COL216 Lab Assignment-2 (S3)

- 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 3: In previous stage, we constructed a single cycle processor. Now, we design a multi-cycle processor for the same set of instructions.

The report for the stage 3 is given below:

2 Program Information

This program has been test on "eda playground" using "-2019 -o" flags.
The program is in four parts: All designs, myTypes, glue code and the FSM. 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 work.MyTypes.all;
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 the designs

We have shown the implementation of all the components in previous stage. We have some changes in two components: Program Counter and Memories.
In Program counter, we return the input PC except when the reset is on at rising edge of the clock.
Also, the memories are now combined together. This means we have only one component for the program and data memory.
The changes in components are given below.

2.1.1 Program Counter

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

```
entity ProgramCounter is
    port (
        clock, reset: in std_logic;                 -- clock
        input_to_pc: in word;                       -- input pc
        output_from_pc: out word                    -- output pc
    );
end ProgramCounter;
```

The inputs to this are:

1. clock
2. current pc
3. reset bit

The outputs are:

1. new pc value

On every clock, it gives the output pc same as input pc or 0 in case of reset.

```
architecture implement_pc of ProgramCounter is
begin
    process(clock, reset)
    begin
        if(rising_edge(clock)) then
            if(reset='1') then          -- if reset then reset to 0
                output_from_pc <= X"00000000";
            else                        -- else output the input pc
                output_from_pc <= input_to_pc;
            end if;
        end if;
    end process;
end implement_pc;
```

2.1.2 Memory

This component stores the data. It has two memory arrays. One for Program Data and one for Data. It implements both the memories using a MUX as input

```
entity Memory is
    port(
        clock, is_instr:in std_logic;        -- clock
        address_to_mem: in std_logic_vector(5 downto 0);-- read/write address
        input_to_mem: in word;              -- input data
        mem_write_enable: in nibble;        -- write enable
        output_from_mem: out word           -- output data
    );
end Memory;
```

The inputs to this are:

1. clock (single bit)
2. a Boolean that tells if we need program or data
3. address to memory
4. data that is to be written, 32 bit vector
5. 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. We add an if-else statement that uses the boolean value and gives us the data from corresponding memory.

```
architecture implement_dm of Memory is
    -- signal for the data in data memory
    type data_mem is array (0 to 63) of word; -- mem array of data
    signal data_memory: data_mem:= (others => (others => '0'));
    -- signal for the data in program memory
    type prog_mem is array (0 to 63) of word; -- mem array of data
    signal program_memory: prog_mem:= (others => (others => '0'));
begin
    process(clock)                          -- process with change in clock
    begin
        -- calculate the index where the input data is to be written and write accordingly
        -- if the rising edge of the clock then we do as write enables
        if(rising_edge(clock)) then
            if mem_write_enable= "0001" then
                data_memory(to_integer(unsigned(address_to_mem)))(7 downto 0)
                <= input_to_mem(7 downto 0);
            elsif mem_write_enable="0010" then
                data_memory(to_integer(unsigned(address_to_mem)))(15 downto 8)
                <= input_to_mem(7 downto 0);
            elsif mem_write_enable="0100" then
                data_memory(to_integer(unsigned(address_to_mem)))(23 downto 16)
                <= input_to_mem(7 downto 0);
            elsif mem_write_enable="1000" then
                data_memory(to_integer(unsigned(address_to_mem)))(31 downto 24)
                <= input_to_mem(7 downto 0);
            elsif mem_write_enable = "0011" then
                data_memory(to_integer(unsigned(address_to_mem)))(15 downto 0)
                <= input_to_mem(15 downto 0);
            elsif mem_write_enable = "1100" then
                data_memory(to_integer(unsigned(address_to_mem)))(31 downto 16)
                <= input_to_mem(15 downto 0);
            elsif mem_write_enable = "1111" then
                data_memory(to_integer(unsigned(address_to_mem))) <= input_to_mem ;
            end if;
        end if;
    end process;
    -- calculate the index that is to be read and read the data at that index
    process(address_to_mem, is_instr)   -- process on change in address to memory
    begin
        if(is_instr='0') then       -- if not instruction
            output_from_mem <= data_memory(to_integer(unsigned(address_to_mem)));
        else                        -- else from data
            output_from_mem <= program_memory(to_integer(unsigned(address_to_mem)));
        end if;
    end process;
end implement_dm;
```

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);
        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 Finite State Machine

This component gives the control signals that are used to activate the various components in glue code depending on the state of the system. The entity of the Finite state machine can be defined as

```
entity FiniteStateMachine is
    port(
        clock, branch_cond: std_logic; -- clock and the boolean for condition being true
        decoded_dpos: in DP_operand_src_type;   -- source of the operand in DP instruction
        decoded_insc: in instr_class_type;      -- instruction class
        decoded_opc: in optype;                 -- the operation code in case of DP
        decoded_dtos: in DT_offset_sign_type;   -- the sign of offset in DT instruction
        decoded_load_store: in load_store_type; -- load or store
        -- control signals
        PW, IorD, MW, IW, DW, M2R, Rsrc, BW, RW, AW, Asrc1, F_set, ReW: out std_logic;
        Asrc2: out std_logic_vector(1 downto 0);
        alu_opc: out optype -- the operation that is to be performed by ALU
    );
end FiniteStateMachine;
```

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

The inputs to this are:

1. clock (single bit)
2. a Boolean that tells if the branch condition is true
3. source of the operand iin DP instruction
4. instruction class
5. operation that is given in DP instruction
6. sign of the offset in case of DT instruction
7. load or store

The outputs are:

1. the different control signals
2. the operation that the alu needs to operate on the operands

The architecture of the Finite state machine can be shown as

```
architecture implement_fsm of FiniteStateMachine is
    -- internal signal used for storing the next state
    signal next_state: nibble:= (others => '0');
begin
    process(clock)                                      -- process of clock
    begin
        -- we update the control signals according to the state
        if(rising_edge(clock)) then                     -- on rising edge
            if(next_state="0000") then          -- if on state 0 then
                next_state <= "0001";-- next state
                PW <= '1';          -- write PC+4 in PC
                IW <= '1';          -- write in instruction register
                alu_opc <= add;     -- add operation in ALU
                Asrc1 <= '0';       -- PC as operand 1
                Asrc2 <= "01";      --  4 as operand 2
                -- else 0
                MW <= '0'; DW <= '0'; BW <= '0'; RW <= '0'; AW <= '0'; M2R <= '0';
                ReW <= '0'; IorD <= '0'; Rsrc <= '0'; F_set <= '0';
            elsif(next_state="0001") then          -- if on state 1 then
                -- check for the type of instruction
                if(decoded_insc=DP) then    -- if DP then the next state is 2
                    next_state <= "0010"; Rsrc <= '0';
                elsif(decoded_insc=DT) then-- if DT then the next state is 3 also the Rsrc is 1
                    next_state <= "0011"; Rsrc <= '1';
                else    -- if BRN then the next state is 4 also the Rsrc is 1
                    next_state <= "0100"; Rsrc <= '1';
                end if;
                AW <= '1';      -- write in A
                BW <= '1';      -- write in B
                -- else 0
                PW <= '0'; IW <= '0'; MW <= '0'; DW <= '0'; RW <= '0'; M2R <= '0';
                ReW <= '0'; IorD <= '0'; F_set <= '0'; Asrc1 <= '0'; Asrc2 <= "00";
            elsif(next_state="0010") then          -- if on state 2 then
                next_state <= "0101"; -- next state
                Asrc1 <= '1';   -- A as operand 1
                F_set <= '1';
                ReW <= '1';     -- write in RES
                if(decoded_dpos = reg) then -- check for the source of the operand 2
                    Asrc2 <= "00";
                else Asrc2 <= "10"; end if;
                -- in this case the alu operation is same that is decoded from the instruction
                alu_opc <= decoded_opc;
                -- else 0
                PW <= '0'; IW <= '0'; MW <= '0'; DW <= '0'; BW <= '0';
                RW <= '0'; AW <= '0'; M2R <= '0'; IorD <= '0'; Rsrc <= '0';
            elsif(next_state="0011") then          -- if on state 3 then
                Rsrc <= '1';        -- read address 2 in register file
                Asrc1 <= '1';       -- A as operand 1
                ReW <= '1';     -- write in RES
                Asrc2 <= "10";      -- second operand as the offset in instruction
                if(decoded_dtos = minus) then   -- check the sign for offset
                    alu_opc <= sub;
                else alu_opc <= add; end if;
                if(decoded_load_store=load) then -- check load store for next state
                    next_state <= "0111";
                else next_state <= "0110"; end if;
                -- else 0
                PW <= '0'; IorD <= '0'; MW <= '0'; IW <= '0'; DW <= '0';
                M2R <= '0'; BW <= '0'; RW <= '0'; AW <= '0'; F_set <= '0';
            elsif(next_state="0100") then          -- if on state 4 then
                next_state <= "0000";   -- next state
                if(branch_cond='1') then -- if the branch condition is true then
                    PW <= '1';              -- we write in PC else don't
                else PW <= '0'; end if;
                alu_opc <= adc;         -- add with carry is to be performed
                Asrc2 <= "11";      -- second operand is offset
                Rsrc <= '1';        -- read address 2 in register file
                -- else 0
                IorD <= '0'; MW <= '0'; IW <= '0'; DW <= '0'; M2R <= '0';
                BW <= '0'; RW <= '0'; AW <= '0'; Asrc1 <= '0'; F_set <= '0';
                ReW <= '0';
            elsif(next_state="0101")then          -- if on state 5 then
                next_state <= "0000";       -- next state
                RW <= '1';               -- write the output to the register
                -- else 0
                PW <= '0'; IorD <= '0'; MW <= '0'; IW <= '0'; DW <= '0';
                M2R <= '0'; Rsrc <= '0'; BW <= '0'; AW <= '0';
                Asrc1 <= '0'; F_set <= '0'; ReW <= '0'; Asrc2 <= "00";
            elsif(next_state="0110") then          -- if on state 0 then
                next_state <= "0000";   -- next state
                IorD <= '1';    -- input the data memory address
                MW <= '1';      -- write in mem
                Rsrc <= '1';    -- read address 2 in register file
                -- else 0
                PW <= '0'; IW <= '0'; DW <= '0'; M2R <= '0'; BW <= '0'; RW <= '0';
                AW <= '0'; Asrc1 <= '0'; F_set <= '0'; ReW <= '0'; Asrc2 <= "00";
            elsif(next_state="0111") then          -- if on state 0 then
                next_state <= "1000";   -- next state
                IorD <= '1';    -- input the data memory address
                Rsrc <= '1';    -- read address 2 in register file
                DW <= '1';      -- write in data register
                PW <= '0'; MW <= '0'; IW <= '0'; M2R <= '0'; BW <= '0'; RW <= '0';
                AW <= '0'; Asrc1 <= '0'; F_set <= '0'; ReW <= '0'; Asrc2 <= "00";
            elsif (next_state="1000") then          -- if on state 0 then
                next_state <= "0000";   -- next state
                M2R <= '1';     -- the MUX m2r
                Rsrc <= '1';    -- read address 2 in register file
                RW <= '1';      -- write in register file
                -- else 0
```

```
                PW <= '0'; IorD <= '0'; MW <= '0'; IW <= '0'; DW <= '0'; BW <= '0';
                AW <= '0'; Asrc1 <= '0'; F_set <= '0'; ReW <= '0'; Asrc2 <= "00";
            end if;
        end if;
    end process;
end implement_fsm;
```

2.4 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(
        clock, reset: in std_logic  -- reset and clock option
    );
end GlueCode;
```

It takes the input as clock that is synchronized with all the components.
The architecture imports all the components, connects all the components and activate them according to the control signals that are given from the finite state machine. The implementation is shown below:

```
begin
    -- map the ports that does not include the multiplexers
    arithematic: ALU port map(alu_operand_1, alu_operand_2, alu_opc, alu_cin, alu_cout,
                alu_result, alu_operand_1_msb, alu_operand_2_msb);
    pcounter: ProgramCounter port map(clock, reset, input_to_pc, output_from_pc);
    memory_dut: Memory port map(clock, not IorD, mem_address, B, mem_write_enable,
                output_from_mem);
    condition: ConditionChecker port map(N, Z, C, V, IR(31 downto 28), output_bool);
    four_flags: flags port map(clock, set, alu_operand_1_msb, alu_operand_2_msb, '0',
                alu_cout,'0', F_set, alu_result, decoded_dpc, N, Z, C, V);
    decoder: InstructionDecoder port map(IR, decoded_opc, decoded_insc, decoded_dpc,
            decoded_dpos, decoded_load_store, decoded_dtos);
    registerf: RegisterFile port map(clock, RW, IR(19 downto 16), read_address_2,
                IR(15 downto 12), write_data_reg, reg1data, reg2data);
    fsm: FiniteStateMachine port map(clock, output_bool, decoded_dpos, decoded_insc,
        decoded_opc, decoded_dtos, decoded_load_store, PW, IorD, MW, IW, DW, M2R,
        Rsrc, BW, RW, AW, Asrc1, F_set, ReW, Asrc2, alu_opc);
    -- now we control the components by the control signals that we get from the FSM
    process(output_from_mem, reg1data, reg2data, alu_result, output_from_pc, C, PW,
            IorD, MW, IW, DW, M2R, Rsrc, BW, RW, AW, Asrc1, F_set, ReW, A, B, IR, DR, RES)
    begin
        -- writing the registers when write enable is 1
        if(IW = '1') then IR <= output_from_mem; -- Instruction Register
        end if;
        if(DW = '1') then DR <= output_from_mem; -- Data Register
        end if;
        if(AW = '1') then A <= reg1data;         -- A Register
        end if;
        if(BW = '1') then B <= reg2data;         -- B Register
        end if;
        if(ReW = '1') then RES <= alu_result;    -- Result Register
        end if;
        if(PW = '1') then   -- if the PC write is 1 then input is given to PC
            input_to_pc <= alu_result(29 downto 0)&"00";
        end if;
        if(M2R='0') then    -- implementing the MUX that checks the input data to register
            write_data_reg <= RES;
        else
            write_data_reg <= DR;
        end if;
        if(Rsrc = '0') then -- implement the MUX to check the input read address in register
            read_address_2 <= IR(3 downto 0);
        else
            read_address_2 <= IR (15 downto 12);
        end if;
        if(Asrc1='0') then  -- implementing the MUX that checks the operand 1 to ALU
            alu_operand_1 <= "00"&output_from_pc(31 downto 2);
        else
            alu_operand_1 <= A;
        end if;
        if(Asrc2="00") then -- implementing the MUX that checks the operand 2 to ALU
            alu_operand_2 <= B;
        elsif(Asrc2 ="01") then
            alu_operand_2 <= X"00000001";
        elsif(Asrc2 ="10") then
            alu_operand_2 <= "00000000000000000000"&IR(11 downto 0);
        else
            if(IR(23)='1') then
                alu_operand_2 <= "11111111"&IR(23 downto 0);
            else
                alu_operand_2 <= "00000000"&IR(23 downto 0);
            end if;
        end if;
        if(IorD='0') then       -- MUX for input address to memory
            mem_address <= output_from_pc(7 downto 2);
        else
            mem_address <= RES(5 downto 0);
        end if;

        if(MW ='1') then    -- update the memory write enable
            mem_write_enable <= "1111";
        else
            mem_write_enable <= "0000";
        end if;
        if(Asrc2 = "11") then   -- carry in for ALU
            alu_cin <= '1';     -- if branch then cin is 1
        else
            alu_cin <= C;       -- else cin is from C flag
        end if;
        if( decoded_insc=DP ) then -- updating the s bit
            set <= IR(20);    -- if DP instruction then 20th bit
        else
            set <= '0';                       -- else set is 0
        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(
            clock, reset: in std_logic
```

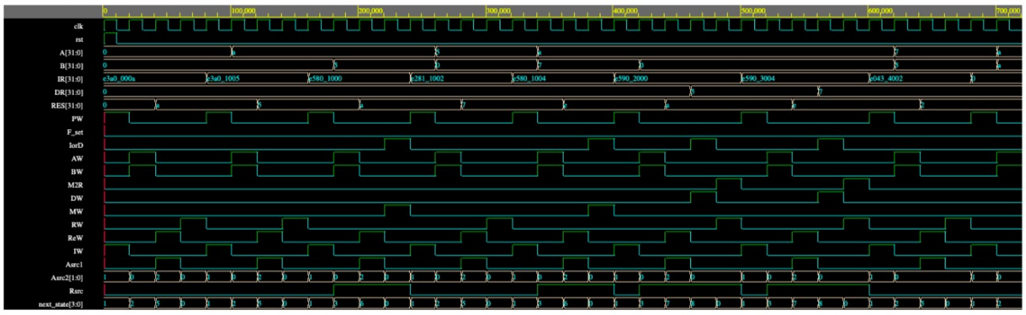
[2]Instructions are hard coded in the program memory

```
        );
    end component;
    signal clk,rst: std_logic:='0';
begin
    gc: GlueCode port map(clk, rst);
    process
    begin
        wait for 1 ns;  -- starting
        clk <= '1';     -- initialize the clock
        rst<= '1';      -- and reset the pc
        wait for 10 ns;
        rst<= '0';      -- turn off the reset
        for i in 0 to 173 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;
```

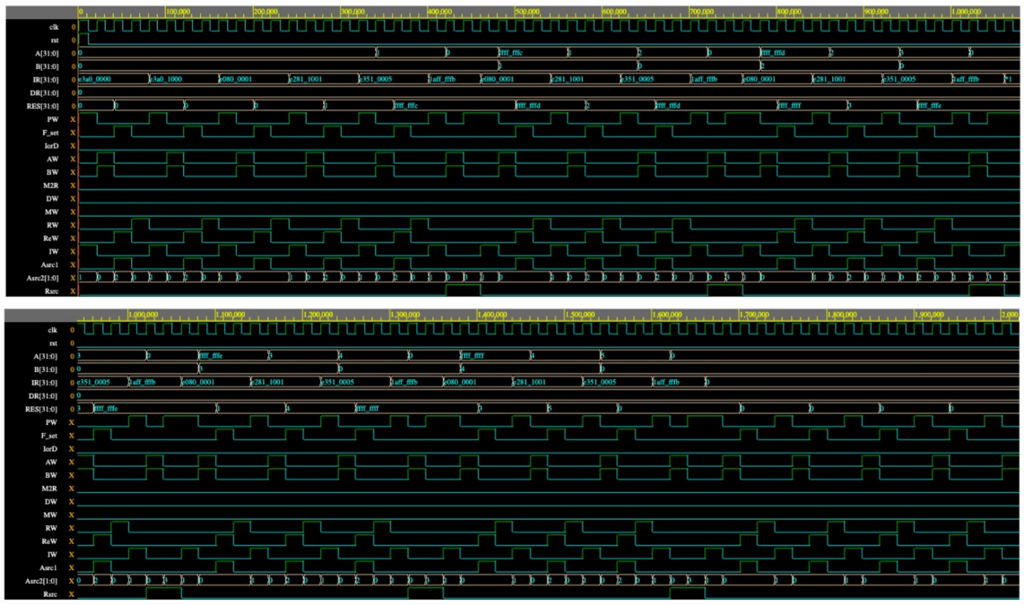
3.2 Resulting EP Wave

This test bench has been run on two sample inputs (given in the assignment pdf). The output signals of the both are given below:[3]

1. Non - loop instruction set



2. Loop instruction set



[3]the images contains the control signals and the registers that store data in different stages

3.3 Logs

The logs of the program are:

```
# 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: ***************************************************************
```

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.
Seven 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 Finite state machine (fsm_design.vhd), one contains the data structures (given along with the assignment) (myTypes.vhd) and one is run.do file that has been used to synthesis.