seen from the schematic, we have made several changes to the reference schematic. These changes can be listed as below.

- We have used a single data bus for the register connections to reduce power consumption. Thus, the production cost of the chip is lowered.
- Number of instructions are increased to 31. Thus, opcodes are extended to 5 bits. As a result, source registers are taken one by one, and the destination register is chosen according to the IR8-IR10 bits of the instruction register for three operand instructions. This means that the destination register for the three operand instructions can be chosen from R0 to R6 using these bits.
- Immediate data and the memory address are taken as 16-bit using an extra cycle.
- A stack segment and a stack pointer are added to the design. To ensure that the code segment and the stack segment do not crosses addresses, stack is located at the end of the RAM.

- An improved FSM is used in the CU to control the timing of the enable signals dynamically. This way, instruction life cycle depends on the type of the instruction. As a result, faster CPU operation is acquired. Details will be given later.

2.3 OUR DESIGN

Top level schematic is made up of two components, CPU and RAM. Connections between these two is established using separate address and data buses. Schematic of the Controller (top level entity) is as shown below.

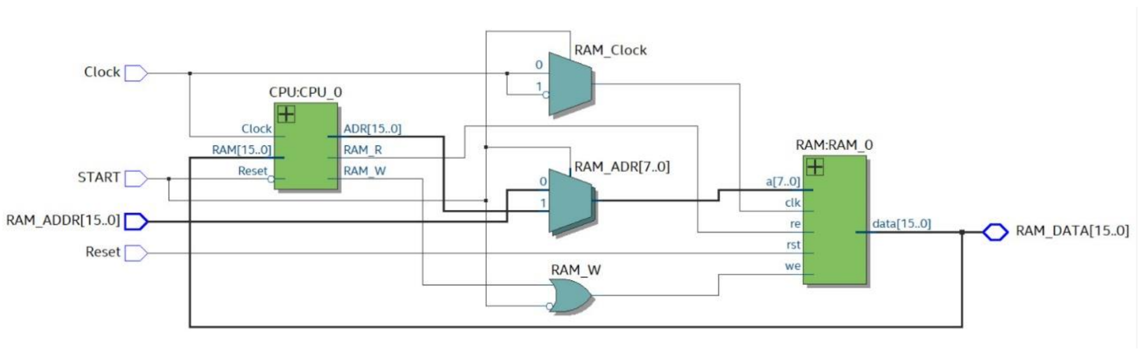


Figure 2.4: Controller Schematic

Data bus and address bus are 16 bits, and the RAM is configured before the program starts. To configure the RAM, START signal is disabled and the specified values are loaded to the specified addresses located in the RAM synchronously. Memory capacity of the unit is 64K x 16 bits. Thus, CPU can address up to 64K locations.

CPU is designed in a structural fashion. Control Unit (CU), Arithmetic-Logic Unit (ALU), Register File, Program Counter, Stack Pointer, Temporary Registers (ATR and TR), Accumulator, Instruction Register (IR), Memory Address Register (MAR) and Flag Register (FR) are the components of this structure. There is a phase difference of 180 degrees between the clock of the CU and the other components of the structure.

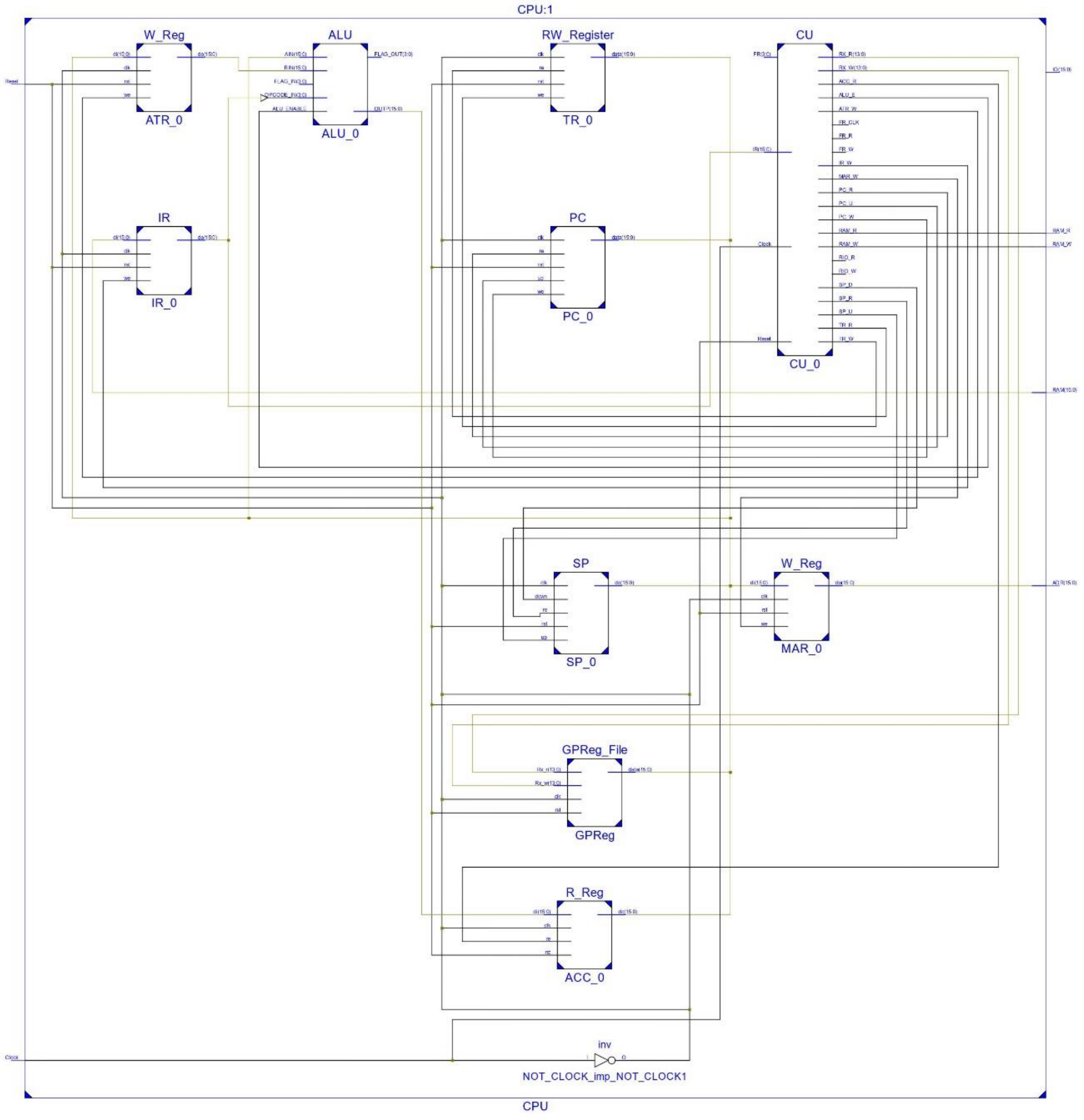


Figure 2.5: RTL Schematic of the Processor

CONTROL UNIT

Control unit is the most important building block of the CPU. System is controlled using the control signals outputted from the unit and the execution time of the instructions are made dynamic using an FSM named as "Stepper".

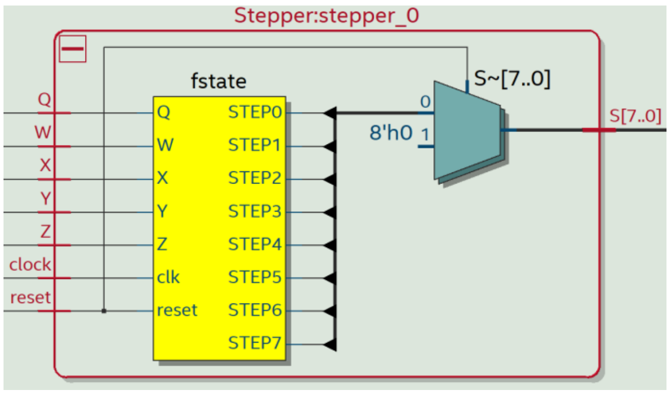


Figure 2.6 Inside of the Stepper

Details of each step is as given below.

STEP 0	Data on program counter(PC) to memory address register(MAR).
STEP 1	Data on ram to instruction register(IR). As the same time increase PC.
STEP 2	If an operand is needed to be taken as immediate data or from RAM, repeat step 0. After repeating the step 0, go to step 3. If the instruction is STORE or LOAD, go to step 4. If none of the above is necessary, go to step 5.
STEP 3	Data on RAM to temp register(TR)*.As the same time increase PC. **Temp register(TR) is a register which stores operands (Immediate data, RAM)*
STEP 4	If the operand is RAM address of data set TR to MAR. If the instruction is STORE or LOAD, set GP register to MAR.
STEP 5	Enable ALU or Control Unit Decoder (CUD) by checking instruction type (IR15); (1 → CU instruction , 0 → ALU instruction). After decoding if the instruction; - is a CU instruction, execute and return step 0. If stack instruction, data on stack pointer (SP) to MAR and go to step 6. - Is an ALU instruction, write destination data to ALU Temp Register(ATR).
STEP 6	Read source data. If PUSH, move source register to stack and decrement SP. If POP, increment SP. Return to step 0 if stack instruction (W input = 1).
STEP 7	If an ALU instruction is being executed, write the data on accumulator to destination register. If POP, move data on stack to destination register.

Stepper state diagram is as shown below.

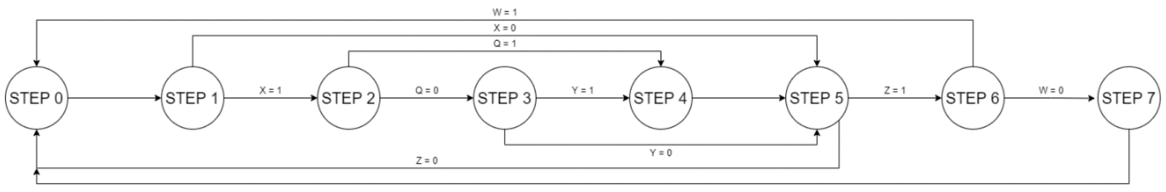


Figure 2.7 Stepper State Diagram

Instruction set of our design is as given on the next two pages.

INSTRUCTION SET																
MNEMONIC	INSTRUCTION CODE															
	D15	D14	D13	D12	D11	D10	D9	D8	D3	D3	D3	D3	D9	D9	D1	D0
ADD	0	0	0	0	0	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Add source 1 to source 2 and store in source 1 or destination																
SUB	0	0	0	0	1	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Subtract source 2 from source 1 and store in source 1 or destination																
AND	0	0	0	1	0	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Logical AND source 1 with source 2 and store in source 1 or destination																
OR	0	0	0	1	1	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Logical OR source 1 with source 2 and store in source 1 or destination																
NOT	0	0	1	0	0	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Logical NOT source 1 with source 2 and store in source 1 or destination																
XOR	0	0	1	0	1	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Logical XOR source 1 with source 2 and store in source 1 or destination																
CMP	0	0	1	1	0	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Compare source 1 with source 2 and store the result in source 1 or destination																
SHL	0	0	1	1	1	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Shift source 1 by 1 bit to the left and store the result in source 1 or destination																
SHR	0	1	0	0	0	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Shift source 1 by 1 bit to the right and store the result in source 1 or destination																
INC	0	1	0	0	1	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Increment source 1 by 1 and store the result in source 1 or destination																
DEC	0	1	0	1	0	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Decrement source by 1 and store the result in source 1 or destination																
ROR	0	1	0	1	1	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Rotate source 1 from left to the right and store the result in source 1 or destination																
ROL	0	1	1	0	0	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Rotate source 1 from right to the left and store the result in source 1 or destination																
ADC	0	1	1	0	1	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Add soruce 1 to source 2 with carry and store the result in source 1 or destination																
SBB	0	1	1	1	0	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Subtract source 1 from source 2 with borrow and store the result in source 1 or destination																
MDL	0	1	1	1	1	D	D	D	S1(D)	S1(D)	S1(D)	S1(D)	S2	S2	S2	S2
Take the modulo of source 1 and store the result in source 1 or destination																

MOV/STORE/LOAD	1	0	0	0	0	X	0/1	0/1	D	D	D	D	S	S	S	S
Move source to destination (D(8) = 0). STORE if D(9) = 1 and D(8) = 1, stores the content of the source register on the memory address shown by the destination register. LOAD if D(9) = 0 and D(8) = 1, loads the content on the memory address shown by the source register to the destination register.																
XCHG	1	0	0	0	1	X	X	X	D	D	D	D	S	S	S	S
Exchange source with destination																
JMP	1	0	0	1	0	X	X	X	0	0	0	0	S	S	S	S
Unconditional jump to memory address (source)																
JZ	1	0	0	1	1	X	X	X	0	0	0	0	S	S	S	S
Jump to memory address (source) if zero flag is set																
JNZ	1	0	1	0	0	X	X	X	0	0	0	0	S	S	S	S
Jump to memory address (source) if zero flag is not set																
JG	1	0	1	0	1	X	X	X	0	0	0	0	S	S	S	S
Jump to memory address (source) if carry flag and zero flag are not set																
JLE	1	0	1	1	0	X	X	X	0	0	0	0	S	S	S	S
Jump to memory address (source) if carry flag or zero flag is set																
	1	0	1	1	1	X	X	X	X	X	X	X	X	X	X	X
Empty																
PUSH	1	1	0	0	0	X	X	X	X	X	X	X	S	S	S	S
Push the contents of the source register to the stack																
POP	1	1	0	0	1	X	X	X	D	D	D	D	X	X	X	X
Pop the last contents stored on the stack to the destination register																
PUSHF	1	1	0	1	0	X	X	X	X	X	X	X	X	X	X	X
Push the contents of the flag register to the stack																
POPF	1	1	0	1	1	X	X	X	X	X	X	X	X	X	X	X
Pop the last content stored on the stack to the flag register																
IN	1	1	1	0	0	X	X	X	D	D	D	D	X	X	X	X
Take input from GPIO pin(source) to destination (Not implemented)																
OUT	1	1	1	0	1	X	X	X	X	X	X	X	S	S	S	S
Output to GPIO pin (destination) from source (Not implemented)																
NOP	1	1	1	1	0	X	X	X	0	0	0	0	X	X	X	X
No operation																
HLT	1	1	1	1	1	0	0	0	X	X	X	X	X	X	X	X
Enter the stopped state.																

		Destination/Source			
SRC1/SRC2	D7/D3	D6/D2	D5/D1	D4/D0	DESCRIPTION
0-13	X	X	X	X	Register
14	1	1	1	0	Immediate Data
15	1	1	1	1	Memory

	Destination/Source			
DST	D10	D9	D8	DESCRIPTION
0-6	X	X	X	Register
7	1	1	1	Use only SRC1 and SRC2

3. RESULTS

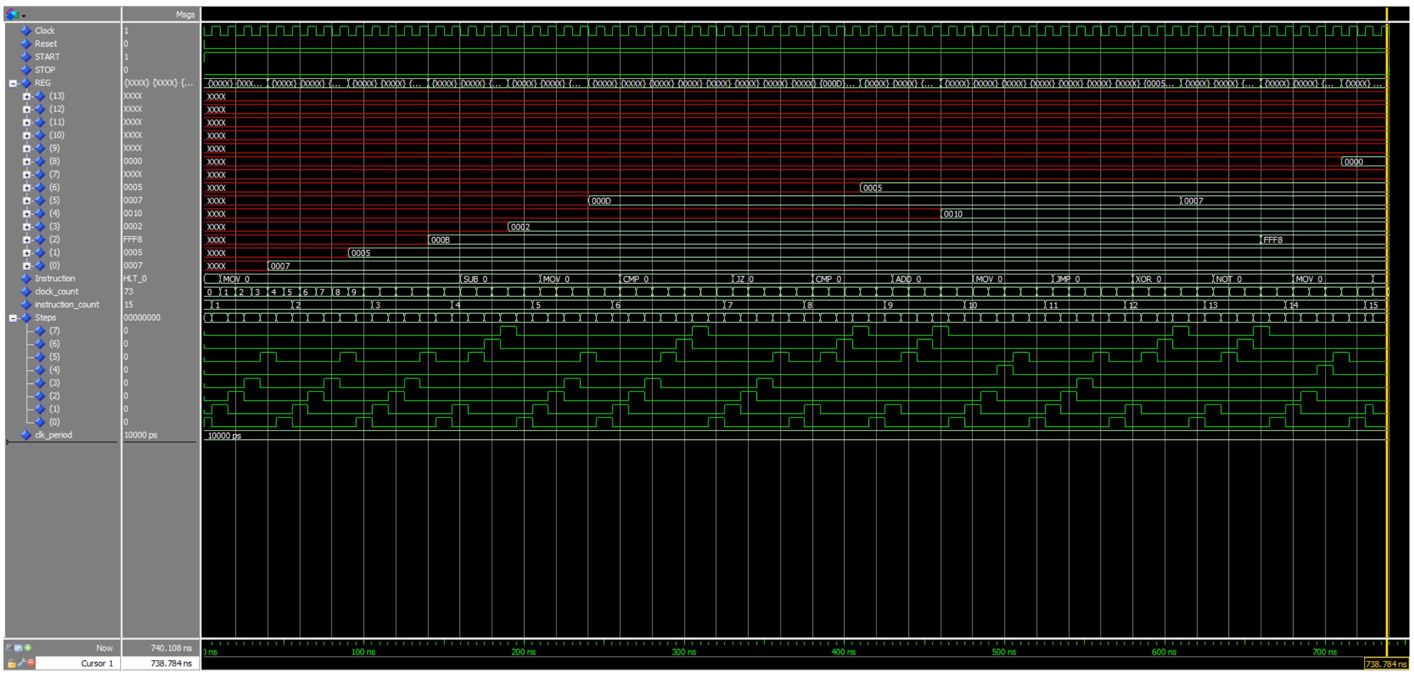
We have implemented all the instructions shown in the above table successfully. To easily test all the aspects of our CPU, a Python script has been written that outputs a VHDL test bench. An assembly code is given to this script as input. The code that is given in project details document is given below.

```
[CODE]
        MOV R0, 07h
        MOV R1, 05h
        MOV R2, 0Bh
        SUB R3, R1, R0
        MOV R5, 0Dh
        CMP R4, 0
        JZ jmp1
        NOP
jmp1:
        CMP R6, R4, R3
        ADD R4, R6, R2
        STORE R5, R4
        JMP jmp2
        NOP
        NOP
jmp2:
        XOR R5, R3, R6
        NOT R2, R5
        LOAD R8, R2
        HLT
```

Resulting test bench output is given below.

```
wait for fast_clk_period / 2;

Reset <= '0';

-----------------------------------------------------------------
TMP_ADR <= x"0000";       WAIT FOR fast_clk_period; TMP_ADR(0) <= 'Z';
-- CODE SEGMENT
-----------------------------------------------------------------

TMP_DATA <= x"800E";      WAIT FOR fast_clk_period;
TMP_DATA <= x"0007";      WAIT FOR fast_clk_period; -- MOV R0, 07h
TMP_DATA <= x"801E";      WAIT FOR fast_clk_period;
TMP_DATA <= x"0005";      WAIT FOR fast_clk_period; -- MOV R1, 05h
TMP_DATA <= x"802E";      WAIT FOR fast_clk_period;
TMP_DATA <= x"000B";      WAIT FOR fast_clk_period; -- MOV R2, 0Bh
TMP_DATA <= x"0B10";      WAIT FOR fast_clk_period; -- SUB R3, R1, R0
TMP_DATA <= x"805E";      WAIT FOR fast_clk_period;
TMP_DATA <= x"000D";      WAIT FOR fast_clk_period; -- MOV R5, 0Dh
TMP_DATA <= x"374E";      WAIT FOR fast_clk_period;
TMP_DATA <= x"0000";      WAIT FOR fast_clk_period; -- CMP R4, 0
TMP_DATA <= x"980E";      WAIT FOR fast_clk_period;
TMP_DATA <= x"000E";      WAIT FOR fast_clk_period; -- JZ jmp1
TMP_DATA <= x"F000";      WAIT FOR fast_clk_period; -- NOP
TMP_DATA <= x"3643";      WAIT FOR fast_clk_period; -- CMP R6, R4, R3
TMP_DATA <= x"0462";      WAIT FOR fast_clk_period; -- ADD R4, R6, R2
TMP_DATA <= x"8354";      WAIT FOR fast_clk_period; -- STORE R5, R4
TMP_DATA <= x"900E";      WAIT FOR fast_clk_period;
TMP_DATA <= x"0015";      WAIT FOR fast_clk_period; -- JMP jmp2
TMP_DATA <= x"F000";      WAIT FOR fast_clk_period; -- NOP
TMP_DATA <= x"F000";      WAIT FOR fast_clk_period; -- NOP
TMP_DATA <= x"2D36";      WAIT FOR fast_clk_period; -- XOR R5, R3, R6
TMP_DATA <= x"2250";      WAIT FOR fast_clk_period; -- NOT R2, R5
TMP_DATA <= x"8182";      WAIT FOR fast_clk_period; -- LOAD R8, R2
TMP_DATA <= x"F800";      WAIT FOR fast_clk_period; -- HLT


TMP_DATA <= "ZZZZZZZZZZZZZZZZ";
START <= '1';

wait for clk_period * 300;

STOP <= '1';
```

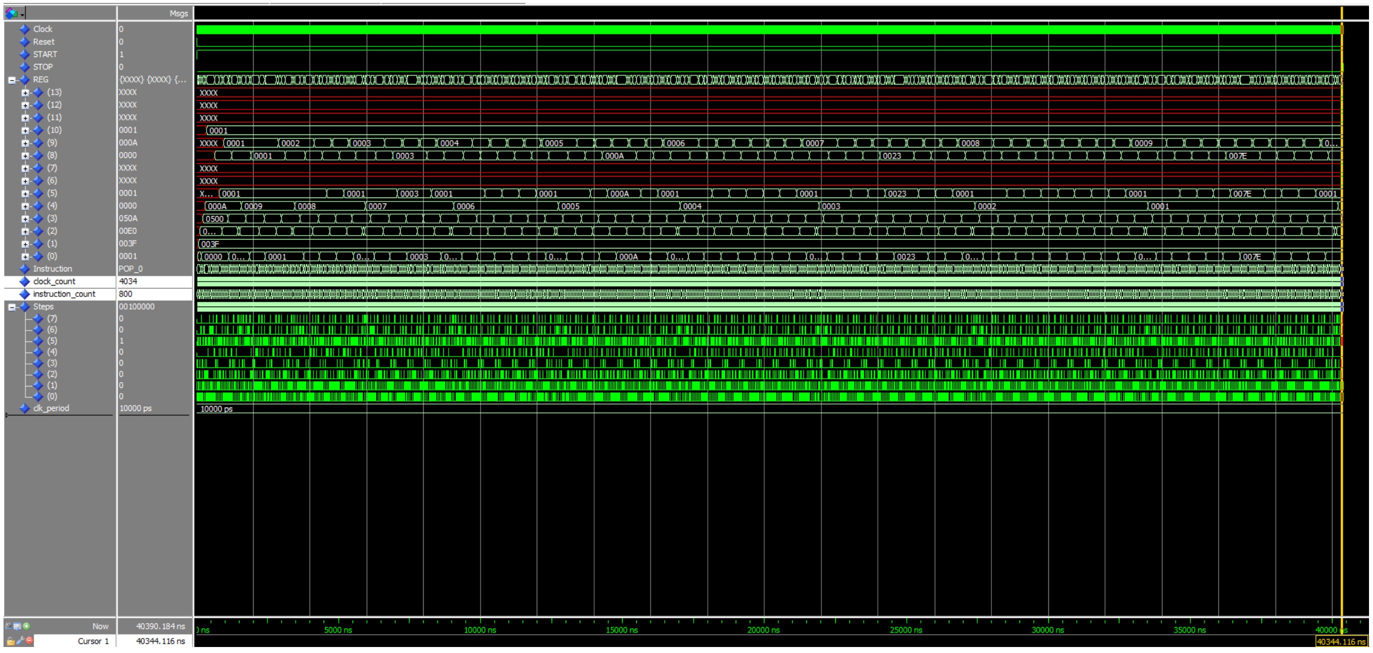
The resulting waveform of the simulation is shown below.



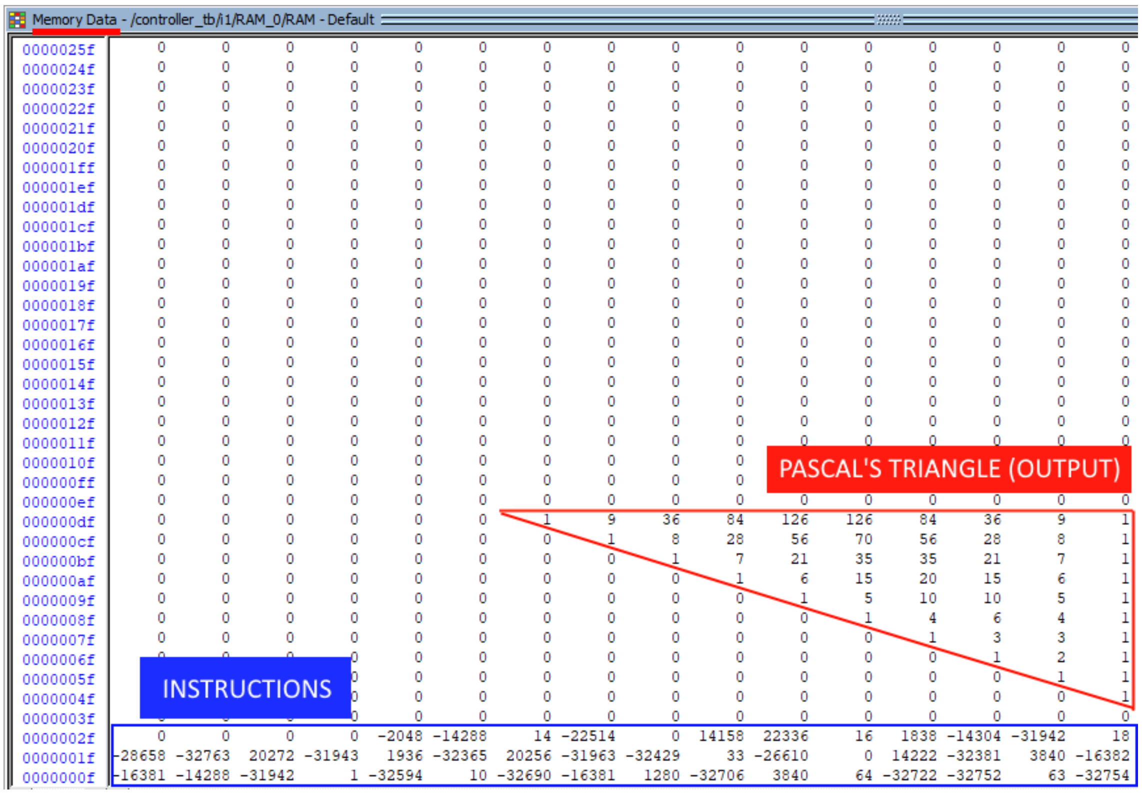
Another program has been executed on the CPU designed to show the capabilities and advantages of our design. A Pascal triangle is drawn on the memory with this program. The assembly code is given below.

```
[CODE]
process:
    MOV R0, 3FH
    MOV R1, R0
    MOV R2, 40H
    SUB R0, R0          ; Set AL as Previous Temp Value (Initial Zero)
    MOV R3, 500h        ; Allocate 500h location to store next line
                        ; data temporarily.
    PUSH R3                     ; Store First R3 location.
    MOV R4, 10                  ; Count of line.
    MOV R10, 1
    STORE R3, R10               ; Start temporary data with 1
process_line:
    POP R3                    ; Pop R3 to return first temp location.
    PUSH R3                     ; Store First R3 location again.
    PUSH R2                   ; Store R2 Initial offset location to
                              ; start with next block.
    SUB R0, R0                  ; Refresh Previous Temp Value as 0.

    current_line_loop:          ; Start write current line values.
        LOAD R8, R3
        CMP R8, 0
        JZ current_line_last    ; Break if current line has finished.
        LOAD R5, R3           ; Load value on R3 memory address
                              ; temporarily on R5
        STORE R2, R5            ; Set current pointer as R5
        INC R2                  ; Go to next pointer.
        LOAD R9, R3
        ADD R9, R0
        STORE R3, R9          ; Add to temp data, previous temp value
                              ; to prepare next line.
        INC R3
        MOV R0, R5
        JMP current_line_loop

    current_line_last:
        STORE R3, R10         ; Set temp data as 1 because of the
                              ;line's last values already 1.
        POP R2
        ADD R2, 10h             ; Pass to next line.
        DEC R4
        CMP R4, 0
        JG process_line     ; To process next line, jump process line.

POP R3
HLT
```

The results are shown below.



Memory view:


Memory Data - /controller_tb/i1/RAM_0/RAM - Default
PASCAL'S TRIANGLE (OUTPUT)
INSTRUCTIONS

0000025f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000024f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000023f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000022f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000021f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000020f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
000001ff	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
000001ef	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
000001df	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
000001cf	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
000001bf	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
000001af	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000019f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000018f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000017f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000016f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000015f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000014f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000013f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000012f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000011f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0000010f	0	0	0	0	0	0	0	0	0	0						
000000ff	0	0	0	0	0	0	0	0	0	0						
000000ef	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
000000df	0	0	0	0	0	0	1	9	36	84	126	126	84	36	9	1
000000cf	0	0	0	0	0	0	0	1	8	28	56	70	56	28	8	1
000000bf	0	0	0	0	0	0	0	0	1	7	21	35	35	21	7	1
000000af	0	0	0	0	0	0	0	0	0	1	6	15	20	15	6	1
0000009f	0	0	0	0	0	0	0	0	0	0	1	5	10	10	5	1
0000008f	0	0	0	0	0	0	0	0	0	0	0	1	4	6	4	1
0000007f	0	0	0	0	0	0	0	0	0	0	0	0	1	3	3	1
0000006f				0	0	0	0	0	0	0	0	0	0	1	2	1
0000005f				0	0	0	0	0	0	0	0	0	0	0	1	1
0000004f				0	0	0	0	0	0	0	0	0	0	0	0	1
0000003f				0	0	0	0	0	0	0	0	0	0	0	0	0
0000002f	0	0	0	0	-2048	-14288	14	-22514	0	14158	22336	16	1838	-14304	-31942	18
0000001f	-28658	-32763	20272	-31943	1936	-32365	20256	-31963	-32429	33	-26610	0	14222	-32381	3840	-16382
0000000f	-16381	-14288	-31942	1	-32594	10	-32690	-16381	1280	-32706	3840	64	-32722	-32752	63	-32754

4. COMMENTS AND CONCLUSION

Elements of the CPU are researched and explained in detail. Control Unit and its elements are designed to obtain a simple and efficient control signal structure. Fetch, decode and execute cycles are implemented according to the step structure given in the design section considering economic purposes. Connections between the CPU and the RAM is implemented using separate data buses and address buses for simplicity. Final design of the system is tested and the result for each function is explained in detail.

Example table given on the project template is implemented successfully and the final value of each parameter is shown for better readability.

Each student took part in each phase of the project equally.

$Selçuk\ Holep = CU, ALU, Registers\ and\ RAM$

$Burak\ Celal\ Kan = CU, ALU, Registers\ and\ RAM$

$Özgürcan\ Ediş = CU, ALU, Registers\ and\ RAM$

$Mehmet\ Bahadır\ Maktav = CU, ALU, Registers\ and\ RAM$

5. REFERENCES

1) But How Do It Know? The Basic Principles of Computers for Everyone, J. Clark Scott, 2009

2) Intel 4004, 8008 and 8086 CPU Datasheet

3) ELE336 Microprocessor Architecture and Programming Lecture Notes