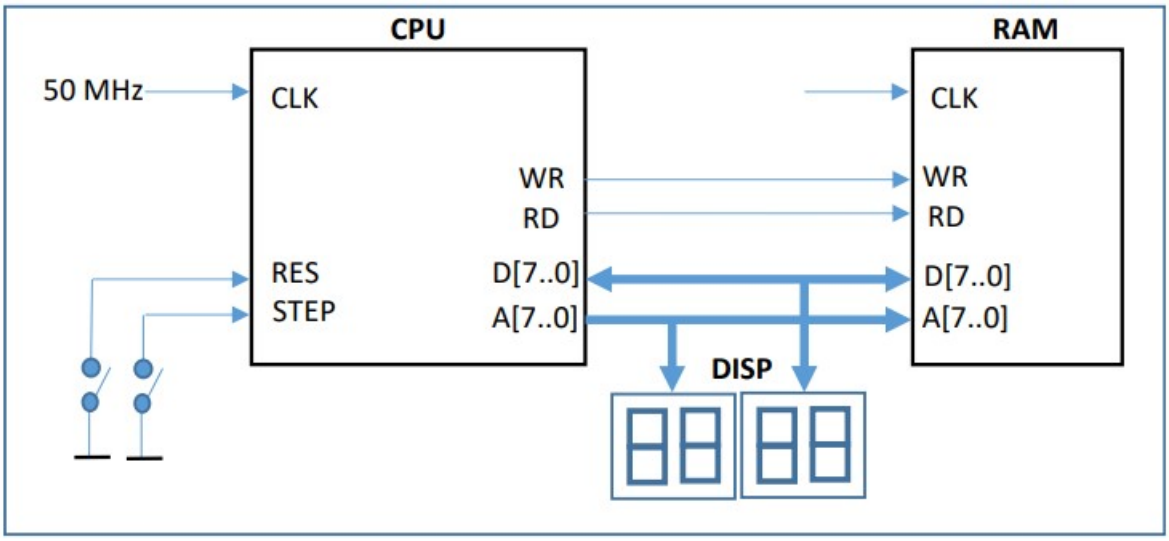
UPPSALA
UNIVERSITET

Project 1FA326

8-bit microprocessor



Author:

Isak Bolin

Uppsala

May 8, 2023

Contents

1 Introduction

Microprocessors have had a big impact on technical development in the world during the last few decades. The first microprocessor: Intel's 4004 was a 4-bit processor introduced in 1971 [1]. Nowadays, we find microprocessors in all kinds of systems ranging all the way from cars to fridges. The purpose of this project is to develop skills and increase knowledge of VHDL programming and FPGA:s, building on the skills gathered in the course: 1FA326 at Uppsala University. In this report, an implementation of an 8-bit microprocessor system (consisting of a cpu and ram) on the DE1-SoC development board is presented. The system has a predefined instruction set and runs on a 50MHz clock.

The design is implemented in VHDL with the program Quartus Prime. Initially, the different subsystems of the CPU and ram were created and tested. After that, the subsystems were connected with the help of a schematic interface. To test the complete design, a lot of simulations were made with ModelSim. Lastly, the design was implemented and tested on the hardware. An important feature is the possibility to step through the code with a button for debugging and display purposes.

2 Project description

The goal of the project is to design an 8 bit microprocessor system that consists of a CPU and RAM (see figure 1). This design is supposed to be implemented on the DE1-SoC development board (see figure 2).

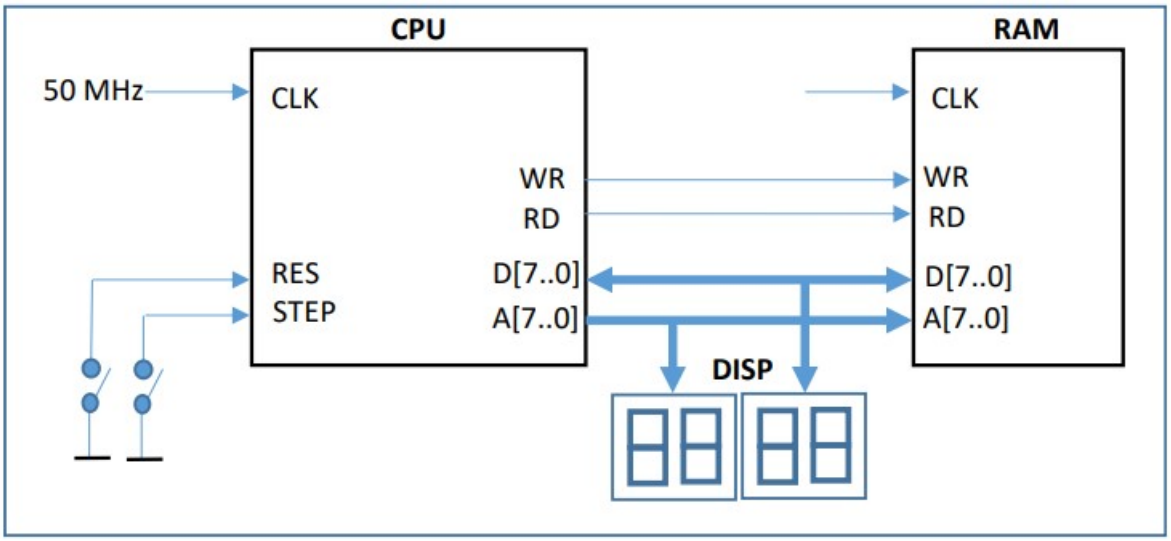


Figure 1: 8-bit microprocessor system.

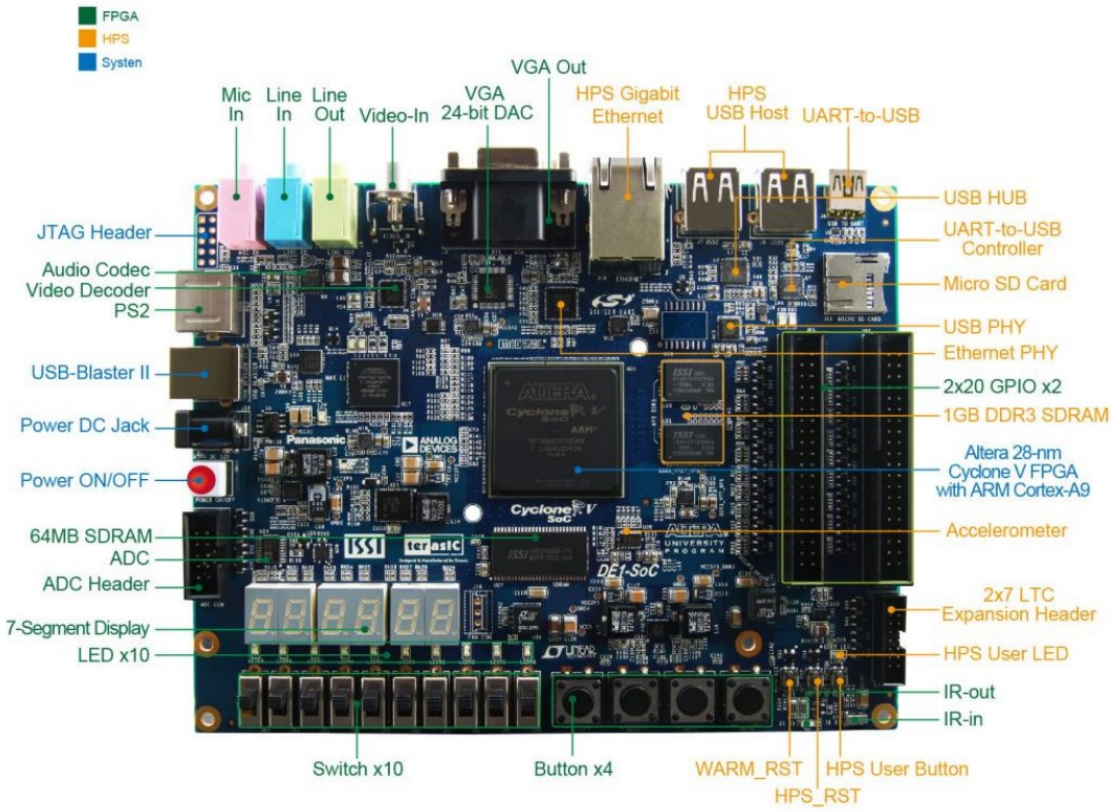


Figure 2: The DE1-SoC development board.

3 Theory

A microprocessor is the central unit of a computer system and usually consists of a control unit, registers, program counter, and an alu [2]. It has a defined instruction set with instructions that the processor can understand and perform. The bandwidth of the processor is the number of bits that can be processed in a single instruction (8 in this case).

The CPU in this project is formed by 5 modules: a control unit, an arithmetic logic unit, a program counter, and two address registers that are used as pointers (see figure 3). All of these run on the development board's 50MHz clock. The control unit is controlling the RAM memory and the other modules by adjusting the control_bus. For debugging purposes, the code on the microprocessor has the ability to be stepped through with a button. At the same time, the address and data bus can be displayed (in hex) on the board's 7-segment displays.

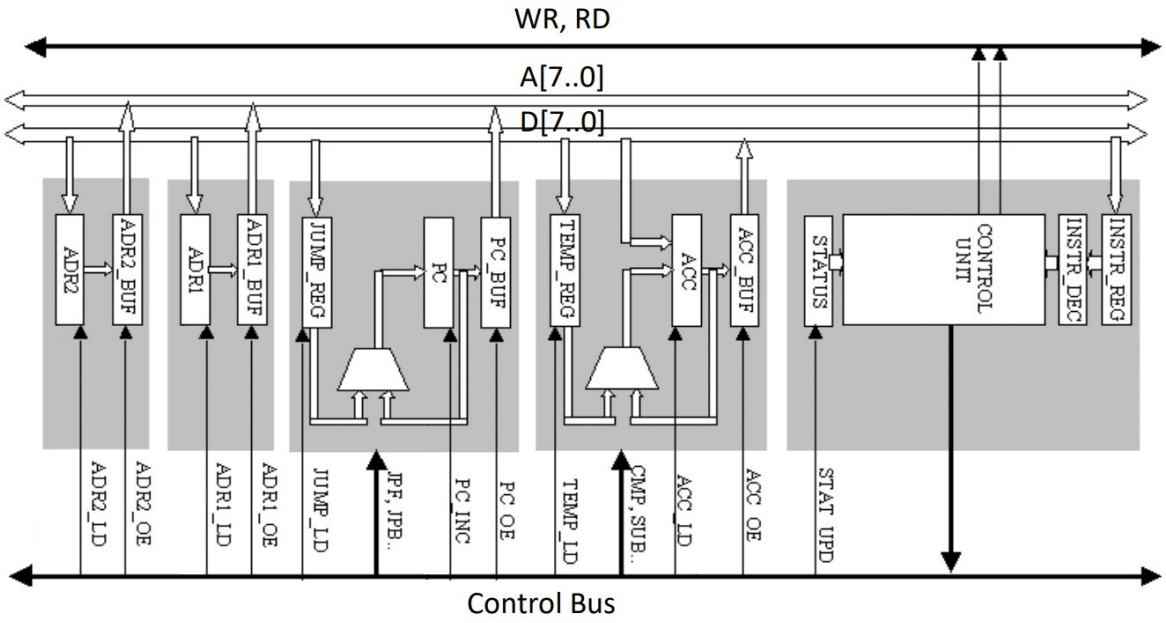


Figure 3: All submodules of the CPU. The signal JPF, JPB also contain JPF_G and JPF_Z. CMP, SUB also includes ADD. Note that this image is modified compared to the one in the original project description document.

The functionality of the CPU compromises operations in two clock cycles: FETCH and EXECUTE. Throughout the FETCH cycle, the instruction is read from memory (pointed by the program counter) and decoded by the control unit. During the EXECUTE cycle, the operation is performed based on the output signals from the control unit on the control_bus.

Instruction-wise, the system features the instruction set displayed in figure 4. A double-length instruction consists of both an instruction and an argument taking up two addresses in the memory. This requires the program counter to be incremented one extra time in the case of a double-length instruction.

Instruction set		
Double-length instructions		
LD_ADR1	Loads address register 1	ADR1 <= MEM(PC+1)
LD_ADR2	Loads address register 2	ADR2 <= MEM(PC + 1)
LD_JUMPREG	Loads JUMP-register	JUMP_REG <= MEM(PC + 1)
Single-length instructions		
LD_ACC	Loads accumulator	ACC <= MEM(ADR1)
LD_TEMP	Loads temporary register	TEMP <= MEM(ADR2)
ST_ACC1	Store accumulator	MEM(ADR1) <= ACC
ST_ACC2	Store accumulator	MEM(ADR2) <= ACC
JPF	Unconditional jump forward	PC <= PC + JUMP_REG
JPB	Unconditional jump back	PC <= PC - JUMP_REG
JPF_G	Conditional jump forward	PC <= PC + JUMP_REG if GT
JPF_Z	Conditional jump forward	PC <= PC + JUMP_REG if Z
CMP	Compare	(GT, Z) <= ACC - TEMP
ADD	Addition	ACC <= ACC + TEMP
SUB	Subtraction	ACC <= ACC - TEMP

Figure 4: The instruction set of the microprocessor.

4 Implementation

4.1 Overview

The microprocessor is realized with a VHDL-project in the program Quartus Prime. For structuring purposes, the project comprises several files (both block diagram and VHDL files). These are: *main.bdf*, *ram.vhd*, *code_package.vhd*, *control_unit.vhd*, *decoder_package.vhd*, *alu.vhd*, *programcounter.vhd*, *address1.vhd*, *address2.vhd*, *hex_disp4.vhd*. To begin with, the whole design was built for simulation purposes, thereby waiting with the implementation of the hardware. The complete system with all its subsystems can be seen in figure 5. Worth noting at this point is the address bus (A[7..0]), data bus (D[7..0]), and control_bus[24..0] connected to many of the blocks. The hex_disp4 block is used later for displaying the busses on the development board's display.

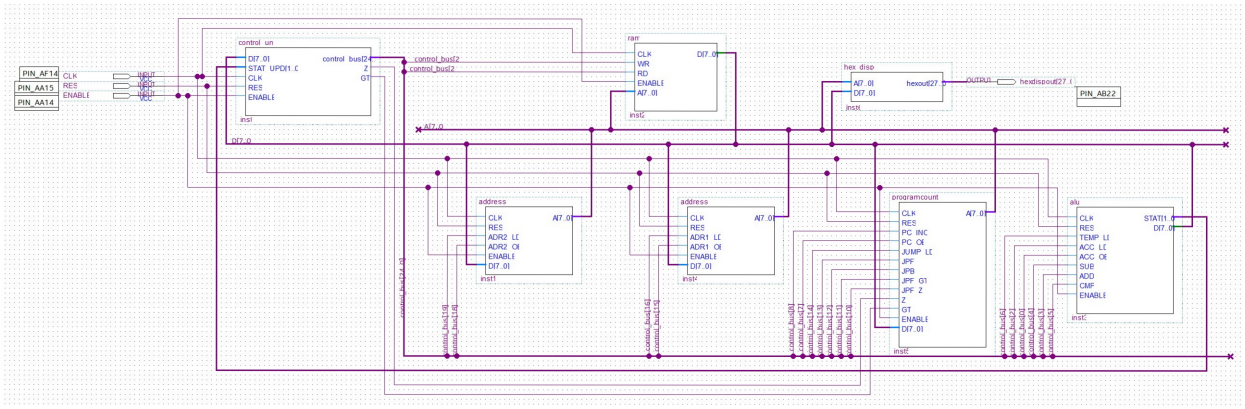


Figure 5: Block diagram from the file *main.bdf* showing the system as a whole.

4.2 RAM

A RAM-memory was the first part of the system to be created. Data on D[7..0] is written to the memory (at the address value on A[7..0]) when WR = 1. At the same time, this block gives the data port a high impedance state to let the rest of the system decide its value. This is to allow a bidirectional data port. When RD = 1, the module outputs the value in the memory (address decided by A[7..0]) on the data bus. Read is immediate and write synchronous. See code below (*ram.vhd*).

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.code_package.all;

entity ram is

        port
        (       CLK, WR, RD : in std_logic;
                A : in std_logic_vector(7 downto 0);
                D : inout std_logic_vector(7 downto 0));

end entity;

architecture behavioral of ram is
signal ram_array : ram_type := code4; --choose which code you want to run

begin
        process(CLK)
        begin
                if rising_edge(CLK) then
                        if WR='1' then
                                ram_array(conv_integer(A)) <= D;
                        end if;
                end if;
        end process;

        D <= ram_array(conv_integer(A)) when RD='1' else "ZZZZZZZZ";

end behavioral;
```

To pre-initialize the memory in the desired way, a package was created in the file *code_package.vhd* (see code below). Here, the ram_type is declared which is a memory that is 8-bit wide and 64-bit deep (arbitrarily chosen). The use of this package allows the creation of several ram_type constants making it easy to load the memory with different programs. In this example, the program loaded is the constant code4. As of now the program is not important and can be ignored. However, in case of curiosity, it does the following: if A>=0 then B<=C.

```
library ieee;
use ieee.std_logic_1164.all;

package code_package is
```

```
        --arbitrary deep memory
        type ram_type is array(63 downto 0) of std_logic_vector(7 downto 0);
        constant code4 : ram_type;

end code_package;

package body code_package is

                --Example project 2
                constant code4 : ram_type :=
                  (0 => x"C0", --Load ADR1 with adress 32
                   1 => x"20",
                   2 => x"C1", --Load ADR2 with adress 33
                   3 => x"21",
                   4 => x"80", --Load ACC with value on adress 32
                   5 => x"81", --Load TEMP with value on adress 33
                   6 => x"88", --CMP

                   7 => x"C2", --Load JUMP_REG with 5
                   8 => x"05",
                   9 => x"87", --JPF_Z
                   10 => x"C2", --Load JUMP_REG with 2
                   11 => x"02",
                   12 => x"86", --JPF_G

                   13 => x"00",
                   14 => x"00",

                   15 => x"C0", --Load ADR1 with adress 34
                   16 => x"22",
                   17 => x"C1", --Load ADR2 with adress 35 (B)
                   18 => x"23",
                   19 => x"80", --Load ACC with value on adress 34 (C)
                   20 => x"83", --ST ACC on adress 35 (B)

                   32 => x"08", --A = 8
                   33 => x"00", --Ref
                   34 => x"14", --C = 20

                   others => (others => '0'));

end code_package;
```

4.3 Control unit

The control unit is the brain of the system. It decodes instructions read from memory using the function *decoder()* in the file *decoder_package.vhd* (see code below). The output of the function is a 25-bit std_logic_vector (control_bus). All its signals can be seen in the package body below.

```
library ieee;
use ieee.std_logic_1164.all;

package decoder_package is

        function decoder(instruction : std_logic_vector(7 downto 0))
        return std_logic_vector;

end decoder_package;


package body decoder_package is

                function decoder(instruction : std_logic_vector(7 downto 0))
                return std_logic_vector is
                        variable temp_control_bus : std_logic_vector(24 downto 0)
                        := (others => '0');

                begin

                --control_bus: 24 RD, 23 WR, 22 Z, 21 GT, 20 DOUBLE_INS,
                --19 ADR2_LD, 18 ADR2_OE, 17 STOP, ADR1_LD, 15 ADR1_OE,
                --14 JUMP_LD, 13 JPF, 12 JPB, 11 JPF_GT, 10 JPF_Z, 9 -,
                --8 PC_INC, 7 PC_OE, 6 TEMP_LD, 5 CMP, 4 SUB, 3 ADD,
                --2 ACC_LD, 1 -, 0 ACC_OE
                --Double-length instructions start with 11
                --and single-length with 10.
        case instruction is

                when "11000000" => --LD_ADR1 hxC0, RD, ADR1_LD, PC_INC, PC_OE,
                                   --DOUBLE_INS
                        temp_control_bus := "10001000100000000110000000";
                when "11000001" => --LD_ADR2 hxC1, RD, ADR2_LD, PC_INC, PC_OE,
                                   --DOUBLE_INS
                        temp_control_bus := "10001100000000000110000000"; --
                when "11000010" => --LD_JUMPREG hxC2, RD, JUMP_LD, PC_INC, PC_OE,
                                   --DOUBLE_INS
                        temp_control_bus := "10001000001000000110000000";

                when "10000000" => --LD_ACC hx80, RD, ADR1_OE, ACC_LD
                        temp_control_bus := "10000000010000000000000100";
                when "10000001" => --LD_TEMP hx81, RD, ADR2_OE, TEMP_LD
                        temp_control_bus := "10000010000000000001000000";
                when "10000010" => --ST_ACC1 hx82, WR, ADR1_OE, ACC_OE
                        temp_control_bus := "01000000010000000000000001";
```

```
            when "10000011" => --ST_ACC2 hx83, WR, ADR2_OE, ACC_OE
                    temp_control_bus := "01000010000000000000000001";
            when "10000100" => --JPF hx84, (RD), JPF, PC_OE
                    temp_control_bus := "10000000000010000010000000";
            when "10000101" => --JPB hx85, (RD), JPB, PC_OE
                    temp_control_bus := "10000000000001000010000000";
            when "10000110" => --JPF_G hx86, (RD), JPF_GT, PC_OE
                    temp_control_bus := "10000000000000100010000000";
            when "10000111" => --JPF_Z hx87, (RD), JPF_Z, PC_OE
                    temp_control_bus := "10000000000000010010000000";
            when "10001000" => --CMP hx88, (RD), PC_OE, CMP
                    temp_control_bus := "10000000000000000010100000";
            when "10001001" => --ADD hx89, (RD), PC_OE, ADD
                    temp_control_bus := "10000000000000000010001000";
            when "10001010" => --SUB hx8A, (RD), PC_OE, SUB
                    temp_control_bus := "10000000000000000010010000";

            when others => --code loops when done
                    temp_control_bus := "10000001000000000010000000";
    end case;

                            return temp_control_bus;
            end function;

end decoder_package;
```

In the file *control_unit.vhd* (code below), the decoder function is used to decode the instruction on the data bus (input port D). It does this only during the FETCH cycle. To decide if the processor is in the FETCH or EXECUTE cycle, the FETCH signal is introduced. It's toggled each rising edge of the clock. If this signal is 1 at the rising edge of the clock, an instruction from memory is decoded and the control_bus is set, determining which operations to be performed during the EXECUTE cycle. If not 1, the control_bus is set to be able to read a new instruction after the EXECUTE cycle. At the same time, PC is incremented. The control_bus is set as a buffer to be able to read the STOP signal in the entity which is set to 1 when the instruction is 00000000. When this signal is 1, the PC is not incremented anymore which makes the code loop in the same place.

Furthermore, the input port STAT_UPD is connected to the alu. This signal is set when the compare operation (CMP) is performed in the alu. The output ports Z and GT follows the STAT_UPD signal and are connected to the program counter block. These signals are used in the instructions JPF_G and JPF_Z (conditional jump forward).

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.decoder_package.all;

entity control_unit is

        port
        (       D : in std_logic_vector(7 downto 0);
```

```
                STAT_UPD : in std_logic_vector(1 downto 0);
                CLK, RES: in std_logic;
                control_bus : buffer std_logic_vector(24 downto 0);
                Z, GT : out std_logic);

end entity;

architecture behavioral of control_unit is
signal STATUS : std_logic_vector(1 downto 0);
signal FETCH : std_logic := '0';

begin
process(CLK,RES)
begin
        if RES = '0' then
                --first two bits is RD and WR.
                --bit 7 is PC_OE, bit 8 is PC_INC
                control_bus <= "10000000000000000110000000";
                FETCH <= '1';
        elsif rising_edge(CLK) then
            if FETCH = '1' then
                control_bus <= decoder(D);
            else
                if control_bus(17) = '1' then
                    --do not increment PC if done
                    --control_bus(17) = STOP
                    control_bus <= "10000000000000000010000000";
                else
                    --just prepare to read instruction again.
                    --PC_INC = '1', PC_OE = '1'
                    control_bus <= "10000000000000000110000000";
                end if;
            end if;
            FETCH <= not FETCH;
        end if;

end process;
Z <= STAT_UPD(1);
GT <= STAT_UPD(0);


end behavioral;
```

4.4 Address registers

In this microcontroller, there are two equivalent address registers implemented: *address1.vhd* (code below) and *address2.vhd*. These registers are used to store addresses that are used as pointers for loading other registers with data from memory. When the signal ADR1_LD on the control_bus is set to 1, the ADR1 register is loaded with the data on D[7..0]. This loading operation is performed in a double-length instruction, so the value to be loaded is stored on the address after the load instruction itself. The first line efter the end process statement indicates that the ADR1 buffer is concurrent. When ADR1_OE is 1, the address stored in the register is extracted to the address bus for use in other operations. When ADR1_OE = 0, the address bus is set to a high impedance state from this module disallowing it to interfere when other operations are executed.

```
library ieee;
use ieee.std_logic_1164.all;

entity address1 is

        port
        (       CLK, RES, ADR1_LD, ADR1_OE : in std_logic;
                D : in std_logic_vector(7 downto 0);
                A : out std_logic_vector(7 downto 0));

end entity;

architecture behavioral of address1 is
signal ADR1 : std_logic_vector(7 downto 0);
signal ADR1_BUF : std_logic_vector(7 downto 0);

begin

        process(CLK,RES)
        begin
                if RES = '1' then
                        A <= "ZZZZZZZZ";
                elsif rising_edge(CLK) then
                        if ADR1_LD = '1' then
                                ADR1 <= D;
                        end if;
                end if;
        end process;
        ADR1_BUF <= ADR1;
        A <= ADR1_BUF when ADR1_OE='1' else "ZZZZZZZZ";

end behavioral;
```

4.5 Program counter

The program counter holds the address of the instruction executed at the time. It is incremented when an instruction is fetched. In the case of reset, it is set to 0, and the program restarts. The basic function of this submodule is when PC_INC on the control bus is 1, the register is incremented. PC_BUF and PC_OE on the address bus function identically as in the address registers.

Additionally, the program counter block handles jump operations which allows modification of the PC and thereby jumps in memory. The first step is to load an argument (size of jump) in JUMP_REG. This is a double-length instruction and works in the same way as the load address instructions. The second step is to increase or decrease the PC with the value in JUMP_REG based on the signals JPF, JPB, JPF_GT, JPF_Z, Z, and GT. Note that Z and GT are connected to the control unit.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity programcounter is

        port
        (       CLK, RES, PC_INC, PC_OE, JUMP_LD, JPF, JPB,
                JPF_GT, JPF_Z, Z, GT : in std_logic;
                D : in std_logic_vector(7 downto 0);
                A : out std_logic_vector(7 downto 0));

end entity;

architecture behavioral of programcounter is
signal JUMP_REG : std_logic_vector(7 downto 0);
signal PC : std_logic_vector(7 downto 0) := "00000000";
signal PC_BUF : std_logic_vector(7 downto 0) := "00000000";


begin
        process(CLK,RES)
        begin
                if RES = '1' then
                        PC <= "00000000";

                elsif rising_edge(CLK) then

                        if PC_INC = '1' then
                                PC <= PC + 1;
                        end if;

                        --JUMP load
                        if JUMP_LD = '1' then
                                JUMP_REG <= D;
                        end if;

                        --JUMP instructions
```

```
                        if JPF = '1' then
                                PC <= PC + JUMP_REG;
                        elsif JPB = '1' then
                                PC <= PC - JUMP_REG;
                        elsif JPF_GT = '1' then
                                if GT = '1' then
                                        PC <= PC + JUMP_REG;
                                end if;
                        elsif JPF_Z = '1' then
                                if Z = '1' then
                                        PC <= PC + JUMP_REG;
                                end if;
                        end if;

                end if;
        end process;

        PC_BUF <= PC;
        A <= PC_BUF when PC_OE='1' else "ZZZZZZZZ";

end behavioral;
```

4.6 Arithmetic logic unit (alu)

It is in the alu that all arithmetic and logic operations are performed. This microprocessor's alu contains two registers: ACC (accumulator) and TEMP_REG (temporary). After completing the operation on the values stored in the accumulator and the temporary register, the result is stored back in the accumulator. Note the use of the *ieee.numeric_std* library, since the alu needs to manage negative numbers. The data port is bidirectional since there is a need to be able to both load the two registers with data and output the accumulator value. A similar solution to the data port in the ram memory is implemented with the difference of adding a buffer. The ACC register will be loaded with the argument in memory pointed by the address in address register 1. Equivalently TEMP register will correspond to address register 2.

Now let's investigate the different operations. ADD will clearly add the values in the two registers and SUB will leave the accumulator with the difference: ACC - TEMP_REG. The CMP (compare) operation performs the operation ACC - TEMP_REG and checks whether the ACC value equals the value in TEMP_REG or if it is greater. If either of this happens the two-bit STAT vector will be updated. Since this port is connected to the control unit, it can in turn note the program counter when using conditional jumps. In microprocessors, this is usually called flag handling.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity alu is

        port
        (       CLK, RES, TEMP_LD, ACC_LD, ACC_OE, SUB, ADD, CMP : in std_logic;
                STAT : out std_logic_vector(1 downto 0);
                D : inout std_logic_vector(7 downto 0));

end entity;

architecture behavioral of alu is
signal TEMP_REG : signed(7 downto 0);
signal ACC : signed(7 downto 0);
signal ACC_BUF : signed(7 downto 0);

begin
        process(CLK,RES)
        begin
                if RES = '1' then
                        D <= "ZZZZZZZZ";
                elsif rising_edge(CLK) then
                        --Load register
                        if ACC_LD = '1' then
                                ACC <= signed(D);
                        elsif TEMP_LD ='1' then
                                TEMP_REG <= signed(D);
                        end if;

                        --Operations
                        if SUB = '1' then
```

```
                        ACC <= ACC - TEMP_REG;
                    elsif ADD = '1' then
                        ACC <= ACC + TEMP_REG;
                    end if;
                    if CMP = '1' then
                        if ACC = TEMP_REG then
                            STAT <= "10";
                        elsif ACC > TEMP_REG then
                            STAT <= "01";
                        end if;
                    end if;

                end if;
            end process;

            ACC_BUF <= ACC;
            D <= std_logic_vector(ACC_BUF) when ACC_OE = '1' else "ZZZZZZZZ";


end behavioral;
```

4.7 Hardware

Up until now, the code created is designed for simulation. To make the code compatible with the requirement specification and hardware a few things need to be added. Firstly, the address and data buses need to be displayed in hex on the development board's seven-segment displays. It is realised with the block *hex_ disp4* in figure 5, which is built up from 4 single seven-segment display function blocks. The code of one of these can be seen below. It converts a 4-bit vector signal to a 7-bit output for control of the display.

```
library ieee;
use ieee.std_logic_1164.all;

PACKAGE SevenSegmentDecoding is
        function to_SevenSegment (symbol: in std_logic_vector(3 downto 0))
        return std_logic_vector;
end PACKAGE SevenSegmentDecoding;

PACKAGE BODY SevenSegmentDecoding is
        function to_SevenSegment (symbol: in std_logic_vector(3 downto 0))
        return std_logic_vector is
                variable input : std_logic_vector(3 downto 0);
                variable output: std_logic_vector(6 downto 0);
        begin
        input := symbol;
                case input is

                        when "0000" => output := "1000000";
                        when "0001" => output := "1111001";
                        when "0010" => output := "0100100";
                        when "0011" => output := "0110000";
                        when "0100" => output := "0011001";
                        when "0101" => output := "0010010";
                        when "0110" => output := "0000010";
                        when "0111" => output := "1111000";
                        when "1000" => output := "0000000";
                        when "1001" => output := "0010000";
                        when "1010" => output := "0001000";
                        when "1011" => output := "0000011";
                        when "1100" => output := "1000110";
                        when "1101" => output := "0100001";
                        when "1110" => output := "0000110";
                        when "1111" => output := "0001110";
                        when others => output := "0110110";

                end case;

                return output;
        end function;

end PACKAGE BODY SevenSegmentDecoding;
```

```
library ieee;
use ieee.std_logic_1164.all;
use work.SevenSegmentDecoding.to_SevenSegment;

entity hex_display is

        port
        (       counts : in std_logic_vector(3 downto 0);
                displayout : out std_logic_vector(6 downto 0));

end entity;

architecture rtl of hex_display is
begin

        displayout <= to_SevenSegment(counts);

end rtl;
```

Moreover, the code on the microprocessor is required to have the ability to be stepped through for display and debugging purposes. It is impossible when the code is executed with the speed of the 50MHz clock. To realize this, an enable signal (connected to a physical active low button) is created in all blocks except the ram memory. When the button is pressed once, only one clock cycle of the code should be performed, independent of how long the button is pressed. The code below shows how the signal is implemented in the address register 1 block. For one clock cycle to be performed, the button need to be pressed and enablesignal need to be 1. The enablesignal is set to 0 right after this and the button needs to go back to a high state before another clock cycle is allowed to be run again.

```
library ieee;
use ieee.std_logic_1164.all;

entity address1 is

        port
        (       CLK, RES, ADR1_LD, ADR1_OE, ENABLE : in std_logic;
                D : in std_logic_vector(7 downto 0);
                A : out std_logic_vector(7 downto 0));

end entity;

architecture behavioral of address1 is
signal ADR1 : std_logic_vector(7 downto 0);
signal ADR1_BUF : std_logic_vector(7 downto 0);
signal enablesignal : std_logic := '1';

begin

        process(CLK,RES)
        begin
                if RES = '0' then
                        A <= "ZZZZZZZZ";
```

```
                enablesignal <= '1';
            elsif rising_edge(CLK) then
                if ENABLE = '0' and enablesignal = '1' then
                    enablesignal <= '0';
                    if ADR1_LD = '1' then
                        ADR1 <= D;
                    end if;
                elsif ENABLE = '1' then
                    enablesignal <= '1';
                end if;
            end if;
        end process;
        ADR1_BUF <= ADR1;
        A <= ADR1_BUF when ADR1_OE='1' else "ZZZZZZZZ";

end behavioral;
```

Lastly, all the signals connected to the buttons and display need to be assigned to the desired pins on the development board according to the DE1-SoC manual [3].

5 Test & Result

5.1 Simulations

All simulations were made in ModelSim. To test and show the workings of signals and instructions, several test programs were written. Below is the program that will be the example in this report (the constant code4 is preloaded into ram). It does the following: IF A>=0 THEN B<=C.

```
constant code4 : ram_type :=
  (0 => x"C0", --Load ADR1 with adress 32
   1 => x"20",
   2 => x"C1", --Load ADR2 with adress 33
   3 => x"21",
   4 => x"80", --Load ACC with value on adress 32
   5 => x"81", --Load TEMP with value on adress 33
   6 => x"88", --CMP

   7 => x"C2", --Load JUMP_REG with 5
   8 => x"05",
   9 => x"87", --JPF_Z
   10 => x"C2", --Load JUMP_REG with 2
   11 => x"02",
   12 => x"86", --JPF_G

   13 => x"00",
   14 => x"00",

   15 => x"C0", --Load ADR1 with adress 34
   16 => x"22",
   17 => x"C1", --Load ADR2 with adress 35 (B)
   18 => x"23",
   19 => x"80", --Load ACC with value on adress 34 (C)
   20 => x"83", --ST ACC on adress 35 (B)

   32 => x"08", --A = 8
   33 => x"00", --Ref
   34 => x"14", --C = 20

   others => (others => '0'));
```

To be able to test the design, the file *main.bdf* were converted into VHDL file: *main.vhd*. Therafter, a testbench was created. Below is the code added to the vht file. Note that the reset here is active high while it is active low on the hardware setup.

```
init : PROCESS
-- variable declarations
BEGIN
        RES <= '1';
        wait for 10 ns;
        RES <= '0';
```

```
    for i in 0 to 200 loop

        CLK <= '0';
        wait for 10 ns;
        CLK <= '1';
        wait for 10 ns;

    end loop;

WAIT;
END PROCESS init;
```

The first part of the simulation is shown in figure 6. Note that the memory is preloaded with A = 8 on address 32, reference value = 0 on address 33, and C = 20 on address 34. Reset is set in the begging for initialization. The FETCH signal is alternating between 0 and 1. It equals 0 during the FETCH cycle and 1 during the EXECUTE cycle. The important thing is that it is 1 exactly in the moment of the rising edge of the clock before the FETCH cycle. The program counter (PC_BUF) is counting up each clock cycle because of double-length instructions. It can be seen that the control bus goes back to its standard value during the executes value which prepares for reading of the next instruction (RD, PC_INC, and PC_OE are 1). Initially, ADR1 is loaded with address 32. The databus first contains 1100000 (hxCO) LD ADR1 instruction and then 00100000 (address 32) which is loaded into ADR1 register. When the decoding of the instruction is performed, the signals that are high on the databus are RD, DOUBLE_INS, ADR1_LD, PC_INC, and PC_OE. After that, the same procedure occurs with ADR2.

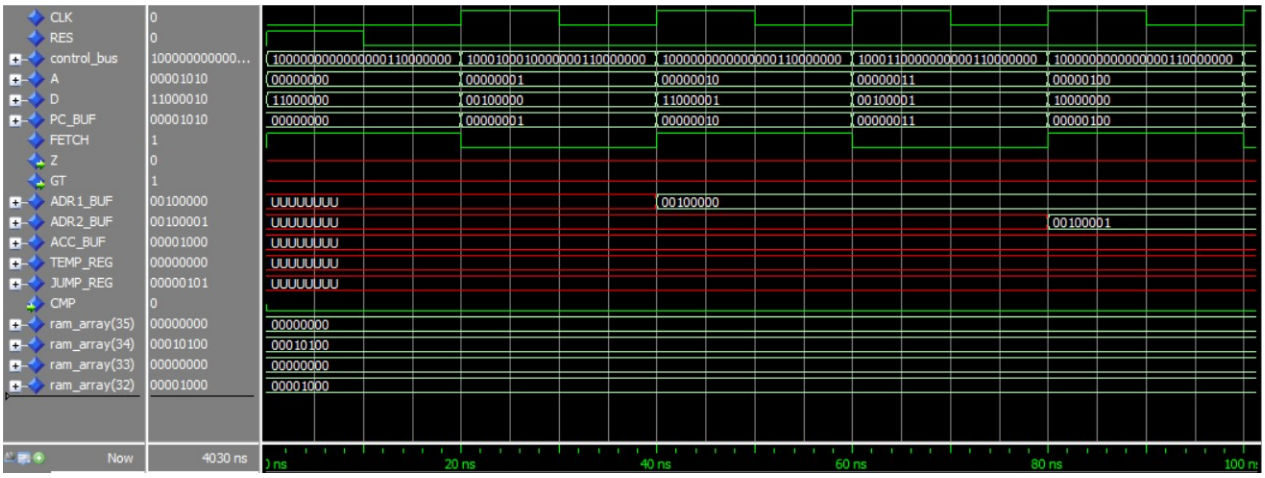


Figure 6: Simulation, 0-100ns.

The next instruction is LD ACC (see figure 7) which loads the accumulator with the value (8) pointed by the value in ADR1. Note that ADR1_OE is set high on the control bus. Hereafter, the same thing occurs with TEMP_REG and ADR2. Note that the PC only increments at the beginning of the FETCH cycle and not during the EXECUTE cycle since these are single-length instructions.

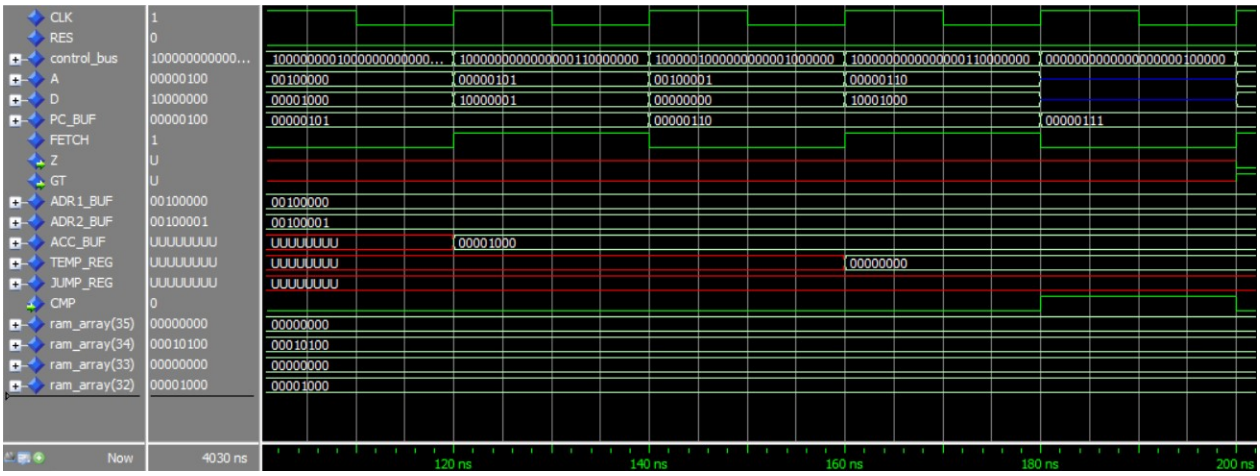


Figure 7: Simulation, 100-200ns.

Following the loading process of the registers in the alu, is the compare (CMP) operation. In figure 8 the signal GT has been set to 1 since A = 8 (ACC) is greater than 0 (TEMP_REG). The code is written so that lines 13 and 14 will be jumped over if A is greater or equal to 0. With that in mind, the next step is to prepare for a jump of the PC if the CMP operation would set the Z flag. JUMP_REG is loaded with 5. It is a double-length instruction so PC will increment both during the FETCH and EXECUTE cycle. After this the current instruction is JPF_Z. Since the Z flag is not set, nothing will happen during this clock cycle.

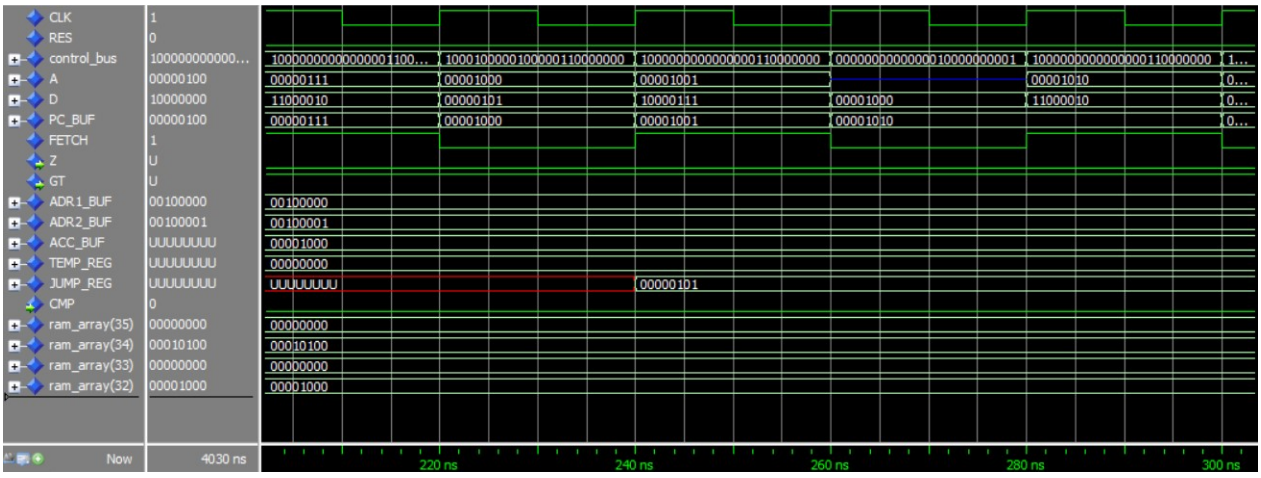


Figure 8: Simulation, 200-300ns.

In figure 9, the program on the microprocessor continues with a load of the JUMP_REG with 2 in case A is greater than 0. Since this is true, JPF_G is executed and 2 is added to the program counter, skipping addresses 13 and 14 in memory. If the control unit would be loaded with 00000000, the code would stop by continuously looping in the same place.

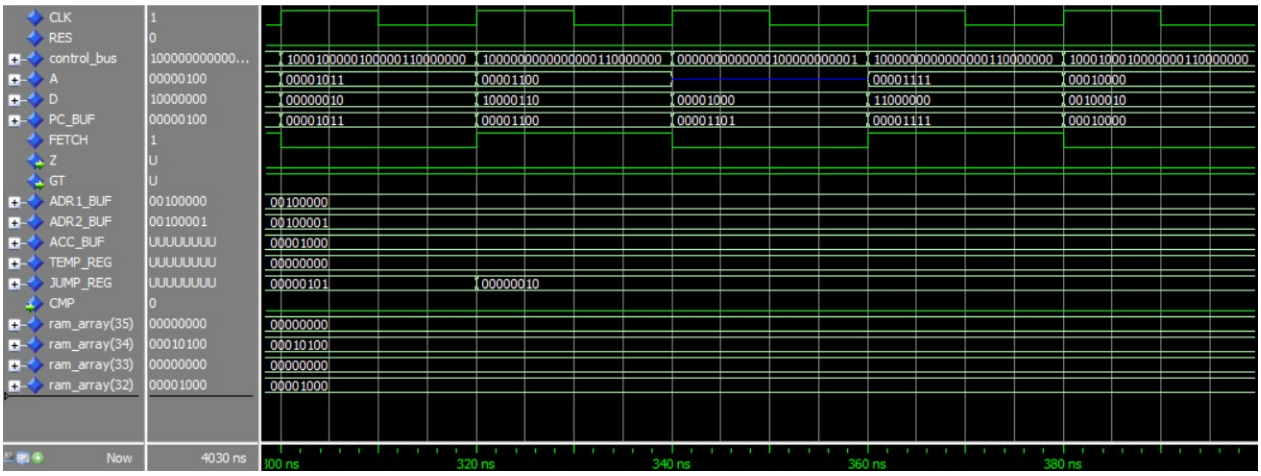


Figure 9: Simulation, 300-400ns.

The next phase in the code is to set B equal to C by storing the same value as C in address 35 (B) in memory. ADR1 is loaded with address 34 (where C is stored) and ADR2 with address 35 (where B are going to be stored). After this, the accumulator is loaded with C (20).

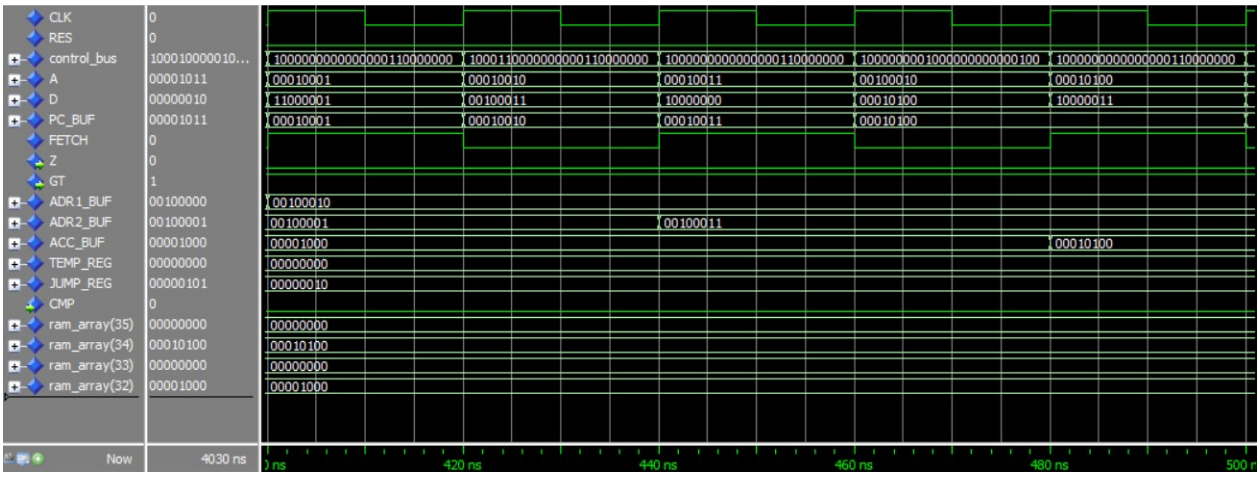


Figure 10: Simulation, 400-500ns.

Finally, 20 is stored in address 35 in memory making B equal to C (see figure 11). The last thing happening is that the instruction loaded is 00000000 and the incrementation of the program counter is put to a halt. The code will loop in the same place, stopping the code.

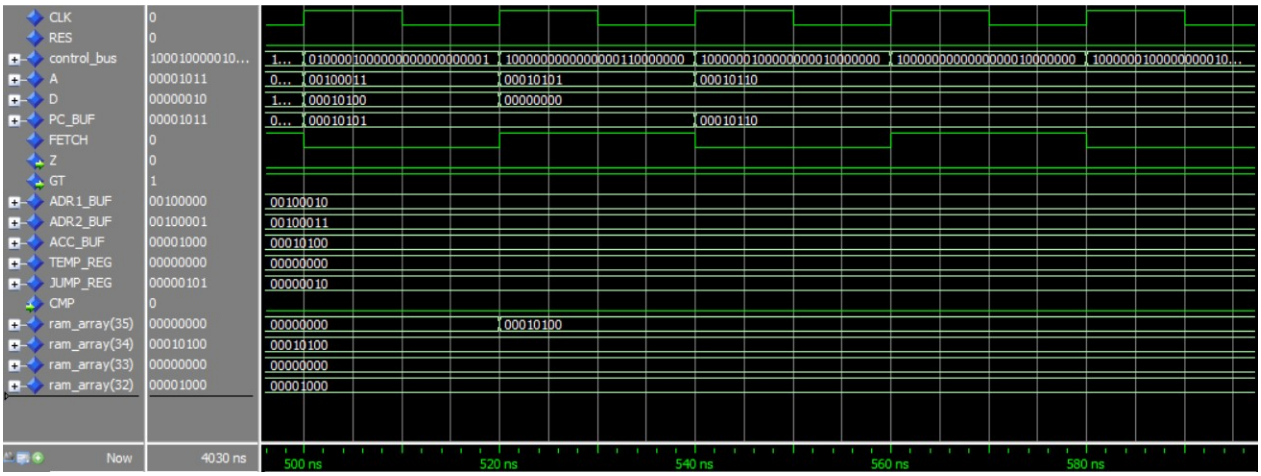


Figure 11: Simulation, 500-600ns.

5.2 Hardware

After the additions in the code created for operation on hardware, the code could be uploaded to the DE1-SoC. The microprocessor worked as intended. In figure 12, you can see the board with the RES and ENABLE buttons. By stepping through the program with the enable button, the address (two left digits) and the databus (two right digits) were displayed on the board. In this picture, the address bus displays the program counter = 10 and the instruction stored on that address in memory (hxC2 = LD JUMP_REG). In the case of reset, the code starts from the beginning (PC = 0).

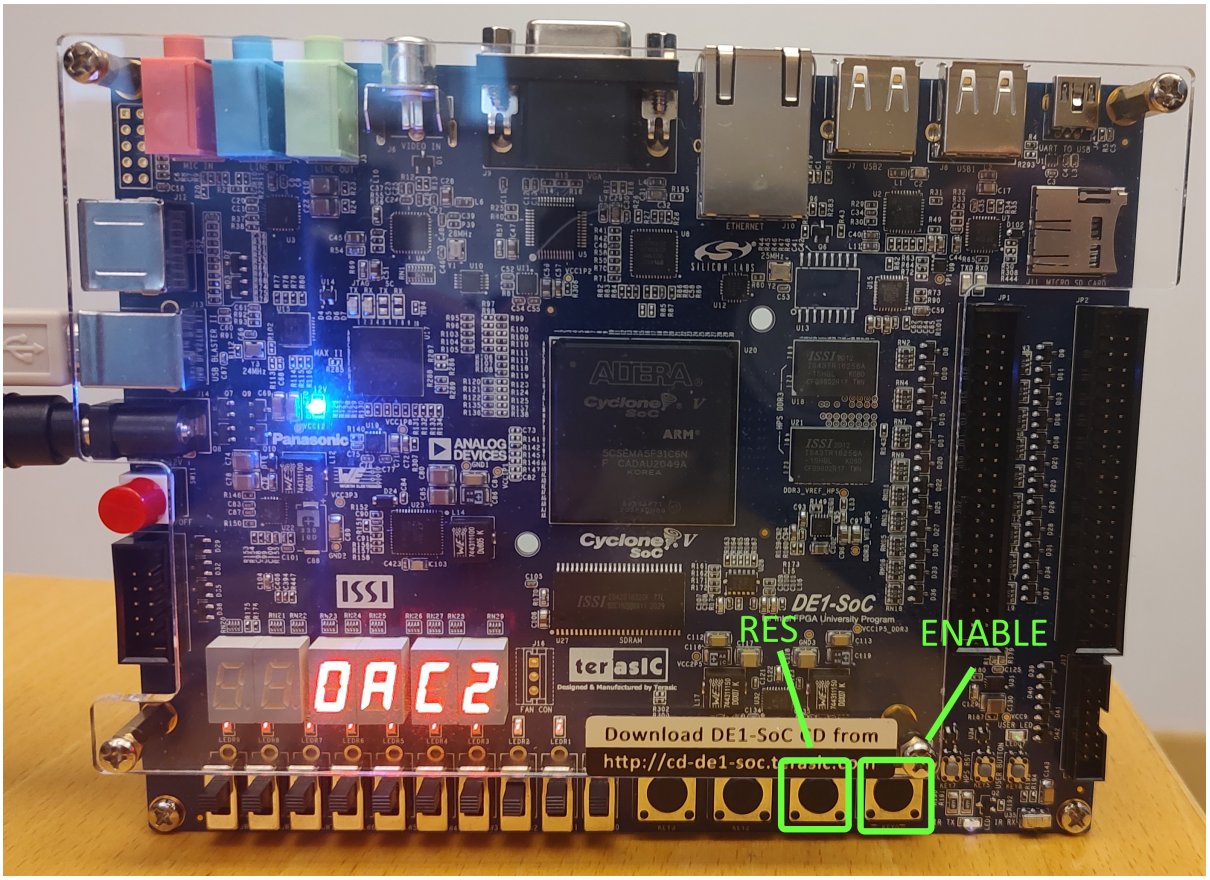


Figure 12: The designed microprocessor running on the DE1-SoC development board.

5.2.1 SignalTap

To further test the design, SignalTap was used to probe some of the inputs and outputs while the microprocessor runs on the FPGA. To be able to do this, the ENABLE signal is removed and RES is used to trigger the analysis. In figure 13, data acquired from SignalTap is displayed. The outputs are the segments on the 7-segment display showing the left digit on the address bus. In this particular case, the display goes from a 1 to a 0 which makes sense since the program counter goes from hx16 (end of code on microprocessor) to hx00 (beginning of code) when the RES button is pressed.

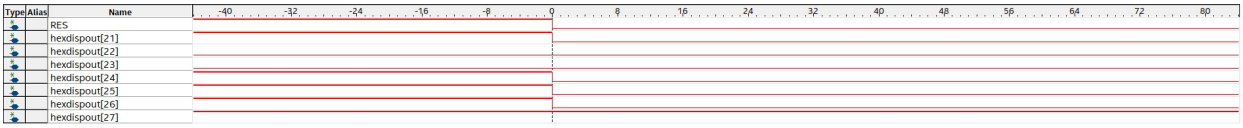


Figure 13: Data from SignalTap.

6 Conclusion and evaluation

The designed microprocessor meets the initial requirements with the complete instruction set in figure 4. The microprocessor has clear limitations but the instruction set could easily be extended. A few instructions that could be implemented for simplicity and performance are ACC INC (simplifies loop and comparison of arguments), conditional jump backward (good when using loops), and multiple move and store operations (making it possible to move and store arguments to different places in memory easily).

Implementing the ENABLE button and the display on the board makes it easy to troubleshoot the microprocessor. In addition, it makes it easier to get an understanding of the working of the design and microprocessors in general.

7 References

[1] K. Shirriff, "The surprising story of the first microprocessors," in IEEE Spectrum, vol. 53, no. 9, pp. 48-54, September 2016, doi: 10.1109/MSPEC.2016.7551353.

[2] Tutorialspoint. *Microprocessor - Overview* [online]. Accessed May 7, 2023, Availible: https://www.tutorialspoint.com/microprocessor/microprocessor_overview.htm

[3] Terasic Technologies, "DE1-SoC User Manual," April 2015