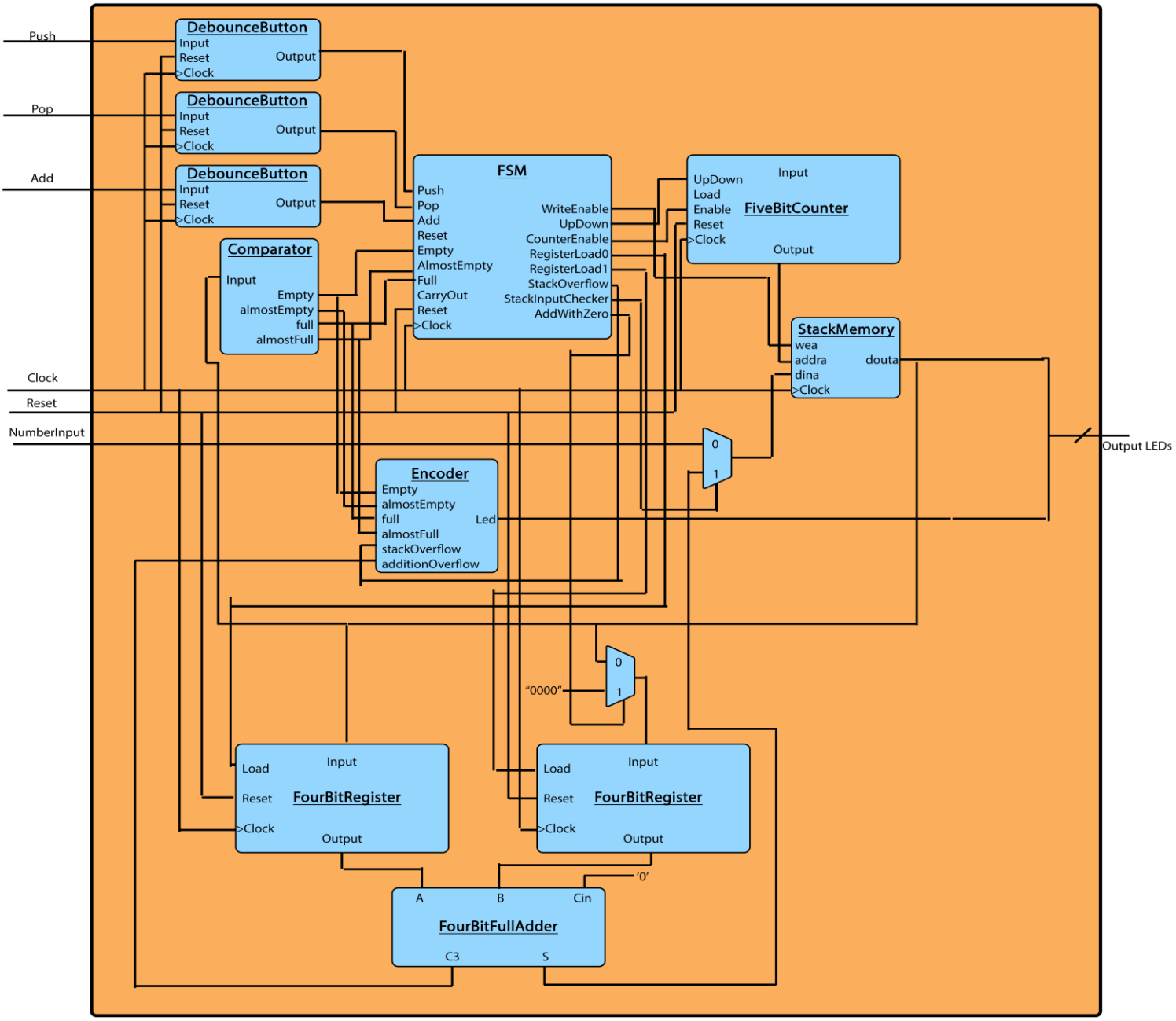
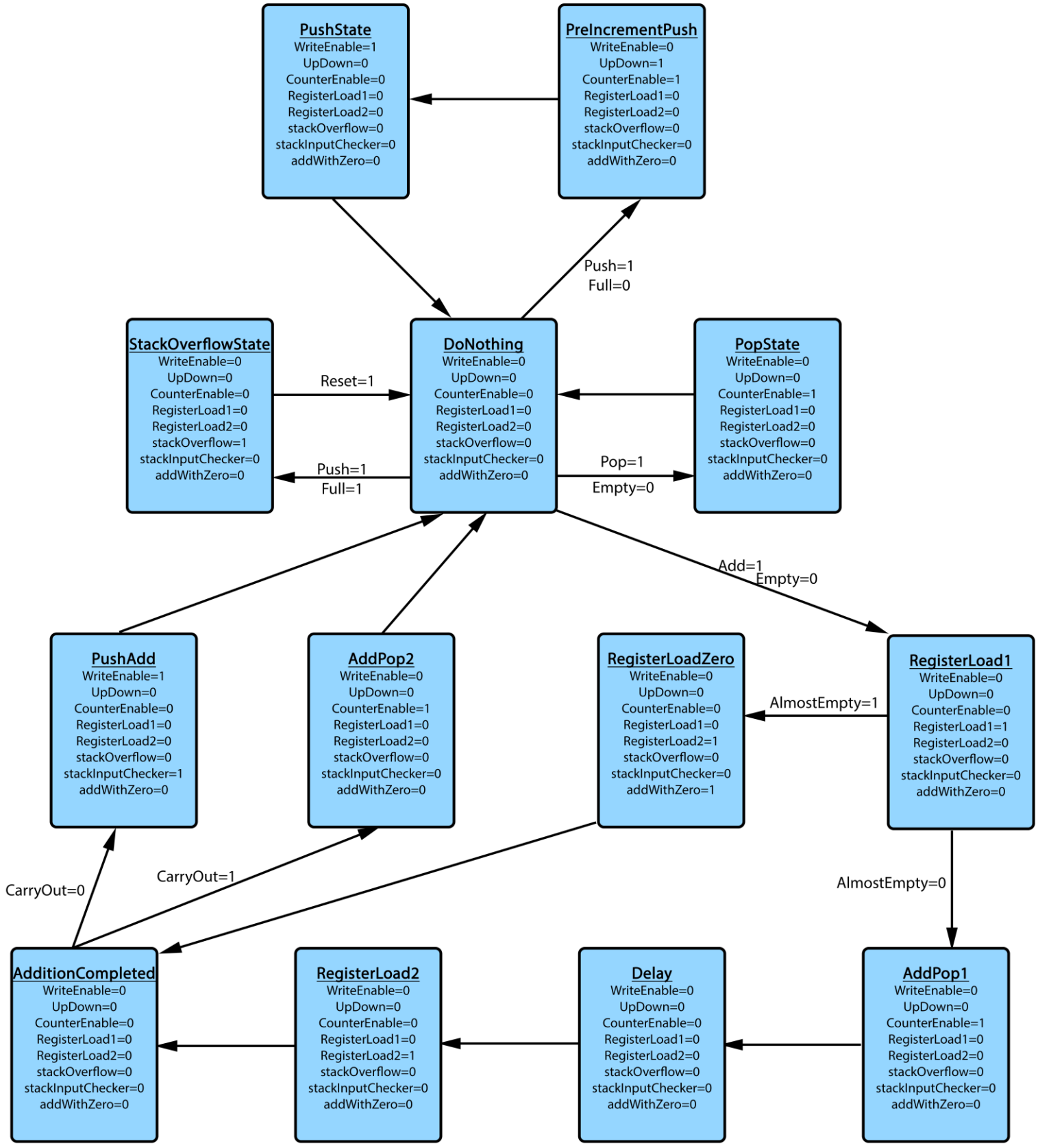
Report of Lab 4
Advanced Logic Design

Team: Κριθαράκης Εμμανουήλ, Φωτάκης Τζανής

Preparation

To begin with, the lab was about an implementation of a memory stack. All we had to do as a preparation was the block diagram of whole circuit, in which inputs, outputs and internal connections should be as specific as possible. Furthermore, we ought to have designed a Finite State Machine state diagram and how each and every part interacts with the others. Last but not least was the implementation of our whole design in code. All the above diagrams are displayed below in picture 1 and 2 respectively.





Overview

Having realized through the last lab the “structural design technique” and the importance of the fsm, designing a stack implementation, which connects with an fsm was simple. First things first, we started decomposing each part and then we composed a top unit using these 11 parts: one 5 bit counter, one 4bit adder, one stack memory, one encoder, one comparator, two registers, one Moore-fsm and three debounce units).

Next, we changed the components of the counter and the adder of last the lab in order to be functional in this project. To be more specific, we added another implementation of flip-flop in counter to reach up to 32 numbers in decimal).

After that, we implemented the comparison module, where inputs were the counter bits (output of the counter) and the outputs were the signals, whose logic '1' means the activation of a specific case such as empty, full, almost full and almost empty. These signals were the inputs of the encoder, where the outputs were a series of 4bits (0000 means empty, 1111 means full, 1110 almost full and 0001 almost empty).

Having done everything working till then we proceed with the memory. We created one through the accurate instructions which had been given to us seeing its implementation through the test bench as we should have.

The most interesting part was fsm and especially the number of its states. We started to "build" the control of pop and push buttons through states having already taken into consideration the pre-increment and post-decrement characteristics of the memory. To handle these, on the one hand we created a state which increases the address pointer and after pushes the 4-bit information in the stack and on the other hand we just popped the information as the post-decrement was going to happened after the pop.

For the stack overflow signal we made another state where if we pushed more than 31 4-bit information, the system would be able to be reset only through the reset button.

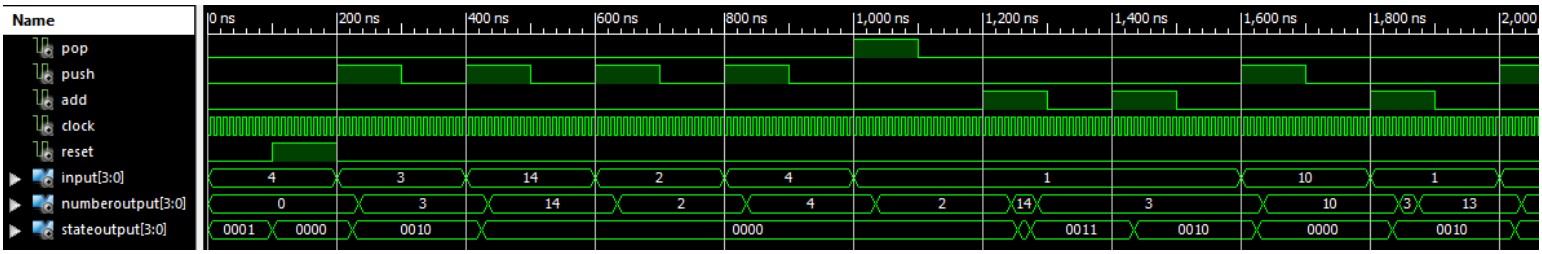
For the add function we made 6 different states in order to work under any condition. First, we thought that we need to pop twice and after to push the addition result if it was necessary. So, if the user pushed the addition button we would have to pop the first 4-bit information (if we have one) and would have to decrease the address pointer. Done that, if the signal almost empty was high we should escape the second push then add the first number with zero and re-pushed it in stack. If the signal almost empty was low, we ought to pop the second information and put it in a register (as the first one). Because one clock period was not enough to load the second number in the register we made a delay state just to make sure that everything is going to be where they should be when the addition state was going to take place. After the addition, we make clear that only when addition overflow signal was high (to elaborate it the carry out of adder) we do not push any 4-bit information back in the stack memory. When addition overflow is low, we push the result of addition in the stack normally.

Finally, we made a top module, which includes 6 components (including debouncebutton for push pop and add buttons). This module has on the one hand inputs add, push, pop, reset, upDown, switches to insert the 4-bit information and clock and on the other hand outputs 4 bits (Out 3…0), whose value shows the current data-out of the stack and the another 4bit output (Out 7…4), which presents the condition of the stack (empty, full, almost full, almost empty or normal), to be accurate, the outputs of the encoder.

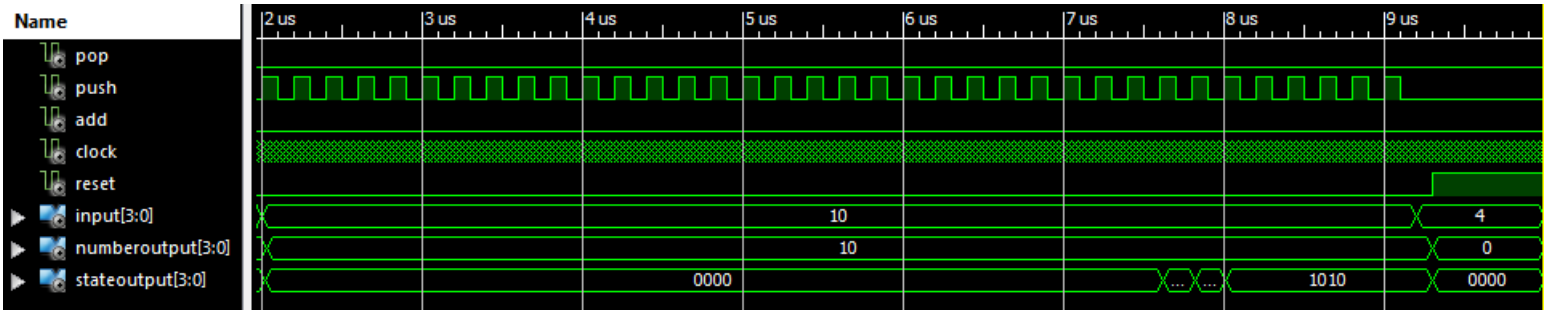
Waveforms –Simulation

Having done the vhd files for final stack implementation all we had to do was to create a test bench to check if the circuit works as we wanted to. In order to do that, we had taken into consideration some of the possible inputs and see if the outputs are the correct ones. The test benches helped us as they produced simulations of how the circuit was going to work with specific inputs. Here is the top module simulation.

In the first part of the simulation we test the behavior of our design through all its functions (push, pop, add) in all their possible states that the functions can be called.



In the second part of the simulation we test the behavior of our design in its limits by pushing 35 numbers in the stack memory.



Conclusion

This lab exercise helped us a lot to understand better the vhdl language and the fpga. First of all, it was the first time to create a stack memory, which was connected to a Moore fsm and other components like the counter, the adder and the registers. Another comment worth mentioning is the fsm machine. Though this project we realized even better not only the importance of it but also the special characteristics between a Moore and a Mealy one and how they differ the way of designing a project like this. Lastly, this was the first time to design a project of that scale, making it difficult to debug but a good way to learn debugging such designs in vhdl code.

Code

TopModule

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity FourBitStack is
   Port ( pop                  : in  STD_LOGIC;
          push                 : in  STD_LOGIC;
          add                  : in  STD_LOGIC;
          clock                : in  STD_LOGIC;
          reset                : in  STD_LOGIC;
          input                : in  STD_LOGIC_VECTOR (3 downto 0);
          numberOutput         : out  STD_LOGIC_VECTOR (3 downto 0);
          stateOutput          : out  STD_LOGIC_VECTOR (3 downto 0));

end FourBitStack;
architecture Structural of FourBitStack is

        component Comparator
                Port ( input                    : in  STD_LOGIC_VECTOR (4 downto 0);
                       empty                    : out  STD_LOGIC;
                       almostEmpty              : out  STD_LOGIC;
                       full                     : out  STD_LOGIC;
                       almostFull               : out  STD_LOGIC);
        end component;
```

```
	component FSM
		Port ( push : in  STD_LOGIC;
		       pop : in  STD_LOGIC;
		       add : in  STD_LOGIC;
		       empty : in  STD_LOGIC;
		       full : in  STD_LOGIC;
		       almostEmpty : in  STD_LOGIC;
		       carryOut : in  STD_LOGIC;
		       reset : in  STD_LOGIC;
		       clock : in  STD_LOGIC;
		       writeEnable : out  STD_LOGIC;
		       upDown : out  STD_LOGIC;
		       counterEnable : out  STD_LOGIC;
		       registerLoad : out  STD_LOGIC_VECTOR (1 downto 0);
		       stackOverflow : out  STD_LOGIC;
		       stackInputChecker : out  STD_LOGIC;
		       addWithZero: out  STD_LOGIC);
	end component;

	component FiveBitCounter
		Port ( Input	: in  STD_LOGIC_VECTOR (4 downto 0);
		       Enable : in  STD_LOGIC;
		       Load	: in  STD_LOGIC;
		       UpDown : in  STD_LOGIC;
		       Reset	: in  STD_LOGIC;
		       Clock	: in  STD_LOGIC;
		       Output : out  STD_LOGIC_VECTOR (4 downto 0));
	end component;

	component FourBitFulladder
		Port ( A	: in  STD_LOGIC_VECTOR (3 downto 0);
		       B	: in  STD_LOGIC_VECTOR (3 downto 0);
		       Cin : in  STD_LOGIC;
		       S	: out  STD_LOGIC_VECTOR (3 downto 0);
		       C3	: out  STD_LOGIC);
	end component;

	component FourBitRegister
		Port ( input	: in  STD_LOGIC_VECTOR (3 downto 0);
		       output : out  STD_LOGIC_VECTOR (3 downto 0);
		       load	: in  STD_LOGIC;
		       reset	: in  STD_LOGIC;
		       clock	: in  STD_LOGIC);
	end component;

	component debouncebutton
		Port ( clk	: in std_logic;
		       rst		: in std_logic;
		       input	: in std_logic;
		       output	: out std_logic);
	end component;

	component StackMemory
	 Port (	clka	: IN STD_LOGIC;
		       wea	: IN STD_LOGIC_VECTOR(0 DOWNTO 0);
		       addra : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
		       dina	: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
		       douta : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
	end component;

	component Encoder
Port ( empty		: in  STD_LOGIC;
       almostEmpty	: in  STD_LOGIC;
       full		: in  STD_LOGIC;
       almostFull	: in  STD_LOGIC;
       stackOverflow	: in  STD_LOGIC;
```

```
        additionOverflow            : in  STD_LOGIC;
        Led                                             : out  STD_LOGIC_VECTOR (3 downto 0));
            end component;

            signal      writeEnableSignal: STD_LOGIC_VECTOR(0 DOWNTO 0);
            signal      registerLoadSignal: STD_LOGIC_VECTOR(1 DOWNTO 0);
            signal      counterOutput: STD_LOGIC_VECTOR(4 DOWNTO 0);
            signal      stackOutput, Register1Input, Register2Input, register1Output, register2Output,
stackInputSignal,adderBSignal,adderOutput: STD_LOGIC_VECTOR(3 DOWNTO 0);
            signal
            counterEnable,upDown,emptySignal,almostEmptySignal,fullSignal,almostFullSignal,carryOutSignal,popSignal,pushSignal,addSignal,st
ackOverflowSignal,stackInputCheckerSignal,addWithZeroSignal: STD_LOGIC;
begin
            comparatorInstance: Comparator Port map( input=>counterOutput,

                        empty=>emptySignal,

                        almostEmpty=>almostEmptySignal,

                        full=>fullSignal,

                        almostFull=>almostFullSignal);

            fsmInstance: FSM Port map(      push=>pushSignal,
                                                                                                    pop=>popSignal,
                                                                                                    add=>addSignal,
                                                                                                    empty=>emptySignal,
                                                                                                    full=>fullSignal,

            almostEmpty=>almostEmptySignal,
                                                                                                    carryOut=>carryOutSignal,
                                                                                                    reset=>reset,
                                                                                                    clock=>clock,

            writeEnable=>writeEnableSignal(0),
                                                                                                    upDown=>upDown,
                                                                                                    counterEnable=>counterEnable,

            registerLoad=>registerLoadSignal,

            stackOverflow=>stackOverflowSignal,

            stackInputChecker=>stackInputCheckerSignal,

            addWithZero=>addWithZeroSignal);

            fiveBitCounterInstance: FiveBitCounter Port map(Input=>"00000",

                                    Enable=>counterEnable,

                                    Load=>'0',

                                    UpDown=>upDown,

                                    Reset=>reset,

                                    Clock=>clock,

                                    Output=>counterOutput);

            fourBitFulladderInstance: FourBitFulladder Port map(    A=>register1Output,

                                                                    B=>register2Output,

                                                                    Cin=>'0',

                                                                    S=>adderOutput,

                                                                    C3=>carryOutSignal);

            fourBitRegisterInstance1: FourBitRegister Port map(input=>stackOutput,

                                                output=>register1Output,
```

```
                                                load=>registerLoadSignal(0),
                                                reset=>reset,
                                                clock=>clock);

fourBitRegisterInstance2: FourBitRegister Port map(input=>Register2Input,
                                                output=>register2Output,
                                                load=>registerLoadSignal(1),
                                                reset=>reset,
                                                clock=>clock);

debouncePop: debouncebutton Port map(   clk=>clock,
rst=>reset,
input=>pop,
output=>popSignal);

debouncePush: debouncebutton Port map(  clk=>clock,
rst=>reset,
input=>push,
output=>pushSignal);

debounceAdd: debouncebutton Port map(   clk=>clock,
rst=>reset,
input=>add,
output=>addSignal);
stackMemoryInstance: StackMemory Port map(clka=>clock,
                wea=>writeEnableSignal,
                addra=>counterOutput,
                dina=>stackInputSignal,
                douta=>stackOutput);

encoderInstance: Encoder Port map(          empty=>emptySignal,
almostEmpty=>almostEmptySignal,
full=>fullSignal,
almostFull=>almostFullSignal,
stackOverflow=>stackOverflowSignal,
additionOverflow=>carryOutSignal,
Led=>StateOutput);

numberOutput<=stackOutput;
process(addWithZeroSignal,clock)
                begin
                        if addWithZeroSignal='0' then
```

```
                                        Register2Input<=stackOutput;
                                else
                                        Register2Input<="0000";
                                end if;
        end process;

        process (stackInputCheckerSignal,clock)
                begin
                        if stackInputCheckerSignal='0' then
                                stackInputSignal<=input;
                        else
                                stackInputSignal<=adderOutput;
                        end if;
        end process;
end Structural;
```

FSM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity FSM is
   Port ( push : in  STD_LOGIC;
          pop : in  STD_LOGIC;
          add : in  STD_LOGIC;
          empty : in  STD_LOGIC;
          full : in  STD_LOGIC;
          almostEmpty : in  STD_LOGIC;
          carryOut : in  STD_LOGIC;
          reset : in  STD_LOGIC;
          clock : in  STD_LOGIC;
          writeEnable : out  STD_LOGIC;
          upDown : out  STD_LOGIC;
          counterEnable : out  STD_LOGIC;
                                  registerLoad : out  STD_LOGIC_VECTOR (1 downto 0);
                                  stackOverflow : out  STD_LOGIC;
                                  stackInputChecker : out  STD_LOGIC;
                                  addWithZero: out  STD_LOGIC);
end FSM;
architecture Behavioral of FSM is

        type StateType is(DoNothing, PreIncrementPush, PushState, PopState, AddPop1, RegisterLoad1, AddPop2, RegisterLoad2, PushAdd,
StackOverflowState, AdditionCompleted, Delay, RegisterLoadZero);
        signal currentState,nextState:StateType;

begin

        process(clock,reset)
                begin
                        if reset='1' then currentState<=DoNothing;
                        elsif rising_edge(clock) then currentState<=nextState;
                        end if;
        end process;

        process(currentState,pop,push,add,carryOut,full)
        begin
                case currentState is
                        when DoNothing=>                                        if pop='1' then

        if empty='0' then

        nextState<=PopState;

        else

        nextState<=DoNothing;

        end if;
                                                                                                                elsif
push='1' then
```

```
            if full='0' then
            nextState<=PreIncrementPush;
            else
            nextState<=StackOverflowState;
            end if;
                                                                                                                elsif
add='1' then
            if empty='0' then
            nextState<=RegisterLoad1;
            else
            nextState<=DoNothing;
            end if;
                                                                                                                else
            nextState<=DoNothing;
                                                                                                                end if;
                        when PopState=>                         nextState<=DoNothing;
                        when PreIncrementPush=>         nextState<=PushState;
                        when PushState=>                        nextState<=DoNothing;
                        when AddPop1=>                                  nextState<=Delay;
                        when Delay=>                                    nextState<=RegisterLoad2;
                        when RegisterLoad1=>                    if almostEmpty='0' then
            nextState<=AddPop1;
                                                                                                                else
            nextState<=RegisterLoadZero;
                                                                                                                end if;
                        when AddPop2=>                                  nextState<=DoNothing;
                        when RegisterLoad2=>                    nextState<=AdditionCompleted;
                        when RegisterLoadZero=>         nextState<=AdditionCompleted;
                        when AdditionCompleted=>        if carryOut='0' then
            nextState<=PushAdd;
                                                                                                                else
            nextState<=AddPop2;
                                                                                                                end if;
                        when PushAdd=>                                  nextState<=DoNothing;
                        when StackOverflowState=>       nextState<=StackOverflowState;
                        when others=>                                   nextState<=DoNothing;
                end case;
            end process;

            with currentState select
                writeEnable<=           '1' when PushState,
                                                                '1' when PushAdd,
                                                                '0' when others;

            with currentState select
                upDown<=                        '1' when PreIncrementPush,
                                                                '0' when others;

            with currentState select
                counterEnable<='1' when PreIncrementPush,
                                                                '1' when PopState,
```

```
                                                    '1' when AddPop1,
                                                    '1' when AddPop2,
                                                    '0' when others;

        with currentState select
                registerLoad<=      "01" when RegisterLoad1,
                                                    "10" when RegisterLoad2,
                                                    "10" when RegisterLoadZero,
                                                    "00" when others;

        with currentState select
                stackOverflow<='1' when StackOverflowState,
                                                    '0' when others;

        with currentState select
                stackInputChecker<='1' when PushAdd,
                                                            '0' when others;

        with currentState select
                addWithZero<=       '1' when RegisterLoadZero,
                                                    '0' when others;

end Behavioral;
```

Comparator

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Comparator is
   Port ( input : in  STD_LOGIC_VECTOR (4 downto 0);
          empty : out  STD_LOGIC;
          almostEmpty : out  STD_LOGIC;
          full : out  STD_LOGIC;
          almostFull : out  STD_LOGIC);
end Comparator;
architecture Structural of Comparator is
begin
        empty<=                 not (input(4) or input(3) or input(2) or input(1) or input(0));
        almostEmpty<=   not (input(4) or input(3) or input(2) or input(1) or(not input(0)));
        full<=                  (input(4) and input(3) and input(2) and input(1) and input(0));
        almostFull<=    (input(4) and input(3) and input(2) and input(1) and (not input(0)));
end Structural;
```

Encoder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Encoder is
   Port ( empty : in  STD_LOGIC;
          almostEmpty : in  STD_LOGIC;
          full : in  STD_LOGIC;
          almostFull : in  STD_LOGIC;
          stackOverflow : in  STD_LOGIC;
          additionOverflow : in  STD_LOGIC;
          Led : out  STD_LOGIC_VECTOR (3 downto 0));
end Encoder;
architecture Behavioral of Encoder is
begin
        process(empty,almostEmpty,full,almostFull,stackOverflow,additionOverflow)
                begin
                        if additionOverflow='0' then
                                if empty='1' then
                                        Led<="0000";
```

```
                elsif almostEmpty='1' then
                        Led<="0010";
                elsif full='1' and stackOverflow='1' then
                        Led<="1010";
                elsif almostFull='1' then
                        Led<="1000";
                elsif full='1' then
                        Led<="1110";
                else
                        Led<="0000";
                end if;
        else
                if empty='1' then
                        Led<="0001";
                elsif almostEmpty='1' then
                        Led<="0011";
                elsif full='1' and stackOverflow='1' then
                        Led<="1011";
                elsif almostFull='1' then
                        Led<="1001";
                elsif full='1' then
                        Led<="1111";
                else
                        Led<="0001";
                end if;
        end if;
    end process;
end Behavioral;
```