COL216 Lab Assignment-2 (S2)

- Mayank Mangla (2020CS50430)

1 Introduction

The aim of this project is to design hardware for implementing a processor that can execute a subset of ARM instructions described below. Starting with a skeleton design, the hardware is built in several stages, adding some functionality at every stage. The designs are to be expressed in VHDL and then simulated and synthesized.

Stage 2: In previous stage, many basic modules were built. Now, some other important components of a processor are designed and then connected together to form a single cycle processor. Program Counter, Flags and Associated circuit, condition checker and instruction decoder are designed and glue code is designed to connect these components together.

The report for the stage 2 is given below:

2 Program Information

This program has been test on "eda playground" using "-2019 -o" flags.
The program is in three parts: All designs, myTypes and glue code. Also there is a test bench for the processor and a run.do file used for synthesis.
All the files have some same part of the code:

```
library IEEE;
use IEEE.numeric_std.all;
use IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

These are the header files that are used in the program. Other than this, each file has an entity declaration and its architecture implementation.
Run.do file has a common code that is the command to synthesis the module.

```
setup_design -manufacturer Xilinx -family Artix-7 -part 7A100TCSG324
foreach arg $::argv {
  add_input_file $arg
}
compile
synthesize
auto_write precision.v
report_output_file_list
report_area
report_timing
#exec cat precision.v
```

Next is the part wise understanding of the code:

2.1 all_design

This module consists all the components' design: ALU, Condition checker, Data Memory, Flags, Instruction Decoder, Program Counter, Program Memory and Register File. Implementation of each component is described below.

2.1.1 ALU

This component does all the data processing instructions. This module responds to 16 different instructions.

```
entity ALU is
    port (
        op1,op2: in word;           -- 2 operands
        opc:in optype;              -- operation code
        cin: in std_logic;          -- carry in
        cout: out std_logic;        -- carry out
        result: out word;           -- final output
        msb1, msb2 : out std_logic  -- msb of the inputs
    );
end ALU;
```

The inputs to this are:

1. two operand, each 32 bit unsigned integer
2. 4 bit instruction
3. carry in (single bit)

The outputs are:

1. carry out (single bit)
2. result of the instruction on the two input operands giving 32 bit unsigned integer
3. most significant bit of the two operands that would be required to calculate the carry

The different instructions have different 4 bit standard op code that is followed by the arm instruction set. Accordingly we perform the operation on the operands.

```
architecture implement_alu of ALU is
begin
    process(op1,op2,cin,opc)    -- triggered by the change change in inputs
    variable temp: std_logic_vector(32 downto 0);   -- temporary variables
    variable tempop2: std_logic_vector(32 downto 0);
    variable tempop3: std_logic_vector(32 downto 0);
    variable tempop4: std_logic_vector(32 downto 0);
    begin
        tempop2 := (not op2) + 1;
        tempop3 := (not op2) + cin;
        tempop4 := op2 + cin;
        case(opc) is                        -- case switches for operations
            when andop => temp := (op1(31) & op1) and (op2(31) & op2);
            when eor => temp := (op1(31) & op1) xor (op2(31) & op2);
            when sub => temp := (op1(31) & op1) + (not (op2(31) & op2)) + 1;
            when rsb => temp := (op2(31) & op2) + (not (op1(31) & op1)) + 1;
            when add => temp := (op1(31) & op1) + (op2(31) & op2);
            when adc => temp := (op1(31) & op1) + (op2(31) & op2) + cin;
            when sbc => temp := (op1(31) & op1) + (not (op2(31) & op2)) + cin;
            when rsc => temp := (op2(31) & op2) + (not (op1(31) & op1)) + cin;
            when tst => temp := (op1(31) & op1) and (op2(31) & op2);
            when teq => temp := (op1(31) & op1) xor (op2(31) & op2);
            when cmp => temp := (op1(31) & op1) + (not (op2(31) & op2)) + 1;
            when cmn => temp := (op1(31) & op1) + (op2(31) & op2);
            when orr => temp := (op1(31) & op1) or (op2(31) & op2);
            when mov => temp := op2(31) & op2;
            when bic => temp := (op1(31) & op1) and (not (op2(31) & op2));
            when mvn => temp := (not (op2(31) & op2));
            when others => temp := "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        end case;
        result <= temp(31 downto 0);                -- updating the result
        cout <= temp(32);                                   -- carry out
        -- updating the msb of operands for different conditions
```

```
        if(opc = "0010" or opc = "0011" or opc = "1010") then msb2 <= tempop2(31);
        elsif (opc = "0110" or opc = "0111") then msb2 <= tempop3(31);
        elsif (opc = "0101") then msb2 <= tempop4(31);
        else msb2 <= op2(31);
        end if;
        msb1 <= op1(31);
    end process;                                        -- ending process
end implement_alu;                                    -- ending implementation
```

2.1.2 Condition Checker

This component checks for the condition is true or not by looking at the corresponding flags.

```
entity ConditionChecker is
    port (
        N_flag: in STD_LOGIC;               -- flags
        Z_flag: in STD_LOGIC;
        C_flag: in STD_LOGIC;
        V_flag: in STD_LOGIC;
        cond: in nibble;        -- input condition that is to be checked
        is_true: out std_logic              -- if the condition is true
    );
end ConditionChecker;
```

The inputs to this are:

1. the flag values
2. condition which is to checked

The outputs are:

1. boolean value that tells whether the condition is true or not

The condition and the flags are checked and accordingly the result is given. Required flags for the different condition is given in the lectures.

```
architecture implement_cc of ConditionChecker is
begin
    process(cond)         -- process on the change of flags or cond
    begin
        if(-- check for the condition value and the corresponding flags
            (cond="0000" and Z_flag='1') or     -- eq
            (cond="0001" and Z_flag='0') or     -- ne
            (cond="0010" and C_flag='1') or     -- hs/cs
            (cond="0011" and C_flag='0') or     -- lo/cc
            (cond="0100" and N_flag='1') or     -- mi
            (cond="0101" and N_flag='0') or     -- pl
            (cond="0110" and V_flag='1') or     -- vs
            (cond="0111" and V_flag='0') or     -- vc
            (cond="1000" and Z_flag='0' and C_flag='1') or  -- hi
            (cond="1001" and Z_flag='1' and C_flag='0') or  -- ls
            (cond="1010" and N_flag=V_flag) or  -- ge
            (cond="1011" and not(N_flag=V_flag)) or         -- lt
            (cond="1100" and Z_flag='0' and N_flag=V_flag) or -- gt
            (cond="1101" and Z_flag='1' and not(N_flag=V_flag)) or -- le
            (cond="1110")                       -- all flags ignored (al)
        ) then                  -- if they match then true is returned
            is_true <= '1';
        else                                    -- else false is returned
            is_true <= '0';
        end if;
    end process;
end implement_cc;
```

2.1.3 Data Memory

This component stores the data. This is different from program memory in a way that the data in data memory can be overwritten. This has one read and one write port.

```
entity DataMemory is
    port(
        add: in std_logic_vector(5 downto 0);-- read/write address
        clk:in std_logic;          -- clock
        data: in word;             -- input data
        wen: in nibble;            -- write enable
        dout: out word             -- output data
    );
end DataMemory;
```

The inputs to this are:

1. data that is to be written, 32 bit vector
2. address where the data is to be written and read, 6 bit unsigned number
3. clock (single bit)
4. a 4 bit write enable

The outputs are:

1. data that is being read, 32 bit vector

The different enables have different 4 bit code which is used to write in the memory to any one byte, half word or full word. Accordingly we write to the memory.

```
architecture implement_dm of DataMemory is
    type mem is array (0 to 63) of word; -- mem array of data
    signal memory: mem:= (others => (others => '0'));       -- signal for the data in data memory
begin
    process(clk)                                            -- process with change in clock
    begin
        -- calculate the index where the input data is to be written and write accordingly
        if(rising_edge(clk)) then   -- if the rising edge of the clock then we do as write enables
            if wen= "0001" then
                memory(to_integer(unsigned((add))))(7 downto 0) <= data(7 downto 0);
            elsif wen="0010" then
                memory(to_integer(unsigned((add))))(15 downto 8) <= data(7 downto 0);
            elsif wen="0100" then
                memory(to_integer(unsigned((add))))(23 downto 16) <= data(7 downto 0);
            elsif wen="1000" then
                memory(to_integer(unsigned((add))))(31 downto 24) <= data(7 downto 0);
            elsif wen = "0011" then
                memory(to_integer(unsigned((add))))(15 downto 0) <= data(15 downto 0);
            elsif wen = "1100" then
                memory(to_integer(unsigned((add))))(31 downto 16) <= data(15 downto 0);
            elsif wen = "1111" then
                memory(to_integer(unsigned((add)))) <= data ;
            end if;
        end if;
    end process;
    -- calculate the index that is to be read and read the data at that index
    dout <= memory(to_integer(unsigned((add))));
end implement_dm;
```

2.1.4 Flags

This component, on every clock, conditionally updates the Flags.

```
entity flags is
    port (
        set: in std_logic;                  -- if the flags are to be set
        clock: in std_logic;                -- clock
        dp_class : in DP_subclass_type;     -- type of dp instruction
        is_shift: in std_logic;             -- if shift or not
        carry_from_alu: in std_logic;       -- the carry form alu
        carry_from_shift: in std_logic;     -- carry from shift
        msb1, msb2: std_logic;              -- msb of the operands
        result: in word;                    -- result from the ALU
        out_flags: out nibble               -- outputed value of flags
    );
end flags;
```

The inputs to this are:

1. Set bit: whether to set the flags or not in some DP instructions
2. clock
3. Class of DP instruction it belongs to
4. if there is shift or not
5. carry from the ALU
6. carry from the shift
7. Most Significant Bits of the two operands from the ALU
8. Result from the ALU

The outputs are:

1. The updated value of the flags

Flags are update with respect to the inputs to the entity. Also this entity stores the value of the flags internally.

```
architecture implement_flags of flags is
    -- N Z C V locally store the values of the flags
    signal N_local, C_local,Z_local,V_local: std_logic:= '0';
begin
    process(clock)                          -- on the process of the clock
    begin
        if (rising_edge(clock)) then        -- on rising edge of the clock
            -- check for different instructions and set value and update the required flags
            if ((set = '1' and dp_class = arith) or (dp_class = comp) ) then
                if(result = X"00000000") then Z_local <= '1';
                else Z_local <= '0';
                end if;
                N_local <= result(31);
                C_local <= carry_from_alu;
                V_local <= (msb1 and msb2 and (not result(31))) or ((not msb1) and (not msb2) and res
            else
                if (set = '1' and is_shift = '1') then
                    if(result = X"00000000") then Z_local <= '1';
                    else Z_local <= '0';
                    end if;
                    N_local <= result(31);
                    C_local <= carry_from_shift;
```

```
                elsif ((set = '1' and is_shift = '0') or dp_class = test) then
                    N_local <= result(31);
                    if(result = X"00000000") then Z_local <= '1';
                    else Z_local <= '0';
                    end if;
                end if;
            end if;
        end if;
    end process;                            -- end the process
    -- assigning the new values to the output flags
    N_flag <= N_local;
    C_flag <= C_local;
    V_flag <= V_local;
    Z_flag <= Z_local;
end implement_flags;                        -- end the implementation
```

2.1.5 Instruction Decoder

This component looks on the 32 bit instruction and gives information about the instruction: class of instruction, registers involved, offset and its sign (if any), etc.

```
entity InstructionDecoder is
    port(
        instruction: in word;               -- input instruction
        -- output the various fields that are decoded from instruction code
        oper: out optype;                   -- dp operation
        ins_class: out instr_class_type;    -- instruction class
        dp_class: out DP_subclass_type;     -- if dp class then dp sub class
        dp_operand_src: out DP_operand_src_type;    -- source of the operands
        ls: out load_store_type;            -- load or store
        dt_offset_sign: out DT_offset_sign_type-- the offset needs to be added or subtracted
    );
end InstructionDecoder;
```

The inputs to this are:

1. the instruction from the Instruction Memory (Program Memory)

The outputs are:

1. The type of DP operation (if DP instruction)
2. The class of the instruction
3. The subclass of the DP instruction (if DP instruction)
4. The source of operands in the DP instruction
5. Whether to load or store in case of DT instruction
6. the sign of the offset in case of the DT instruction

This component looks at the various bits of the instruction and decides the type of instruction and various other fields.

```
architecture implement_id of InstructionDecoder is
    type oparraytype is array (0 to 15) of optype;  -- the array of different operations
    constant oparray : oparraytype := (andop, eor, sub, rsb, add, adc, sbc, rsc, tst, teq, cmp, cmn,
begin
    process (instruction)
        variable opc: std_logic_vector(2 downto 0); -- variable operation code
    begin
        opc := instruction(24 downto 22);
        if(instruction(27 downto 26)="00") then     -- DP or multiplication instruction
```

```
            oper <= oparray(to_integer(unsigned(instruction(24 downto 21))));
            if(instruction(7 downto 4)="1001") then -- multiplication instruction
                ins_class<= mul;
            else                                    -- else DP instruction
                ins_class<=DP;
            end if;
            if ( (opc="001") or (opc="010") or (opc="011") ) then   -- subclasses
                dp_class <= arith;          -- arithematic (add/sub/rsb/adc/sbc/rsc)
            elsif ( (opc="000") or (opc="110") or (opc="111") ) then
                dp_class <= logic;          -- logical (mov/mvn/and/orr/bic/eor)
            elsif (opc="101") then
                dp_class <= comp;           -- comparison (cmp/cmn)
            elsif (opc="100") then
                dp_class <= test;           -- test (tst/teq)
            else
                dp_class <= none;           -- else none
            end if;
            if (instruction(25)='0') then   -- operand source is immidiate or register
                dp_operand_src <= reg;
            else
                dp_operand_src <= imm;
            end if;
        elsif(instruction(27 downto 26)="01") then                  -- DT instructions
            ins_class <= DT;
            if(instruction(20)='1') then
                ls <= load;                 -- if the L bit is 1 then load
            else
                ls <= store;                -- else store
            end if;
            if(instruction(23)='1') then    -- add/sub the offset
                dt_offset_sign <= plus;
            else
                dt_offset_sign <= minus;
            end if;
        elsif(instruction(27 downto 26)="10") then                  -- branch instruction
            ins_class <=BRN;
        else
            ins_class <= none;              -- if nothing then none
        end if;
    end process;
end implement_id;
```

2.1.6 Program Counter

This component tells the program which instruction is to be implemented.

```
entity ProgramCounter is
    port (
        clock: in std_logic;                        -- clock
        -- the offset relative to pc in case of branch instruction to be implemented
        offset: in std_logic_vector(23 downto 0);
        branch: in std_logic;                       -- whether to branch or not
        rst: in std_logic;                          -- to reset the program to 0
        out_pc: out word                            -- output pc
    );
end ProgramCounter;
```

The inputs to this are:

1. clock
2. offset relative to current pc in case of branch instruction to be implemented

3. whether to branch
4. reset bit

The outputs are:

1. new pc value

On every clock, it updates the pc by pc+4 or pc+offset+4, depending on whether to branch or not

```
architecture implement_pc of ProgramCounter is
    signal pc: word:= X"00000000";              -- locally storing the value of pc
begin
    process(clock, rst)
    begin
        if(rising_edge(clock)) then
            if(rst='1') then
                out_pc <= X"00000000";
                pc <= X"00000000";
            elsif(branch='1') then                  -- if we need to branch
                if(offset(23) = '0') then
                    out_pc <= std_logic_vector(signed(pc) + signed("000000"&offset&"00") + 8);-- ther
                    pc <= std_logic_vector(signed(pc) + signed("000000"&offset&"00") + 8);
                else
                    out_pc <= std_logic_vector(signed(pc) + signed("111111"&offset&"00") + 8);-- ther
                    pc <= std_logic_vector(signed(pc) + signed("111111"&offset&"00") + 8);
                end if;
            else
                out_pc <= std_logic_vector(4 + signed(pc)); -- if no branch instruction then increme
                pc <= std_logic_vector(signed(pc) + 4);
            end if;
        end if;
    end process;
end implement_pc;
```

2.1.7 Program Memory

This component contains all the instructions stored in the form of 32 bit number each. We read the data from the this component and perform the instruction that is read. This is Read Only Memory and hence there is no port for writing into this memory.

```
entity ProgramMemory is
    port(
        radd: in std_logic_vector(5 downto 0);             -- reading address
        dout: out word           -- the data that is read
    );
end ProgramMemory;
```

The inputs to this are:

1. one address of the register of which the data is to be read (4 bit unsigned integer)

The outputs are:

1. one data vector that has been read from the addresses (32 bit vector)

Here we have a memory data type that is an array of length 64 each element is of 32 bits *(This memory s hard coded in the code, can be modified according to the testing by changing its initialization in the code).*

```
architecture implement_pm of ProgramMemory is
    type mem is array (0 to 63) of word;-- mem is array of instruction in program memory
    signal memory: mem;             -- signal that is accessing the instruction array
begin
    process(radd)                           -- process on read address
    begin
        dout <= memory(to_integer(unsigned(radd))); -- getting the data at the given address
    end process;
end implement_pm;
```

2.1.8 Register File

This component contains all the registers' addresses. This component enables us to read and write data from or to registers.

```
entity RegisterFile is
    port (
        rad1, rad2, wad: in nibble;     -- read and write addresses
        data: in word;                  -- data that is to be written
        wen,clk: in std_logic;          -- write enable and clock
        dout1,dout2: out word           -- data that is being read
    );
end RegisterFile;
```

The inputs to this are:

1. two addresses of the register of which the data is to be read, each 4 bit unsigned integer
2. one address of the register which is to be overwritten, each 4 bit unsigned integer
3. clock and write enable (single bit)

The outputs are:

1. two data vectors that have been read from the two addresses (32 bit each)

Here we have a memory data type that is an array of length 16 each element is of 32 bit. It stores the registers data.Then we read the data ignoring the condition of the clock. Then on the rising edge of the clock we write the data on the memory

```
architecture implement_rf of RegisterFile is
    type mem is array (0 to 15) of word;
    -- defining a type that is array of 16 std_logic_vector (32 bits)
    signal memory: mem:=(others => (others => '0'));    -- internal signal that is of mem type
begin
    dout1 <= memory(to_integer(unsigned(rad1)));        -- reading the register and outputing the sa
    dout2 <= memory(to_integer(unsigned(rad2)));        -- without any intervention of clock
    process(clk)                                        -- process when clock or writing is triggere
    begin
        if (rising_edge(clk)) then                      -- at rising edge of the clock
            if wen = '1' then                           -- and write enable set
                memory(to_integer(unsigned(wad))) <= data;-- write the data to the register address
            end if;
        end if;
    end process;                                        -- end process
end implement_rf;                                       -- end implementation
```

2.2 myTypes

This is a class of some data types defined to make our work easier and for better understanding.[1]

```
package MyTypes is
        subtype word is std_logic_vector (31 downto 0);
        subtype hword is std_logic_vector (15 downto 0);
        subtype byte is std_logic_vector (7 downto 0);
        subtype nibble is std_logic_vector (3 downto 0);
        subtype bit_pair is std_logic_vector (1 downto 0);
        type optype is (andop, eor, sub, rsb, add, adc, sbc, rsc,
                 tst, teq, cmp, cmn, orr, mov, bic, mvn);
        type instr_class_type is (DP, DT, MUL, BRN, none);
        type DP_subclass_type is (arith, logic, comp, test, none);
        type DP_operand_src_type is (reg, imm);
```

[1] This has been taken from program given with assignment

```
        type load_store_type is (load, store);
        type DT_offset_sign_type is (plus, minus);
end MyTypes;
package body MyTypes is
end MyTypes;
```

The data types that are defined in the file are:

- word: it is a vector of std_logic of length 32 (31 downto 0). All the data and instructions use this data type.
- hword: it is a vector of std_logic of length 16 (15 downto 0).
- byte: it is a vector of std_logic of length 8 (7 downto 0).
- nibble: it is a vector of std_logic of length 4 (3 downto 0). Some enables and all register numbers are represented using this data type.
- bit_pair: it a pair of std_logic (1 downto 0).
- optype: this is used to represent the type of DP instruction (out of 16)
- instr_class_type: type of instruction
- DP_subclass_type: Type of DP instruction
- DP_operand_src_type: the operand value is stored in register or is passed as immediate value
- load_store_type: load or store
- DT_offset_sign_type: the offset needs to be added or subtracted

2.3 Glue Code

This is the glue code for the processor that connects all the components to execute the single cycle processor.The entity of the glue code can be defined as

```
entity GlueCode is
    port(
        reset, clock: in std_logic
    );
end GlueCode;
```

It takes the input as clock that is synchronized with all the components.
The architecture imports all the components and connects all the components. The connections are show as:

```
begin
    -- map the ports that does not include the multiplexers
    arithematic: ALU port map(d1, b_in, op_mod, C, c2_out, d_out, ms1, ms2);
    condition: ConditionChecker port map(N, Z, C, V, instr(31 downto 28), output_bool);
    data: DataMemory port map(d_out(5 downto 0), clock, d2, mw, data_out);
    four_flags: flags port map(s, clock, dpc,'0', c2_out,'0', ms1, ms2, d_out, N, Z, C, V);
    decoder: InstructionDecoder port map(instr, op, insc, dpc, dpos, load_store, dtos);
    pcounter: ProgramCounter port map(clock, instr(23 downto 0), br, reset, pcout);
    program: ProgramMemory port map(pcout(7 downto 2), instr);
    registerf: RegisterFile port map(instr(19 downto 16), r2, instr(15 downto 12), d, s2, clock, d1,
    process(dpos, insc, d2, instr, d_out, data_out, load_store, output_bool, clock)
    begin
        -- defining the second input of the ALU
        if(dpos = reg) then b_in <= d2; -- in case of operand from register
        else b_in <= "00000000000000000000"&instr(11 downto 0); -- if immediate
        end if;
        -- defining the read address port 2 in the register file
        if (insc = DP) then r2 <= instr(3 downto 0);-- if the instruction is DP then read the operan
```

```
        else r2 <= instr(15 downto 12); -- else read the destination
        end if;
        -- defining the data that is to be written in register file
        if(insc=DT) then d <= data_out; -- data from data memory in case of date transfer instructio
        else d <= d_out;        -- else data from data memory
        end if;
        -- write enable of the data memory (only word write is there)
        if(load_store=load) then mw <= "0000";  -- in case of the load instruction don't write
        else mw <= "1111";      -- else write
        end if;
        -- set bit in flags
        if(insc=DP) then s <= instr(20);    -- if DP instruction then 20th bit
        else s <= '0';                      -- else set is 0
        end if;
        -- defining the write enable of the register file
        if(insc=DT) then s2 <= instr(20);   -- if DT then 20th bit
        elsif (insc=DP) then                -- if DP then
            if(dpc=comp) then               -- if comp then 0
                s2 <= '0';
            else                            -- else 1
                s2 <= '1';
            end if;
        else s2 <= '0';
        end if;
        -- setting the branch value of pc
        if(insc=BRN) then br <= output_bool; -- if branch instruction and condition is true
        else br <= '0';                     -- then branch else 0
        end if;
        -- operation being performed in the ALU
        if(insc =DP) then   -- if DP instruction then
            op_mod <= op;   -- the operation decided by decoder
        elsif(dtos = plus) then -- else if the offset is to be added then
            op_mod <= add;  -- add operation
        else                -- else the offset is to be subtracted
            op_mod <= sub;  -- => sub operation
        end if;
    end process;
end implement_gc;
```

3 Testing

3.1 Test Bench

The code for testing the program is given here:[2]

```
architecture implement_tb of processor_tb is
    component GlueCode is
        port(
            reset, clock: in std_logic
        );
    end component;
    signal clk,rst: std_logic:='0';
begin
    gc: GlueCode port map(rst,clk);
    process
    begin
        wait for 1 ns; -- starting
        clk <= '1';     -- initialize the clock
```

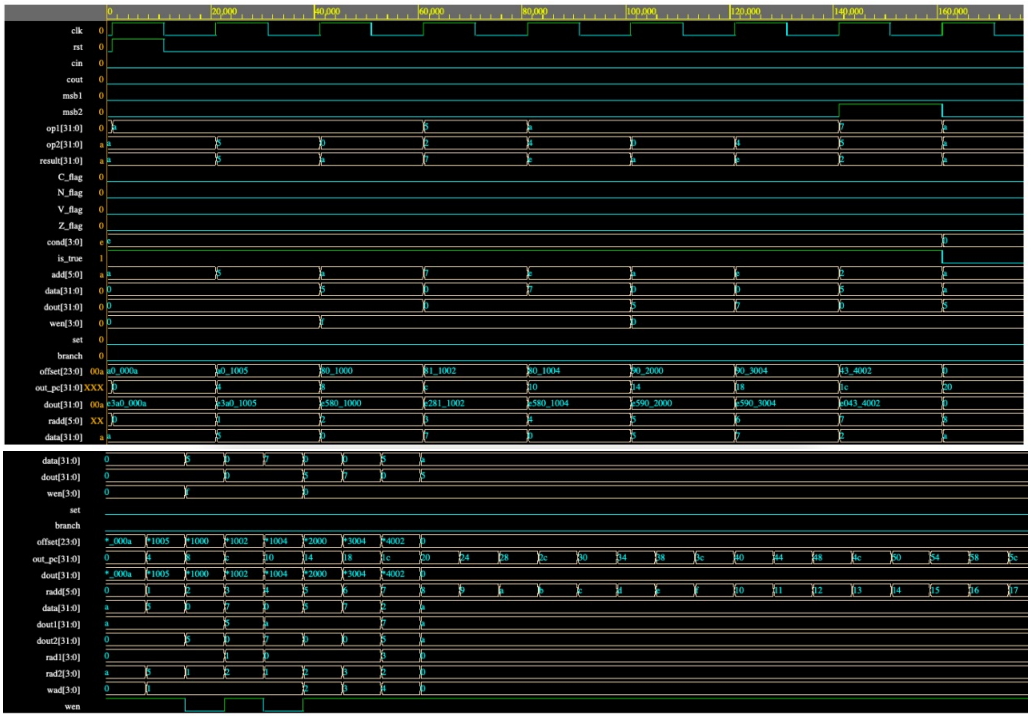
[2]Instructions are hard coded in the program memory

```
        rst<= '1';      -- and reset the pc
        wait for 10 ns;
        rst<= '0';      -- turn off the reset
        for i in 0 to 45 loop  -- loop till all the instructions are over
            clk <= not clk; -- in each loop reverse the clock
            wait for 10 ns; -- after 10 ns
        end loop;
        wait;
    end process;
end implement_tb;
```

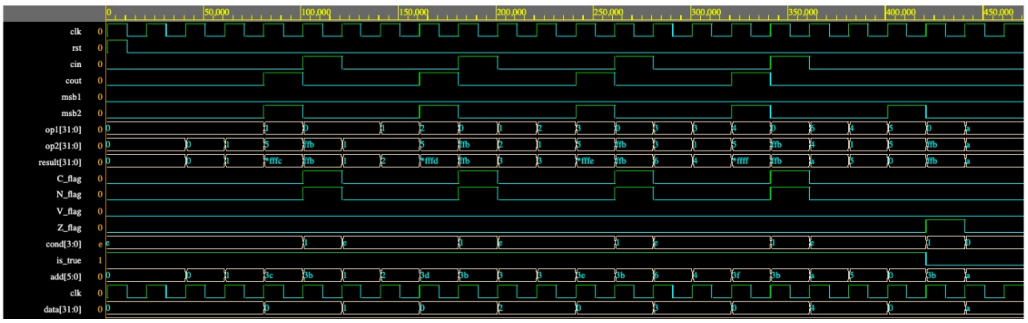
3.2 Resulting EP Wave

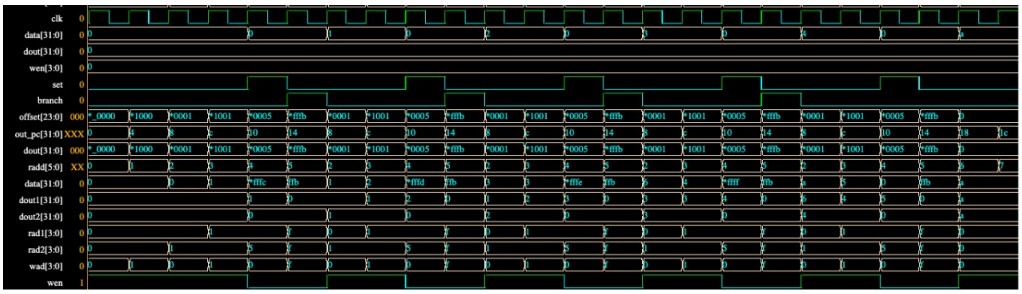
This test bench has been run on two sample inputs (given in the assignment pdf). The output signals of the both are given below:

1. Non - loop instruction set



2. Loop instruction set





3.3 Logs

The logs of the program are:

```
# Info: ************************************************************
# Info: Device Utilization for 7A100TCSG324
# Info: ************************************************************
# Info: Resource                          Used    Avail   Utilization
# Info: ------------------------------------------------------------
# Info: IOs                               2       210       0.95%
# Info: Global Buffers                    0       32        0.00%
# Info: LUTs                              0       63400     0.00%
# Info: CLB Slices                        0       15850     0.00%
# Info: Dffs or Latches                   0       126800    0.00%
# Info: Block RAMs                        0       135       0.00%
# Info: DSP48E1s                          0       240       0.00%
# Info: ------------------------------------------------------------
# Info: ************************************************************
# Info: Library: work    Cell: GlueCode    View: implement_gc
# Info: ************************************************************
# Info:  Number of ports :                          2
# Info:  Number of nets :                           0
# Info:  Number of instances :                      0
# Info:  Number of references to this view :        0
# Info: Total accumulated area :
# Info:  Number of gates :                          0
# Info:  Number of accumulated instances :          0
# Info: ************************************************************
```

4 Conclusion

The program has given the correct result for the above test cases. The submission file (.zip) contains 12 files other than this report file.
Eight files each one for one component (e.g. alu_desgin.vhd), one contains the glue code (glue_code.vhd), one contains the test bench for the program (testbench.vhd), one contains the data structures (given along with the assignment) and one is run.do file that has been used to synthesis.