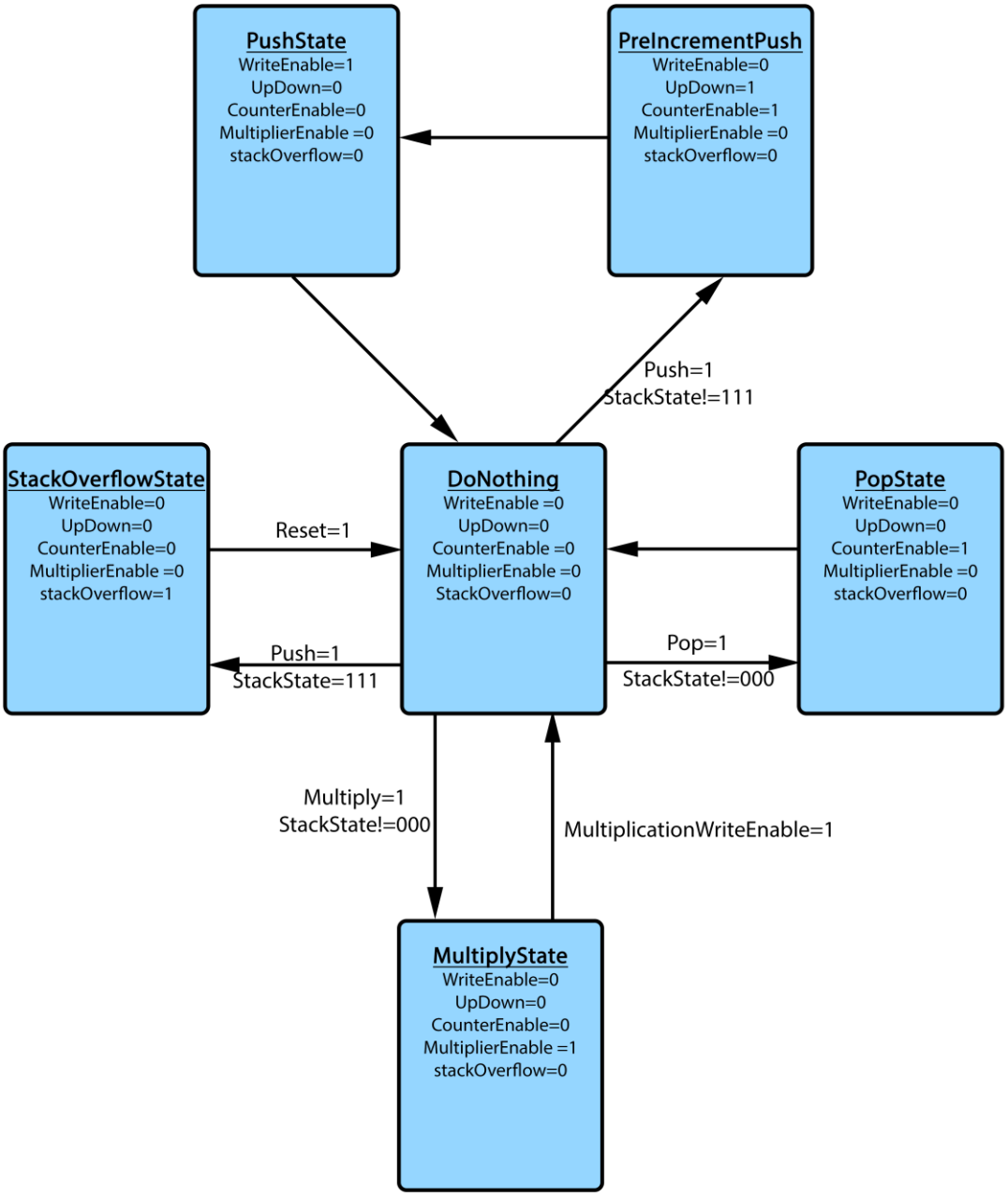
Lab Report 5

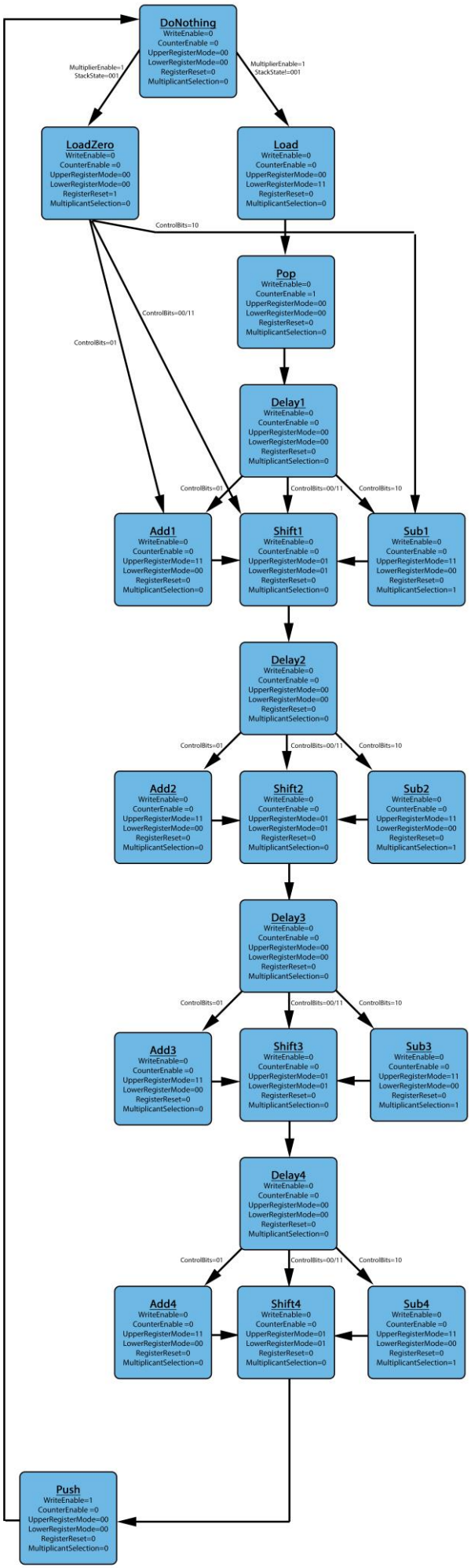
Team

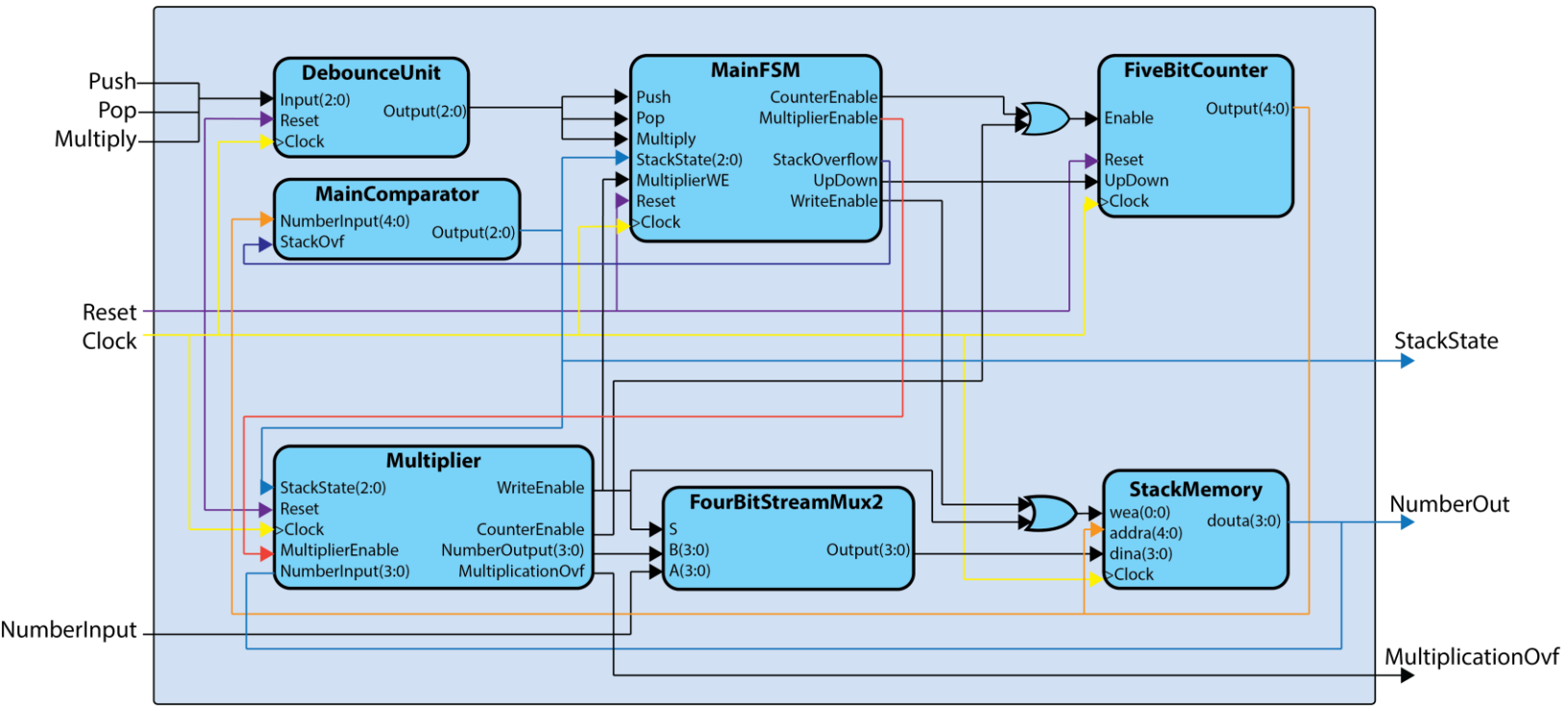
Kritharakis Emmanouil
Fotakis Tzanis

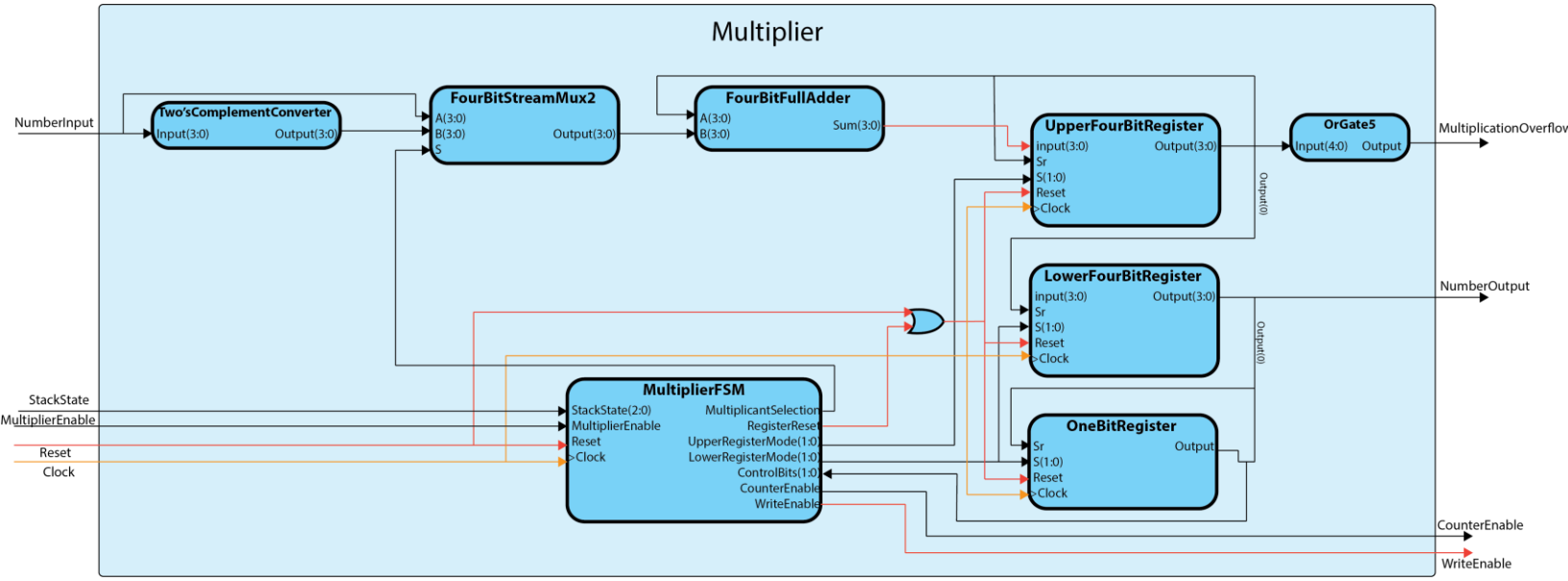
Preparation

To begin with, the lab was about an implementation of a multiplier whose numbers was taken by 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




DoNothing
WriteEnable=0
CounterEnable =0
UpperRegisterMode=00
LowerRegisterMode=00
RegisterReset=0
MultiplicantSelection=0
MultiplierEnable=1
StackState=001
MultiplierEnable=1
StackState!=001
LoadZero
WriteEnable=0
CounterEnable =0
UpperRegisterMode=00
LowerRegisterMode=00
RegisterReset=1
MultiplicantSelection=0
Load
WriteEnable=0
CounterEnable =0
UpperRegisterMode=00
LowerRegisterMode=11
RegisterReset=0
MultiplicantSelection=0
ControlBits=10
Pop
WriteEnable=0
CounterEnable =1
UpperRegisterMode=00
LowerRegisterMode=00
RegisterReset=0
MultiplicantSelection=0
ControlBits=00/11
ControlBits=01
Delay1
WriteEnable=0
CounterEnable =0
UpperRegisterMode=00
LowerRegisterMode=00
RegisterReset=0
MultiplicantSelection=0
ControlBits=01
ControlBits=00/11
ControlBits=10
Add1
WriteEnable=0
CounterEnable =0
UpperRegisterMode=11
LowerRegisterMode=00
RegisterReset=0
MultiplicantSelection=0
Shift1
WriteEnable=0
CounterEnable =0
UpperRegisterMode=01
LowerRegisterMode=01
RegisterReset=0
MultiplicantSelection=0
Sub1
WriteEnable=0
CounterEnable =0
UpperRegisterMode=11
LowerRegisterMode=00
RegisterReset=0
MultiplicantSelection=1
Delay2
WriteEnable=0
CounterEnable =0
UpperRegisterMode=00
LowerRegisterMode=00
RegisterReset=0
MultiplicantSelection=0
ControlBits=01
ControlBits=00/11
ControlBits=10
Add2
WriteEnable=0
CounterEnable =0
UpperRegisterMode=11
LowerRegisterMode=00
RegisterReset=0
MultiplicantSelection=0
Shift2
WriteEnable=0
CounterEnable =0
UpperRegisterMode=01
LowerRegisterMode=01
RegisterReset=0
MultiplicantSelection=0
Sub2
WriteEnable=0
CounterEnable =0
UpperRegisterMode=11
LowerRegisterMode=00
RegisterReset=0
MultiplicantSelection=1
Delay3
WriteEnable=0
CounterEnable =0
UpperRegisterMode=00
LowerRegisterMode=00
RegisterReset=0
MultiplicantSelection=0
ControlBits=01
ControlBits=00/11
ControlBits=10
Add3
WriteEnable=0
CounterEnable =0
UpperRegisterMode=11
LowerRegisterMode=00
RegisterReset=0
MultiplicantSelection=0
Shift3
WriteEnable=0
CounterEnable =0
UpperRegisterMode=01
LowerRegisterMode=01
RegisterReset=0
MultiplicantSelection=0
Sub3
WriteEnable=0
CounterEnable =0
UpperRegisterMode=11
LowerRegisterMode=00
RegisterReset=0
MultiplicantSelection=1
Delay4
WriteEnable=0
CounterEnable =0
UpperRegisterMode=00
LowerRegisterMode=00
RegisterReset=0
MultiplicantSelection=0
ControlBits=01
ControlBits=00/11
ControlBits=10
Add4
WriteEnable=0
CounterEnable =0
UpperRegisterMode=11
LowerRegisterMode=00
RegisterReset=0
MultiplicantSelection=0
Shift4
WriteEnable=0
CounterEnable =0
UpperRegisterMode=01
LowerRegisterMode=01
RegisterReset=0
MultiplicantSelection=0
Sub4
WriteEnable=0
CounterEnable =0
UpperRegisterMode=11
LowerRegisterMode=00
RegisterReset=0
MultiplicantSelection=1
Push
WriteEnable=1
CounterEnable =0
UpperRegisterMode=00
LowerRegisterMode=00
RegisterReset=0
MultiplicantSelection=0





Overview

Having realized through the last lab the importance of a well-defined finite state machine and how a memory stack is implemented properly we designed our circuit. This time each and every module ought to have been designed with structural mode. So the early first thing we did, was to make the gate modules and, or, not and xor . Then we made the multiplexers 2 to 1 in order to make a bigger one (4 to 1). The central fsm was quite same with the previous of last lab as the pop and pop states was exactly the same (and every state which helps to make pop and push). The only alteration was the multiplication. To elaborate it, the previous fsm has a part for addition. Now, the addition has been changed with multiplication. Despite this change, in order to make a "clear" design the whole control for multiplication was taken into consideration through another fsm which was part of a whole module called "multiplier"." Multiplier" had all the sub-modules for multiplication. These were the shift register, 2's compliment converter, multiplexers to check if we have addition of subtraction, multiplier fsm, or gate and an adder.

About shift register, we thought that it would be easier to design our multiplier if we "cut" our register into 3 different parts. In the first part we had the upper 4 bits register which was the one where all the additions and subtractions results was

written and in the shift mode we shift in msb bit the previous msb value. In the second part we had the lower 4 bits register which was the one in shift mode we shift its msb bit with lsb bit of upper register .In last part we have a 1 bit register which was the one whose bit was taken into consideration with lsb bit of lower 4 bits register in order to see if we have subtraction,addition or doNothing state(in Booth algorithm).I should point out that when the multiplication was going to start shift register was loaded with 0000(upper register) ,value of one number for multiplication (lower number) and 0 for (1 bit register) as algorithm says.

About multiplier fsm, it is triggered with multiply button and if stack is not empty we load the first number and we pop it as long as it isn't the last number of stack(almost empty).If it's not in order to pop it we give another clock cycle to be sure that the circuit was settled down and every number was where it should has. Then we start the Booth algorithm. We took advantage the fact that we multiplied 4 bits numbers so the alterations was every time 4, so we created 4 states for addition as well as for subtraction and shift. We saw that after shift states the circuit wasn't settled down properly so we add delay states in order to make sure that everything was going smoothly. After the Booth algorithm we had the result and the last thing was to push it back to memory stack and send a signal if we have a multiplication overflow. Then as the multiplier fsm has finished what it ought to have done we returned back to do Nothing state of central fsm.

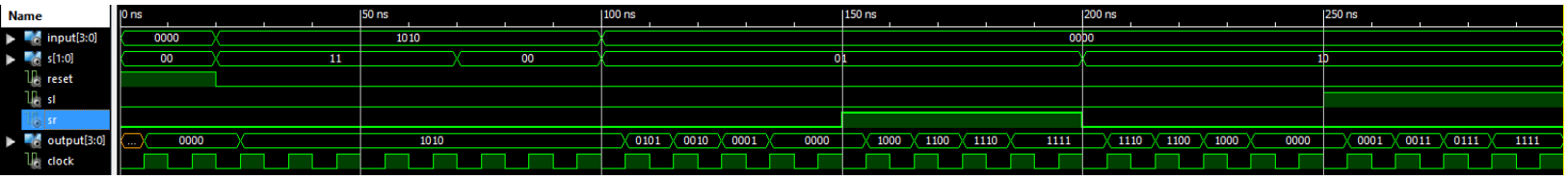
About the leds, we had to make a functional encoder which will take signals like empty , almost empty , full, almost full, stack2ateleast and other signals and show them with a 3 bits representation. In order to do that we took all the signals and with or and gates we achieved to represent them led by led as we should have. Last but not least we have to add that these signals were result of a main comparator whose "job" was to check the address of memory and return signals for equality or if it greater or less than a specific number of address. Main comparator was a module with sub-modules for checking equality, greater or less value checker and the encoder we referred above.

Finally, we made a top module, which includes 6 components (including debouncebuttonUnit for push pop and multiply buttons). This module has on the one hand inputs multiply, push, pop, reset, 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 stack2atleast, multiplication Overflow and StackOverflow), to be accurate, the outputs of the encoder(sub-module of mainComparator).

Waveforms –Simulation

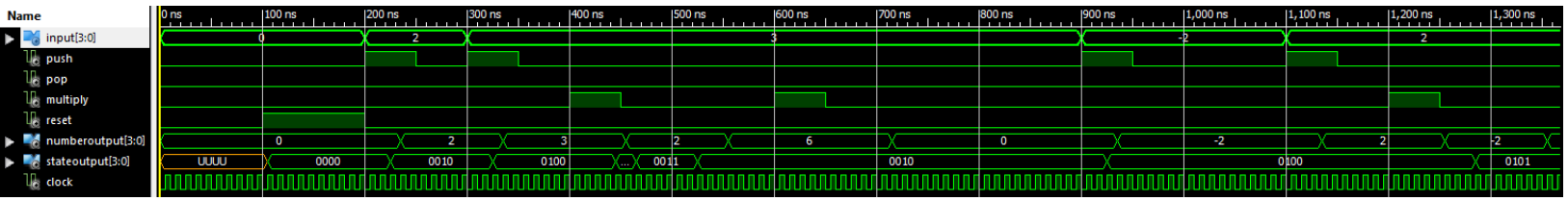
Having done the vhd files for final stack implementation for multiplication all we had to do was to create a test bench to check if the circuit works as we wanted to. In order to do that, we had taken into consideration some of the possible inputs and see if the outputs are the correct ones. The test 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 shift register in its limits by checking if it could shift its output as it ought to have.

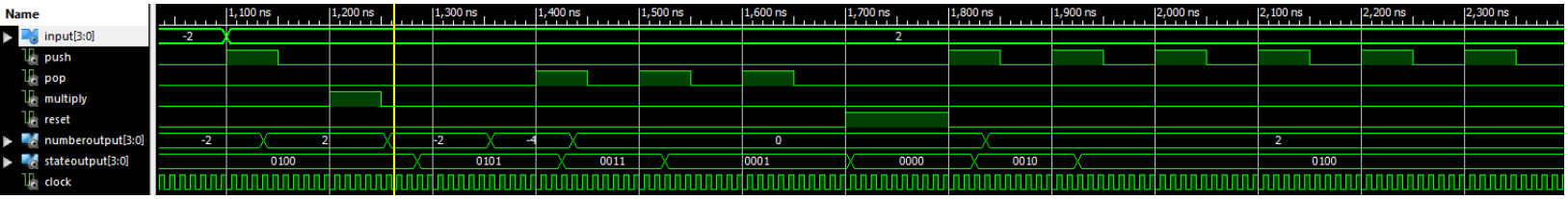


In the second part of the simulation we test the behavior of our design through all its functions (push, pop, multiply) in same of their possible states that the functions can be called.

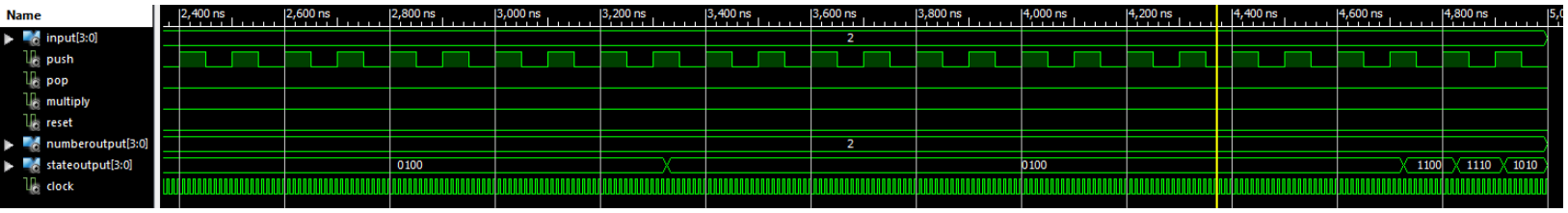
Part 1



Part 2



Part



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 second fsm which interact with the central one and all the way to make it work was more than beneficial for us. Another comment worth mentioning is the structural mode. Being obligatory to design each and every module structural helped us to make a “clear” design ,which is going to be key for our designs in the foreseeable future .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

We only demonstrate the new modules of our design because there are others that are the same from the last lab.

Top Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity TopModule is
   Port ( Push : in  STD_LOGIC;
          Pop : in  STD_LOGIC;
          Multiply : in  STD_LOGIC;
          Reset : in  STD_LOGIC;
          Input : in  STD_LOGIC_VECTOR (3 downto 0);
                                       Clock : in  STD_LOGIC;
          NumberOutput : out  STD_LOGIC_VECTOR (3 downto 0);
          StateOutput : out  STD_LOGIC_VECTOR (3 downto 0));
end TopModule;
architecture Structural of TopModule is
-----------------------------------------------------------------Components-------------------------------------------------
               component DebounceUnit
                              Port(          Input : in  STD_LOGIC_VECTOR (2 downto 0);
                                                  Reset : in   STD_LOGIC;
                                                  Clock : in   STD_LOGIC;
                                                  Output: out STD_LOGIC_VECTOR (2 downto 0));
               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 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 MainFSM
		Port(	Push : in  STD_LOGIC;
				Pop : in  STD_LOGIC;
				Multiply : in  STD_LOGIC;
				Reset : in  STD_LOGIC;
				StackState: in STD_LOGIC_VECTOR (2 downto 0);
				MultiplierWriteEnable: in STD_LOGIC;
				Clock : in  STD_LOGIC;
				CounterEnable : out  STD_LOGIC;
				WriteEnable : out  STD_LOGIC;
				UpDown : out STD_LOGIC;
				MultiplierEnable : out  STD_LOGIC;
				StackOverflow : out  STD_LOGIC;
				StackInputChecker : out  STD_LOGIC);
	end component;

	component Multiplier
		Port(	NumberInput : in  STD_LOGIC_VECTOR (3 downto 0);
				Reset : in  STD_LOGIC;
				MultiplierEnable : in STD_LOGIC;
				StackState : in STD_LOGIC_VECTOR (2 downto 0);
				Clock : in  STD_LOGIC;
				WriteEnable : out STD_LOGIC;
				CounterEnable : out STD_LOGIC;
				NumberOutput : out  STD_LOGIC_VECTOR (3 downto 0);
				MultiplicationOverflow : out STD_LOGIC);
	end component;

	component FourBitStreamMux2
		Port(	A : in  STD_LOGIC_VECTOR (3 downto 0);
				B : in  STD_LOGIC_VECTOR (3 downto 0);
				S : in  STD_LOGIC;
				Output : out  STD_LOGIC_VECTOR (3 downto 0));
	end component;

	component MainComparator
  Port (		NumberInput : in  STD_LOGIC_VECTOR (4 downto 0);
				StackOverflow : in  STD_LOGIC;
				Output : out  STD_LOGIC_VECTOR (2 downto 0));
	end component;

----------------------------------------------------------------------Signals---------------------------------------------------------------------

	signal PushSignal,PopSignal,MultiplySignal, MultiplierWriteEnableSignal, FSMWriteEnableSignal, CounterEnableSignal, UpDownSignal, MultiplierEnableSignal,
StackInputCheckerSignal, StackOverflowSignal, MultiplierCounterEnableSignal, CounterEnable: std_logic;

	signal StackStateSignal: std_logic_vector (2 downto 0);
	signal StackInput, StackOutput, MultiplierOutput: std_logic_vector (3 downto 0);
	signal CounterOutput: std_logic_vector (4 downto 0);
	signal WriteEnableSignal: std_logic_vector (0 downto 0);

-------------------------------------------------------------Instantiations-----------------------------------------------------------
begin

	MainComparatorInstance: MainComparator
  Port Map(			NumberInput=>CounterOutput,
					StackOverflow=>StackOverflowSignal,
					Output=>StackStateSignal);
```

```
	StackInputMux: FourBitStreamMux2
		Port Map(  A=>Input,
					B=>MultiplierOutput,
					S=>MultiplierWriteEnableSignal,
					Output=>StackInput);

	MultiplierInstance: Multiplier
		Port Map(  NumberInput=>StackOutput,
					Reset=>Reset,
					MultiplierEnable=>MultiplierEnableSignal,
					StackState=>StackStateSignal,
					Clock=>Clock,
					WriteEnable=>MultiplierWriteEnableSignal,
					CounterEnable=>MultiplierCounterEnableSignal,
					NumberOutput=>MultiplierOutput,
					MultiplicationOverflow=>StateOutput(0));

	MainFSMInstance: MainFSM
		Port Map(  Push=>PushSignal,
					Pop=>PopSignal,
					Multiply=>MultiplySignal,
					Reset=>Reset,
					StackState=>StackStateSignal,
					MultiplierWriteEnable=>MultiplierWriteEnableSignal,
					Clock=>Clock,
					CounterEnable=>CounterEnableSignal,
					WriteEnable=>FSMWriteEnableSignal,
					UpDown=>UpDownSignal,
					MultiplierEnable=>MultiplierEnableSignal,
					StackOverflow=>StackOverflowSignal,
					StackInputChecker=>StackInputCheckerSignal);

	DebounceUnitInstance: DebounceUnit
		Port Map     (                  Input(0)=>Push,
					Input(1)=>Pop,
					Input(2)=>Multiply,
					Reset=>Reset,
					Clock=>Clock,
					Output(0)=>PushSignal,
					Output(1)=>PopSignal,
					Output(2)=>MultiplySignal);

	CounterEnable<=CounterEnableSignal or MultiplierCounterEnableSignal;
	FiveBitCounterInstance: FiveBitCounter
		Port Map(  Input=>"00000",
					Enable=>CounterEnable,
					Load=>'0',
					UpDown=>UpDownSignal,
					Reset=>Reset,
					Clock=>Clock,
					Output=>CounterOutput);

	WriteEnableSignal(0)<=FSMWriteEnableSignal or MultiplierWriteEnableSignal;
	StackMemoryInstance: StackMemory
		Port Map(  clka=>Clock,
					wea=>WriteEnableSignal,
					addra=>CounterOutput,
					dina=>StackInput,
					douta=>StackOutput);
	StateOutput(3 downto 1)<=StackStateSignal;
	NumberOutput<=StackOutput;
end Structural;
```

Main Comparator module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity MainComparator is
   Port ( NumberInput : in  STD_LOGIC_VECTOR (4 downto 0);
          StackOverflow : in  STD_LOGIC;
          Output : out  STD_LOGIC_VECTOR (2 downto 0));
end MainComparator;
architecture Structural of MainComparator is
component EqualityFiveBitComparator
```

```
   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 GreaterLessFiveBitComparator
   Port ( A : in  STD_LOGIC_VECTOR (4 downto 0);
          B : in  STD_LOGIC_VECTOR (4 downto 0);
          Greater : out  STD_LOGIC;
          Less : out  STD_LOGIC);
end component;

component Encoder
   Port ( Empty : in  STD_LOGIC;
          AlmostEmpty : in  STD_LOGIC;
          Stack2AtLeast : in  STD_LOGIC;
          AlmostFull : in  STD_LOGIC;
          Full : in  STD_LOGIC;
          StackOverflow : in  STD_LOGIC;
          Output : out  STD_LOGIC_VECTOR (2 downto 0));
end component;

          signal EmptySignal, AlmostEmptySignal, Stack2AtLeastSignal, AlmostFullSignal, FullSignal:std_logic;

begin
          EncoderInstance: Encoder
                    Port Map(Empty=>EmptySignal,
                                                  AlmostEmpty=>AlmostEmptySignal,
                                                  Stack2AtLeast=> Stack2AtLeastSignal,
                                                  AlmostFull=> AlmostFullSignal,
                                                  Full=>FullSignal,
                                                  StackOverflow=>StackOverFlow,
                                                  Output=>Output);

          EqualityFiveBitComparatorInstance: EqualityFiveBitComparator
                    Port Map(input=> NumberInput,
                                                  Empty=>EmptySignal,
                                                  AlmostEmpty=>AlmostEmptySignal,
                                                  Full=>FullSignal,
                                                  AlmostFull=>AlmostFullSignal);

          GreaterLessFiveBitComparatorInstance: GreaterLessFiveBitComparator
                    Port Map(A=>NumberInput,
                                                  B=>"00001",
                                                  Greater=>Stack2AtLeastSignal);
end Structural;
```

Encoder module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Encoder is
   Port ( Empty : in  STD_LOGIC;
          AlmostEmpty : in  STD_LOGIC;
          Stack2AtLeast : in  STD_LOGIC;
          AlmostFull : in  STD_LOGIC;
          Full : in  STD_LOGIC;
          StackOverflow : in  STD_LOGIC;
          Output : out  STD_LOGIC_VECTOR (2 downto 0));
end Encoder;

architecture Structural of Encoder is
          component orGate
                    Port( inputA : in  STD_LOGIC;
                                        inputB : in  STD_LOGIC;
```

```
                                        output : out  STD_LOGIC);
        end component;

        component notGate
                Port( input : in  STD_LOGIC;
                                        output : out  STD_LOGIC);
        end component;

        component andGate
                Port(       inputA : in  STD_LOGIC;
                                        inputB : in  STD_LOGIC;
                                        output : out  STD_LOGIC);
        end component;
        signal Led0or1Output, Led1OrOutput, NotStackOvf, Led2or1Output: std_logic;
begin
        Led0Or1: orGate Port Map(inputA=>Full, inputB=>AlmostEmpty, output=> Led0or1Output);
        Led0Or2: orGate Port Map(inputA=>Led0or1Output, inputB=>StackOverflow, output=> Output(0));

        Led1Or: orGate Port Map(inputA=>Full, inputB=>Stack2AtLeast, output=>Led1OrOutput);
        NotStackOverflow: notGate Port Map(input=>StackOverflow, output=>NotStackOvf);
        Led1And: andGate Port Map(inputA=>Led1OrOutput, inputB=>NotStackOvf,output=>Output(1));

        Led2Or1: orGate Port Map(inputA=>Full, inputB=>AlmostFull, output=> Led2or1Output);
        Led2Or2: orGate Port Map(inputA=>Led2or1Output, inputB=>StackOverflow, output=> Output(2));
end Structural;
```

Equality checker module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity EqualityFiveBitComparator 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 EqualityFiveBitComparator;

architecture Structural of EqualityFiveBitComparator is
 component EqualityOneBitComparator
    Port (A : in  STD_LOGIC;
                                        B : in  STD_LOGIC;
                                        result : out  STD_LOGIC);
 end component;

 component andGate5
  Port ( input : in  STD_LOGIC_VECTOR (4 downto 0);
       output : out  STD_LOGIC);
        end component;

 signal almostFullSignal,fullSignal,emptySignal,almostEmptySignal : STD_LOGIC_VECTOR (4 downto 0);
begin
        --Check if sp is equal to "00000"  Empty
        Empty0Bit:EqualityOneBitComparator port map(A=>input(0),B=>'0',result=>emptySignal(0));
        Empty1Bit:EqualityOneBitComparator port map(A=>input(1),B=>'0',result=>emptySignal(1));
        Empty2Bit:EqualityOneBitComparator port map(A=>input(2),B=>'0',result=>emptySignal(2));
        Empty3Bit:EqualityOneBitComparator port map(A=>input(3),B=>'0',result=>emptySignal(3));
        Empty4Bit:EqualityOneBitComparator port map(A=>input(4),B=>'0',result=>emptySignal(4));

        --Check if sp is equal to "00001"  AlmostEmpty
        AlEmpty0Bit:EqualityOneBitComparator port map(A=>input(0),B=>'1',result=>almostEmptySignal(0));
        AlEmpty1Bit:EqualityOneBitComparator port map(A=>input(1),B=>'0',result=>almostEmptySignal(1));
        AlEmpty2Bit:EqualityOneBitComparator port map(A=>input(2),B=>'0',result=>almostEmptySignal(2));
```

```
        AlEmpty3Bit:EqualityOneBitComparator port map(A=>input(3),B=>'0',result=>almostEmptySignal(3));
        AlEmpty4Bit:EqualityOneBitComparator port map(A=>input(4),B=>'0',result=>almostEmptySignal(4));

        --Check if sp is equal to "11111"  Full
        Full0Bit:EqualityOneBitComparator port map(A=>input(0),B=>'1',result=>fullSignal(0));
        Full1Bit:EqualityOneBitComparator port map(A=>input(1),B=>'1',result=>fullSignal(1));
        Full2Bit:EqualityOneBitComparator port map(A=>input(2),B=>'1',result=>fullSignal(2));
        Full3Bit:EqualityOneBitComparator port map(A=>input(3),B=>'1',result=>fullSignal(3));
        Full4Bit:EqualityOneBitComparator port map(A=>input(4),B=>'1',result=>fullSignal(4));

        --Check if sp is equal to "11110"  AlmostFull
        AlFull0Bit:EqualityOneBitComparator port map(A=>input(0),B=>'0',result=>almostFullSignal(0));
        AlFull1Bit:EqualityOneBitComparator port map(A=>input(1),B=>'1',result=>almostFullSignal(1));
        AlFull2Bit:EqualityOneBitComparator port map(A=>input(2),B=>'1',result=>almostFullSignal(2));
        AlFull3Bit:EqualityOneBitComparator port map(A=>input(3),B=>'1',result=>almostFullSignal(3));
        AlFull4Bit:EqualityOneBitComparator port map(A=>input(4),B=>'1',result=>almostFullSignal(4));

        --Outputs
        EmptyAndGate:                   andGate5 port map(input=>emptySignal,                   output=>Empty);
        AmlostEmptyAndGate:     andGate5 port map(input=>almostEmptySignal,     output=>AlmostEmpty);
        FullAndGate:                    andGate5 port map(input=>fullSignal,                    output=>Full);
        AlmostFullAndGate:      andGate5 port map(input=>almostFullSignal,              output=>AlmostFull);
end Structural;
```

Greater or less checker module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity GreaterLessFiveBitComparator is
   Port ( A : in  STD_LOGIC_VECTOR (4 downto 0);
          B : in  STD_LOGIC_VECTOR (4 downto 0);
          Greater : out  STD_LOGIC;
          Less : out  STD_LOGIC);
end GreaterLessFiveBitComparator;

architecture Structural of GreaterLessFiveBitComparator is
                component EqualityOneBitComparator
                        Port (          A : in  STD_LOGIC;
                                        B : in  STD_LOGIC;
                                        result : out  STD_LOGIC);
                end component;

                component GreaterOneBitComparator
                        Port (          A : in  STD_LOGIC;
                                        B : in  STD_LOGIC;
                                        Result : out  STD_LOGIC);
                end component;

                component andGate5
                        Port (          input : in  STD_LOGIC_VECTOR (4 downto 0);
                                        output : out  STD_LOGIC);
                end component;

                component orGate5
                        Port (          input : in  STD_LOGIC_VECTOR (4 downto 0);
                                        output : out  STD_LOGIC);
                end component;

                component norGate
                        Port (          inputA : in  STD_LOGIC;
                                        inputB : in  STD_LOGIC;
                                        output : out  STD_LOGIC);
```

```
                                    end component;

                                    signal EResult, GResult: std_logic_vector (4 downto 0);
                                    signal andResult: std_logic_vector (3 downto 0);
                                    signal orResult,equalityAndGateResult: std_logic;
begin
                                    E0: EqualityOneBitComparator        Port Map(A=>A(0), B=>B(0), Result=>EResult(0));
                                    G0: GreaterOneBitComparator         Port Map(A=>A(0), B=>B(0), Result=>GResult(0));

                                    E1: EqualityOneBitComparator        Port Map(A=>A(1), B=>B(1), Result=>EResult(1));
                                    G1: GreaterOneBitComparator         Port Map(A=>A(1), B=>B(1), Result=>GResult(1));

                                    E2: EqualityOneBitComparator        Port Map(A=>A(2), B=>B(2), Result=>EResult(2));
                                    G2: GreaterOneBitComparator         Port Map(A=>A(2), B=>B(2), Result=>GResult(2));

                                    E3: EqualityOneBitComparator        Port Map(A=>A(3), B=>B(3), Result=>EResult(3));
                                    G3: GreaterOneBitComparator         Port Map(A=>A(3), B=>B(3), Result=>GResult(3));

                                    E4: EqualityOneBitComparator        Port Map(A=>A(4), B=>B(4), Result=>EResult(4));
                                    G4: GreaterOneBitComparator         Port Map(A=>A(4), B=>B(4), Result=>GResult(4));

                                    orGate:  orGate5     Port Map(input(0)=>GResult(4), input(4 downto 1)=>andResult, output=>orResult);
                                    andGate0: andGate5 Port Map(input(0)=>GResult(3),Input(1)=>EResult(4), input(2)=>'1',                                    input(3)=>'1',
                                                input(4)=>'1',                                  output=>andResult(0));
                                    andGate1: andGate5 Port Map(input(0)=>GResult(2),Input(1)=>EResult(4), input(2)=>EResult(3),  input(3)=>'1',
                                    input(4)=>'1',                                  output=>andResult(1));
                                    andGate2: andGate5 Port Map(input(0)=>GResult(1),Input(1)=>EResult(4), input(2)=>EResult(3),  input(3)=>EResult(2),input(4)=>'1',
                                                output=>andResult(2));
                                    andGate3: andGate5 Port Map(input(0)=>GResult(0),Input(1)=>EResult(4), input(2)=>EResult(3),
                                    input(3)=>EResult(2),input(4)=>EResult(1),output=>andResult(3));
                                    Greater<=orResult;
                                    equalityAndGate: andGate5 Port Map(input=>EResult, output=>equalityAndGateResult);
                                    LessNorGate: norGate Port Map(inputA=>orResult, inputB=>equalityAndGateResult, output=>Less);
end Structural;
```

Multiplier module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Multiplier is
   Port (     NumberInput : in  STD_LOGIC_VECTOR (3 downto 0);
                                    Reset : in  STD_LOGIC;
                                    MultiplierEnable : in STD_LOGIC;
                                    StackState : in STD_LOGIC_VECTOR (2 downto 0);
                                    Clock : in  STD_LOGIC;
                                    WriteEnable : out STD_LOGIC;
                                    CounterEnable : out STD_LOGIC;
                                    NumberOutput : out  STD_LOGIC_VECTOR (3 downto 0);
                                    MultiplicationOverflow : out STD_LOGIC);
end Multiplier;

architecture Structural of Multiplier is
-------------------------------------------------------------components-----------------------------------------
            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);
```

```
                                        S : in  STD_LOGIC_VECTOR (1 downto 0);
                                        reset : in  STD_LOGIC;
                                        clock : in  STD_LOGIC;
                                        sl : in STD_LOGIC;
                                        sr : in STD_LOGIC);
        end component;

        component OneBitRegister
                Port(       Input : in  STD_LOGIC;
                                        Reset : in  STD_LOGIC;
                                        sl : in  STD_LOGIC;
                                        sr : in  STD_LOGIC;
                                        S : in STD_LOGIC_VECTOR (1 downto 0);
                                        Clock : in  STD_LOGIC;
                                        Output : out  STD_LOGIC);
        end component;

        component FourBitStreamMux2
                Port(       A : in  STD_LOGIC_VECTOR (3 downto 0);
                                        B : in  STD_LOGIC_VECTOR (3 downto 0);
                                        S : in  STD_LOGIC;
                                        Output : out  STD_LOGIC_VECTOR (3 downto 0));
        end component;

        component FourBitTwosComplementConverter
                Port(       Input : in  STD_LOGIC_VECTOR (3 downto 0);
                                        Output : out  STD_LOGIC_VECTOR (3 downto 0));
        end component;

        component MultiplierFSM
                Port(       MultiplierEnable : in  STD_LOGIC;
                                        Reset : in  STD_LOGIC;
                                        StackState : in STD_LOGIC_VECTOR (2 downto 0);
                                        ControlBits : in STD_LOGIC_VECTOR (1 downto 0);
                                        Clock : in  STD_LOGIC;
                                        WriteEnable : out  STD_LOGIC;
                                        CounterEnable : out  STD_LOGIC;
                                        UpperRegisterMode : out  STD_LOGIC_VECTOR (1 downto 0);
                                        LowerRegisterMode : out  STD_LOGIC_VECTOR (1 downto 0);
                                        RegisterReset : out  STD_LOGIC;
                                        MultiplicantSelection :out  STD_LOGIC);
        end component;

        component orGate5
                Port(       input: in std_logic_vector (4 downto 0);
                                        output: out std_logic);
        end component;
-------------------------------------------------------------Signals-----------------------------------------------

        signal MultiplicantSelectionSignal, PreviousBitOutput, RegisterResetSignal, RegisterReset: std_logic;
        signal UpperRegisterModeSignal, LowerRegisterModeSignal: std_logic_vector (1 downto 0);
        signal InvertedNumberInput, MultiplicantAdder, UpperProductRegisterOutput, LowerProductRegisterOutput, Sum: std_logic_vector (3 downto 0);

-------------------------------------------------------------Instances---------------------------------------------
begin
        MultiplierFSMInstance: MultiplierFSM
                Port Map(  MultiplierEnable=>MultiplierEnable,
                                                Reset=>Reset,
                                                StackState=>StackState,
                                                ControlBits(0)=>PreviousBitOutput,
                                                ControlBits(1)=>LowerProductRegisterOutput(0),
                                                Clock=>Clock,
                                                WriteEnable=>WriteEnable,
                                                CounterEnable=>CounterEnable,
                                                UpperRegisterMode=>UpperRegisterModeSignal,
                                                LowerRegisterMode=>LowerRegisterModeSignal,
                                                RegisterReset=>RegisterResetSignal,
```

```
                                        MultiplicantSelection=>MultiplicantSelectionSignal);

        FourBitFullAdderInstance: FourBitFulladder
                Port Map(A=>UpperProductRegisterOutput,
                                        B=>MultiplicantAdder,
                                        Cin=>'0',
                                        S=>Sum);

        RegisterReset<=Reset or RegisterResetSignal;
        UpperProductRegister: FourBitRegister
                Port Map(  input=>Sum,
                                        output=>UpperProductRegisterOutput,
                                        S=>UpperRegisterModeSignal,
                                        reset=>RegisterReset,
                                        clock=>Clock,
                                        sl=>'0',
                                        sr=>UpperProductRegisterOutput(3));

        LowerProductRegister: FourBitRegister
                Port Map(  input=>NumberInput,
                                        output=>LowerProductRegisterOutput,
                                        S=>LowerRegisterModeSignal,
                                        reset=>RegisterReset,
                                        clock=>Clock,
                                        sl=>'0',
                                        sr=>UpperProductRegisterOutput(0));

        PreviousBitRegister: OneBitRegister
                Port Map(  Input=>'0',
                                        Reset=>Reset,
                                        sl=>'0',
                                        sr=>LowerProductRegisterOutput(0),
                                        S=>LowerRegisterModeSignal,
                                        Clock=>Clock,
                                        Output=>PreviousBitOutput);

        FourBitStreamMux2Instance: FourBitStreamMux2
                Port Map(  A=>NumberInput,
                                        B=>InvertedNumberInput,
                                        S=>MultiplicantSelectionSignal,
                                        Output=>MultiplicantAdder);

        FourBitTwosComplementConverterInstance: FourBitTwosComplementConverter
                Port Map(  Input=>NumberInput,
                                        Output=>InvertedNumberInput);

        NumberOutput<=LowerProductRegisterOutput;

        MultiplicationOverflowOr: orGate5 port map(input(3 downto 0)=> UpperProductRegisterOutput, input(4)=>'0', output=>MultiplicationOverflow);
end Structural;
```

Multiplier FSM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity MultiplierFSM is
        Port (      MultiplierEnable : in  STD_LOGIC;
                        Reset : in  STD_LOGIC;
                        StackState : in STD_LOGIC_VECTOR (2 downto 0);
                        ControlBits : in STD_LOGIC_VECTOR (1 downto 0);
                        Clock : in  STD_LOGIC;
                        WriteEnable : out  STD_LOGIC;
                        CounterEnable : out  STD_LOGIC;
                        UpperRegisterMode : out  STD_LOGIC_VECTOR (1 downto 0);
```

```
                        LowerRegisterMode : out  STD_LOGIC_VECTOR (1 downto 0);
                        RegisterReset : out  STD_LOGIC;
                        MultiplicantSelection : out  STD_LOGIC);
end MultiplierFSM;
architecture Behavioral of MultiplierFSM is
        type StateType is(DoNothing, Load, LoadZero, Pop, Add1, Sub1, Shift1, Add2, Sub2, Shift2, Add3, Sub3, Shift3, Add4, Sub4, Shift4, Push, Delay1, Delay2, Delay3,
Delay4);
        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,MultiplierEnable,StackState,ControlBits)
                begin
                        case currentState is
                        when DoNothing=>    if MultiplierEnable='1' then
                                                                        if StackState="001" then
                                                                                nextState<=LoadZero;
                                                                        else
                                                                                nextState<=Load;
                                                                        end if;
                                                                else
                                                                        nextState<=DoNothing;
                                                                end if;

                        when Load=>                             nextState<=Pop;
                        when Pop=>                              nextState<=Delay1;
                        when Delay1=>                   if ControlBits="01" then
                                                                        nextState<=Add1;
                                                                elsif ControlBits="10" then
                                                                        nextState<=Sub1;
                                                                else
                                                                        nextState<=Shift1;
                                                                end if;
                        when LoadZero=>     if ControlBits="01" then
                                                                        nextState<=Add1;
                                                                elsif ControlBits="10" then
                                                                        nextState<=Sub1;
                                                                else
                                                                        nextState<=Shift1;
                                                                end if;
                        when Add1=>                             nextState<=Shift1;
                        when Sub1=>                             nextState<=Shift1;

                        when Shift1=>                   nextState<=Delay2;
                        when Delay2=>                   if ControlBits="01" then
                                                                        nextState<=Add2;
                                                                elsif ControlBits="10" then
                                                                        nextState<=Sub2;
                                                                else
                                                                        nextState<=Shift2;
                                                                end if;
                        when Add2=>                             nextState<=Shift2;
                        when Sub2=>                             nextState<=Shift2;

                        when Shift2=>                   nextState<=Delay3;
                        when Delay3=>                   if ControlBits="01" then
                                                                        nextState<=Add3;
                                                                elsif ControlBits="10" then
                                                                        nextState<=Sub3;
                                                                else
                                                                        nextState<=Shift3;
                                                                end if;
```

```
                when Add3=>                     nextState<=Shift3;
                when Sub3=>                     nextState<=Shift3;

                when Shift3=>           nextState<=Delay4;
                when Delay4=>           if ControlBits="01" then
                                                        nextState<=Add4;
                                                elsif ControlBits="10" then
                                                        nextState<=Sub4;
                                                else
                                                        nextState<=Shift4;
                                                end if;
                when Add4=>                     nextState<=Shift4;
                when Sub4=>                     nextState<=Shift4;

                when Shift4=>           nextState<=Push;
                when Push=>                     nextState<=DoNothing;
                when others=>           nextState<=DoNothing;
        end case;
end process;

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

with currentState select
        counterEnable<=                 '1' when Pop,
                                                '0' when others;

with currentState select
        UpperRegisterMode<=             "01" when Shift1,
                                                "01" when Shift2,
                                                "01" when Shift3,
                                                "01" when Shift4,
                                                "11" when Add1,
                                                "11" when Sub1,
                                                "11" when Add2,
                                                "11" when Sub2,
                                                "11" when Add3,
                                                "11" when Sub3,
                                                "11" when Add4,
                                                "11" when Sub4,
                                                "00" when others;

with currentState select
        LowerRegisterMode<=             "01" when Shift1,
                                                "01" when Shift2,
                                                "01" when Shift3,
                                                "01" when Shift4,
                                                "11" when Load,
                                                "00" when others;

with currentState select
        RegisterReset<=                 '1' when LoadZero,
                                                '0' when others;

with currentState select
        MultiplicantSelection<=         '1' when Sub1,
                                                '1' when Sub2,
                                                '1' when Sub3,
                                                '1' when Sub4,
                                                '0' when others;
end Behavioral;
```

2's compliment converter module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity FourBitTwosComplementConverter is
   Port ( Input : in  STD_LOGIC_VECTOR (3 downto 0);
          Output : out  STD_LOGIC_VECTOR (3 downto 0));
end FourBitTwosComplementConverter;
architecture Structural of FourBitTwosComplementConverter is
            component TwoBitMultiplexer is
   Port ( Input:          in  STD_LOGIC_VECTOR (1 downto 0);
          S:              in STD_LOGIC;
          Output:out  STD_LOGIC);
            end component;

            component orGate
                        port(       inputA: in std_logic;
                                    inputB: in std_logic;
                                    output: out std_logic);
            end component;

            component notGate
                        Port(       input: in std_logic;
                                    output: out std_logic);
            end component;

            signal Outputs: std_logic_vector (3 downto 0);
            signal selectors, notInput: std_logic_vector (2 downto 0);
begin
            Outputs(0)<=Input(0);
            selectors(0)<=Input(0);
            selectorsOr1: orGate Port Map(inputA=>selectors(0), inputB=>input(1), output=>selectors(1));
            selectorsOr2: orGate Port Map(inputA=>selectors(1), inputB=>input(2), output=>selectors(2));
            notGate0: notGate Port Map(input=>input(1), output=>notInput(0));
            notGate1: notGate Port Map(input=>input(2), output=>notInput(1));
            notGate2: notGate Port Map(input=>input(3), output=>notInput(2));
            TwoBitMultiplexer0: TwoBitMultiplexer Port Map(Input(0)=>Input(1),Input(1)=>notInput(0),s=>selectors(0),Output=>Outputs(1));
            TwoBitMultiplexer1: TwoBitMultiplexer Port Map(Input(0)=>Input(2),Input(1)=>notInput(1),s=>selectors(1),Output=>Outputs(2));
            TwoBitMultiplexer2:   TwoBitMultiplexer Port Map(Input(0)=>Input(3),Input(1)=>notInput(2),s=>selectors(2),Output=>Outputs(3));
            Output<=Outputs;
end Structural;
```

Main fsm

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity MainFSM is
            Port(       Push : in  STD_LOGIC;
                        Pop : in  STD_LOGIC;
                        Multiply : in  STD_LOGIC;
                        Reset : in  STD_LOGIC;
                        StackState: in STD_LOGIC_VECTOR (2 downto 0);
                        MultiplierWriteEnable: in STD_LOGIC;
                        Clock : in  STD_LOGIC;
                        CounterEnable : out  STD_LOGIC;
                        WriteEnable : out  STD_LOGIC;
                        UpDown : out STD_LOGIC;
                        MultiplierEnable : out  STD_LOGIC;
                        StackOverflow : out  STD_LOGIC;
                        StackInputChecker : out  STD_LOGIC);
end MainFSM;
architecture Behavioral of MainFSM is
            type StateType is(DoNothing, PreIncrementPush, PushState, PopState, MultiplyState, StackOverflowState);
```

```
	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,Multiply,MultiplierWriteEnable,StackState)
		begin
			case currentState is
	when DoNothing=>	if pop='1' then
								if StackState="000" then
								nextState<=DoNothing;
								else
					nextState<=PopState;
								end if;
								elsif push='1' then
								if StackState="111" then
								nextState<=StackOverflowState;
			else
								nextState<=PreIncrementPush;
								end if;
								elsif Multiply='1' then
								if StackState="010" then
				nextState<=MultiplyState;
								elsif StackState="111" then
				nextState<=MultiplyState;
								elsif StackState="100" then
				nextState<=MultiplyState;
								elsif StackState="001" then
			nextState<=MultiplyState;
								else
					nextState<=DoNothing;
								end if;
								else
				nextState<=DoNothing;
								end if;
	when PopState=>				nextState<=DoNothing;			when PreIncrementPush=>
	nextState<=PushState;
	when PushState=>			nextState<=DoNothing;
	when MultiplyState=>		if MultiplierWriteEnable='0' then
	nextState<=MultiplyState;
								else
						nextState<=DoNothing;
								end if;
when StackOverflowState=>		nextState<=StackOverflowState;
when others=>				nextState<=DoNothing;
end case;
	end process;

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

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

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

	with currentState select
		stackOverflow<=		'1' when StackOverflowState,
```

```
                                                    '0' when others;

        with currentState select
                MultiplierEnable<=     '1' when MultiplyState,
                                                    '0' when others;

        with currentState select
                StackInputChecker<=  '1' when MultiplyState,
                                                    '0' when others;

end Behavioral;
```

4bits Shift register module

(We made 2 of them upper 4 bits and lower 4 bits)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity FourBitRegister is
   Port (     input : in  STD_LOGIC_VECTOR (3 downto 0);
                        output : out  STD_LOGIC_VECTOR (3 downto 0);
                        S : in  STD_LOGIC_VECTOR (1 downto 0);
                        reset : in  STD_LOGIC;
                        clock : in  STD_LOGIC;
                        sl : in STD_LOGIC;
                        sr : in STD_LOGIC);
end FourBitRegister;

architecture Structural of FourBitRegister is
        component OneBitRegister
   Port (     Input : in  STD_LOGIC;
                        Reset : in  STD_LOGIC;
                        sl : in  STD_LOGIC;
                        sr : in  STD_LOGIC;
                        S : in STD_LOGIC_VECTOR (1 downto 0);
                        Clock : in  STD_LOGIC;
                        Output : out  STD_LOGIC);
        end component;
        signal RegisterOutput: std_logic_vector (3 downto 0);

begin
        OneBitRegister0: OneBitRegister Port Map(Input=>Input(0), Reset=>Reset, sl=>sl,                                                 sr=>RegisterOutput(1),
        S=>S, Clock=>Clock, Output=>RegisterOutput(0));
        OneBitRegister1: OneBitRegister Port Map(Input=>Input(1), Reset=>Reset, sl=>RegisterOutput(0),   sr=>RegisterOutput(2),        S=>S, Clock=>Clock,
Output=>RegisterOutput(1));
        OneBitRegister2: OneBitRegister Port Map(Input=>Input(2), Reset=>Reset, sl=>RegisterOutput(1),   sr=>RegisterOutput(3),        S=>S, Clock=>Clock,
Output=>RegisterOutput(2));
        OneBitRegister3: OneBitRegister Port Map(Input=>Input(3), Reset=>Reset, sl=>RegisterOutput(2),   sr=>sr,                                                S=>S,
Clock=>Clock, Output=>RegisterOutput(3));
        Output<=RegisterOutput;
end Structural;
```

Previous 1 bit shift register module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity OneBitRegister is
   Port(      Input : in  STD_LOGIC;
```

```
                                Reset : in  STD_LOGIC;
                                sl : in  STD_LOGIC;
                                sr : in  STD_LOGIC;
                                S : in STD_LOGIC_VECTOR (1 downto 0);
                                Clock : in  STD_LOGIC;
                                Output : out  STD_LOGIC);
end OneBitRegister;

architecture Structural of OneBitRegister is

        component DFlipFlop
                Port (  D : in  STD_LOGIC;
                                Reset : in  STD_LOGIC;
                                Clock : in  STD_LOGIC;
                                Qp : out  STD_LOGIC;
                                Qn : out  STD_LOGIC);
        end component;

        component Mux4
                Port (  input : in  STD_LOGIC_VECTOR (3 downto 0);
                                output : out  STD_LOGIC;
                                s : in  STD_LOGIC_VECTOR (1 downto 0));
        end component;

        signal MuxOutput,QSignal: std_logic;
begin
        FlipFlop: DFlipFlop port map (D=>MuxOutput,Reset=>Reset,Clock=>Clock,Qp=>QSignal);
        Multiplexer: Mux4 port map (input(0)=>QSignal, input(1)=>sr, input(2)=>sl,input(3)=>Input, output=>MuxOutput, s=>S);
        Output<=QSignal;
end Structural;
```