Algorithmic State Machine (Binary Multiplier)

Introduction:

As we learned in Digital 2 in the theoretical part about ASM (Algorithmic state machine), we can design how the system works. By using hardware description language we can convert **asm** to code and we can separate the code to several processes (two or three). The first process to check reset and clk, second one to choose the next stage and give the output or we can separate it into two processes, one to choose the next stage and one to give the output. In this experiment we want to design the asm for the multiplier because the software (vivado) can't implement and synthesize * multiple. So we want to make some operator like counting, shifting to implement multiple.

Abstract:

We'll learn how to build an asm chart for a multiplier then test it by simulation then we'll implement and synthesize the code and generate bitstream to install it on ZedBoard to show the result.

Apparatus:

1-Vivado Design Suite HL WebPACK™ Edition.
2-ZedBoard.

Procedure:

Part 1: ASM Simulation:

1-ASM for multiplier

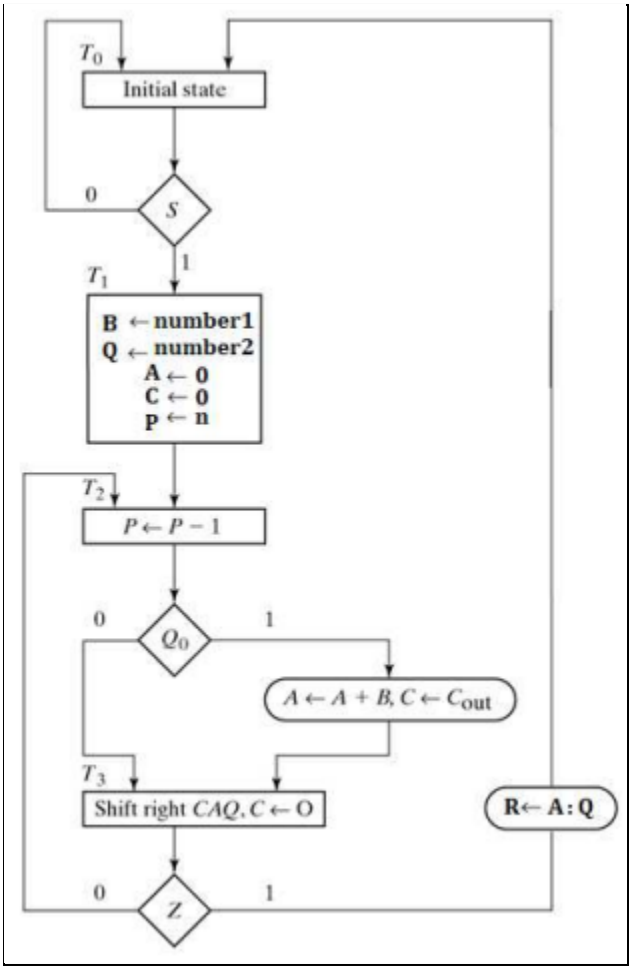


Fig1: ASM For 4-bit multiplier

2-Implement the ASM chart in VHDL using 3 processes.
3-Simulate the code and make sure it behaves correctly.

ASM multiplier implementation:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;


entity Binary_Multiplier_ASM is
    Port ( S : in STD_LOGIC;
           Reset : in STD_LOGIC;
```

```
        CLK : in STD_LOGIC;
        number1 : in STD_LOGIC_VECTOR (3 downto 0);
        number2 : in STD_LOGIC_VECTOR (3 downto 0);
        R : out STD_LOGIC_VECTOR (7 downto 0));
end Binary_Multiplier_ASM;

architecture Behavioral of Binary_Multiplier_ASM is
type state_type is (T0, T1, T2, T3);
signal current_state, next_state :state_type ;
signal p : integer ;
signal flag : integer :=0 ;
begin

state_register : process( clk, reset)
begin
   if(reset = '1') then
    current_state <= T0;
   elsif (clk'event and clk = '1') then
    current_State <= next_state;
   end if;
end process state_register;

next_state_process : process( s, current_State)
begin
case current_state is
when T0 =>
   if(S = '0')then
      next_state <= T0;
   elsif(S = '1') THEn next_state <= T1;
   end if;
when T1 =>  next_state <= T2;
when T2 => next_state <= T3;
when T3 =>
   if(p = 0)then
      flag<=1;
      next_state <= T0;
   else next_state <= T2;
   end if;
when others => NULL;
end case;
end process next_state_process;


output_process : process( clk)
```

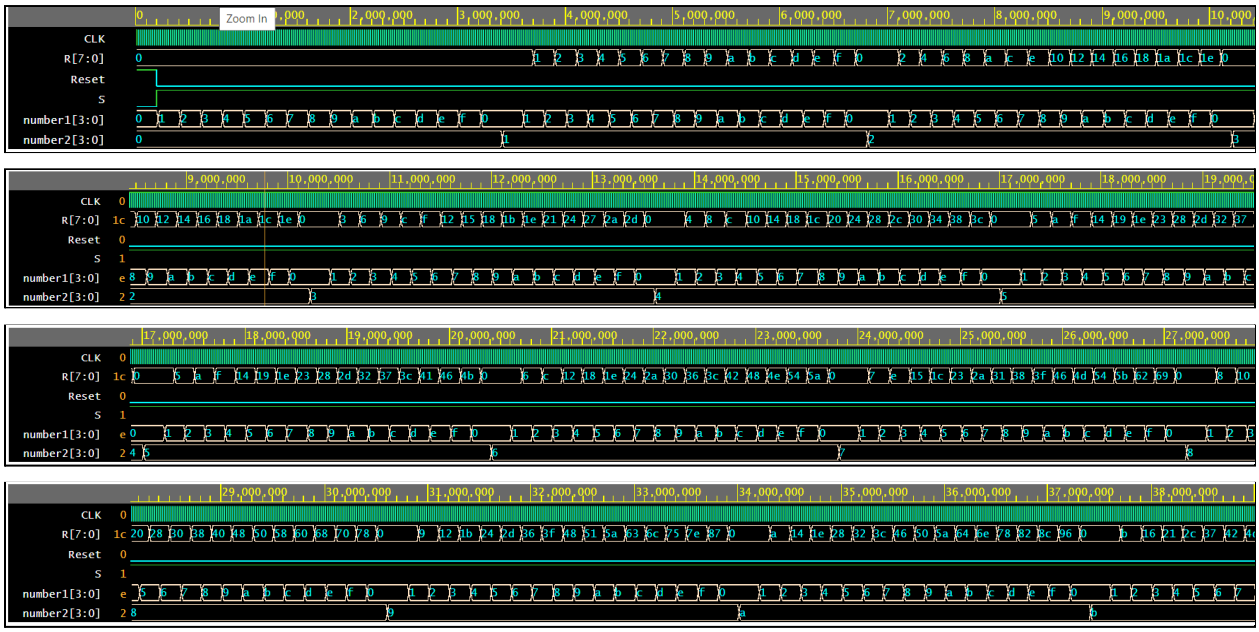
```
variable Q, B : STD_LOGIC_VECTOR (3 downto 0);
variable A : STD_LOGIC_VECTOR (3 downto 0):="0000";
variable c : STD_LOGIC :='0';
variable temp_a : STD_LOGIC_VECTOR (4 downto 0):="00000";
begin
if(clk'event and clk = '1') then
 case current_state is
 when T0=>
     if(flag=1)then
     R <= (A & Q);
     end if;
     if(reset='1')then
     R<="00000000";
     end if;
 when T1 =>
    B := number1 ;
    Q := number2 ;
    A := "0000" ;
    c := '0';
    p <= 4;
 when T2 =>
    p <= p - 1 ;
        if(Q(0) = '1')then
        temp_a := ('0'&A) + ('0'&B);
         A := temp_a(3 downto 0);
         c := temp_a(4);
    end if;
when T3 =>
    Q := A(0) & Q(3 downto 1);
    A := C & A(3 downto 1);
    C:='0';

when others => NULL;
END CASE;
end if;
end process output_process;

end Behavioral;
```

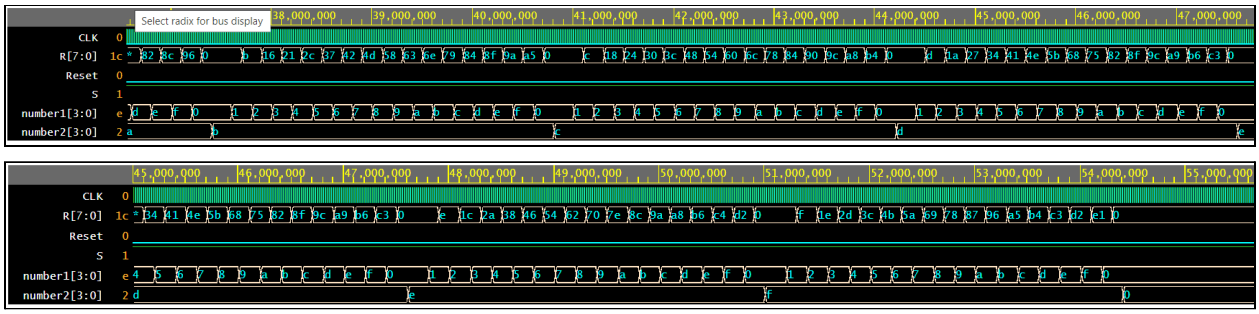
Multiplier testbench:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;


entity binary_multiplier_asm_tb is
--  Port ( );
end binary_multiplier_asm_tb;

architecture Behavioral of binary_multiplier_asm_tb is
component Binary_Multiplier_ASM
    Port ( S : in STD_LOGIC;
           Reset : in STD_LOGIC;
           CLK : in STD_LOGIC;
           number1 : in STD_LOGIC_VECTOR (3 downto 0);
           number2 : in STD_LOGIC_VECTOR (3 downto 0);
           R : out STD_LOGIC_VECTOR (7 downto 0));
end component;

signal S, Reset, CLK : std_logic:='0';
signal number1, number2 : STD_LOGIC_VECTOR (3 downto 0);
signal R : STD_LOGIC_VECTOR (7 downto 0);
constant period : time := 10 ns;
begin
u1: Binary_Multiplier_ASM
port map(S=>S,
 CLK=>CLK,
 Reset=>Reset,
 number1=>number1,number2=>number2,R=>R);

clk_gen: process
begin
clk<='0';
wait for period/2;
clk<='1';
wait for period/2;
end process clk_gen;

stimulus: process
begin
reset <= '1';
number1<="0000";
number2<="0000";
```

```
S<= '0';
wait for period*20;

reset <= '0';
S<= '1';

wait for 10 ns;
        for i in 0 to 15 loop
        for j in 0 to 15 loop
            number1<= number1 + '1';
            wait for period*20 ;
                end loop;
                number2<= number2 + '1';
                wait for period*20;
        end loop;

wait;
end process stimulus;
end Behavioral;
```

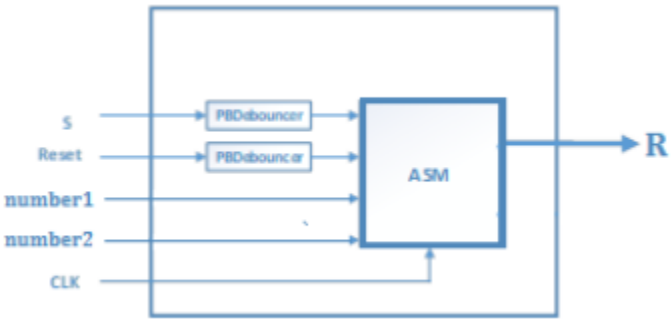
Part of Results in simulation:





Part 2: Top-level Entity:

In the top level entity we'll use the entity that has the code we implemented in the previous part, and we'll use PDBouncer that just a delay that ensures the correct state of the button. We used two PDBouncer because we want to use two push buttons.



Then we'll synthesize, implemente, and generate bitstream to install it on ZedBoard and show our results.
note: the reset button does not work!!!

Conclusion:

In the experiment we learned how to convert the asm chart for binary multiplier to code that we can simulate it, then make a top level entity that includes the asm entity and two entity of PDBouncer then we implement and synthesize the top level entity then generate bitstream to show our results in ZedBoard. We learned how to implement the multiple without using * (multiple symbol) because our software can't implement and synthesize it.