Fall 2019

California State University, Northridge

Department of Electrical & Computer Engineering



Lab Experiment 6

*Arithmetic*

*High Radix Multiplier (Radix 8)*

Juan Silva & Tumen Khureltulga

(Group 16)

October 29, 2019

ECE 524L

**Table of Contents**

**I. Introduction and Problem Statements…….………………………...…Page 2**

**II. Procedure……..…………………………………………….…………...Page 3**

**III. Testing Strategy………………………………….....………………......Page 4**

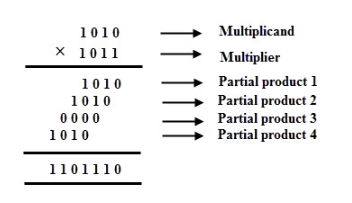
**IV. Results…………....…………………………………....…………….......Page 5**

**V. Analysis…………....……………………………....………………..……Page 7**

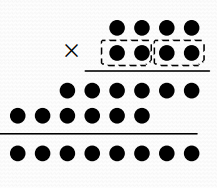
**VI. Appendix……………...………………………....………………….….Page 8**

1. **Introduction and Problem Statements**

In this lab experiment, an arithmetic operation known as the Radix-8 multiplier is designed. This operation will take two 32-bit signed numbers and output its product. This is a conventional sequential multiplier. The basic algorithm of multiplying decimal number is based on calculating partial products, left-shifting them, and then adding them together. This algorithm applies to binary number as well. The binary multiplication is much easier as it contains only 0s and 1s. For example, a multiplier procedure shown below in *Figure 6.1.*

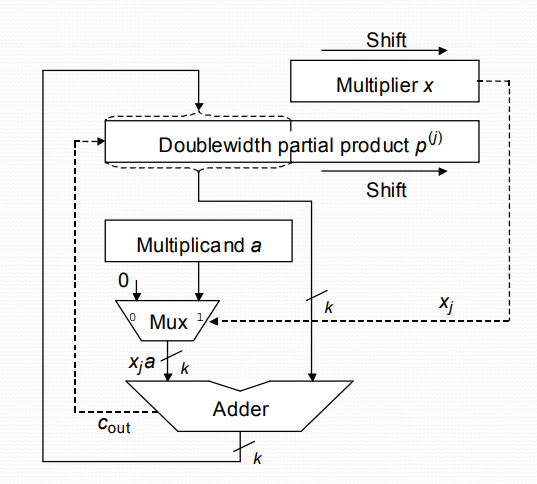


**Figure 6.1 -** Basic Multiplication Algorithm

Based on the algorithm shown above, partial products are generated for each digit in the multiplier. This process continues until there are no more numbers left to shift. In partial product multiplication, multiplication with a zero results in a partial product of zero whereas multiplication results in a partial product equal to the multiplicand. This operation is known as a Radix-2 Multiplier because partial products are shifted one position to the left.   
 The figure on the left shows a Radix-4 multiplier. Two bits from the partial product are selected and shifted two positions to the left. In a Radix-8 multiplier, partial products are shifted three positions to the left. Instead of multiplying the multiplicand by 2 bits of the multiplier each time, radix 8 multiplier uses three bits starting from LSB to MSB. 

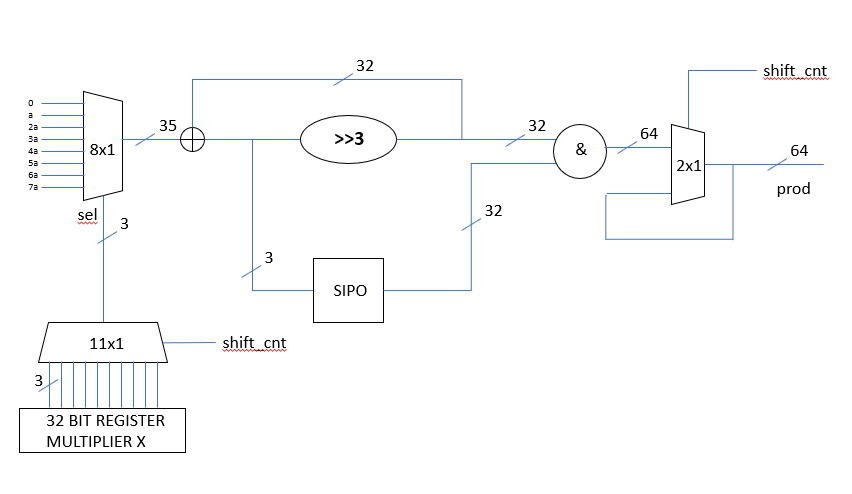
1. **Procedure**

A Radix-8 design is based on the basic hardware multiplier implementation shown in *Figure 6.2* given a few adjustments. Two 32-bit registers are used for the multiplier and multiplicand, one 64-bit register are used for the product, and a 32-bit adder with an 8x1 MUX are used. The addition acts as a counter that controls which bits are being selected from the multiplier register. As a result, the shift operation on multiplier is not needed and removed.



**Figure 6.2 -** Basic Hardware Multipliers

The top level diagram of the designed circuit is shown in *Figure 6.3.* The multiplier and multiplicand receives data in a parallel fashion. Three bits are selected from multiplicand with a counter control that will serve as a control signal for the 8x1 MUX. Inputs are preloaded with 35-bit wide multiples of A (from 0 to 7A) and produce a 35-bit sum based on the data that is selected. The output of MUX and partial products are added. The three LSB of the sum are saved and fed to a 32-bit Serial-In Parallel-Out register. The sum is shifted three bits right and becomes the partial product. This operation continues until the maximum number of bits are shifted out which is 11 times for a 32-bit number. Once the operation is complete, the product of two unsigned number is output and the circuit is ready to take new values for multiplicand and multiplier. The previous output is held until it’s overwritten by the next multiplication cycle. The new data **must be** submitted after 120 ns from the previous data. If any data is submitted within 120 ns from the previous data, the result will be incorrect.



**Figure 6.3 -** Block Diagram of Radix-8 Multiplier

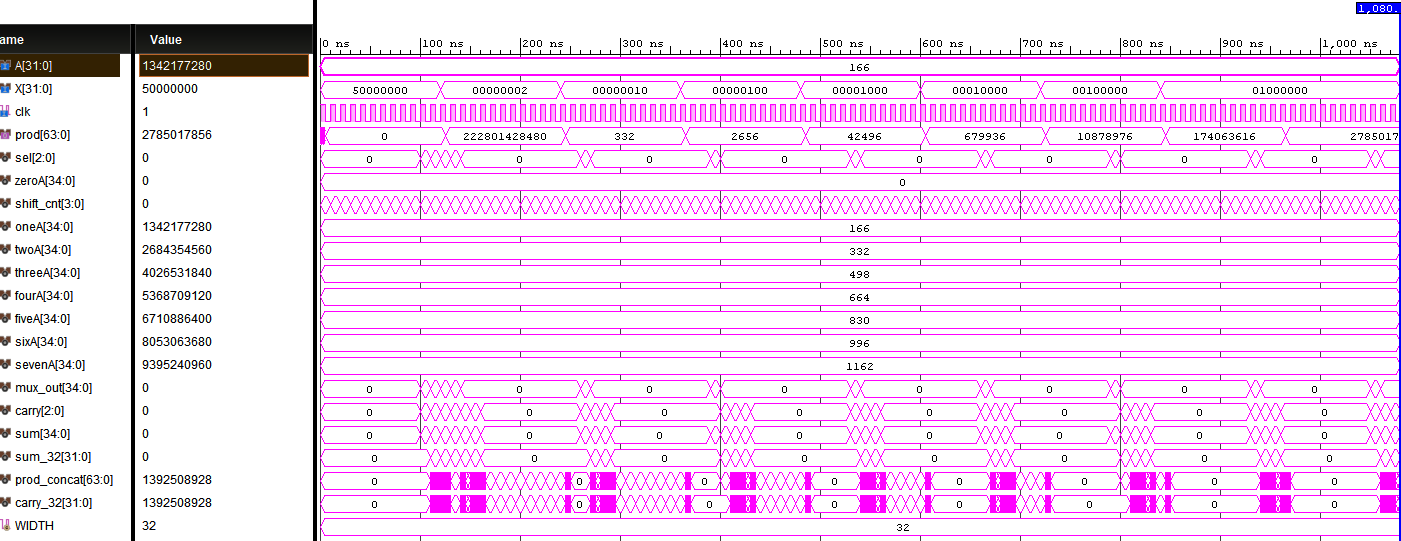
1. **Testing Strategy**

To verify the functionality of the circuit, a handful of test vectors were chosen. These vectors are calculated and compared with the output of the circuit. For example, when A=166 and X=1342177280, then the expected output is 222801428480. If the observed result matches with expected output, then the circuit design is valid. 16 test vectors were used, each having a different case tested. For the first 10 cases, value A remained constant while value X varied. In the next 10 cases, both values A and X varied every 120 ns. In addition, both maxterms and minterms of A and X were tested. The results are shown in *Table 6.1.*

1. **Results & Data**

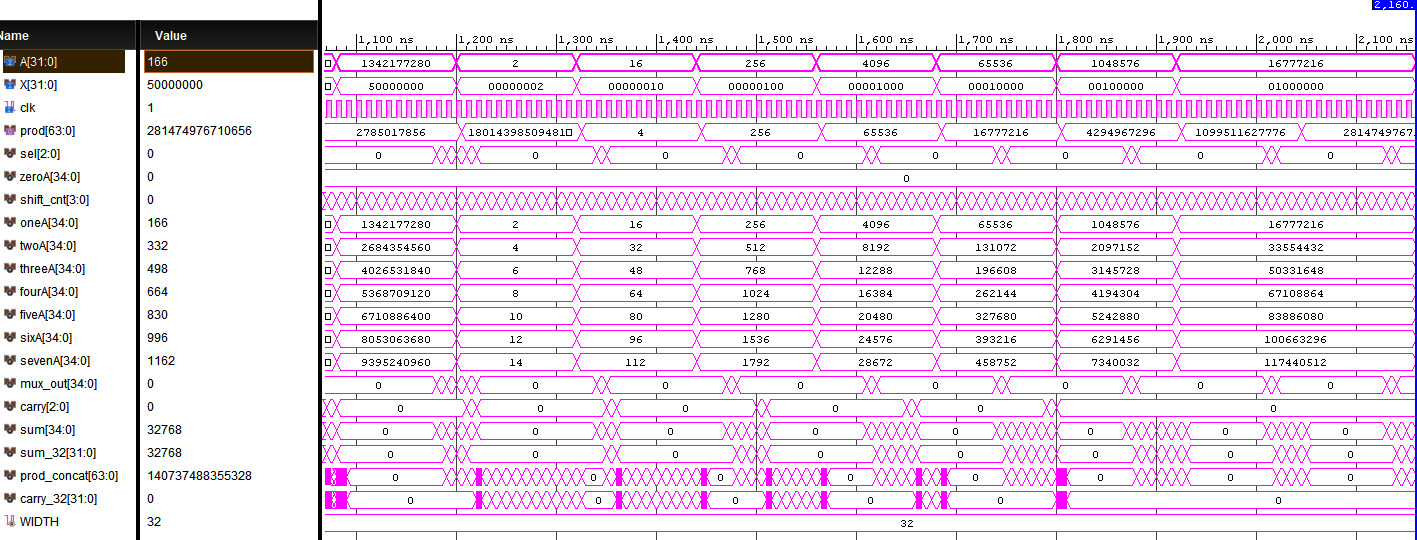
| **A** | **X** | **Expected Output** | **Observed Result** |
| --- | --- | --- | --- |
| 166 | 1342177280 | 222801428480 | 222801428480 |
| 166 | 2 | 332 | 332 |
| 166 | 16 | 2656 | 2656 |
| 166 | 256 | 42496 | 42496 |
| 166 | 4096 | 679936 | 679936 |
| 166 | 65536 | 10878976 | 10878976 |
| 166 | 1048576 | 174063616 | 174063616 |
| 166 | 16777216 | 2785017856 | 2785017856 |
| 1342177280 | 1342177280 | 1801439850948198400 | 1801439850948198400 |
| 2 | 2 | 4 | 4 |
| 16 | 16 | 256 | 256 |
| 256 | 256 | 65536 | 65536 |
| 4096 | 4096 | 16777216 | 16777216 |
| 65536 | 65536 | 4294967296 | 4294967296 |
| 1048576 | 1048576 | 1099511627776 | 1099511627776 |
| 16777216 | 16777216 | 281474976710656 | 281474976710656 |

**Table 6.1** - Expected Values versus Observed Results



**Figure 6.4 -** Test Vectors Where Value **“**A” Remained Constant.

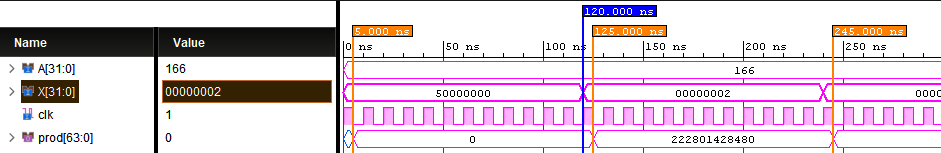
Bel

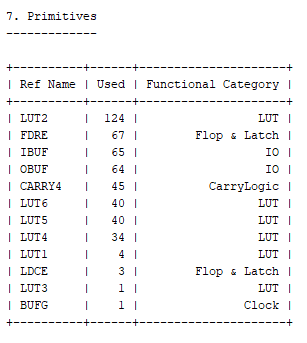


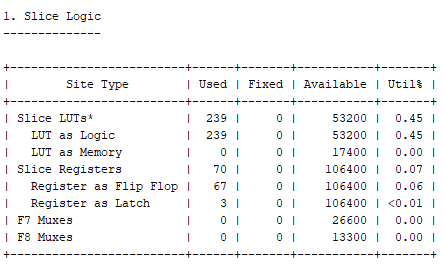
**Figure 6.5 -** Test Vectors Where Value **“**A” Varied.

1. **Analysis**

First output comes out at 5 ns. It is zero because initially the 64 bit registers are loaded with zero. The actual first output comes out at 125 ns. It takes 120 ns to calculate the first product only. From then on it takes 125 ns to calculate the product. Notice that the value of the X is changed at 120 ns, then the output is displayed at 245 ns. See below



**Figure 6.6** - Waveforms of the Multiplier design up to 250 ns.



**Figure 5.3** - Slice and Primitives section of the Report Utilization.

It turns out that the design uses 67 register: 31 register for the carry, 32 register for sum, and 4 for the counter. 3 latches for MUX select line. It uses 45 carry logic to generate the product. Previous statement about 32 bit registers for multiplier and multiplicand was false. As a result, the design saved 64 registers. In total the design uses 239 LUTs.

1. **Appendix**

| **Source Codes** | **Testbench** |
| --- | --- |
| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use IEEE.NUMERIC\_STD.ALL;  entity lab6 is  generic(WIDTH : integer := 32);  port( A : in unsigned(WIDTH-1 downto 0);  X : in unsigned(WIDTH-1 downto 0);  clk:in std\_logic;  prod : out unsigned(2\*WIDTH-1 downto 0)  );  end lab6;  architecture Behavioral of lab6 is  --MUX signals  signal sel : std\_logic\_vector(2 downto 0);  signal shift\_cnt : unsigned(3 downto 0);  --MUX I/O will be 35 bits when applying Radix 16  signal zeroA, oneA : unsigned((WIDTH+3)-1 downto 0);  signal twoA, threeA, fourA : unsigned((WIDTH+3)-1 downto 0);  signal fiveA, sixA, sevenA : unsigned((WIDTH+3)-1 downto 0);  signal mux\_out : unsigned((WIDTH+3)-1 downto 0);  signal carry : unsigned(WIDTH-30 downto 0):=(others=>'0');  --Shift and add  signal sum : unsigned((WIDTH+3)-1 downto 0);  signal sum\_32 : unsigned(WIDTH-1 downto 0):=(others=>'0');  signal prod\_concat : unsigned(2\*WIDTH-1 downto 0);  signal carry\_32 : unsigned(WIDTH-1 downto 0):=(others=>'0');  begin  zeroA <= (others => '0');  oneA <= resize(A, (WIDTH+3));  twoA <= oneA sll 1;  threeA <= twoA + oneA;  fourA <= oneA sll 2;  fiveA <= fourA + oneA;  sixA <= fiveA + oneA;  sevenA <= sixA + oneA;    carry<=(sum(WIDTH-30 downto WIDTH-WIDTH));  sum<=mux\_out+sum\_32;  prod\_concat <= sum\_32(WIDTH-1 downto 0) & carry\_32(WIDTH-1 downto 0) ;    sum\_shift: process(clk)  begin  if clk'event and clk='1' then  if (shift\_cnt/=0) then  sum\_32<=sum(WIDTH+2 downto 3);  else  sum\_32<=(others=>'0');  end if;  end if;  end process;  cnt:process(clk)  begin  if clk'event and clk='1' then  -- carry\_32(WIDTH-1 downto WIDTH-3)<=carry;  if(shift\_cnt /= 11) then  shift\_cnt <= shift\_cnt + 1; --shift\_cnt=0 at t=0  else  shift\_cnt <= (others => '0');  end if;    end if;  end process;  out\_data:process(clk)  begin  if falling\_edge (clk) then  if(shift\_cnt=0) then  prod <= prod\_concat sll 1;  end if;  end if;  end process;      carry\_mem:process(clk)  begin  if (clk='1') then  if (shift\_cnt/=0) then  carry\_32(WIDTH-1 downto WIDTH-3)<=carry;  else  carry\_32<=(others=>'0');  end if;  else  carry\_32<=carry\_32 srl 3;  end if;  end process;  mux: process(sel)  begin  -- carry31<=carry\_32 srl 3;  case sel is  --Pass P0  when "000" =>  mux\_out <= zeroA;    --Pass A  when "001" =>  mux\_out <= oneA;    --Pass 2A (shift by 1)  when "010" =>  mux\_out <= twoA;    ---Pass 3A (2A + A)  when "011" =>  mux\_out <= threeA;    ---Pass 4A (shift by 2)  when "100" =>  mux\_out <= fourA;    ---Pass 5A (4A + A)  when "101" =>  mux\_out <= fiveA;    --Pass 6A (5A + A)  when "110" =>  mux\_out <= sixA;    ---Pass 7A (6A + A)  when "111" =>  mux\_out <= sevenA;    when others =>  mux\_out <= mux\_out;  end case;  end process;  sel\_generate: process(shift\_cnt)  begin  --Observe LSBs of X input using Radix 8  case shift\_cnt is    when "0000" =>  sel <= (others=>'0');  when "0001" =>  sel <= std\_logic\_vector(X(2 downto 0));  --Pass A  when "0010" =>  sel <= std\_logic\_vector(X(5 downto 3));  --Pass 2A (shift by 1)  when "0011" =>  sel <= std\_logic\_vector(X(8 downto 6));  ---Pass 3A (2A + A)  when "0100" =>  sel <= std\_logic\_vector(X(11 downto 9));  ---Pass 4A (shift by 2)  when "0101" =>  sel <= std\_logic\_vector(X(14 downto 12));  ---Pass 5A (4A + A)  when "0110" =>  sel <= std\_logic\_vector(X(17 downto 15));  when "0111" =>  sel <= std\_logic\_vector(X(20 downto 18));  --Pass 6A (5A + A)  when "1000" =>  sel <= std\_logic\_vector(X(23 downto 21));  ---Pass 3A (2A + A)  when "1001" =>  sel <= std\_logic\_vector(X(26 downto 24));  ---Pass 4A (shift by 2)  when "1010" =>  sel <= std\_logic\_vector(X(29 downto 27));  ---Pass 5A (4A + A)  when "1011" =>  sel <= '0' & std\_logic\_vector(X(31 downto 30));  when others =>  sel <=sel;    end case;  end process;  end Behavioral; | library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use IEEE.NUMERIC\_STD.ALL;  entity tb\_lab6 is  -- Port ( );  end tb\_lab6;  architecture Behavioral of tb\_lab6 is  component lab6 is  generic(WIDTH : integer := 32);  port( A : in unsigned(WIDTH-1 downto 0);  X : in unsigned(WIDTH-1 downto 0);  clk:in std\_logic;  prod : out unsigned(2\*WIDTH-1 downto 0)  );  end component;  constant WIDTH : integer := 32;  constant PERIOD:time:=10 ns;  signal A,X : unsigned(WIDTH-1 downto 0);  signal prod : unsigned(2\*WIDTH-1 downto 0);  signal clk:std\_logic;  begin  UUT: lab6  port map(A => A, X => X, prod => prod,clk=>clk);  process  begin  X <= to\_unsigned(16#50000000#, A'length);  A <= to\_unsigned(16#A6#, A'length);  wait for 120ns;  X <= to\_unsigned(16#2#, A'length);  wait for 120 ns;  X <= to\_unsigned(16#10#, A'length);  wait for 120 ns;  X <= to\_unsigned(16#100#, A'length);  wait for 120 ns;    X <= to\_unsigned(16#1000#, A'length);  wait for 120 ns;  X <= to\_unsigned(16#10000#, A'length);  wait for 120 ns;    X <= to\_unsigned(16#100000#, A'length);  wait for 120 ns;  X <= to\_unsigned(16#1000000#, A'length);  wait for 120 ns;  X <= to\_unsigned(16#1000000#, A'length);  wait for 120 ns;    X <= to\_unsigned(16#50000000#, A'length);  A <= to\_unsigned(16#50000000#, A'length);  wait for 120ns;  A <= to\_unsigned(16#2#, A'length);  X <= to\_unsigned(16#2#, A'length);  wait for 120 ns;  A <= to\_unsigned(16#10#, A'length);  X <= to\_unsigned(16#10#, A'length);  wait for 120 ns;  A <= to\_unsigned(16#100#, A'length);  X <= to\_unsigned(16#100#, A'length);  wait for 120 ns;  A <= to\_unsigned(16#1000#, A'length);  X <= to\_unsigned(16#1000#, A'length);  wait for 120 ns;  A <= to\_unsigned(16#10000#, A'length);  X <= to\_unsigned(16#10000#, A'length);  wait for 120 ns;  A <= to\_unsigned(16#100000#, A'length);  X <= to\_unsigned(16#100000#, A'length);  wait for 120 ns;  A <= to\_unsigned(16#1000000#, A'length);  X <= to\_unsigned(16#1000000#, A'length);  wait for 120 ns;  A <= to\_unsigned(16#1000000#, A'length);  X <= to\_unsigned(16#1000000#, A'length);  wait for 120 ns;  A <= to\_unsigned(16#10000000#, A'length);  X <= to\_unsigned(16#10000000#, A'length);  end process;  --X <= to\_unsigned(16#AD#, A'length);  process  begin  clk<='1';  wait for PERIOD/2;  clk<= not clk;  wait for PERIOD/2;  end process;    end Behavioral; |