Digital System Design Lab Report, Lab6

Author: Ying Yiwen
Number: 12210159

Abstract

This lab report investigates three distinct multiplier architectures—combinational, repetitive-addition, and pipelined designs—for FPGA-based digital systems. The study aims to evaluate their trade-offs in speed, hardware resource utilization, and throughput. Combinational logic achieves maximum speed through parallel partial product generation but incurs exponential area complexity. Repetitive-addition minimizes resource usage via sequential addition and shift operations, albeit with slower computation. Pipelined design balances speed and throughput by segmenting operations into clock-synchronized stages, enabling parallel processing. Experimental results quantify these trade-offs: combinational design exhibits the fastest single-operation latency, pipelined design maximizes throughput, and repetitive-addition optimizes space efficiency. This analysis provides empirical insights for selecting multiplier architectures based on application-specific constraints in reconfigurable computing systems.

Contents

1 Introduction

Digital arithmetic circuits form the cornerstone of modern FPGA-based computing systems, with multiplier units being particularly critical in applications ranging from signal processing to machine learning accelerators. This experiment explores three distinct multiplier implementations in VHDL: combinational logic, repetitive-addition, and pipelined architectures, each embodying fundamental trade-offs between temporal efficiency, hardware resource utilization, and power consumption.

Theoretical analysis suggests that combinational multipliers achieve maximal throughput through parallel partial product generation and carry propagation, albeit at the cost of exponential area complexity $O(n)^2$. In contrast, repetitive-addition designs leverage sequential finite state machines to iteratively accumulate results, sacrificing speed for linear area scaling $O(n)$. Pipelined architectures introduce temporal segmentation through register staging, theoretically enabling sub-linear latency amortization while maintaining moderate resource overhead.

This systematic comparison aims to quantify how these architectural paradigms manifest in physical FPGA implementations. Key metrics include LUT (Look-Up Table) consumption, maximum clock frequency, and pipeline initiation interval – parameters directly influencing application-specific design choices. Through this experimental framework, we establish empirical correlations between algorithmic complexity, hardware pragmatism, and performance scalability in reconfigurable computing environments.

2 Combinational Design

The combinatorial design is based on the principles of Bool algebra and combinational logic circuits. Multiplication operations can be realized directly through logic gates. Multiplication can be decomposed into generation and accumulation of partial products, and these operations can be represented by Bool expressions and realized by combinational logic circuits.

$$P = A \times B = \sum_{i=0}^{n-1} (A_i \times B) \times 2^i \tag{1}$$

where A_i is the i-th bit of A, B is the multiplier, and 2^i represents the shift operation.

The main goal of the combinatorial design is to achieve the fastest multiplication operation. By reducing the latency of logic gates and optimizing the circuit structure, the operation time can be minimized. However, this design usually sacrifices hardware resources because a large number of logic gates are required to implement complex multiplication logic.

The following is the VHDL code for the combinational design:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity combinational_design is
  port (
    a, b : in std_logic_vector(4 downto 0); -- Input vectors a and b, both 5 bits
        wide
    y : out std_logic_vector(9 downto 0)   -- Output vector y, 10 bits wide to hold
        the product
  );
end combinational_design;
```

```
architecture Behavioral of combinational_design is

  constant WIDTH : integer := 5; -- Define the width of the input vectors
  signal au, bv0, bv1, bv2, bv3, bv4: std_logic_vector (WIDTH-1 downto 0); --
     Intermediate signals
  signal pp0, pp1, pp2, pp3, pp4: std_logic_vector (WIDTH downto 0); -- Signals for
     partial products
  signal prod: std_logic_vector (2*WIDTH-1 downto 0); -- Signal to hold the final
     product

begin

  au <= a; -- Assign input a to signal au

  -- Create vectors bv0 to bv4 where each bit is replicated from the corresponding
     bit of b
  bv0 <= (others => b(0));
  bv1 <= (others => b(1));
  bv2 <= (others => b(2));
  bv3 <= (others => b(3));
  bv4 <= (others => b(4));

  -- Generate the first partial product pp0, Perform bitwise AND between bv0 and au,
      and prepend a '0'
  pp0 <= '0' & (bv0 and au);

  -- Generate the second partial product pp1, Shift pp0 right by one bit and add it
     to the result of bv1 AND au
  pp1 <= ('0' & pp0(WIDTH downto 1)) + ('0' & (bv1 and au));

  -- Generate the third partial product pp2, Shift pp1 right by one bit and add it
     to the result of bv2 AND au
  pp2 <= ('0' & pp1(WIDTH downto 1)) + ('0' & (bv2 and au));

  -- Generate the fourth partial product pp3, Shift pp2 right by one bit and add it
     to the result of bv3 AND au
  pp3 <= ('0' & pp2(WIDTH downto 1)) + ('0' & (bv3 and au));

  -- Generate the fifth partial product pp4, Shift pp3 right by one bit and add it
     to the result of bv4 AND au
  pp4 <= ('0' & pp3(WIDTH downto 1)) + ('0' & (bv4 and au));

  -- Concatenate the least significant bits of all partial products to form the
     final product
  prod <= pp4 & pp3(0) & pp2(0) & pp1(0) & pp0(0);

  -- Assign the final product to the output y
  y <= prod;

end Behavioral;
```

After synthesis and implementations, we can get the schematic diagram as follows:

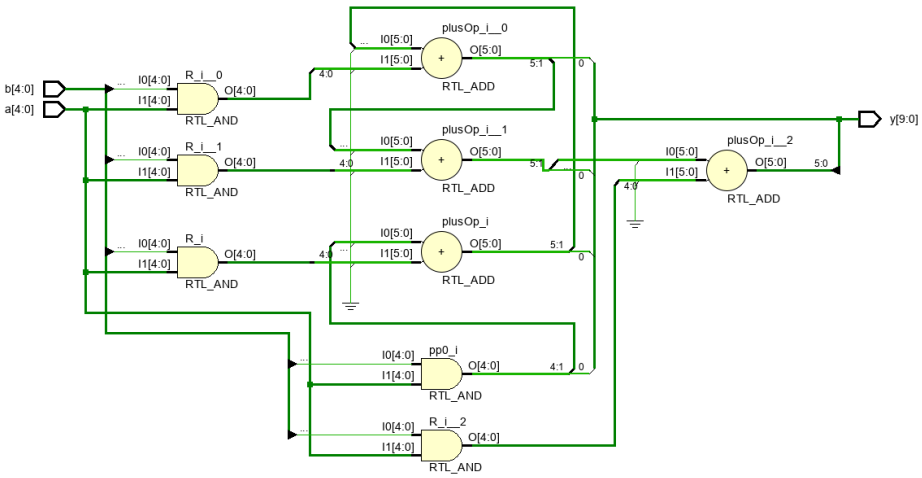

Figure 1: Combinational Design Schematic

Using the testbench in the appendix, we can get the result. It computes rightly.

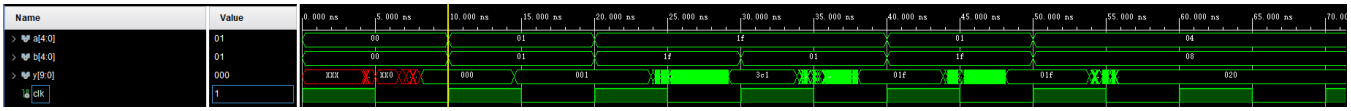

Figure 2: Combinational Design Result

3 Repetitive-Addition Design

Repeated addition design is based on basic addition and shift operations. Multiplication can be viewed as a repeated execution of addition. $A \times B$ can be represented as adding A to B times. The weight assignment in multiplication can be realized by shift operation.

The main goal of a repetitive-addition design is to reduce the use of hardware resources. By using simple adders and shift registers, a smaller chip area and lower power consumption can be achieved, but at the expense of computing speed.

The following is the VHDL code for the repetitive-addition design:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Entity declaration
entity repetitve_addtion is
  port (
    CLK     : in  std_logic;                     -- System clock
    RESET   : in  std_logic;                     -- Active-high reset
    start   : in  std_logic;                     -- Start computation signal
    a_in    : in  std_logic_vector(4 downto 0);  -- Multiplicand input
    b_in    : in  std_logic_vector(4 downto 0);  -- Multiplier input
    r       : out std_logic_vector(9 downto 0); -- 16-bit result output
    ready   : out std_logic                      -- Computation complete flag
```

```
  );
end entity repetitve_addtion;

architecture Behavioral of repetitve_addtion is
  constant WIDTH : integer := 5;                  -- Data width configuration
  type state_type is (idle,  ab0, load, op);     -- FSM states:
    -- idle: Initial/waiting state
    -- ab0:  Handle zero input case
    -- load: Initialize registers
    -- op:   Multiplication operation state

  -- State and data registers
  signal state_reg, state_next : state_type;
  signal a_reg, a_next         : std_logic_vector(WIDTH-1 downto 0); -- Multiplicand
      storage
  signal n_reg, n_next         : std_logic_vector(WIDTH-1 downto 0); -- Counter
     storage
  signal r_reg, r_next         : std_logic_vector(2*WIDTH-1 downto 0); -- Result
     accumulation

begin
  -- Synchronous State and Data Registers Update Process
  process (CLK, RESET) is
  begin
    if RESET = '1' then           -- Asynchronous reset
      state_reg <= idle;        -- Return to initial state
      a_reg     <= (others => '0');  -- Clear multiplicand register
      n_reg     <= (others => '0');  -- Clear counter register
      r_reg     <= (others => '0'); -- Clear result register
    elsif CLK'event and CLK = '1' then  -- Clock edge triggered update
      state_reg <= state_next;  -- Update state register
      a_reg     <= a_next;      -- Update multiplicand register
      n_reg     <= n_next;      -- Update counter register
      r_reg     <= r_next;      -- Update result register
    end if;
  end process;

  -- Combinational Next-State and Data Path Logic
  process (start, state_reg, a_reg, n_reg, r_reg, a_in, b_in) is
  begin
    -- Default register values (prevent latches)
    a_next  <= a_reg;
    n_next  <= n_reg;
    r_next  <= r_reg;
    ready   <= '0';  -- Default ready signal state

    case state_reg is
      when idle =>
        ready <= '1';  -- System ready for new operation
        if start = '1' then
          -- Handle zero input special case
          if (a_in = x"00" or b_in = x"00") then
            state_next <= ab0;
          else
```

```
            state_next <= load;  -- Proceed to normal operation
          end if;
        else
          state_next <= idle;  -- Maintain idle state
        end if;

      when ab0 =>
        -- Immediate result for zero input case
        a_next   <= a_in;        -- Store multiplicand
        n_next   <= b_in;        -- Store multiplier
        r_next   <= (others => '0');  -- Clear result
        state_next <= idle;      -- Return to idle

      when load =>
        -- Initialize registers for multiplication
        a_next   <= a_in;         -- Load multiplicand
        n_next   <= b_in;         -- Initialize counter
        r_next   <= (others => '0');  -- Reset accumulator
        state_next <= op;         -- Proceed to operation

      when op =>
        -- Core multiplication iteration
        n_next   <= n_reg - 1;          -- Decrement counter
        r_next   <= ("0000" & a_reg) + r_reg;  -- Accumulate sum

        -- Check completion condition
        if (n_reg = x"01") then  -- Last iteration
          state_next <= idle;  -- Return to idle
        else
          state_next <= op;    -- Continue iterations
        end if;
    end case;
  end process;

  -- Connect internal register to output port
  r <= r_reg;

end architecture Behavioral;
```

After synthesis and implementations, we can get the schematic diagram as follows:

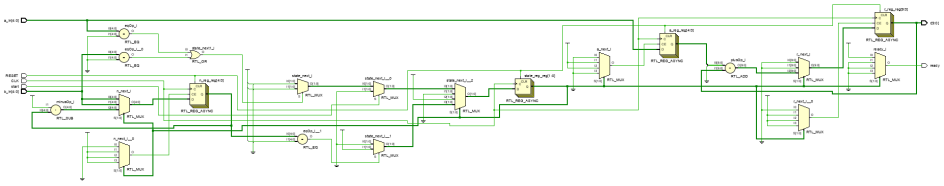

Figure 3: Repetitive-Addition Design Schematic

Using the testbench in the appendix, we can get the result. It computes rightly.

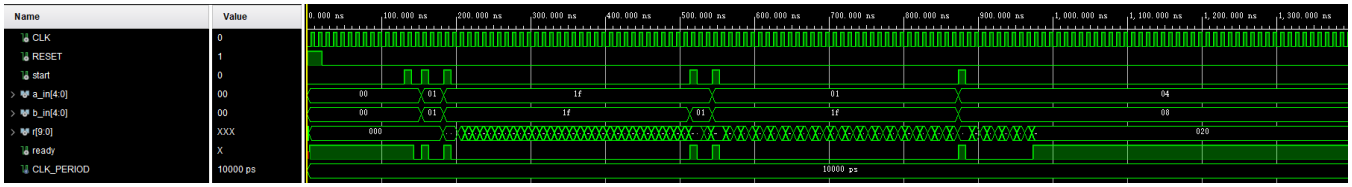

Figure 4: Repetitive-Addition Design Result

4 Pipelined Design

The pipeline design is based on the pipeline principle, which decomposes a complex multiplication operation into several simple stages and inserts registers between each stage. Each stage can be completed in a single clock cycle, thus enabling parallel processing.

The main goal of the pipelined design is to increase the throughput rate, i.e., the number of multiplication operations completed per unit of time. With parallel processing, the pipelined design can significantly improve the performance of the system by outputting a multiplication result in each clock cycle.

For an n-bit multiplier, if it's divided into m pipeline stages, each with a delay of T_{stage}, the delay of a single operation is:

$$T_{total} = T_{stage} \times m \tag{2}$$

The throughput rate of a pipeline design can be expressed as:

$$\text{Throughput} = \frac{1}{T_{clock}} \tag{3}$$

The following is the VHDL code for the pipelined design:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;     -- Legacy arithmetic package
use IEEE.STD_LOGIC_UNSIGNED.ALL; -- Unsigned operations

-- Entity Declaration
entity pipeline_multiplier is
  generic(WIDTH : integer := 5);  -- Default 5-bit operand width
  port(
    clk    : in  std_logic;                      -- System clock
    reset  : in  std_logic;                      -- Active-high reset
    a      : in  std_logic_vector(WIDTH-1 downto 0); -- Multiplicand
    b      : in  std_logic_vector(WIDTH-1 downto 0); -- Multiplier
    y      : out std_logic_vector(9 downto 0)     -- 10-bit product output
  );
end entity;

architecture Behavioral of pipeline_multiplier is
  -- Pipeline registers for propagating multiplicand
  signal a2_reg, a3_reg, a4_reg      : std_logic_vector(WIDTH-1 downto 0);
  signal a1, a2_next, a3_next, a4_next: std_logic_vector(WIDTH-1 downto 0);

  -- Shifted multiplier bits for partial product generation
  signal b1          : std_logic_vector(4 downto 1);  -- Stage1 multiplier slice
  signal b2_next, b2_reg: std_logic_vector(4 downto 2); -- Stage2 multiplier slice
```

```
  signal b3_next, b3_reg: std_logic_vector(4 downto 3); -- Stage3 multiplier slice
  signal b4_next, b4_reg: std_logic_vector(4 downto 4); -- Stage4 multiplier slice

  -- Bit-vector replication and partial products
  signal bv0, bv1, bv2, bv3, bv4: std_logic_vector(4 downto 0); -- Bit replication
     buses
  signal bp0, bp1, bp2, bp3, bp4: std_logic_vector(5 downto 0); -- Partial products

  -- Pipelined accumulated results
  signal pp0              : std_logic_vector(5 downto 0);  -- Stage0 partial product
  signal pp1_next, pp1_reg: std_logic_vector(6 downto 0); -- Stage1 accumulation
  signal pp2_next, pp2_reg: std_logic_vector(7 downto 0); -- Stage2 accumulation
  signal pp3_next, pp3_reg: std_logic_vector(8 downto 0); -- Stage3 accumulation
  signal pp4_next, pp4_reg: std_logic_vector(9 downto 0); -- Stage4 final result

begin
  -- Pipeline Stage 0-1: Initial Partial Product Generation
  -- Generate LSB partial product (b[0] * a)
  bv0 <= (others => b(0));          -- Replicate b[0] for AND operation
  bp0 <= "0" & (bv0 and a);         -- Zero-extended AND result
  pp0 <= bp0;                       -- Direct partial product assignment
  a1  <= a;                         -- First stage multiplicand
  b1  <= b(4 downto 1);             -- Shifted multiplier bits

  -- Pipeline Stage 1-2: First Accumulation Stage
  -- Generate b[1] partial product and accumulate
  bv1 <= (others => b1(1));         -- Replicate b[1] bit
  bp1 <= "0" & (bv1 and a1);        -- Shifted partial product
  pp1_next(6 downto 1) <= ("0" & pp0(5 downto 1)) + bp1; -- Accumulate with shift
  pp1_next(0)          <= pp0(0);   -- Carry LSB through pipeline
  a2_next              <= a1;       -- Propagate multiplicand
  b2_next              <= b1(4 downto 2); -- Shift multiplier slice

  -- Pipeline Stage 2-3: Second Accumulation Stage
  bv2 <= (others => b2_reg(2));     -- Replicate current multiplier bit
  bp2 <= "0" & (bv2 and a2_reg);    -- Generate shifted partial product
  pp2_next(7 downto 2) <= ("0" & pp1_reg(6 downto 2)) + bp2; -- Accumulate
  pp2_next(1 downto 0) <= pp1_reg(1 downto 0); -- Preserve lower bits
  a3_next              <= a2_reg;   -- Continue multiplicand propagation
  b3_next              <= b2_reg(4 downto 3); -- Shift multiplier slice

  -- Pipeline Stage 3-4: Third Accumulation Stage
  bv3 <= (others => b3_reg(3));     -- Current multiplier bit replication
  bp3 <= "0" & (bv3 and a3_reg);    -- Shifted partial product
  pp3_next(8 downto 3) <= ("0" & pp2_reg(7 downto 3)) + bp3; -- Accumulate
  pp3_next(2 downto 0) <= pp2_reg(2 downto 0); -- Preserve lower bits
  a4_next              <= a3_reg;   -- Final multiplicand propagation
  b4_next              <= b3_reg(4 downto 4); -- Final multiplier bit

  -- Pipeline Stage 4-5: Final Accumulation Stage
  bv4 <= (others => b4_reg(4));     -- MSB replication
  bp4 <= "0" & (bv4 and a4_reg);    -- MSB partial product
  pp4_next(9 downto 4) <= ("0" & pp3_reg(8 downto 4)) + bp4; -- Final sum
  pp4_next(3 downto 0) <= pp3_reg(3 downto 0); -- Preserve LSBs
```

```
  -- Pipeline Register Update Process
  pipe_reg: process(clk, reset)
  begin
    if reset = '1' then
      -- Asynchronous reset clears all pipeline registers
      pp1_reg <= (others => '0');
      pp2_reg <= (others => '0');
      pp3_reg <= (others => '0');
      pp4_reg <= (others => '0');
      a2_reg  <= (others => '0');
      a3_reg  <= (others => '0');
      a4_reg  <= (others => '0');
      b2_reg  <= (others => '0');
      b3_reg  <= (others => '0');
      b4_reg  <= (others => '0');
    elsif rising_edge(clk) then
      -- Clock-driven pipeline propagation
      pp1_reg <= pp1_next;
      pp2_reg <= pp2_next;
      pp3_reg <= pp3_next;
      pp4_reg <= pp4_next;
      a2_reg  <= a2_next;
      a3_reg  <= a3_next;
      a4_reg  <= a4_next;
      b2_reg  <= b2_next;
      b3_reg  <= b3_next;
      b4_reg  <= b4_next;
    end if;
  end process;

  -- Connect final pipeline stage to output
  y <= pp4_reg;

end architecture Behavioral;
```

After synthesis and implementations, we can get the schematic diagram as follows:

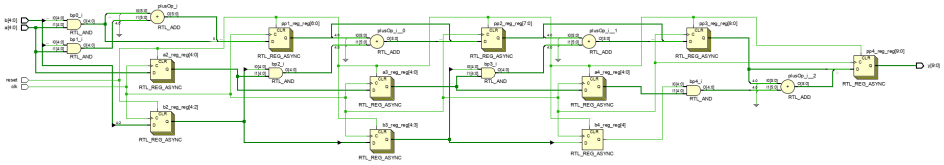

Figure 5: Pipelined Design Schematic

Using the testbench in the appendix, we can get the result. It computes rightly.

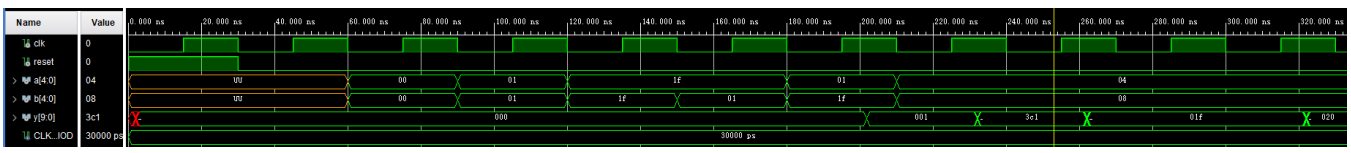

Figure 6: Pipelined Design Result

5 Code Appendix

Testbench codes are as follows, I test 0*0, 1*1, 31*31, 31*1, 1*31, 4*8.

```
-- Initialize and reset
RESET <= '1';
wait for CLK_PERIOD * 2;
RESET <= '0';
wait for CLK_PERIOD;
wait for 100 ns;

-- Test 1: 0 * 0
a_in <= "00000"; b_in <= "00000";
start <= '1';
wait for CLK_PERIOD;
start <= '0';
expected_r := std_logic_vector(to_unsigned(0, 10));
wait until ready = '1';
assert r = expected_r report "Test 1 failed: 0*0" severity error;

-- Test 2: 1 * 1
a_in <= "00001"; b_in <= "00001";
start <= '1';
wait for CLK_PERIOD;
start <= '0';
expected_r := std_logic_vector(to_unsigned(1, 10));
wait until ready = '1';
assert r = expected_r report "Test 2 failed: 1*1" severity error;

-- Test 3: Max value * Max value (31*31)
a_in <= "11111"; b_in <= "11111";
start <= '1';
wait for CLK_PERIOD;
start <= '0';
expected_r := std_logic_vector(to_unsigned(31*31, 10));
wait until ready = '1';
assert r = expected_r report "Test 3 failed: 31*31" severity error;

-- Test 4: Max * 1
a_in <= "11111"; b_in <= "00001";
start <= '1';
wait for CLK_PERIOD;
start <= '0';
expected_r := std_logic_vector(to_unsigned(31, 10));
wait until ready = '1';
assert r = expected_r report "Test 4 failed: 31*1" severity error;

-- Test 5: 1 * Max
a_in <= "00001"; b_in <= "11111";
start <= '1';
wait for CLK_PERIOD;
start <= '0';
expected_r := std_logic_vector(to_unsigned(31, 10));
wait until ready = '1';
assert r = expected_r report "Test 5 failed: 1*31" severity error;
```

```
-- Test 6: Power of 2 multiplication
a_in <= "00100"; b_in <= "01000";  -- 4 * 8
start <= '1';
wait for CLK_PERIOD;
start <= '0';
expected_r := std_logic_vector(to_unsigned(32, 10));
wait until ready = '1';
assert r = expected_r report "Test 6 failed: 4*8" severity error;

-- End of test
```

6 Analysis

6.1 Speed

The multiplier of the **combinatorial design** is based on a purely combinational logic circuit implementation. It generates the output by passing the input signal directly through the combinational logic circuit without the constraints of a clock signal, thus allowing the fastest arithmetic speed in theory. The output signal can be generated almost immediately as long as the input signal is stable and the delay depends mainly on the propagation delay of the combinational logic circuit.

Repetitive-addition design is a multiplication implementation based on addition and shift operations. It accomplishes the multiplication operation by multiple addition operations, each of which corresponds to one bit of the multiplication. Therefore, it is relatively slow and the operation time is proportional to the number of bits of the multiplier.

The **pipeline design** achieves parallel processing by breaking the multiplication operation into multiple stages and inserting registers between each stage. Each stage can be completed in a single clock cycle, so the pipelined design can significantly increase the throughput rate of the multiplier.

Therefore, in terms of operating speed:

$$\text{Combinatorial Design} > \text{Pipeline Design} > \text{Repetitive-Addition Design} \tag{4}$$

Test Case	0*0	1*1	31*31	31*1	1*31	4*8
Combinational	10	20	320	20	320	90
Repetitive-Addition	3.137	4.547	9.162	8.103	8.147	5.693

Table 1: Time of Combinational Design and Repetitive-Addition Design (Unit:ns)

6.2 Space Usage

The **combinatorial design** requires a large number of logic gates to implement multiplication operations, especially for multi-digit numbers. It requires the implementation of the complete multiplication logic, including operations such as partial product generation and summation, and therefore consumes a large amount of hardware resources.

The hardware resource requirements of a **repetitive-addition design** are relatively low because it relies heavily on adders and shift registers. The adder is relatively simple to implement and the shift registers require less logic resources.

A **pipelined design** requires registers to be inserted between each stage and therefore takes up additional hardware resources. In addition, each stage also needs to implement some of the multiplication logic, so the overall hardware resource requirements are higher than for a repeated addition design, but usually lower than for a combinatorial design.

Therefore, in terms of space usage:

$$\text{Combinatorial Design} > \text{Pipeline Design} > \text{Repetitive-Addition Design} \tag{5}$$

6.3 Throughput

The multiplier of a **combinatorial design** can theoretically achieve the fastest arithmetic speed because it is based on a purely combinational logic circuit implementation without the constraint of a clock signal. The output signal can be generated almost immediately as long as the input signal is stable. Therefore, the throughput of a combinational design depends mainly on the propagation delay of the combinational logic circuit.

The multiplier of the **repetitive-addition** design accomplishes the multiplication operation by multiple addition operations, each of which corresponds to one bit of the multiplication. Therefore, its throughput is relatively slow and the operation time is proportional to the number of bits in the multiplier.

The **pipelined design** of the multiplier achieves parallel processing by breaking the complex multiplication operation into multiple simple stages and inserting registers between each stage. Each stage can be completed in a single clock cycle, thus enabling parallel processing. As a result, the throughput of a pipelined design can be significantly increased.

Therefore, in terms of throughput:

$$\text{Pipeline Design} > \text{Combinatorial Design} > \text{Repetitive-Addition Design} \tag{6}$$

7 Conclusion

The comparative analysis of combinational, repetitive-addition, and pipelined multipliers reveals distinct performance characteristics. Combinational design excels in single-operation speed but demands significant hardware resources. Repetitive-addition reduces area usage compared to combinational logic, though at the cost of linearly scaled computation time. Pipelined architectures achieve the highest throughput (1 result per clock cycle after pipeline filling) while maintaining moderate resource overhead. For high-speed real-time applications, combinational or pipelined designs are preferable; resource-constrained systems benefit from repetitive-addition. This study underscores the importance of architectural trade-offs in FPGA-based arithmetic unit design.