Edinburgh Napier
University

**Programmable Logic Design**

**ELE10105**

# Microprocessor Design.

40292302

May 2020

# Table of Content

# Introduction.

Field-Programmable Gate Array (FPGA) chips are a category of Programmable Logic Devices allowing to code digital circuits. They are an alternative to custom integrated circuits allowing for reduced development hence a cost effective option. (Horowitz, Hill and Oxford University Press, 2015) This report details the steps taken to develop an “Application-Specific full-custom Integrated Circuit” (ASIC) “System-on-a-Chip” (SoC). The microprocessor created has only one purpose to realise a series calculation of calculations and return a value. Following a short overview of the processor architecture, each component ( or module) is described with annotated VHDL code (ALU, ROM, RAM, PC and IR). Then, test bench programmes are offered with a simulation time diagram for each component. The next section focuses on the Control Unit and its operations. Then, all components will be connected to test the operation of the microprocessor. Finally, a short discussion centered around the work undertaken in this project will be given.

# I. Overview Schematic and Diagram.

Figure I.1 describes the overall proposed architecture of the microprocessor with the different modules and data paths. The solution, presented in this report, respects the general architecture but differs on the manner by which the CPU controller interacts with the Arithmetic Logic Unit (ALU). The control method is based on "a simple microprocessor" presented by Zwoliński (2004). Instead of sending the "Arithmetic Op Code", the CPU controller will interpret the "Op Code" and trigger the corresponding digital signal (more details on this aspect in the following section). The microprocessor developed differs to Zwoliński's in that it does not use a single system bus to communicate between the components, thus respecting the communication flow present in Figure I.1. Some additional controls were also added to control the flow of data in the system.

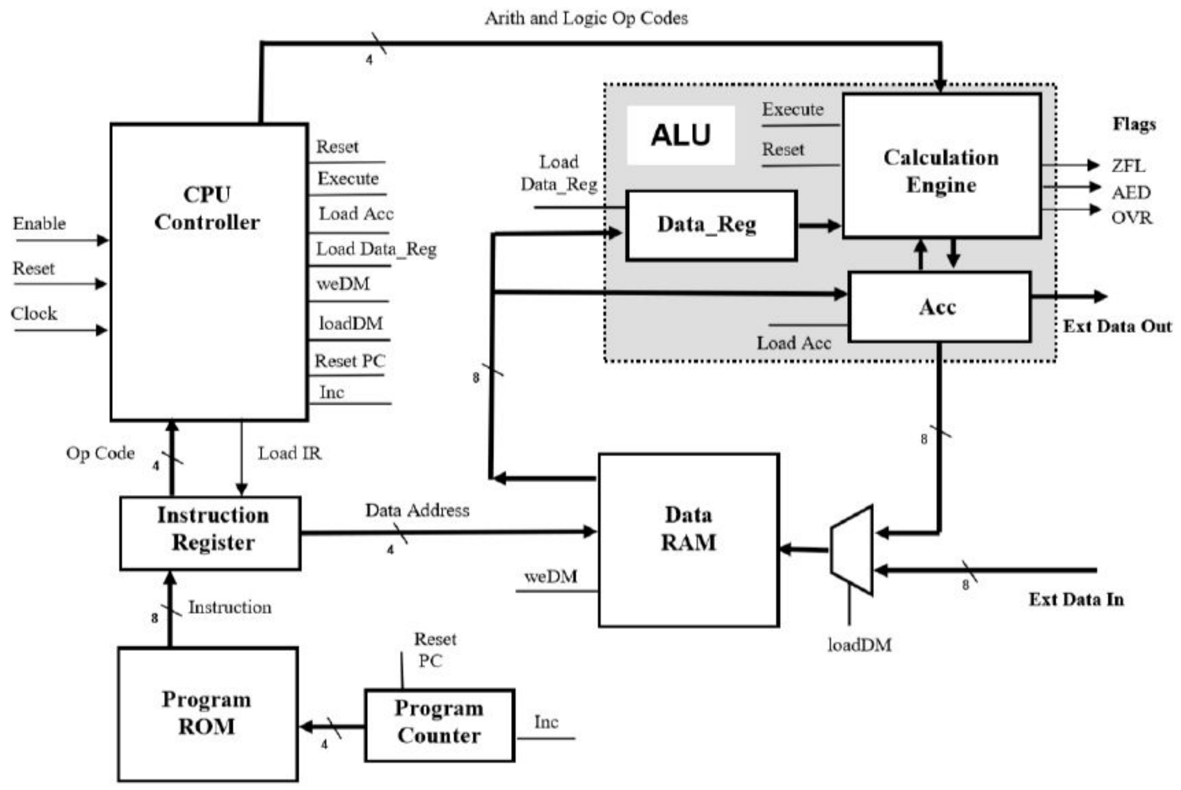


*Figure I.1 Proposed Microprocessor Architecture.*

# II. Description of Modules.

This section presents the different modules and components used in the realisation of the circuit. Each module presentation is accompanied by a VHDL annotated programme.

## A. Arithmetic Logic Unit (ALU).

The ALU is the first component developed in the microprocessor. Its role is to perform arithmetic and logic operations using values loaded to the accumulator and to the data register. The data is loaded, to the ALU, through a single for and then assigned to the right register. Once the operation is realised and needs to be stored the accumulator transfers the data to the output. It is worth mentioning the accumulator is a 9 bit register (for flags management - more on this later) and data is conditioned as a byte therefore the output only uses bits 7 to 0.

The digital controls of the ALU can be split into two categories: the controls managing the flow of data and arithmetic/logic controls.

**Data Controls:**
load_Acc - Loads a value from the input port to the Accumulator.
load_Data_reg - Loads a value from the input port to the Data Register.
ALU_Execute - Enables Arithmetic and Logic Controls.
Acc_bus - Sends the value from the Accumulator to the output port.
ALU_res - In the requirements this was controlled by an “Op code”. Since the Control Unit interprets the codes, it is more sensible to place it in the data control section.

**Arithmetic and Logic Controls:**

ALU_inc - Increments the value in the Accumulator by 1.

ALU_add - Adds the value in the data register to the value held in the Accumulator.

ALU_sub - Subtracts the value in the data register to the value held in the Accumulator.

ALU_mul - Multiplies the value in the Accumulator by the value in the Data Register.

ALU_shl - Shifts the value in the Accumulator left by the value in the Data Register.

ALU_shr - Shifts the value in the Accumulator right by the value in the Data Register.

ALU_nand - Performs a nand logic operation between the Accumulator value and the value in the Data Register.

ALU_xor - Performs an exclusive OR logic operation between the Accumulator value and the value in the Data Register.

ALU_set - Sets the Accumulator to “111111111”.

**Flags operations:**

Those are digital signals which trigger when a condition is met within the ALU.

ZFL - Zero flag triggers when the Accumulator is equal to 0.

AED - This flag triggers when the Accumulator is equal to the Data Register.

OVR - The overflow flag triggers when the Accumulator integer value goes beyond 255 meaning the 9th bit is triggered hence an overflow to a byte.

Additional outputs were added, for testing, to display the value held in the data register and to monitor the overflow integer value during the simulations.

**VHDL Code:**

```
-----------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
------------------------------------------------------------------------
entity ALU is
  Port (clock : in std_logic; -- System Clock.
        ALU_in : in std_logic_vector(7 downto 0); -- ALU Data input.
        load_Acc, load_Data_reg, ALU_Execute, Acc_bus: in std_logic;--
ALU data control code: ALU_in to Acc, ALU_in to Data_reg, ALU operation
to Acc, Acc to ALU_out
     ALU_res, ALU_inc, ALU_add, ALU_sub, ALU_mul, ALU_shl, ALU_shr,
ALU_nand, ALU_xor, ALU_set: in std_logic; -- ALU Operation requests.
        ALU_out : out std_logic_vector(7 downto 0); -- ALU output.
        data_reg_out : out std_logic_vector(8 downto 0); -- for testing
        output_ovr : out integer; -- for testing
        ZFL, OVR, AED : out std_logic); -- Flags: Zero, Overload, Acc =
Data_Reg.
end ALU;
------------------------------------------------------------------------
architecture Behavioral of ALU is
signal Acc: unsigned(8 downto 0); -- Accumulator register.
signal output_Over : integer; -- Acc conversion to integer for Overflow.
signal Data_reg : unsigned(8 downto 0); -- Data register input.
------------Arithmetic/Logic Operations--------------------------------
begin
   Data_reg <= unsigned('0' & ALU_in) when load_Data_reg ='1'; --data
input to Data register
 process (clock) is
    begin
     if ALU_res ='1' then
        Acc <= "000000000"; -- RES 0
     elsif rising_edge(clock) then
           if load_Acc = '1' then
          Acc <= unsigned('0' & ALU_in); --data input to Accumulator
           elsif ALU_Execute = '1' then
               if ALU_inc ='1' then
                   Acc <= (Acc + 1); -- INC 1
               elsif ALU_add ='1' then
                   Acc <= (Acc + Data_reg); -- ADD 2
               elsif ALU_sub ='1' then
                   Acc <= (Acc - Data_reg); -- SUB 3
               elsif ALU_mul ='1' then
                 Acc <= resize(Acc * Data_reg,9); -- MUL 4
```

```
                elsif ALU_shl ='1' then
                  Acc <= (Acc sll to_integer(Data_reg)); -- SHL 5
                elsif ALU_shr ='1' then
                Acc <= (Acc srl to_integer(Data_reg)); -- SHR 6
                elsif ALU_nand ='1' then
                    Acc <= Acc NAND Data_reg; -- NAND 7
                elsif ALU_xor ='1' then
                    Acc <= Acc XOR Data_reg; -- XOR 8
                elsif ALU_set ='1' then
                    Acc <= "111111111"; -- SET 9
                end if;
               end if;
           end if;
      end process;
ALU_out <= std_logic_vector(Acc(7 downto 0)) when ACC_bus = '1'; -- ALU
output
-------------Flags Operations-----------------------------------------
ZFL <= '1' when Acc = "00000000" else '0'; -- Zero in Acc flag.
AED <= '1' when Acc = Data_reg else '0'; -- Acc = Data_reg flag.
output_Over <= to_integer(Acc);
output_ovr <= output_Over; -- for testing
data_reg_out <= std_logic_vector(Data_reg); -- for testing
OVR <= '1' when output_Over > 255 -- Overload flag
    else '0';
end architecture Behavioral;
----------------------------------------------------------------------
```

## B. Read Only Memory (ROM).

The ROM is a fixed 16x8 memory device controlled by the increments of the programme controller. Each increment triggers a memory location of the ROM (8-bit). Its main purpose is to store the “Op codes” (4-bit) and RAM addresses (4-bit). In the example provided below the “Op codes” are matched with the same 4-bit RAM address as shown in Table II.B. This allows for an easy testing of the ROM functionalities. Although ROM and RAM addresses are likely to still be matching, the “Op Codes” may not be stored in this order in the final installment of the device.

*Table II.B ROM Address Content*

| **ROM for Test** | | | **ROM for Debugged Programme** | | |
|---|---|---|---|---|---|
| **ROM address** | **Op Code** | **RAM address** | **ROM address** | **Op Code** | **RAM address** |
| “0000” | “0000” - RES | “0000” | “0000” | “1101” - STE | “0000” |
| “0001” | “0001” - INC | “0001” | “0001” | “1011” - LDD | “0001” |
| “0010” | “0010” - ADD | “0010” | “0010” | “0010” - ADD | “0010” |
| “0011” | “0011” - SUB | “0011” | “0011” | “0011” - SUB | “0011” |
| “0100” | “0100” - MUL | “0100” | “0100” | “1111”-Not used | “0100” |
| “0101” | “0101” - SHL | “0101” | “0101” | “0001” - INC | “0101” |
| “0110” | “0110” - SHR | “0110” | “0110” | “1100” - STA | “0110” |
| “0111” | “0111” - AND | “0111” | “0111” | “1010” - LDA | “0110” |
| “1000” | “1000” - XOR | “1000” | “1000” | “0110” - SHR | “0111” |
| “1001” | “1001” - | “1001” | “1001” | “1100” - | “1000” |

|  | SET |  |  | STA |  |
|---|---|---|---|---|---|
| “1010” | “1010” - LDA | “1010” | “1010” | “0101” - SHL | “1001” |
| “1011” | 1011” - LDD | 1011” | “1011” | “1000” - XOR | “1010” |
| “1100” | “1100” - STA | “1100” | “1100” | MUL | 1011” |
| “1101” | “1101” - STE | “1101” | “1101” | empty | “1100” |
| “1110” | “1110” - INZ | “1110” | “1110” | empty | “1101” |
| “1111” | “1111” - JMP | “1111” | “1111” | empty | “1110” |

**VHDL Code:**

```
------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_unsigned.ALL;
use IEEE.numeric_std.all;
------------------------------------------------------------------------
entity ROM is
    port (ROM_in : in std_logic_vector (3 downto 0); -- input 4 bit
sequence from Programme Controller.
          ROM_out: out std_logic_vector (7 downto 0)); -- 8 bit output
(Instruction/RAM address).
end entity ROM;
```

```
------------------------------------------------------------------------
architecture Behavioral of ROM is
      type rom_array is array (0 to 15) of std_logic_vector (7 downto
0);
         constant rom : rom_array := ( --Instruction/RAM address 4 bit
each
                   "00000000",
                   "00010001",
                   "00100010",
                   "00110011",
                   "01000100",
                   "01010101",
                   "01100110",
                   "01110111",
                   "10001000",
                   "10011001",
                   "10101010",
                   "10111011",
                   "11001100",
                   "11011101",
                   "11101110",
                   "11111111");
begin
    process (ROM_in)
       begin
           ROM_out <= rom(to_integer(unsigned(ROM_in))); -- output byte
from ROM location
           end process;
end architecture Behavioral;
------------------------------------------------------------------------
```

## C. Random-Access Memory (RAM).

The RAM is a write and read 16x8 memory device used by the control unit to store temporary data to be used in the calculations. The device has two inputs, through which the targeted memory address (4-bit) and data (8-bit) are loaded to their respective registers, and one output to send data (8-bit) to the system.

**RAM Controls:**

MDR_bus - Send the content of the data register to the output

load_MDR - Load data from the input to the data register.

load_MAR - Load address from the input to the address register.

CS - Activate read/write of the RAM.

WeDM - Write Enable.

**Ram Registers:**

mdr - 8-bit data register.

mar - 4bit address register.

**VHDL Code:**

```
------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
------------------------------------------------------------------------
entity RAM is
    port (clock, reset : in std_logic; -- system control.
            MDR_bus, load_MDR, load_MAR, CS, -- Send mdr to RAM_out,
load RAM_in to mdr, load RAM_in to mar, activate RAM.
            WeDM : in std_logic; -- Write enable.
            RAM_in_addr : in std_logic_vector (3 downto 0); -- RAM
address input.
            RAM_in_data : in std_logic_vector (7 downto 0); -- RAM data
input.
            RAM_out : out std_logic_vector (7 downto 0));-- RAM output.
end entity RAM;
```

```
----------------------------------------------------------------------
architecture Behavioral of RAM is
    signal mdr : std_logic_vector(7 downto 0); -- 8-bit data register.
    signal mar : unsigned(3 downto 0); -- 4-bit address register.

begin
    RAM_out <= mdr when MDR_bus = '1' else (others => 'Z'); --MDR to bus
activated send data register to RAM_out
    process (clock, reset) is
        type ram_array is array (0 to 15) of std_logic_vector(7 downto
0); -- 16x8 Memory
    variable mem: ram_array;
    begin
        if reset = '1' then -- When reset 0 both registers
            mdr <= (others => '0');
            mar <= (others => '0');
        elsif rising_edge(clock) then
            if load_MAR = '1' then -- RAM_in to address register
                mar <= unsigned(RAM_in_addr);
            elsif load_MDR = '1' then -- RAM_in to data register
                mdr <= RAM_in_data;
            elsif CS = '1' then -- If ram activated
                if WeDM = '1' then -- write is enable ->write data
register to memory location indicated by address register.
                    mem(to_integer(mar)) := mdr;
                else
                    mdr <= mem(to_integer(mar));  -- Read -> assign data
from memory location indicated by address register to data register.
                end if;
           end if;
        end if;
    end process;
end Behavioral;
----------------------------------------------------------------------
```

## D. Programme Controller (PC).

The PC is controlled by the Controller Unit. It increments its counter when requested and will output a 4-bit sequence sent to trigger ROM addresses. Its count can be set to 0 by sending a “Reset_PC” digital signal.

**VHDL Code:**

```
-------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
-------------------------------------------------------------------------
entity PC is
    Port (clock,reset_PC : in Std_logic; -- system control
           Inc : in Std_logic; -- Increment the PC
           ROM_addr : out Std_logic_vector(3 downto 0)); -- PC 4-bit
count output
end PC;
-------------------------------------------------------------------------
architecture Behavioral of PC is
signal PC_count: unsigned(3 downto 0); -- 4-bit PC counter
begin
     process (clock, reset_PC) is
        begin
            if reset_PC = '1' then -- reset the PC to "0000"
                PC_count <= (others => '0');

             elsif rising_edge(clock) then
                if Inc = '1' then -- Increment trigger add "0001" to
current count
                PC_count <= PC_count + "0001";
                end if;
            end if;
         end process;
ROM_addr <= Std_logic_vector(PC_count); -- output 4-bit sequence
corresponding to ROM address.
end Behavioral;
-------------------------------------------------------------------------
```

## E. Instruction Register (IR)

The IR is an information splitter. It receives data from the ROM (8-bit) containing an "Op Code" and RAM address. It will split those two information and send them to the relevant output. The "Op Code" will be sent to the Controller Unit at every Load_IR request. The RAM address is outputted at every turn of the clock.

**VHDL Code:**

```
------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
------------------------------------------------------------------------
entity IR is
    Port (clock, reset: in Std_logic;
          Load_IR : in Std_logic;
          Data_ROM : in Std_logic_vector(7 downto 0); -- 8-bit Input
data from the ROM
          Operand : out Std_logic_vector(3 downto 0); -- 4-bit Op Code
output
          RAM_addr : out Std_logic_vector(3 downto 0));-- 4-bit RAM
address.
end IR;
------------------------------------------------------------------------
architecture Behavioral of IR is
begin
    process (clock, reset) is
        begin
            if reset = '1' then -- if reset 0 the outputs
                Operand <= (others => '0');
                RAM_addr <= (others => '0');
            elsif rising_edge(clock) then
                if load_IR = '1' then -- if load the opcode output bits
7 to 4 from the input
                Operand <= Data_ROM(7 downto 4);
                end if;
                RAM_addr <= Data_ROM(3 downto 0); -- always output the
bits 3 to 0 from the input
            end if;
        end process;
end Behavioral;
------------------------------------------------------------------------
```

## F. Data Mux (DM).

The DM controls the data input to the RAM. If loadDM is activated the Mux will load the data from the system input. If LoadDM is inactive the data will flow from the Accumulator to the RAM. If the calculations are finished the data mux will send the value in the Accumulator to output F.

**VHDL Code:**

```
-------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-------------------------------------------------------------------------
entity Data_in_bus is
    Port (LoadDM, clock, ACC_outSys : in Std_logic; -- Load data control
          Ext_Data_in : in Std_logic_vector (7 downto 0); -- Input
exterior to system
          Acc_in : in Std_logic_vector (7 downto 0); -- Input from the
Accumulator
          DM_out : out Std_logic_vector (7 downto 0); -- Mux Output to
RAM.
          F : out Std_logic_vector (7 downto 0)); -- Mux Output to
System output.
end Data_in_bus;
-------------------------------------------------------------------------
architecture Behavioral of Data_in_bus is
begin
    process (clock) is
        begin
            if rising_edge(clock)then
                if LoadDM ='1' then -- Load the data to the system
                  DM_out <= Ext_Data_in;
                elsif ACC_outSys ='1' then -- If Calculations finished
sent to system output
                  F <= Acc_in;  -- Load the Accumulator
                else
                  DM_out <= Acc_in;  -- Load the Accumulator
                end if;
             end if;
    end process;
end Behavioral;
-------------------------------------------------------------------------
```

## III. Simulation of Modules.

This section will present the test benches codes and comment on the simulation output for each of the main modules. The PC and IR were tested in combination with the ROM to verify the programme control operation from increment to “Op code” - RAM outputs.

### A. Arithmetic Logic Unit (ALU).

**Operating Principle:** The binary value for “3” is put at the ALU input. This value is then loaded to the Data register and to the Accumulator. Calculations are then triggered one by one using their corresponding digital signal.

**Test Bench:**

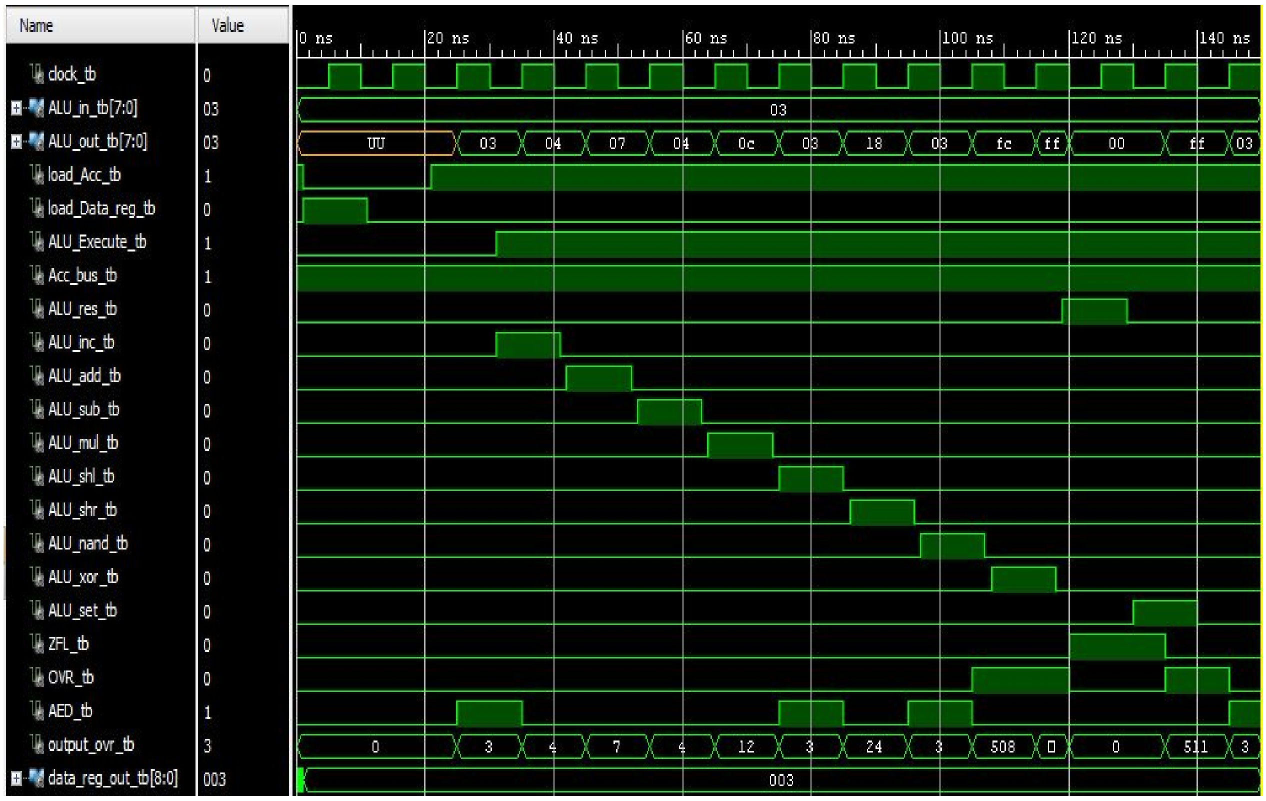
```
------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
------------------------------------------------------------------------
entity ALU_tb is
end ALU_tb;
------------------------------------------------------------------------
architecture testbench of ALU_tb is
------ DUT Declaration:------
    component ALU is
  Port (clock : in std_logic; -- System Clock.
            ALU_in : in std_logic_vector(7 downto 0); -- ALU Data input.
            load_Acc, load_Data_reg, ALU_Execute, Acc_bus: in
std_logic;-- ALU data control code: sysbus to Acc, ALU operation to Acc,
Acc to sysbus
            ALU_res, ALU_inc, ALU_add, ALU_sub, ALU_mul, ALU_shl,
ALU_shr, ALU_nand, ALU_xor, ALU_set: in std_logic; -- ALU Operation
requests: Load_ Data_Reg, Load_Acc, Execute, Reset
            ALU_out : out std_logic_vector(7 downto 0); -- Data register
input.
            output_ovr : out integer; -- data overflow output (for
testing).
            data_reg_out : out std_logic_vector(8 downto 0); -- for
testing
```

```
            ZFL, OVR, AED : out std_logic); -- Flags: Zero, Overload,
Acc = Data_Reg.
      end component;

    ------Signal Declaration:------

    signal clock_tb : std_logic :='0'; -- System Clock.
    signal ALU_in_tb : std_logic_vector(7 downto 0):="00000011"; --
System Bus from/to system.
    signal ALU_out_tb : std_logic_vector(7 downto 0);
    --signal Data_reg_tb : unsigned(7 downto 0);--  Data reg input.
    signal load_Acc_tb, load_Data_reg_tb, ALU_Execute_tb, Acc_bus_tb:
std_logic:='0';-- ALU data control code: sysbus to Acc, ALU operation to
Acc, Acc to sysbus
    signal ALU_res_tb, ALU_inc_tb, ALU_add_tb, ALU_sub_tb, ALU_mul_tb,
ALU_shl_tb, ALU_shr_tb, ALU_nand_tb, ALU_xor_tb, ALU_set_tb:
std_logic:='0'; -- ALU Operation requests
    signal ZFL_tb, OVR_tb, AED_tb : std_logic; -- Flags: Zero, Overload,
Acc = Data_Reg.
    signal output_ovr_tb: integer;
    signal data_reg_out_tb: std_logic_vector(8 downto 0);
begin
    ------ DUT Instantiation:------

        dut: ALU Port map (clock => clock_tb, ALU_out => ALU_out_tb,
ALU_in => ALU_in_tb, load_Data_reg => load_Data_reg_tb, load_Acc =>
load_Acc_tb, ALU_Execute => ALU_Execute_tb, Acc_bus => Acc_bus_tb,
        ALU_res => ALU_res_tb, ALU_inc => ALU_inc_tb, ALU_add =>
ALU_add_tb, ALU_sub => ALU_sub_tb, ALU_mul => ALU_mul_tb, ALU_shl =>
ALU_shl_tb, ALU_shr => ALU_shr_tb,
        ALU_nand => ALU_nand_tb, ALU_xor => ALU_xor_tb, ALU_set =>
ALU_set_tb, ZFL => ZFL_tb, OVR => OVR_tb, AED => AED_tb, output_ovr=>
output_ovr_tb, data_reg_out => data_reg_out_tb);

    ------ Stimuli Generation:------
        clock_tb <= not clock_tb after 5ns;

    stim_proc: process -- After reset switch input every 10ns
        begin
           ACC_bus_tb <= '1';
           load_Acc_tb <= '1';
            wait for 1 ns;
           load_Acc_tb <= '0';
           load_Data_reg_tb <= '1';
             wait for 10 ns;
```

```
 load_Data_reg_tb <= '0';
   wait for 10 ns;
 load_Acc_tb <= '1';
   wait for 10 ns;
  ---Start Calculations-----
 ALU_Execute_tb <= '1';
 ALU_inc_tb <= '1';
   wait for 10 ns;
 ALU_inc_tb <= '0';
   wait for 1 ns;
  ALU_add_tb <= '1';
   wait for 10 ns;
 ALU_add_tb <= '0';
    wait for 1 ns;
 ALU_sub_tb <= '1';
   wait for 10 ns;
 ALU_sub_tb <= '0';
   wait for 1 ns;
 ALU_mul_tb <= '1';
   wait for 10 ns;
 ALU_mul_tb <= '0';
    wait for 1 ns;
 ALU_shl_tb <= '1';
   wait for 10 ns;
 ALU_shl_tb <= '0';
   wait for 1 ns;
 ALU_shr_tb <= '1';
   wait for 10 ns;
 ALU_shr_tb <= '0';
   wait for 1 ns;
 ALU_nand_tb <= '1';
   wait for 10 ns;
 ALU_nand_tb <= '0';
   wait for 1 ns;
 ALU_xor_tb <= '1';
   wait for 10 ns;
 ALU_xor_tb <= '0';
  wait for 1 ns;
 ALU_res_tb <= '1'; -- reset on
  wait for 10 ns;
 ALU_res_tb <= '0'; -- reset off
  wait for 1 ns;
ALU_set_tb <= '1';
 wait for 10 ns;
ALU_set_tb <= '0';
```

```
                wait for 1 ns;
                   wait;
             end process;
end testbench;
------------------------------------------------------------------------
```

**Simulation Output:**



| Name | Value |
|---|---|
| clock_tb | 0 |
| ALU_in_tb[7:0] | 03 |
| ALU_out_tb[7:0] | 03 |
| load_Acc_tb | 1 |
| load_Data_reg_tb | 0 |
| ALU_Execute_tb | 1 |
| Acc_bus_tb | 1 |
| ALU_res_tb | 0 |
| ALU_inc_tb | 0 |
| ALU_add_tb | 0 |
| ALU_sub_tb | 0 |
| ALU_mul_tb | 0 |
| ALU_shl_tb | 0 |
| ALU_shr_tb | 0 |
| ALU_nand_tb | 0 |
| ALU_xor_tb | 0 |
| ALU_set_tb | 0 |
| ZFL_tb | 0 |
| OVR_tb | 0 |
| AED_tb | 1 |
| output_ovr_tb | 3 |
| data_reg_out_tb[8:0] | 003 |

**Comment:** The simulation output demonstrates and follows the operation of the test bench. “Data_reg_out_tb” monitors the value loaded: “3” and output_ovr_tb displays the value in the accumulator as an integer.

No values are displayed in the Accumulator until “load_acc” is triggered and operations only happen once “ALU_execute” is triggered. The “ALU_res” command resets the Accumulator to zero. Each arithmetic operation performs as expected. Appropriate flags are triggered when their conditions are met.

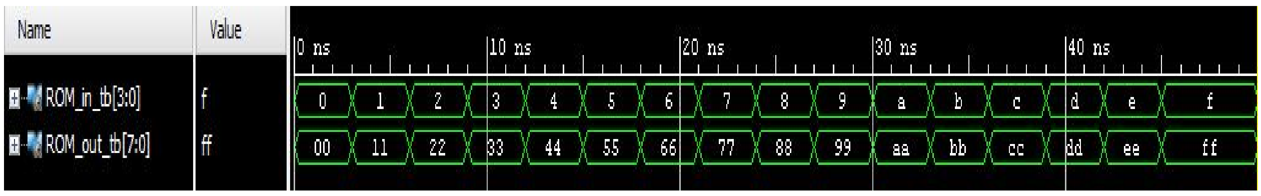
## B. Read Only Memory (ROM).

**Operating Principle:** The test bench loads a 4-bit address sequence from 0 to 15 hence returning the values stored in the corresponding memory locations.

**Test Bench:**

```
------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
------------------------------------------------------------------------
entity ROM_tb is
    end ROM_tb;
------------------------------------------------------------------------
architecture testbench of ROM_tb is
component ROM
     port (ROM_in : in std_logic_vector (3 downto 0);
          ROM_out: out std_logic_vector (7 downto 0));
    end component;
    ------Signal Declaration:------
   signal ROM_in_tb : std_logic_vector (3 downto 0);
   signal ROM_out_tb : std_logic_vector (7 downto 0);
begin
      ------ DUT Instantiation:------
    dut: ROM Port map(ROM_in => ROM_in_tb, ROM_out => ROM_out_tb);
     ----- Stimuli Generation:------
   ROM_in_tb <=        "0000",
                       "0001" AFTER 3ns,
                       "0010" AFTER 6ns,
                       "0011" AFTER 9ns,
                       "0100" AFTER 12ns,
                       "0101" AFTER 15ns,
                       "0110" AFTER 18ns,
                       "0111" AFTER 21ns,
                       "1000" AFTER 24ns,
                       "1001" AFTER 27ns,
                       "1010" AFTER 30ns,
                       "1011" AFTER 33ns,
                       "1100" AFTER 36ns,
                       "1101" AFTER 39ns,
                       "1110" AFTER 42ns,
                       "1111" AFTER 45ns;
end testbench;
```

**Simulation Output:**


Name
Value
ROM_in_tb[3:0]
ROM_out_tb[7:0]
f
ff
0 ns
10 ns
20 ns
30 ns
40 ns
0 1 2 3 4 5 6 7 8 9 a b c d e f
00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff

**Comment:** The simulation demonstrates the data stored being outputted according to its location with the two 4-bit "Op code" and RAM codes.

## C. Random-Access Memory (RAM).

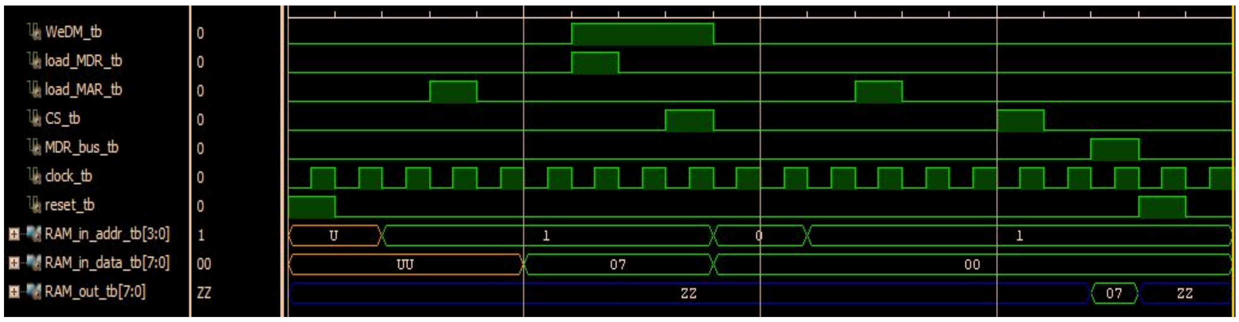
**Operating Principle:** The RAM is first triggered to load the address location "1" and then to store the number "7" from the RAM input . The address is then loaded again to read from the same location.

**Test Bench:**

```
----------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_unsigned.ALL;
use IEEE.numeric_std.all;
----------------------------------------------------------------------
entity Ram_tb is
end Ram_tb;
----------------------------------------------------------------------
------ DUT Declaration:------
architecture testbench OF Ram_tb is

    component RAM
     port (clock, reset : in std_logic;
           MDR_bus, load_MDR, load_MAR, CS,
           WeDM : in std_logic;
           RAM_in_data : in std_logic_vector (7 downto 0);
           RAM_in_addr : in std_logic_vector (3 downto 0);
           RAM_out : out std_logic_vector (7 downto 0));
    end component;

    ------Signal Declaration:------
   signal WeDM_tb, load_MDR_tb, load_MAR_tb, CS_tb: std_logic:= '0';
   signal MDR_bus_tb : std_logic:= '0';
   signal clock_tb, reset_tb : std_logic := '0';
   signal RAM_in_addr_tb : std_logic_vector(3 downto 0);
   signal RAM_in_data_tb : std_logic_vector(7 downto 0);
   signal RAM_out_tb : std_logic_vector(7 downto 0);

begin

     ------ DUT Instantiation:------
    dut: RAM Port map(clock => clock_tb, reset => reset_tb, WeDM =>
WeDM_tb, MDR_bus => MDR_bus_tb,
    load_MDR => load_MDR_tb, load_MAR => load_MAR_tb, CS => CS_tb,
RAM_out=> RAM_out_tb, RAM_in_addr=> RAM_in_addr_tb, RAM_in_data=>
RAM_in_data_tb);

     ----- Stimuli Generation:------
   clock_tb <= NOT clock_tb AFTER 5ns;
   stim_proc: process -- After reset switch input every 10ns
begin
-------------------- write data to memory location 1
    reset_tb <='1';
        wait for 10ns;
    reset_tb <='0';
        wait for 10ns;
    RAM_in_addr_tb <="0001"; -- set address index in bus
        wait for 10ns;
    load_MAR_tb <= '1'; -- load address to MAR
        wait for 10ns;
    load_MAR_tb <= '0';
        wait for 10ns;
    RAM_in_data_tb <= x"07"; -- set data to write in bus
        wait for 10ns;
    WeDM_tb  <= '1';
    load_MDR_tb <= '1'; -- load data to MDR
        wait for 10ns;
    load_MDR_tb <= '0';
        wait for 10ns;
    CS_tb <= '1'; -- write MDR to memory(mar) location.
        wait for 10ns;
    CS_tb <= '0';
    WeDM_tb  <= '0';
    RAM_in_data_tb <="00000000"; -- reset the bus
    RAM_in_addr_tb <="0000"; -- reset address index in bus
        wait for 20ns;

-------------------- read data from memory location 1
    RAM_in_addr_tb <="0001"; -- set address index in bus
```

```
        wait for 10ns;
    load_MAR_tb <= '1'; -- load address to MAR
        wait for 10ns;
    load_MAR_tb <= '0';
    RAM_in_data_tb <="00000000"; -- reset bus to wait for read data
        wait for 10ns;
    WeDM_tb <= '0'; -- set RAM to read
        wait for 10ns;
    CS_tb <= '1'; -- read from memory(mar) location to mdr.
        wait for 10ns;
    CS_tb <= '0'; -- Close RAM
        wait for 10ns;
    MDR_bus_tb <= '1'; -- send mdr to sysbus
        wait for 10ns;
    MDR_bus_tb <= '0'; -- close mdr to sysbus

    end process;
end testbench;
-----------------------------------------------------------------------
```

**Simulation Output:**



**Comment:** The RAM performs as expected. It can store to and load data from a defined address.

## D. Programme Controller, ROM and Instruction Register.

**Operating Principle:** For this test the ALU was not integrated because the "Op codes" required to be interpreted before triggering the required digital signal. The test bench increments the PC counter to trigger readings from the ROM which are then split by the Instruction Register.

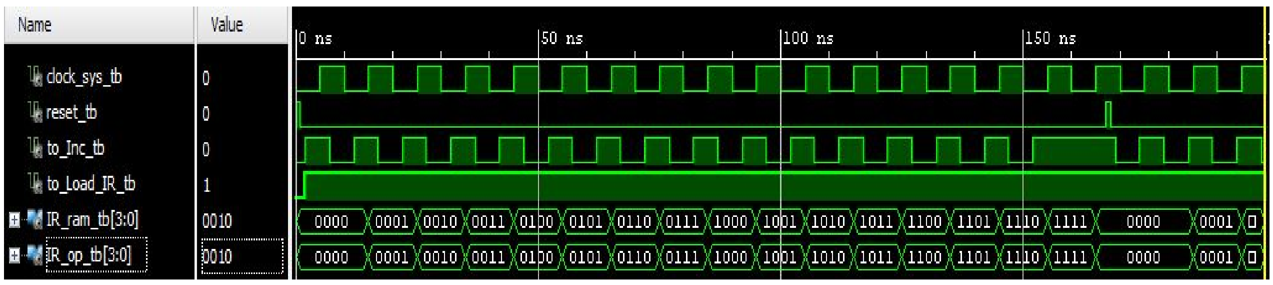
**Data path - VHDL Code:**

```
--------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--------------------------------------------------------
entity Data_path is
    Port (clock_sys, reset : in Std_logic; -- system control.
          to_Inc, to_Load_IR : in Std_logic; -- External Inputs.
          IR_ram : out std_logic_vector (3 downto 0); -- Output RAM
address.
          IR_op : out std_logic_vector (3 downto 0));-- Output Op Code.
end Data_path;
--------------------------------------------------------
architecture Behavioral of Data_path is
--------------Connecting Signals------------------------
signal PC_to_ROM : Std_logic_vector(3 downto 0);
signal ROM_to_IR : Std_logic_vector(7 downto 0);
--------------Connecting Modules------------------------
begin
   PC: entity work.PC port map(clock => clock_sys, reset_PC => reset,
Inc => to_Inc, ROM_addr => PC_to_ROM); -- PC connected to external
inputs and to the ROM
   ROM: entity work.ROM port map(ROM_in =>PC_to_ROM, ROM_out =>
ROM_to_IR);-- Output of the PC to the ROM and output connect to the
Instruction Register.
   -- IR connected to the output of ROM and splitting between Opcode and
RAM
   IR: entity work.IR port map(clock => clock_sys, reset => reset,
Load_IR => to_Load_IR, Data_ROM => ROM_to_IR, Operand => IR_op, RAM_addr
=> IR_ram);
   --RAM: entity work.ROM port map(clock => clock, reset_PC => reset,
MDR_bus, load_MDR, load_MAR, CS, R_NW, RAM_in, RAM_out);

end Behavioral;
--------------------------------------------------------
```

**Test Bench:**

```
--------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--------------------------------------------------------
entity Data_path_tb is
--  Port ( );
end Data_path_tb;
--------------------------------------------------------
architecture testbench of Data_path_tb is
component Data_path is
  Port (clock_sys, reset : in Std_logic;
         to_Inc, to_Load_IR : in Std_logic;
         IR_ram : out std_logic_vector (3 downto 0);
         IR_op : out std_logic_vector (3 downto 0));
      end component;
------Signal Declaration:------
          signal clock_sys_tb, reset_tb : std_logic :='0'; -- Clock.
          signal to_Inc_tb, to_Load_IR_tb : std_logic :='0';
          signal IR_ram_tb :  std_logic_vector (3 downto 0);
          signal IR_op_tb :  std_logic_vector (3 downto 0);
begin
      ------ DUT Instantiation:------
  dut: Data_path Port map(clock_sys => clock_sys_tb, reset => reset_tb,
to_Inc => to_Inc_tb, IR_ram => IR_ram_tb, IR_op => IR_op_tb, to_Load_IR
=> to_Load_IR_tb);
           clock_sys_tb <= not clock_sys_tb after 5 ns;
  stim_proc: process
    begin
        reset_tb <= '1';
            wait for 1ns;
        reset_tb <= '0';
            wait for 1 ns;
        to_Load_IR_tb <= '1';
         for i in 0 to 30 loop
            to_Inc_tb <= NOT to_Inc_tb;
               wait for 5 ns;
           -- Inc_tb <= '0';
           end loop;
           wait for 10ns;
      end process;

end architecture testbench;
```

**Simulation Output:**



**Comment:** For every increment of the PC the ROM outputs the data from corresponding memory addresses which are then split in 4-bit sequences by the Instruction register. When the PC reset is triggered the addressing restarts from zero.

# IV. Description of the Control Unit (CU) and Operation.

## A. Operation Principle.

The CU is the component realising the overall control of the system by activating the different modules sequentially. The sequence is programmed to take in 5 bytes (V, W, X, Y and Z) of data, realise a series of calculations to return the output:

$F = ([(X - Y) * 2] \text{ XOR } [(W + Z) / 2]) * V$

**Inputs:**

Enable - Allows the CU to operate (a sort of ON/OFF switch).

Clock - The system clock which allows to synchronise the operations of the different components.

Reset - Allows to restart and reset the system.

Op Code - 4-bit sequence from the ROM memory which will determine the instruction to be realised.

**Outputs:** The outputs of the CU correspond to the control inputs of the other components such as the PC, ALU, RAM and Data Mux as well as controlling the output of the system.

**Information specific to proposed programme prior to debugging:**

The information presented below was true prior to debugging the system. The ASM chart, programme, and ROM relating to this version are available in appendices A, B and C. An updated version of this information is offered following this section.

**Finite State Machine:** The system follows the principles of a finite Mealy state machine meaning the outputs are determined by the states and the inputs (Proposed programme: s0 to s10 and the op codes respectively).

**Calculation Cycle Counter:** The calculations requiring the system to load and store data many times, it is clear the overall calculation needed to be broken in steps each with its own flag. Here the calculations are broken down in cycles from 0 to 4 as shown below:

0. (W+Z)/2.
1. (X-Y).
2. (X-Y)*2.
3. [(X-Y)*2] XOR [(W+Z)/2]
4. ([(X-Y)*2] XOR [(W+Z)/2])*V

On completion of the calculation the "Calculation_Cycle" counter is incremented.

**ROM and RAM Counter Selector:** These are available at s4 and s9. They allow the increment of the PC for a specific count in order to target a specific RAM address to store or load data. It can also be used to target the appropriate ROM address to load an instruction.

**Information specific to programme post to debugging:**

Debugging of the proposed programme demonstrated a lot of issues and proved very time consuming. Therefore, the information contained in the following parts only describes a fragment of tasks supposed to be realised by the system. The tasks presented here will realise the operations and calculations to obtain (W+Z)/2 and store the result into the RAM.

Although the same concepts of Calculation cycle counter and ROM/RAM counter selector are used their state may be different.

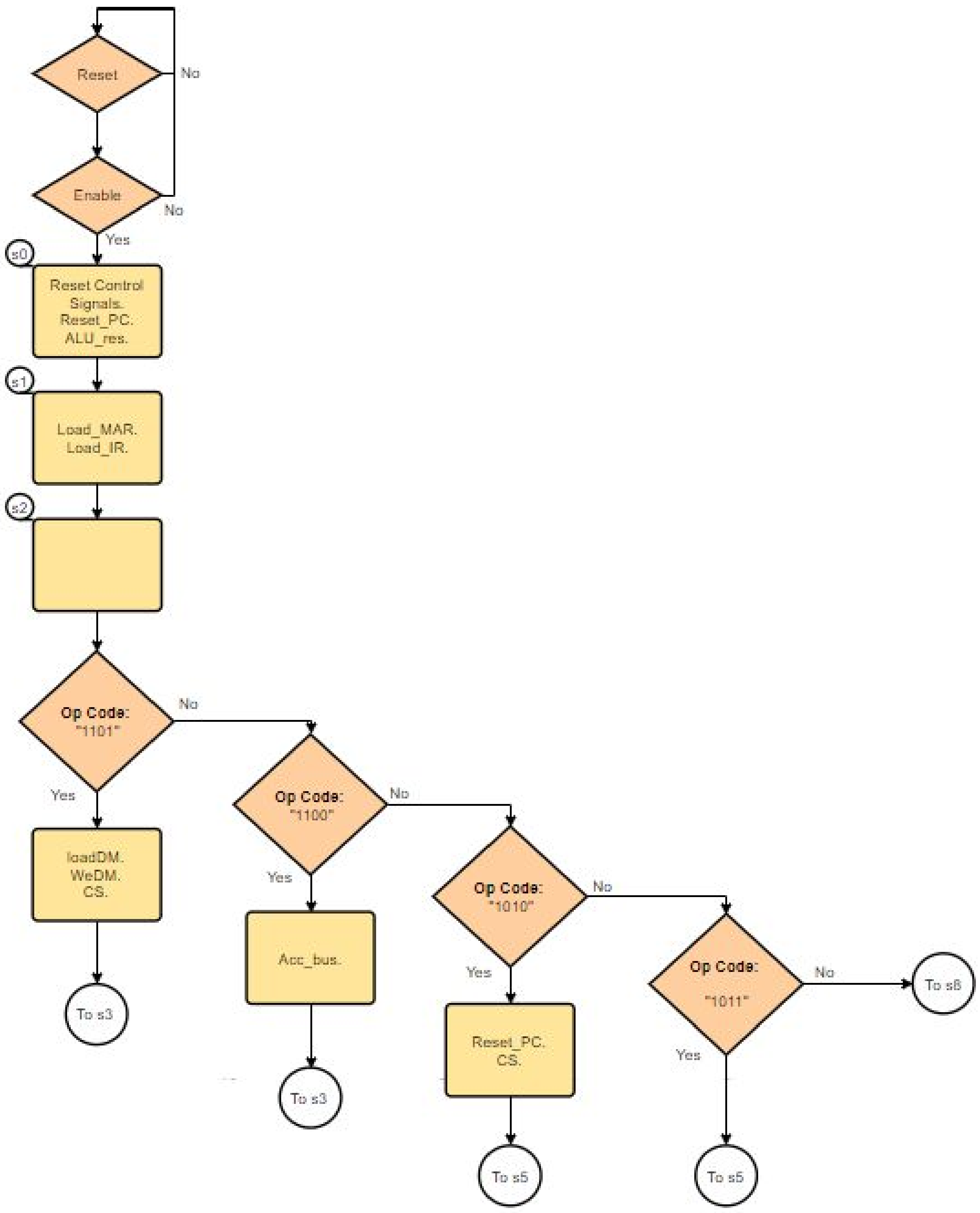
The calculation cycles have been updated as such:

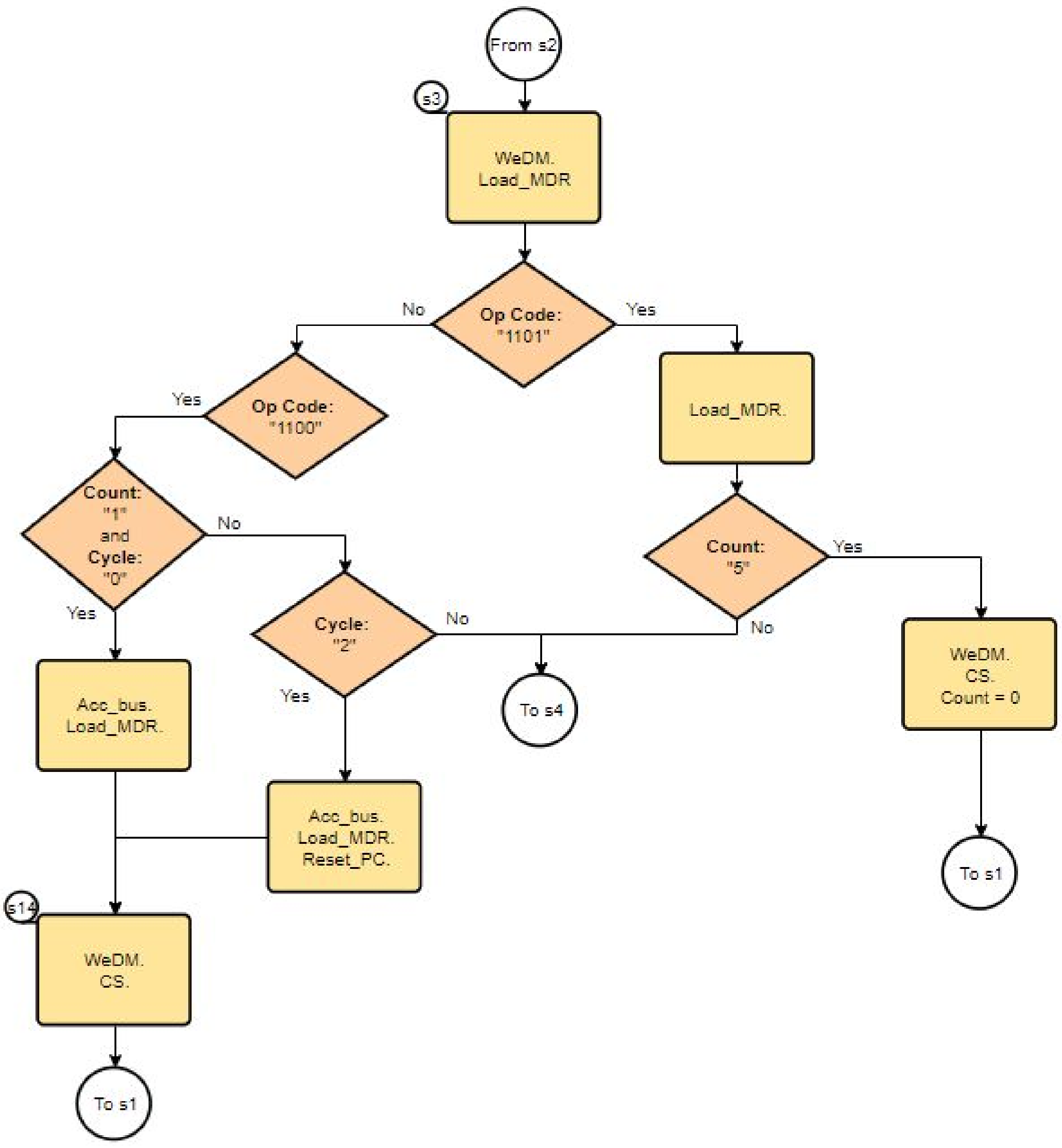
0. (W+Z).
1. (W+Z)/2.

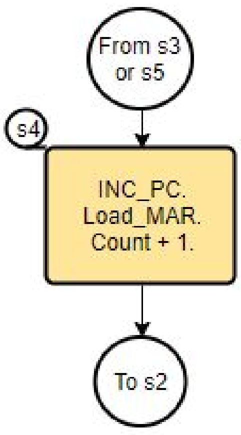
Not presented and still requiring debugging:

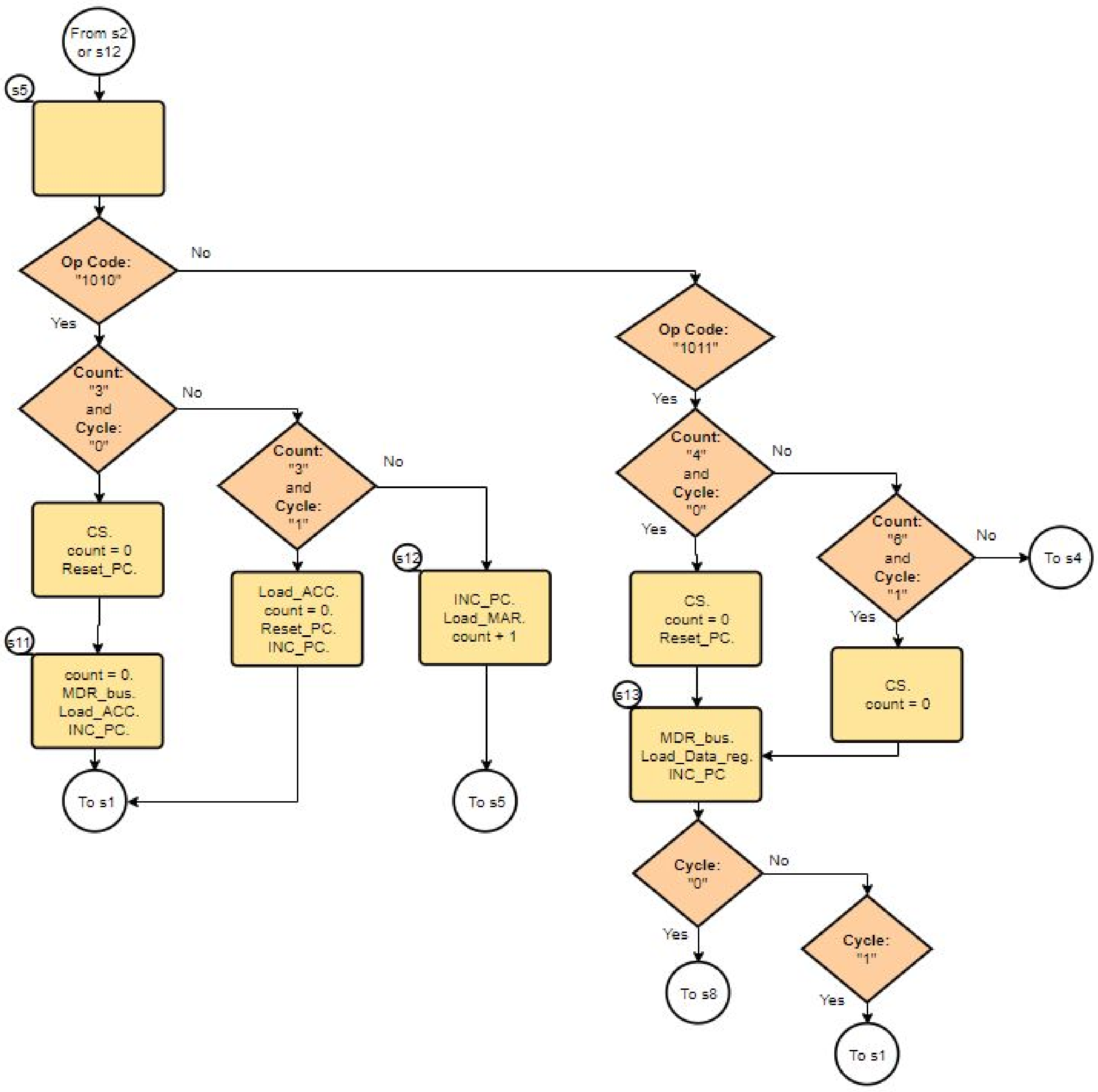
2. (X-Y).
3. (X-Y)*2.
4. [(X-Y)*2] XOR [(W+Z)/2].
5. ([(X-Y)*2] XOR [(W+Z)/2])*V.

**Algorithmic State Machine Chart:** This chart gives an overview of the operation of the CU sequence to load data in the system then perform and store “(W+Z)/2”. The chart has been broken down to allow for better readability.

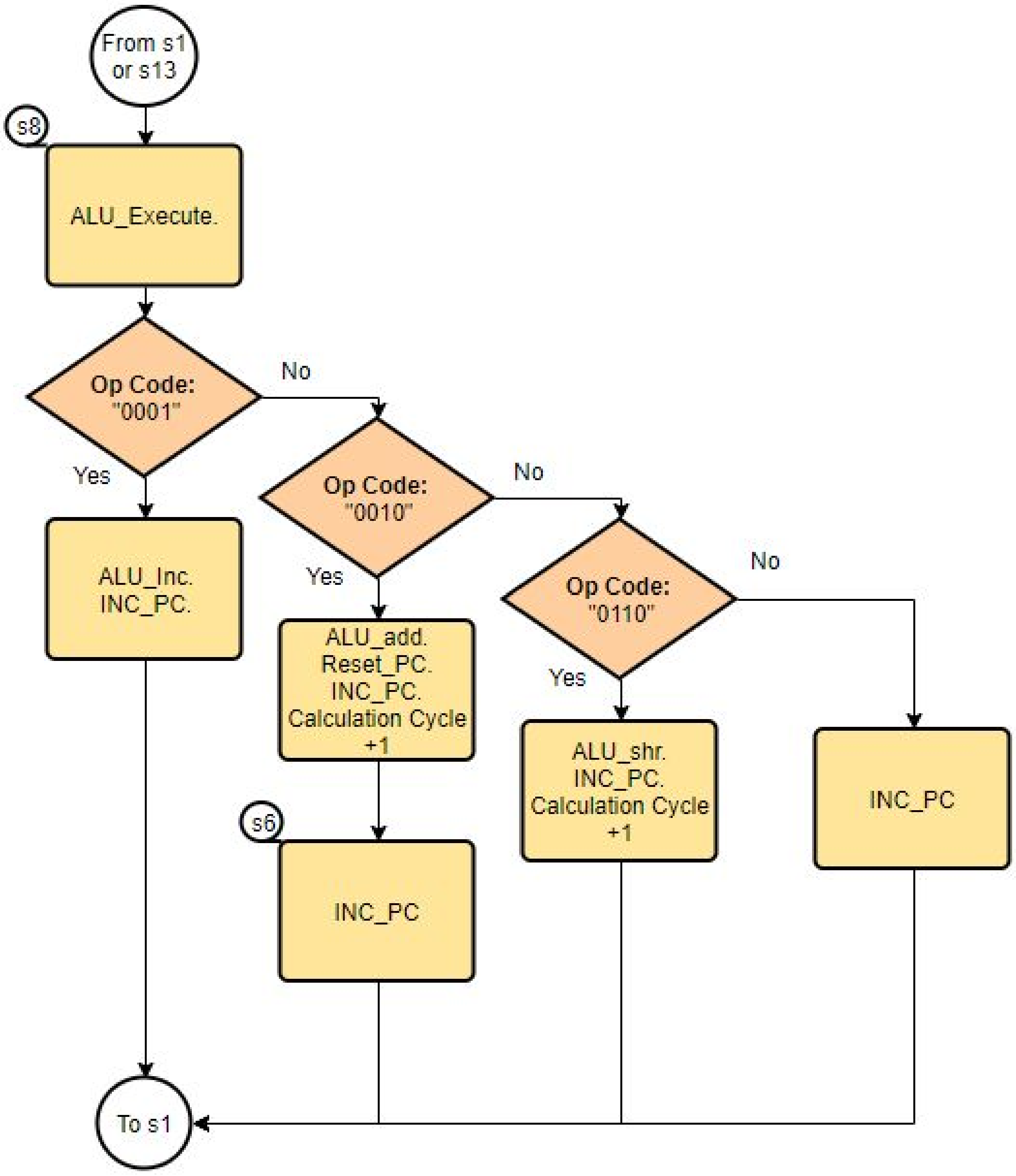



From s2
s3
WeDM.
Load_MDR
Op Code:
"1101"
No
Yes
Op Code:
"1100"
Yes
Load_MDR.
Count:
"1"
and
Cycle:
"0"
No
Count:
"5"
Yes
Yes
Cycle:
"2"
No
No
WeDM.
CS.
Count = 0
Acc_bus.
Load_MDR.
Yes
To s4
Acc_bus.
Load_MDR.
Reset_PC.
To s1
s14
WeDM.
CS.
To s1


From s3
or s5
s4
INC_PC.
Load_MAR.
Count + 1.
To s2



From s2 or s12
s5
Op Code: "1010"
No
Yes
Count: "3" and Cycle: "0"
No
CS. count = 0 Reset_PC.
s11
count = 0. MDR_bus. Load_ACC. INC_PC.
To s1
Count: "3" and Cycle: "1"
No
Load_ACC. count = 0. Reset_PC. INC_PC.
s12
INC_PC. Load_MAR. count + 1
To s5
Op Code: "1011"
Yes
Count: "4" and Cycle: "0"
No
Yes
CS. count = 0 Reset_PC.
s13
MDR_bus. Load_Data_reg. INC_PC
Count: "6" and Cycle: "1"
No
To s4
Yes
CS. count = 0
Cycle: "0"
No
Yes
To s8
Cycle: "1"
Yes
To s1


From s1 or s13
s8
ALU_Execute.
Op Code: "0001"
No
Yes
ALU_Inc. INC_PC.
Op Code: "0010"
No
Yes
ALU_add. Reset_PC. INC_PC. Calculation Cycle +1
s6
INC_PC
Op Code: "0110"
No
Yes
ALU_shr. INC_PC. Calculation Cycle +1
INC_PC
To s1

## B. Control Unit VHDL Code.

The programme provided still requires further debugging. It performs the operations to load data to the system, obtain “(W+Z)/2” and store the result to the RAM. Code still requiring debugging has been highlighted.

```
----------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
----------------------------------------------------------------------
entity sequencer is
port (clock, reset, Enable : in std_logic; -- system controls
      opcode: in Std_logic_vector(3 downto 0); -- Op code input from IR
      load_Acc, load_Data_reg, ALU_Execute, Acc_bus, ACC_outSys, --
Control Outputs
      load_IR, load_MAR, MDR_bus, load_MDR, loadDM,CS, WeDM, INC_PC,
Reset_PC, ALU_res, -- Control Outputs
      ALU_inc, ALU_add, ALU_sub, ALU_mul, ALU_shl, ALU_shr, ALU_xor :
out std_logic);-- Arithmetic Outputs

end entity sequencer;
----------------------------------------------------------------------
architecture Behavior of sequencer is
        type state is (s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11,
s12, s13, s14); -- Programme states
        signal present_state, next_state : state;
begin
    sequence : process (clock, reset) is -----------------------------
Sequencer
    begin
        if reset = '1' then
            present_state <= s0;
        elsif rising_edge(clock) then
            present_state <= next_state;
        end if;
    end process sequence;
----------------------------------------------------------------------
    programme : process (Enable, present_state, opcode) is ------------
Application Programme
    variable count : integer range 0 TO 255;
    variable Calculation_Cycle: integer range 0 TO 255;
    begin
```

```
        if Enable <='1' then
        ACC_bus <= '0';
        ACC_outSys <= '0';
        load_ACC <= '0';
        load_Data_reg <= '0';
        load_IR <= '0';
        load_MAR <= '0';
        loadDM <='0';
        MDR_bus <= '0';
        load_MDR <= '0';
        ALU_Execute <= '0';
        INC_PC <= '0';
        CS <= '0';
        WeDM <= '0';
        ALU_res <= '0';
        ALU_inc <= '0';
        ALU_add <= '0';
        ALU_sub <= '0';
        ALU_mul <= '0';
        ALU_shl <= '0';
        ALU_shr <= '0';
        ALU_xor <= '0';
        Reset_PC <= '0';
        ALU_res <= '0';
         case present_state is
             when s0 => -- Reset state
             Reset_PC <= '1';
             ALU_res <= '1';
             count := 0;
             Calculation_Cycle:= 0;
             next_state <= s1;
             when s1 =>
                 Load_MAR <='1';
                 load_IR  <='1';
                 next_state <= s2;
             when s2 =>
                 if opcode = "1101" then -- STE writing to RAM actions.
                     loadDM <='1'; -- load data from data_mux - RAM
allocation according to ROM address:{ROM 0:V, ROM 1:W, ROM 2:X, ROM 3:Y,
ROM 4:Z)
                     WeDM  <='1';-- Enable writting.
                     CS  <='1';-- Activate RAM.
                     next_state <= s3;
                 elsif opcode = "1100" then --STA writing to RAM actions.
                       Acc_bus <='1'; -- From ROM 8 and PC is 1 - go to
```

```
ram address in ROM 6 and Store result of :"(W plus Z)/2"
                     next_state <= s3;
                elsif opcode = "1010" then -- LDA reading from RAM
actions.
                    if Calculation_cycle = 0 then
                         Reset_PC <='1'; -- PC is 0
                         CS <= '1'; -- Activate RAM
                         next_state <= s5;
                     end if;
                elsif opcode = "1011" then -- LDD reading from RAM
actions.
                    next_state <= s5;
                else
                    next_state <= s8;
                end if;
            when s3 => -- Store.
                if opcode = "1101" then -- if STE - ROM location 0 ->
Store External Data to RAM.
                    load_MDR <='1'; -- load data from RAM_data_in to MDR
                    if count = 5 then
                      WeDM  <='1';-- Enable writting.
                      CS  <='1';-- Activate RAM.
                      count := 0;
                      next_state <= s1;
                    else
                      next_state <= s4;
                    end if;
                elsif opcode = "1100" then -- if STA - ROM location 6 or
7 -> Store Acc to RAM.
                   -- Store "00000001" from Acc in RAM address according
to ROM 6.
                     if count = 1 and Calculation_Cycle = 0  then
                        Acc_bus <='1';
                        load_MDR <='1';
                        next_state <= s14;
                  -- From ROM 0 and PC is 0 - go to ram address in ROM 8
and Store result of :"(W plus Z)/2"
                      elsif Calculation_Cycle = 2  then
                        Acc_bus <='1';
                        load_MDR <='1';
                        Reset_PC <='1';
                        next_state <= s14;
                       else
                          next_state <= s4;
                     end if;
                 end if;
            when s4 => -- increment PC and load address to RAM
                INC_PC <='1';
                Load_MAR <='1';
                count := count + 1;
                next_state <= s2;
            when s5 => -- Load.
                   if opcode = "1010" then -- if LDA - ROM location 7 ->
Load Acc from Data RAM.
                    -- go to ram address in ROM 1 and load W
                    if count = 3 and Calculation_Cycle = 0 then
                       CS <='1';
                       count := 0;
                       Reset_PC <= '1';
                       next_state <= s11;
                    elsif count = 3 and Calculation_Cycle = 1 then
                     -- go to ram address in ROM 2 and load X
                       load_ACC <= '1';
                       count := 0;
                       Reset_PC <= '1';
                       INC_PC <= '1';
                       next_state <= s1;
                    else
                        next_state <= s12;  -- to Load Counter/Selector
                    end if;
                elsif opcode = "1011" then -- if LDD - ROM location 1 ->
Load Data_reg from Data RAM.
                    -- go to ram address in ROM 4 and load Z
                    if count = 4 and Calculation_Cycle = 0 then
                      CS <='1';
                      count := 0;
                      Reset_PC <='1';
                      next_state <= s13; -- to RAM_out->ALU_in->Data_reg
                    elsif count = 6 and Calculation_Cycle = 1 then -- go
to ram address in ROM 6 and load "00000001" (in calculation cycle 1)
                      CS <='1';
                      count := 0;
                      next_state <= s13; -- to RAM_out->ALU_in->Data_reg
        --Still to be Debugged
        --              elsif count = 3 and Calculation_Cycle =2 then
        --              -- go to ram address in ROM 3 and load Y
        --                load_Data_reg <= '1';
        --                count := 0;
        --                next_state <= s1;
        --              elsif count = 4 and Calculation_Cycle =3 then --
```

```
PC is 1 - go to ram address in ROM 5 and load "00000001" (in calculation
cycle 2)
        --              load_Data_reg <= '1';
        --              count := 0;
        --              INC_PC <='1';
        --              next_state <= s9;
        --            elsif count = 5 and Calculation_Cycle =4 then --
PC is 1 - go to ram address in ROM 9 and load "(W+Z)/2" (in calculation
cycle 3)
        --                load_Data_reg <= '1';
        --                count := 0;
        --               INC_PC <='1';
        --                next_state <= s9; -- PC is 7
        --            elsif Calculation_Cycle = 5 then -- PC is 1 - go
to ram address in ROM 0 and load "V" (in calculation cycle 4)
        --                Reset_PC <='1';
        --               Load_MAR <='1';
        --                load_Data_reg <= '1';
        --                INC_PC <='1';
        --                next_state <= s9; -- PC is 1
                    else
                      next_state <= s4;
                    end if;
                end if;
            when s6 =>
                    INC_PC <='1'; -- LDD-ROM 1
                    -- go to ram address in ROM 6 and load "00000001"
                    next_state <= s1;
            when s7 =>
         --Still to be debugged
         --          --Load_IR <='1'; --STA-ROM 9
         --          Reset_PC <='1';
         --          next_state <= s2;
            when s8 => -- Calculations Selection/ALU opcode
interpretation.
               ALU_Execute <= '1';
                   if opcode = "0001" then -- ROM location 5
                        ALU_inc <= '1';
                        INC_PC <='1';
                        next_state <= s1;
                   elsif opcode = "0010" then -- ROM location 2
                        ALU_add <= '1';
                        Reset_PC <= '1'; -- Reset to send to LDD-ROM 1
                        Calculation_Cycle := Calculation_Cycle +1; --
calculation cycle 0 "(W+Z)" done
```

```
                        next_state <= s6;
                   elsif opcode = "0110" then -- ROM location 8
                        ALU_shr <= '1';
                        Calculation_Cycle := Calculation_Cycle +1; --
calculation cycle 1 "(W+Z)/2" done.
                        INC_PC <='1';
                        next_state <= s1;
       --Still to be Debugged
       --             elsif opcode = "0011" then -- ROM location 3
       --                  ALU_sub <= '1';
       --                 Reset_PC <= '1';
       --                  Calculation_Cycle := Calculation_Cycle +1; --
calculation cyle 2 "(X-Y)" done.
       --                  next_state <= s1;
       --             elsif opcode = "0100" then -- ROM location 11
       --                 ALU_mul <= '1';
       --                  Calculation_Cycle := Calculation_Cycle +1; --
calculation cyle 5 "([(X-Y)*2] XOR [(W+Z)/2])*V" done.
       --                  next_state <= s10;
       --             elsif opcode = "0101" then -- ROM location 9
       --                  ALU_shl <= '1';
       --                  Reset_PC <= '1';
       --                  Calculation_Cycle := Calculation_Cycle +1; --
calculation cycle 3 "(X-Y)*2" done.
       --                  next_state <= s1;
       --             elsif opcode = "1000"  then -- ROM location 10
       --                 ALU_xor <= '1';
       --                 Calculation_Cycle := Calculation_Cycle +1; --
calculation cycle 4 "[(X-Y)*2] XOR [(W+Z)/2]" done.
       --                  Reset_PC <= '1';
       --                  next_state <= s1;
                   else
                        INC_PC <='1';
                        next_state <= s1;
                   end if;
            when s9 => -- PC is 6 (in calculation cycle 3) - PC is 7 (in
calculation cycle 4)
        --Still to be Debugged
        --         if count = 3 and (Calculation_Cycle = 2 or
Calculation_Cycle = 3) then -- PC 6 -> 9. -- PC 7 -> 10.
        --            count := 0;
        --            next_state <= s1; -- Load SHL in ROM 9 -- Load XOR
in ROM 10.
        --         elsif count = 10 and Calculation_Cycle = 4 then -- PC
is 1 (in calculation cycle 4)
```

```
        --           count := 0;
        --           next_state <= s1; -- Load MUL in ROM 11.
        --        else
        --          count := count +1;
        --          next_state <= s11;
        --        end if;
            when s10 => -- Acc -> Output F
          --      ACC_bus <= '1';
          --      ACC_outSys <= '1';
            when s11 => -- MDR to RAM output to Acc.
                 count := 0;
                 MDR_bus <= '1';
                 load_ACC <= '1';
                 INC_PC <='1';
                 next_state <= s1;
            when s12 => -- Load Counter/Selector.
                 INC_PC <='1';
                 Load_MAR <='1';
                 count := count + 1;
                 next_state <= s5;
            when s13 => -- MDR to RAM output to Data_Reg.
                  count := 0;
                  MDR_bus <= '1';
                  load_Data_reg <= '1';
                  INC_PC <='1';
                  if Calculation_Cycle = 0 then
                    next_state <= s8; -- send to calculations selection
to hit the INC_PC
                  elsif Calculation_Cycle = 1 then
                    next_state <= s1;
                  end if;
            when s14 => -- write to ram
                  WeDM  <='1';-- Enable writting.
                  CS  <='1';-- Activate RAM.
                  next_state <= s1;     -- In Calculation Cycle 2 this
is the point where debugging stopped
         end case;
        end if;
    end process programme;
end architecture Behavior;
------------------------------------------------------------------------
```

# V. Microprocessor Simulation.

## A. Data Path.

The data path is the module linking all the different components inputs and outputs of the different components to one another using signals. The data path also provides the inputs and output to the whole system.

**Data Path VHDL Code:**

```
--------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--------------------------------------------------------
entity Data_path is
    Port (clock_sys, reset, Enable : in Std_logic; -- system
control.
          System_input : in std_logic_vector (7 downto 0); --
System data input.
          Output_F : out std_logic_vector (7 downto 0));-- System
Output.
end Data_path;
--------------------------------------------------------
architecture Behavioral of Data_path is
--------------Connecting Signals------------------------
signal CU_to_PC : Std_logic;
signal CU_to_IR : Std_logic;
signal CU_to_Reset_PC : Std_logic;
signal CU_LoadDM : Std_logic;
signal CU_to_WeDM : Std_logic;
signal CU_to_CS : Std_logic;
signal CU_to_MAR : Std_logic;
signal CU_to_MDR : Std_logic;
signal CU_MDR_bus : Std_logic;
signal CU_load_Acc : Std_logic;
signal CU_load_Data_reg : Std_logic;
signal CU_ALU_Execute : Std_logic;
signal CU_Acc_bus : Std_logic;
signal CU_ACC_outSys : Std_logic;
signal CU_ALU_res : Std_logic;
signal CU_ALU_inc : Std_logic;
signal CU_ALU_add : Std_logic;
signal CU_ALU_sub : Std_logic;
signal CU_ALU_mul : Std_logic;
signal CU_ALU_shl : Std_logic;
signal CU_ALU_shr : Std_logic;
signal CU_ALU_xor : Std_logic;
signal PC_to_ROM : Std_logic_vector(3 downto 0);
signal ROM_to_IR : Std_logic_vector(7 downto 0);
signal IR_to_CU : Std_logic_vector(3 downto 0); --opcode
signal IR_to_RAM : Std_logic_vector(3 downto 0); --ram
address/data
signal RAM_to_ALU : Std_logic_vector(7 downto 0);
signal ALU_to_DM : Std_logic_vector(7 downto 0);
signal DM_to_RAM : Std_logic_vector(7 downto 0);
--------------Connecting Modules-----------------------
begin
CU: entity work.sequencer port map(clock => clock_sys,
                                      reset => reset,
                                      Enable => Enable,
                                      opcode =>IR_to_CU,
                                      load_Acc => CU_load_Acc,
                                      load_Data_reg =>
CU_load_Data_reg,
                                      ALU_Execute =>
CU_ALU_Execute,
                                      Acc_bus => CU_Acc_bus,
                                      ACC_outSys => CU_ACC_outSys,
                                      load_IR => CU_to_IR,
                                      load_MAR => CU_to_MAR,
                                      MDR_bus => CU_MDR_bus,
                                      load_MDR => CU_to_MDR,
                                      loadDM => CU_loadDM,
                                      CS => CU_to_CS,
                                      WeDM => CU_to_WeDM,
                                      INC_PC => CU_to_PC,
                                      Reset_PC => CU_to_Reset_PC,
                                      ALU_res => CU_ALU_res,
                                      ALU_inc => CU_ALU_inc,
                                      ALU_add => CU_ALU_add,
                                    ALU_sub => CU_ALU_sub,
                                    ALU_mul => CU_ALU_mul,
                                    ALU_shl => CU_ALU_shl,
                                    ALU_shr => CU_ALU_shr,
                                    ALU_xor => CU_ALU_xor);

ALU: entity work.ALU port map(clock => clock_sys,
                              ALU_in => RAM_to_ALU,
                              load_Acc => CU_load_Acc,
                              load_Data_reg => CU_load_Data_reg,
                              ALU_Execute => CU_ALU_Execute,
                              Acc_bus => CU_Acc_bus,
                              ALU_res => CU_ALU_res,
                              ALU_inc => CU_ALU_inc,
                              ALU_add => CU_ALU_add,
                              ALU_sub => CU_ALU_sub,
                              ALU_mul => CU_ALU_mul,
                              ALU_shl => CU_ALU_shl,
                              ALU_shr => CU_ALU_shr,
                              ALU_xor => CU_ALU_xor,
                              ALU_out => ALU_to_DM);


PC: entity work.PC port map(clock => clock_sys,
                              reset_PC => CU_to_Reset_PC,
                              Inc => CU_to_PC,
                              ROM_addr => PC_to_ROM);

ROM: entity work.ROM port map(ROM_in =>PC_to_ROM,
                              ROM_out => ROM_to_IR);

IR: entity work.IR port map(clock => clock_sys,
                              reset => reset,
                              Load_IR => CU_to_IR,
                              Data_ROM => ROM_to_IR,
                              Operand => IR_to_CU,
                              RAM_addr => IR_to_RAM);

RAM: entity work.RAM port map(clock => clock_sys,
                              reset => reset,
                              WeDM => CU_to_WeDM,
                                  RAM_in_addr =>IR_to_RAM,
                                  RAM_in_data => DM_to_RAM,
                                  RAM_out => RAM_to_ALU,
                                  CS => CU_to_CS,
                                  MDR_bus => CU_MDR_bus,
                                  load_MDR => CU_to_MDR,
                                  load_MAR => CU_to_MAR);

DM: entity work.Data_in_bus port map(clock => clock_sys,
                                         LoadDM => CU_LoadDM,
                                         Ext_Data_in =>
System_input,
                                         Acc_in => ALU_to_DM,
                                         DM_out => DM_to_RAM,
                                         ACC_outSys =>
CU_ACC_outSys,
                                         F => Output_F);

end Behavioral;
--------------------------------------------------------
```

## B. Test Bench.

The test bench provides the input stimulus to the system. In this application, reset, the system clock, enable and data input are used to interact with the rest of the system. Various data values are sent to the system data input to simulate: V, W, X, Y and Z.

**Test Bench VHDL code:**

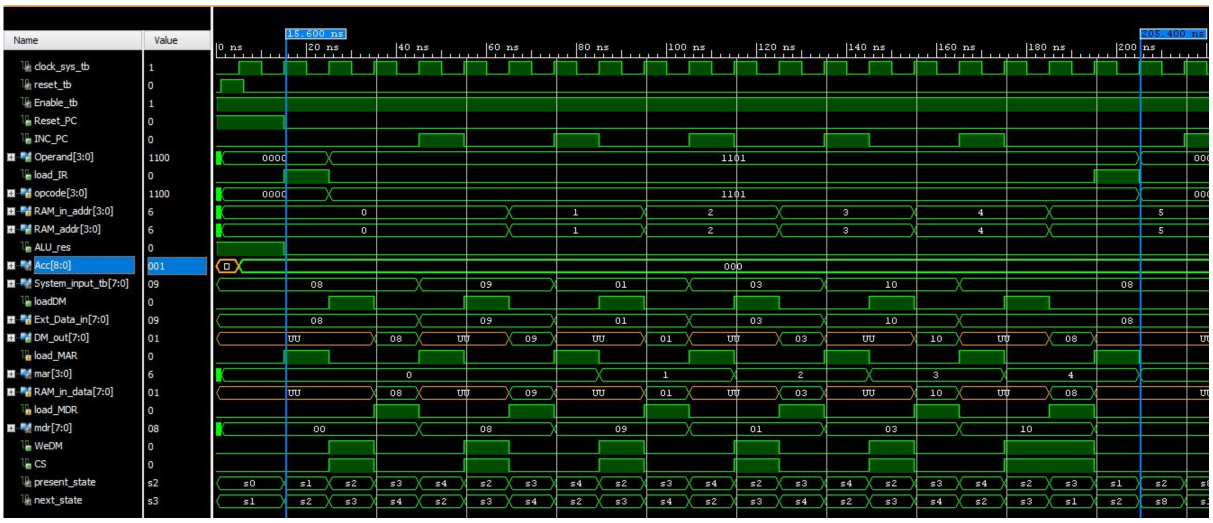
```
--------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--------------------------------------------------------
entity Data_path_tb is
--  Port ( );
end Data_path_tb;
--------------------------------------------------------
architecture testbench of Data_path_tb is
component Data_path is
  Port (clock_sys, reset, Enable : in Std_logic; -- system
control.
          System_input : in std_logic_vector (7 downto 0); --
System data input.
          Output_F : out std_logic_vector (7 downto 0));
      end component;
------Signal Declaration:------

          signal clock_sys_tb : std_logic :='0'; -- System Clock.
          signal reset_tb : std_logic :='0';
          signal Enable_tb : std_logic :='0';
          signal System_input_tb :  std_logic_vector (7 downto
0):="00001000";
          signal Output_F_tb :  std_logic_vector (7 downto 0);
begin

      ------ DUT Instantiation:------
  dut: Data_path Port map(clock_sys => clock_sys_tb, reset =>
reset_tb, Output_F => Output_F_tb, Enable => Enable_tb,
System_input => System_input_tb);
```

```
clock_sys_tb <= not clock_sys_tb after 5 ns;
reset_tb <= '1' after 1ns, '0' after 6ns;
Enable_tb <= '1';
  stim_proc: process
    begin
            wait for 45 ns;
        System_input_tb <= "00001001";
            wait for 30 ns;
        System_input_tb <= "00000001";
             wait for 30 ns;
        System_input_tb <= "00000011";
             wait for 30 ns;
        System_input_tb <= "00010000";
             wait for 30 ns;
        System_input_tb <= "00001000";
             wait for 30 ns;
      end process;

end architecture testbench;
--------------------------------------------------------
```

## C. Microcontroller Simulation output.

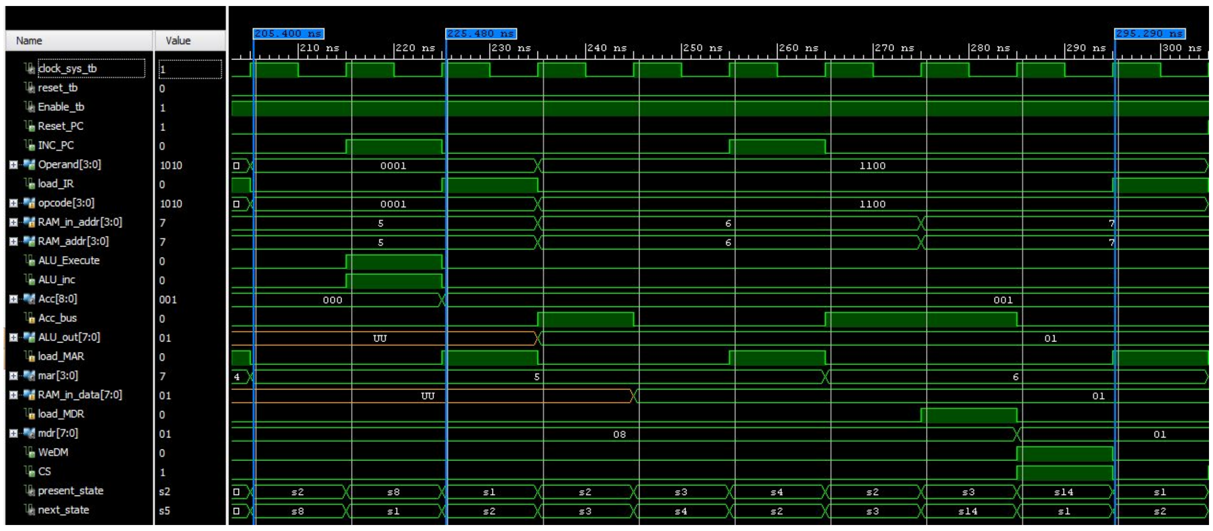
As previously mentioned in its current form CU is only capable to perform operations and calculations for the two first calculation cycles to obtain “(W+Z)/2” and store it to the RAM. The section below will provide time diagrams for each phase.

1. Reset and load data (V, W, X, Y and Z) to the system RAM:



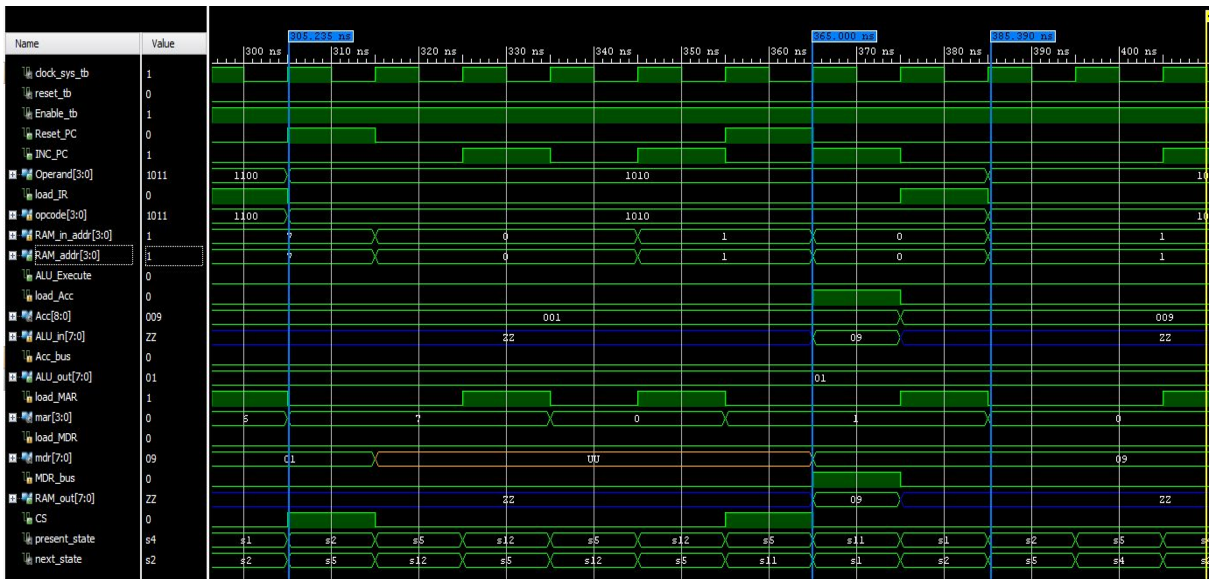
The first marker in the simulation above delimits the reset phase. The present state is at s0 where the accumulator and PC are reset with the next state being s1.
At s1 the first instruction (STE) is loaded from ROM 0. The data mux is activated to load the external data. The programme then loops the sequence of loading the MAR, MDR, enabling the writing and activating the RAM to transit the data from the input to the 5 first RAM locations. As shown in the diagram, RAM addresses are controlled by the increase of the PC allowing the incremental output of RAM addresses (done by transiting through the counter selector in s4). Once the count is at 5 the programme returns to s1 for the next operation (second marker).

2. Increment the accumulator and store the value (“01”) to RAM for later use:



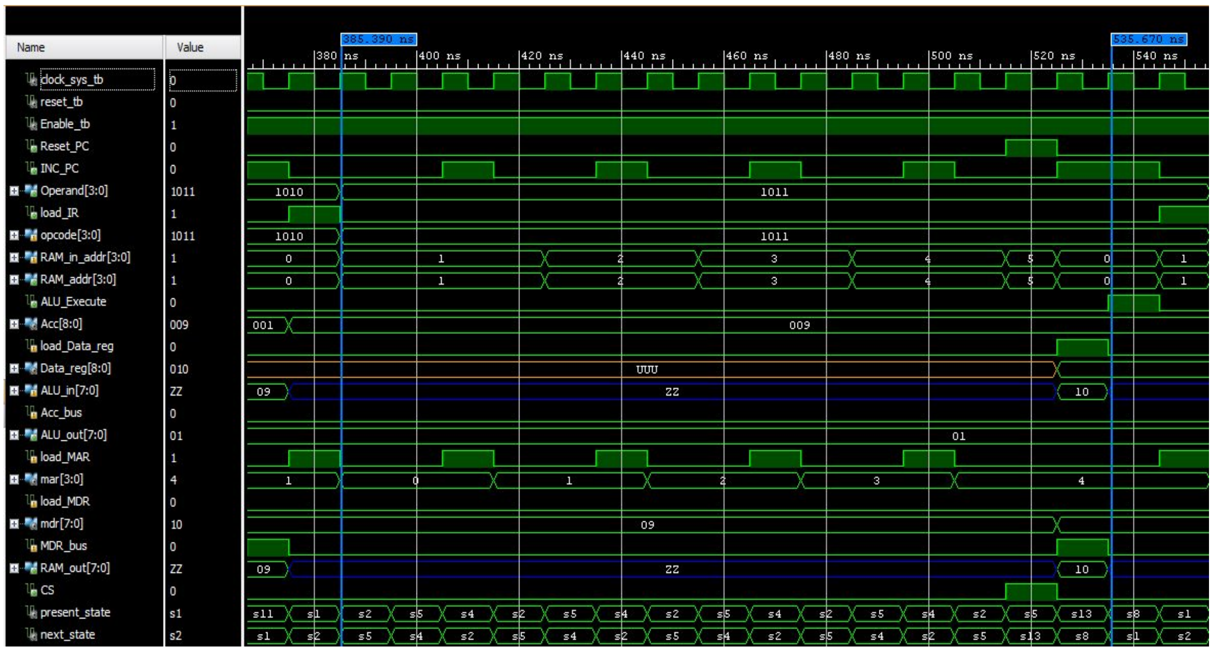
The first marker shows the point at which the next instruction is loaded (INC - increment the accumulator by one). From then the programme goes to s8 where the ALU operations are performed (ALU_inc and ALU_execute). The second marker shows the point at which the increment is done and the next instruction is loaded (STA - store the accumulator value to the RAM). The Acc value is then sent to the ALU output (through the data mux) to the RAM input. The programme then follows the writing to RAM sequence to store the value “01” to RAM location 6 selected using PC. The third marker marks the point at which the CU loads the next instruction.

3. Load the accumulator with W value (“09”) from the RAM:



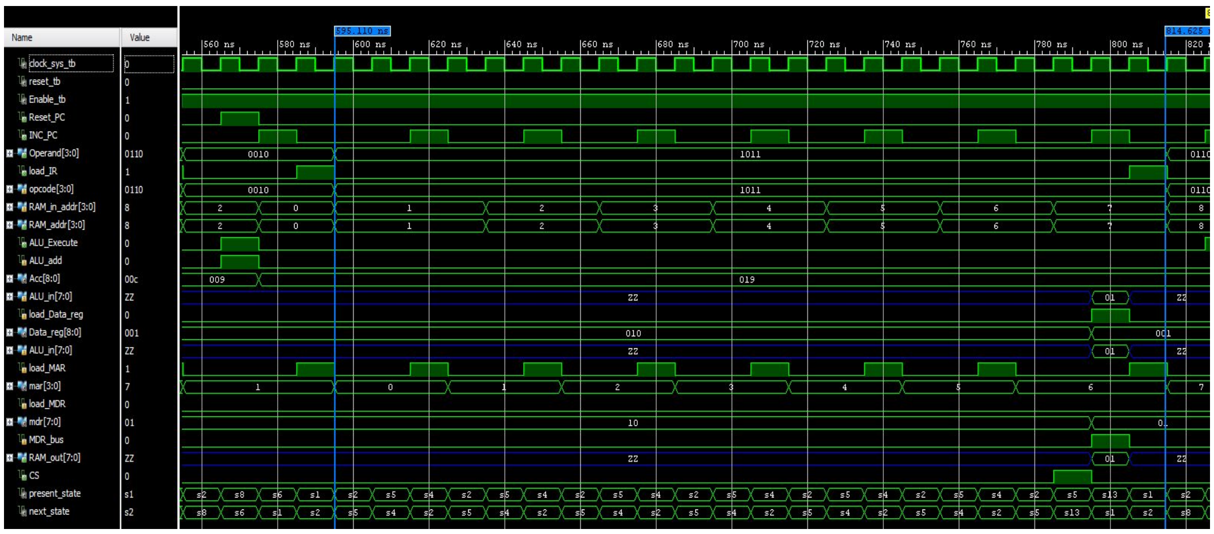
The first marker shows the point at which the next instruction is loaded (LDA - Load data from RAM to the accumulator in the ALU). Previously to the instruction being loaded the MAR was loaded with RAM location 6 and then the data transferred to the MDR by activating CS. The second marker shows the point at which the MDR is sent to the output of the RAM, Load_acc is activated. The third marker shows the point at which the operation is completed and the value from W (“09”) is displayed in the accumulator.

4. Load the data register with Z value (“10) from the RAM:



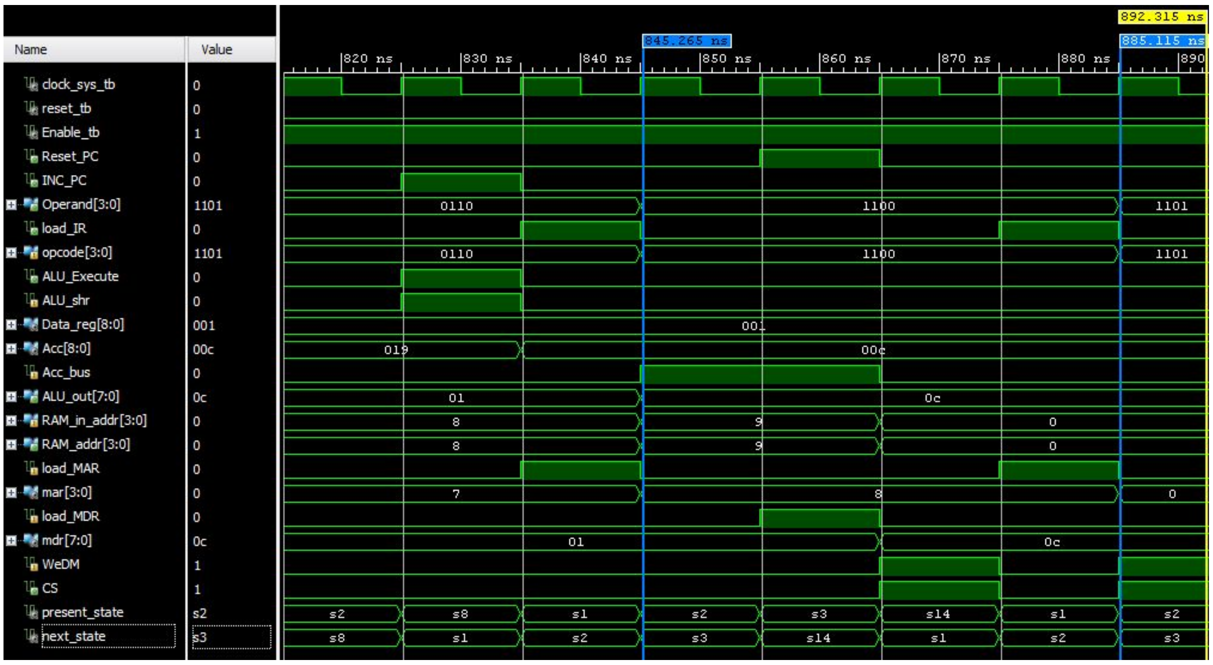
The first marker shows where the next instruction is loaded (LDD - Load the data from RAM to the data register in the ALU). The programme has to use the PC to select the RAM address 5 where the value for Z is stored. As explained previously, the data is loaded from the RAM to the ALU input but this time loaded to the data register.

5. Perform the addition and load “01” to the data register:



The period prior to the first marker shows the addition instruction being performed in the ALU between the W and Z and returning the corresponding value “019” or “25” in decimal. The following period shows the next instruction loaded (LDD) with PC incrementing to retrieve the value “01” previously stored in RAM location 6. The value is then loaded to the data register in the ALU and the next instruction loaded.

6. Perform the division by 2 using the shift right function and store the result to RAM:



The period prior to the first marker, follows the SHR instruction loaded and shows the division, by 2, be realised by shifting the value in the accumulator by the value in the data register “01”. The value returned is “0C” or “12” in decimal. The period between the first and second marker shows the programme storing the value of the calculation in RAM location 8 for later use.

# Discussion: decisions, problems, errors and self-assessment.

This part of the report serves as a discussion on various aspects of the project. It highlights certains points around the decisions made, the problems and errors encountered as well as a personal reflection on the development methodology used.

**Decisions:** As previously mentioned the design was an amalgamation of the proposed architecture in the assignment specifications and of a microprocessor designed by Zwoliński. Instead of using a single bus for the operations, the components are connected to one another according to their operations to avoid errors due to components writing simultaneously to the same bus.
Secondly, based on Zwoliński's design, opcodes are interpreted in the Control Unit, instead of passed on to the ALU, and trigger digital outputs.
Thirdly, it was decided to realise the multiplication and divisions as a basic shift to the right or left operation. More work would be required to allow the ALU to accurately realise those operations using the flags.
Counters were created and used as flags to control various elements in the CU sequence.
Finally, the instructions implementation in the ROM were done so as to allow a more efficient control of the system and avoid resetting and incrementing the PC the least possible. Only 12 ROM locations and 7 RAM locations are necessary for the programme to operate.

**Error and Problems:** Most of the errors and problems were created by matching the sequence between the different components to allow the data to flow logically and be present at the right moment in the sequence. This was especially true for RAM, in tandem with the CU, operations which required a lot of debugging.
At first the counters would seem not to be working because the wrong expression was used to check their value in a condition statement “count<= 4” instead of “count

= 4”. It was also discovered that the counters used during RAM selection required to count to one more than the location desired because the condition would be met and operations would trigger faster than the RAM could associate the right address.

**Personal Reflection:** Although debugging is faster than writing the programme for the CU, a different approach would have been necessary. Indeed, a step by step approach of creating and testing a function to slowly build the programme would have been beneficial. Although, writing the whole CU programme and organising the ROM allocation, was useful to flesh out the main architecture and process of the programme it made debugging very tedious at first.
Now that the process is better understood debugging the rest of the programme would have required less time. It is believed more knowledge of the operations of a microprocessor would have been beneficial to speed up the development process.

# Conclusion

The report presented the steps undertaken to implement an ASIC FPGA system. The development was broken down in several phases and by components. Components were programmed and tested independently to check their basic operations. A large part of the development was focused on the Control Unit. The CU was developed as a Mealy Finite State Machine using an Algorithm State Machine design methodology. Efforts were taken to keep the operations efficient when possible (storing the next instruction in the ROM location matching the last state of the programme controller so to avoid many increments of the PC).
The various components operations made it difficult to control the sequence and a lot of time was spent debugging the system. In the end, the solution only offers part of the operations required by the specifications. Although, now that the methodology and process are better understood it is believed the rest of the operations would be implemented relatively rapidly. More work is required to finalise some operations and optimise/rationalise the present programme code.

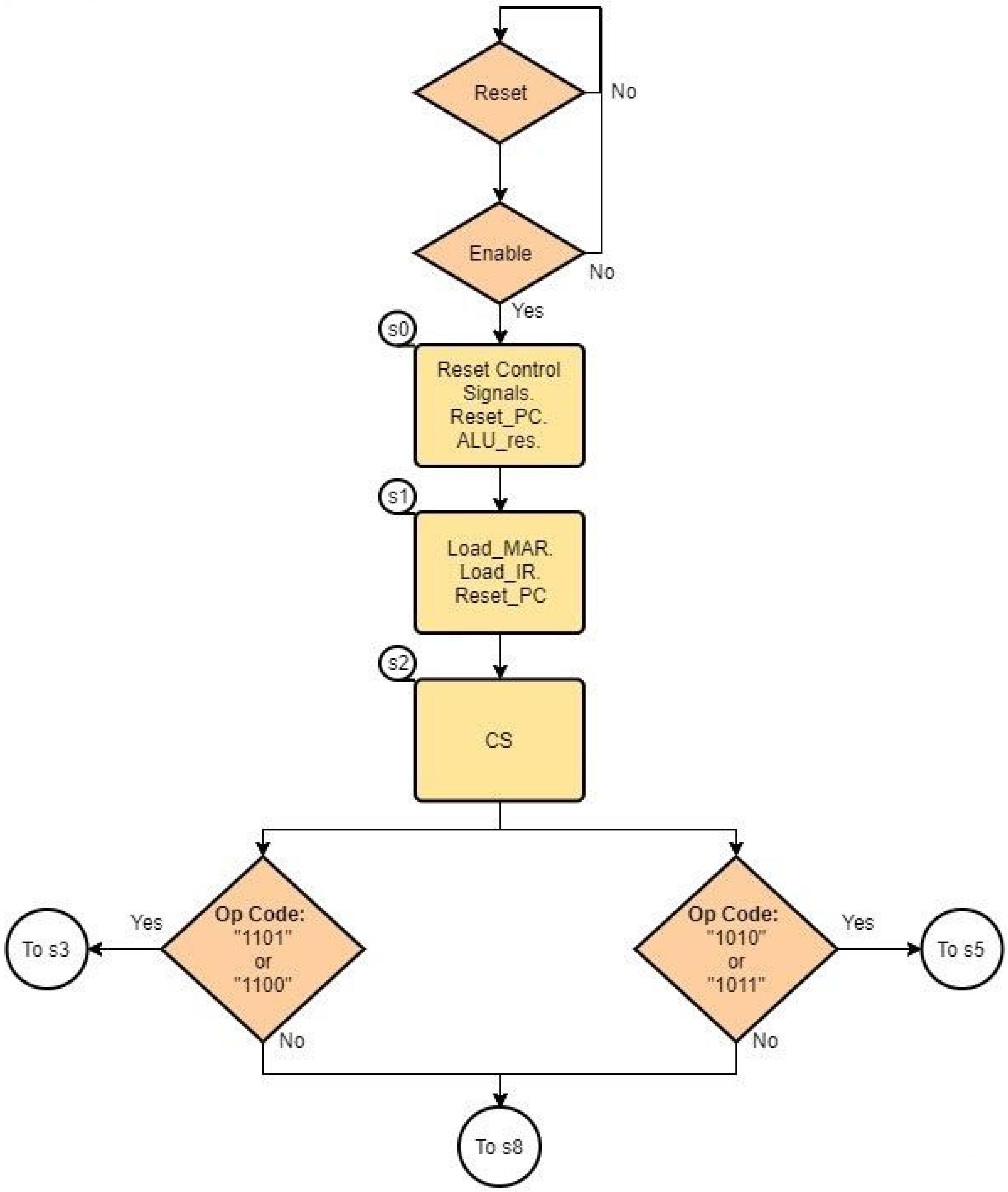
# Bibliography.

Horowitz, P., Hill, W. and Oxford University Press (2015). *The art of electronics*. 3rd ed. New York: Cambridge University Press.

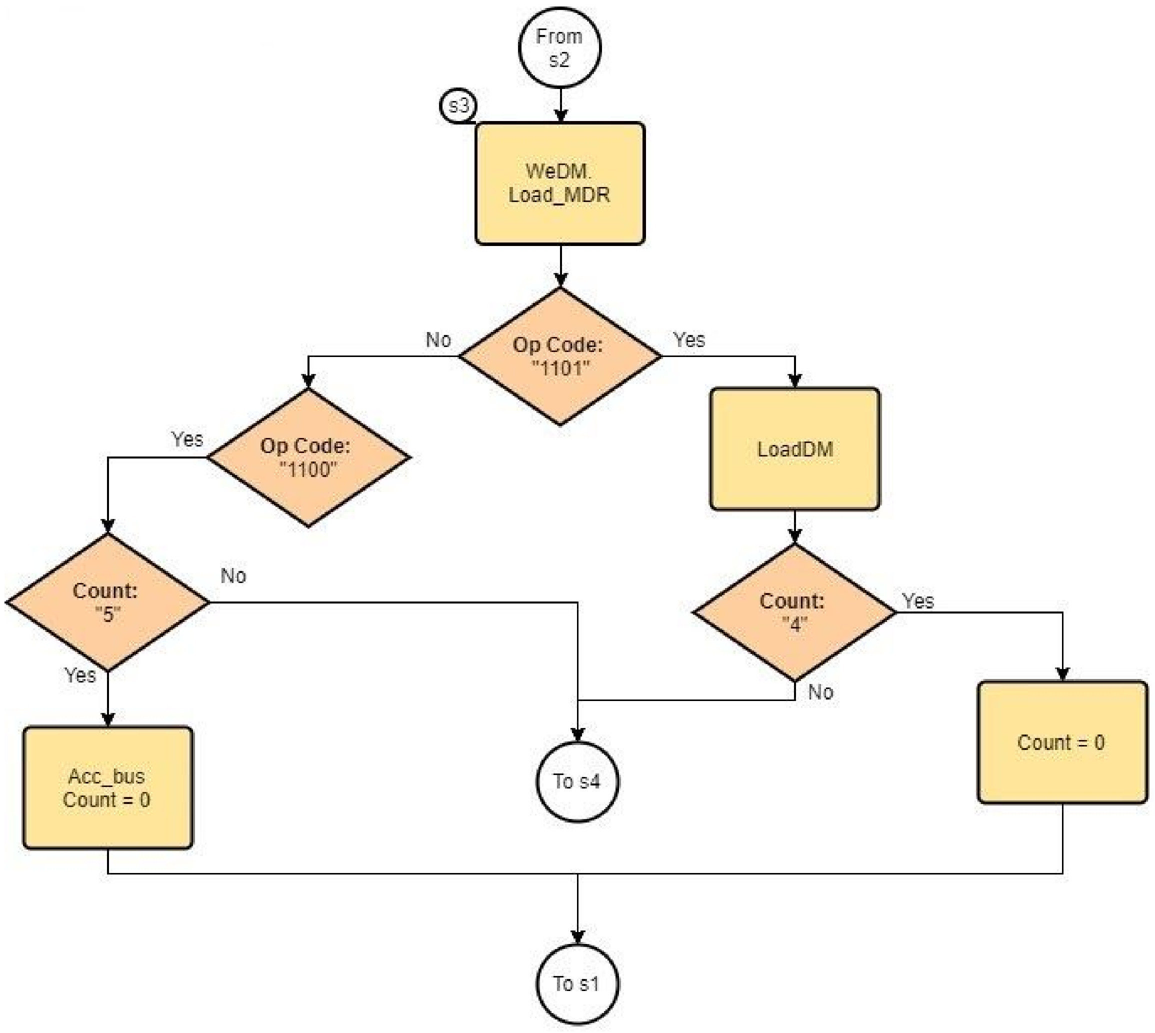
Zwoliński, M. (2004). *Digital system design with VHDL*. 2nd ed. Harlow: Prentice-Hall.

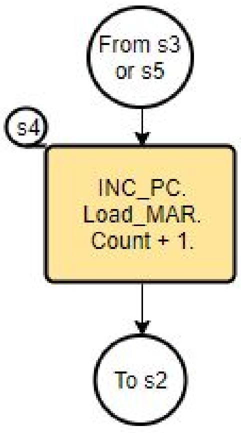
# Appendices

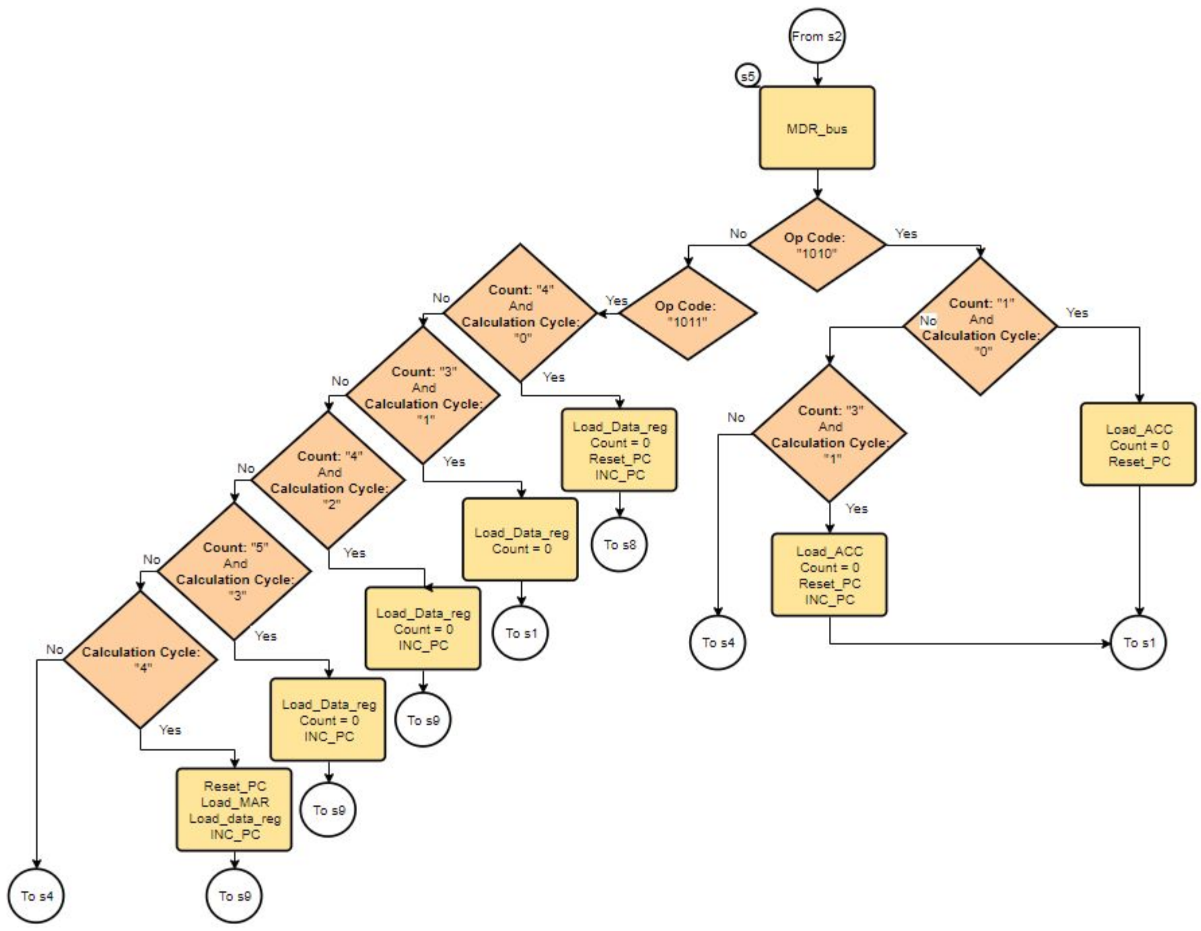
### Appendix A - Proposed Algorithmic State Machine Chart.

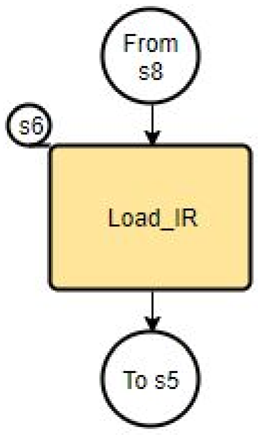


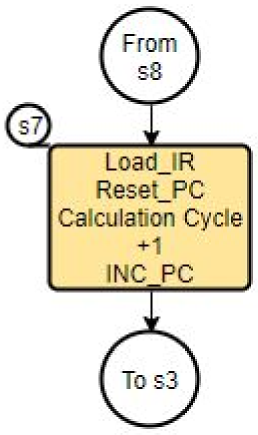
Reset
No
Enable
No
Yes
s0
Reset Control Signals. Reset_PC. ALU_res.
s1
Load_MAR. Load_IR. Reset_PC
s2
CS
Op Code: "1101" or "1100"
Yes
To s3
No
Op Code: "1010" or "1011"
Yes
To s5
No
To s8

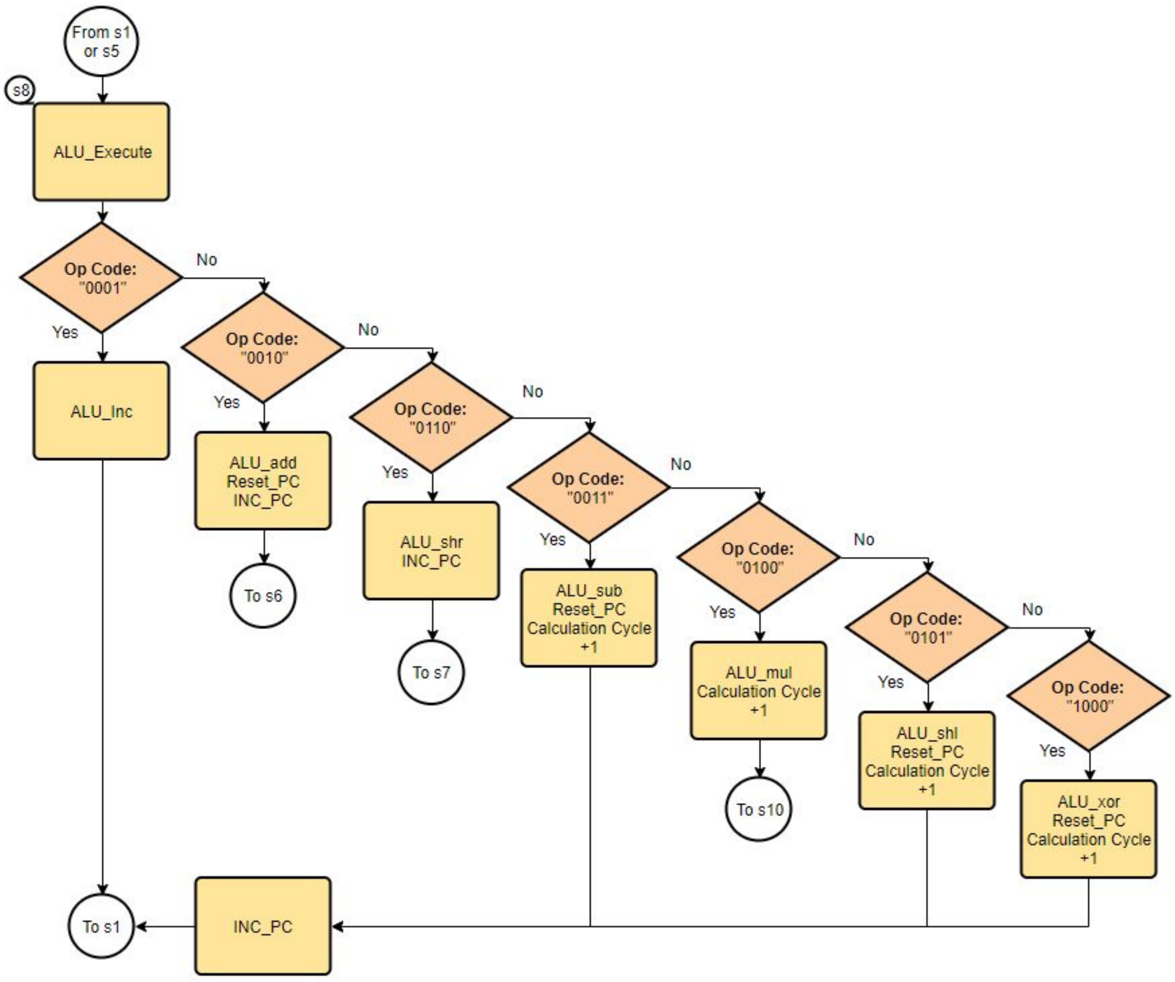

From s2
s3
WeDM.
Load_MDR
Op Code: "1101"
No
Yes
Op Code: "1100"
Yes
LoadDM
Count: "5"
No
Yes
Count: "4"
Yes
No
Acc_bus
Count = 0
To s4
Count = 0
To s1

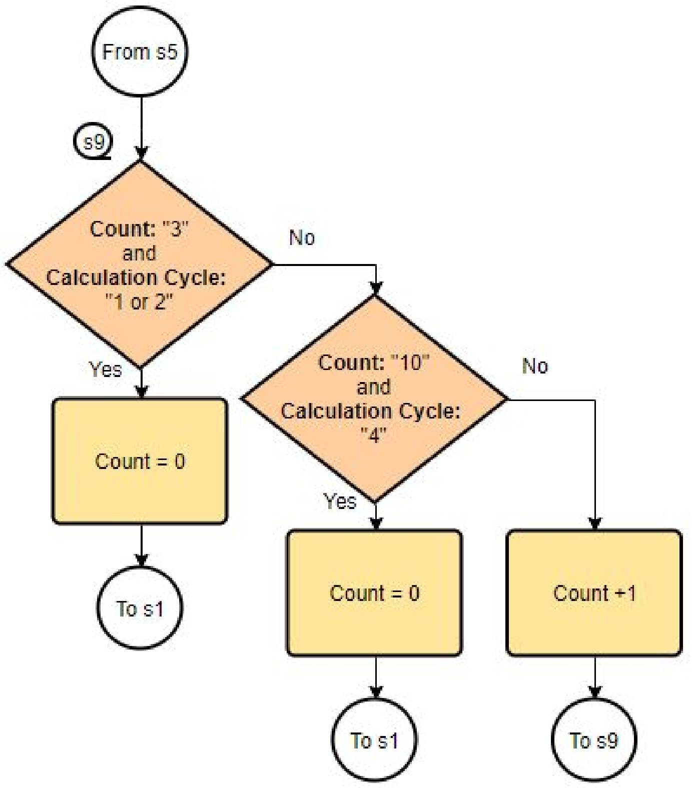

From s3 or s5
s4
INC_PC.
Load_MAR.
Count + 1.
To s2

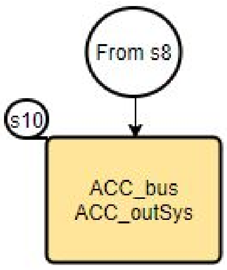

From s2
s5
MDR_bus
Op Code: "1010"
No
Yes
Op Code: "1011"
Count: "1" And Calculation Cycle: "0"
Load_ACC Count = 0 Reset_PC
To s1
Count: "3" And Calculation Cycle: "1"
Load_ACC Count = 0 Reset_PC INC_PC
To s4
Count: "4" And Calculation Cycle: "0"
Load_Data_reg Count = 0 Reset_PC INC_PC
To s8
Count: "3" And Calculation Cycle: "1"
Load_Data_reg Count = 0
To s1
Count: "4" And Calculation Cycle: "2"
Load_Data_reg Count = 0 INC_PC
To s9
Count: "5" And Calculation Cycle: "3"
Load_Data_reg Count = 0 INC_PC
To s9
Calculation Cycle: "4"
Reset_PC Load_MAR Load_data_reg INC_PC
To s9
To s4


From s8
s6
Load_IR
To s5


From s8
s7
Load_IR
Reset_PC
Calculation Cycle +1
INC_PC
To s3


From s1 or s5
s8
ALU_Execute
Op Code: "0001"
Yes
No
ALU_Inc
Op Code: "0010"
Yes
No
ALU_add
Reset_PC
INC_PC
To s6
Op Code: "0110"
Yes
No
ALU_shr
INC_PC
To s7
Op Code: "0011"
Yes
No
ALU_sub
Reset_PC
Calculation Cycle +1
Op Code: "0100"
Yes
No
ALU_mul
Calculation Cycle +1
To s10
Op Code: "0101"
Yes
No
ALU_shl
Reset_PC
Calculation Cycle +1
Op Code: "1000"
Yes
ALU_xor
Reset_PC
Calculation Cycle +1
INC_PC
To s1


From s5
s9
Count: "3" and Calculation Cycle: "1 or 2"
Yes
No
Count = 0
To s1
Count: "10" and Calculation Cycle: "4"
Yes
No
Count = 0
To s1
Count +1
To s9


From s8
s10
ACC_bus
ACC_outSys

## Appendix B - Proposed Control Unit Processor VHDL Code.

```
-----------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
-----------------------------------------------------------------------
entity sequencer is
port (clock, reset, Enable : in std_logic; -- system controls
      opcode: in Std_logic_vector(3 downto 0); -- Op code input from IR
      load_Acc, load_Data_reg, ALU_Execute, Acc_bus, ACC_outSys, --
Control Outputs
      load_IR, load_MAR, MDR_bus, load_MDR, loadDM,CS, WeDM, INC_PC,
Reset_PC, ALU_res, -- Control Outputs
      ALU_inc, ALU_add, ALU_sub, ALU_mul, ALU_shl, ALU_shr, ALU_xor :
out std_logic); -- Arithmetic Outputs
end entity sequencer;
-----------------------------------------------------------------------
architecture Behavior of sequencer is
        type state is (s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10); --
Programme states
        signal present_state, next_state : state;
begin
    sequence : process (clock, reset) is ---------------- Sequencer
    begin
        if reset = '1' then
            present_state <= s0;
        elsif rising_edge(clock) then
            present_state <= next_state;
        end if;
    end process sequence;
-----------------------------------------------------------------------
    programme : process (Enable, present_state, opcode) is --Application
Programme
    variable count : integer;
    variable Calculation_Cycle: integer;
    begin
       if Enable <='1' then
        case present_state is
            when s0 => -- Reset state
         -- reset all the control signals to default and PC and Acc to 0
            ACC_bus <= '0';
            ACC_outSys <= '0';
            load_ACC <= '0';
            load_Data_reg <= '0';
            load_IR <= '0';
            load_MAR <= '0';
            loadDM <='0';
            MDR_bus <= '0';
            load_MDR <= '0';
            ALU_Execute <= '0';
            INC_PC <= '0';
            CS <= '0';
            WeDM <= '0';
            ALU_res <= '0';
            ALU_inc <= '0';
            ALU_add <= '0';
            ALU_sub <= '0';
            ALU_mul <= '0';
            ALU_shl <= '0';
            ALU_shr <= '0';
            ALU_xor <= '0';
            Reset_PC <= '1';
            ALU_res <= '1';
            next_state <= s1;
            when s1 =>
                Load_MAR <='1';
                load_IR  <='1';
                Reset_PC <='1'; -- Reset PC for RAM indexing.
                next_state <= s2;
            when s2 =>
                CS  <='1';-- Activate RAM
                if opcode = "1101" or opcode = "1100" then -- writing to
RAM actions.
                    next_state <= s3;
                elsif opcode = "1010" or opcode = "1011" then -- reading
from RAM actions.
                    next_state <= s5;
                else
                    next_state <= s8;
                end if;
            when s3 => -- Store.
                WeDM  <='1';
                load_MDR <='1';
                if opcode = "1101" then -- if STE - ROM location 0 ->
Store External Data to RAM.
                    loadDM <='1'; -- load data from data_mux - RAM
allocation according to ROM address:{ROM 0:V, ROM 1:W, ROM 2:X, ROM 3:Y,
ROM 4:Z)
                    if count <= 4 then
```

```
                              count := 0;
                              next_state <= s1;
                            else
                              next_state <= s4;
                            end if;
                        elsif opcode = "1100" then -- if STA - ROM location 5 or
8 -> Store Acc to RAM.
                            if count <= 5 then -- Store "00000001" from Acc in
RAM address according to ROM 5.
                                Acc_bus <='1'; -- From ROM 8 and PC is 1 - go to
ram address in ROM 6 and Store result of :"(W plus Z)/2"
                                count := 0;
                                next_state <= s1;
                            else
                                next_state <= s4;
                            end if;
                         end if;
                    when s4 => -- increment PC and load address to RAM
                        INC_PC <='1';
                        Load_MAR <='1';
                        count := count +1;
                        next_state <= s2;
                    when s5 => -- Load.
                        MDR_bus <= '1';
                        if opcode = "1010" then -- if LDA - ROM location 6 ->
Load Acc from Data RAM.
                            -- go to ram address in ROM 1 and load W
                            if count <= 1 and Calculation_Cycle <= 0 then
                               load_ACC <= '1';
                               count := 0;
                               next_state <= s1;
                            elsif count <= 3 and Calculation_Cycle <= 1 then
                             -- go to ram address in ROM 2 and load X
                               load_ACC <= '1';
                               count := 0;
                               Reset_PC <= '1';
                               INC_PC <= '1';
                               next_state <= s1;
                            else
                              next_state <= s4;
                            end if;
                        elsif opcode = "1011" then -- if LDD - ROM location 1 ->
Load Data_reg from Data RAM.
                            -- go to ram address in ROM 4 and load Z
                            if count <= 4 and Calculation_Cycle <=0 then -- PC
```

```
is 1 - go to ram address in ROM 5 and load "00000001" (in calculation
cycle 0)
                      load_Data_reg <= '1';
                      count := 0;
                      Reset_PC <='1';
                      INC_PC <='1';
                      next_state <= s8;
                    elsif count <= 3 and Calculation_Cycle <=1 then
                    -- go to ram address in ROM 3 and load Y
                      load_Data_reg <= '1';
                      count := 0;
                      next_state <= s1;
                    elsif count <= 4 and Calculation_Cycle <=2 then --
PC is 1 - go to ram address in ROM 5 and load "00000001" (in calculation
cycle 2)
                      load_Data_reg <= '1';
                      count := 0;
                      INC_PC <='1';
                      next_state <= s9;
                    elsif count <= 5 and Calculation_Cycle <=3 then --
PC is 1 - go to ram address in ROM 6 and load "(W+Z)/2" (in calculation
cycle 3)
                        load_Data_reg <= '1';
                        count := 0;
                        INC_PC <='1';
                        next_state <= s9; -- PC is 7
                    elsif Calculation_Cycle <= 4 then -- PC is 1 - go to
ram address in ROM 0 and load "V" (in calculation cycle 4)
                        Reset_PC <='1';
                        Load_MAR <='1';
                        load_Data_reg <= '1';
                        INC_PC <='1';
                        next_state <= s9; -- PC is 1
                    else
                      next_state <= s4;
                    end if;
                end if;
            when s6 =>
                    Load_IR <='1'; -- LDD-ROM 1
                    -- go to ram address in ROM 5 and load "00000001"
                    next_state <= s5;
            when s7 =>
                  Load_IR <='1'; --STA-ROM 8
                  Reset_PC <='1';
                  Calculation_Cycle := Calculation_Cycle +1; --
```

```
calculation cycle 0 "(W+Z)/2" done.
                  INC_PC <='1';
                  next_state <= s3;
            when s8 =>
               ALU_Execute <= '1';
                   if opcode = "0001" then -- ROM location 4
                        ALU_inc <= '1';
                   elsif opcode = "0010" then -- ROM location 2
                        ALU_add <= '1';
                        Reset_PC <= '1'; -- Reset to send to LDD-ROM 1
                        INC_PC <='1';
                        next_state <= s6;
                   elsif opcode = "0110" then -- ROM location 7
                        ALU_shr <= '1';
                        INC_PC <='1';
                        next_state <= s7;
                   elsif opcode = "0011" then -- ROM location 3
                        ALU_sub <= '1';
                        Reset_PC <= '1';
                        Calculation_Cycle := Calculation_Cycle +1; --
calculation cycle 1 "(X-Y)" done.
                   elsif opcode = "0100" then -- ROM location 11
                        ALU_mul <= '1';
                        Calculation_Cycle := Calculation_Cycle +1; --
calculation cycle 4 "([(X-Y)*2] XOR [(W+Z)/2])*V" done.
                        next_state <= s10;
                   elsif opcode = "0101" then -- ROM location 9
                        ALU_shl <= '1';
                        Reset_PC <= '1';
                    Calculation_Cycle := Calculation_Cycle +1; --
calculation cycle 2 "(X-Y)*2" done.
                   elsif opcode = "1000"  then -- ROM location 10
                        ALU_xor <= '1';
                        Calculation_Cycle := Calculation_Cycle +1; --
calculation cycle 3 "[(X-Y)*2] XOR [(W+Z)/2]" done.
                        Reset_PC <= '1';
                end if;
                INC_PC <='1';
                next_state <= s1;
             when s9 => -- PC is 6 (in calculation cycle 2) - PC is 7
(in calculation cycle 3)
                if count <= 3 and (Calculation_Cycle <= 1 or
Calculation_Cycle <= 2) then -- PC 6 -> 9. -- PC 7 -> 10.
                   count := 0;
                   next_state <= s1; -- Load SHL in ROM 9 -- Load XOR in
```

```
ROM 10.
                elsif count <= 10 and Calculation_Cycle <= 4 then -- PC
is 1 (in calculation cycle 4)
                   count := 0;
                   next_state <= s1; -- Load MUL in ROM 11.
                else
                   count := count +1;
                   next_state <= s9;
                end if;
            when s10 => -- Acc -> Output F
                ACC_bus <= '1';
                ACC_outSys <= '1';
         end case;
        end if;
    end process programme;
end architecture Behavior;
------------------------------------------------------------------------
```

## Appendix C - ROM for proposed programme.

| ROM for Programme | | |
|---|---|---|
| **ROM address** | **Op Code** | **RAM address** |
| “0000” | “1101” - STE | “0000” |
| “0001” | “1011” - LDD | “0001” |
| “0010” | “0010” - ADD | “0010” |
| “0011” | “0011” - SUB | “0011” |
| “0100” | “0001” - INC | “0100” |
| “0101” | “1100” - STA | “0101” |
| “0110” | “1010” - LDA | “0110” |

| | | |
|---|---|---|
| “0111” | “0110” - SHR | “0111” |
| “1000” | “1100” - STA | “1000” |
| “1001” | “0101” - SHL | “1001” |