Performance and Cost Comparison of Several Multiplier Architectures

Wang Zhuoyang[1]

[1] *12112907, Department of Electrical and Electronic Engineering, SUSTech. Email: glverfer@outlook.com*

Abstract

No abstract.

Keywords: Multiplier; Comparison; VHDL.

Contents

Digital System Designing Laboratory Reports (2023)

Corresponding author
Wang Zhuoyang

1 Introduction

This experiment designs and implements several basic multiplier architectures, including two combinational logic implementations, a state machine implementation based on accumulation, and a pipeline implementation for large amounts of data. The experiment presents the design ideas and VHDL code for each architecture, verifies the correctness and timing delay of the simulation results after behavioral simulation and synthesis, draws RTL diagrams and synthesized schematics, and compares and analyzes other indicators such as device utilization.

2 Multiplier Architectures Comparison

2.1 Combinational Design

Principle

The first design is implemented using pure combinational logic circuits, which has a higher delay than sequential logic circuits within a single clock cycle. The basic idea of the design is to mimic the way humans calculate by listing numbers vertically. Each bit of the second operand is bitwise ANDed with the first operand, shifted accordingly, and then added together to obtain the result (Figure 1).

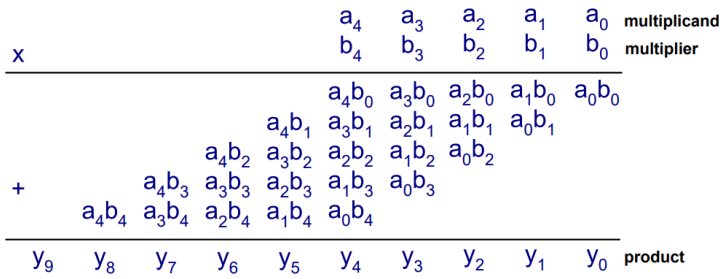

Figure 1. Do multiplication by listing the number vertically.

Implementation

The code is shown in Code 1.

```
...

architecture arch_1 of combinational_multiplier_16 is
    signal au, bv0, bv1, bv2, bv3, bv4, bv5, bv6, bv7, bv8, bv9, bv10, bv11,
     bv12, bv13, bv14, bv15: std_logic_vector(15 downto 0);
    signal p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13, p14,
    p15, prod: std_logic_vector(31 downto 0);
begin
    au <= a;
    bv0 <= (others => b(0));
    bv1 <= (others => b(1));
    bv2 <= (others => b(2));
    bv3 <= (others => b(3));
    bv4 <= (others => b(4));
    bv5 <= (others => b(5));
    bv6 <= (others => b(6));
    bv7 <= (others => b(7));
    bv8 <= (others => b(8));
    bv9 <= (others => b(9));
    bv10 <= (others => b(10));
    bv11 <= (others => b(11));
    bv12 <= (others => b(12));
```

```
    bv13 <= (others => b(13));
    bv14 <= (others => b(14));
    bv15 <= (others => b(15));
    p0 <= "0000000000000000" & (bv0 and au);
    p1 <= "000000000000000" & (bv1 and au) & '0';
    p2 <= "00000000000000" & (bv2 and au) & "00";
    p3 <= "0000000000000" & (bv3 and au) & "000";
    p4 <= "000000000000" & (bv4 and au) & "0000";
    p5 <= "00000000000" & (bv5 and au) & "00000";
    p6 <= "0000000000" & (bv6 and au) & "000000";
    p7 <= "000000000" & (bv7 and au) & "0000000";
    p8 <= "00000000" & (bv8 and au) & "00000000";
    p9 <= "0000000" & (bv9 and au) & "000000000";
    p10 <= "000000" & (bv10 and au) & "0000000000";
    p11 <= "00000" & (bv11 and au) & "00000000000";
    p12 <= "0000" & (bv12 and au) & "000000000000";
    p13 <= "000" & (bv13 and au) & "0000000000000";
    p14 <= "00" & (bv14 and au) & "00000000000000";
    p15 <= "0" & (bv15 and au) & "000000000000000";
    prod <= (((p0 + p1) + (p2 + p3)) + ((p4 + p5) + (p6 + p7))) + (((p8 + p9
    ) + (p10 + p11)) + ((p12 + p13) + (p14 + p15)));
    y <= prod;
end architecture;
```

Listing 1. Code for combinational design (architecture 1).

RTL Diagram and Synthesized Schematic

The RTL diagram is shown in Figure 2 and the synthesized schematic is shown in 3.

Analysis is not provided here. It will be carried out and compared with other methods later in the following sections.

Simulation

An example for the result of the behavioral simulation is shown in Figure 4 and the one for post-synthesized simulation is shown in Figure 5.

The information in the figure confirms the correctness of the calculation results.

Moreover, the timing delay can be observed in the timing simulation. Taking the calculation starting from 269550 ns as an example, the estimated time before the result becomes stable is about 6.816 ns. Next, we will compare its performance with another combinational logic architecture.

Specifications

The device utilization is shown in Figure 6.

And the delay is revealed in the simulation results.

2.2 Another Combinational Design

Principle

The second combinational design mainly optimized the resource consumption of the first one. The basic idea is to reduce the resource consumption caused by the 0 position generated by shifting (Figure 7). However, this solution will obviously reduce the performance because the intermediate results of the original solution are obtained by parallel calculation and then added to get the final result, while the intermediate results of the new solution have a cascading structure, which will increase the circuit delay.

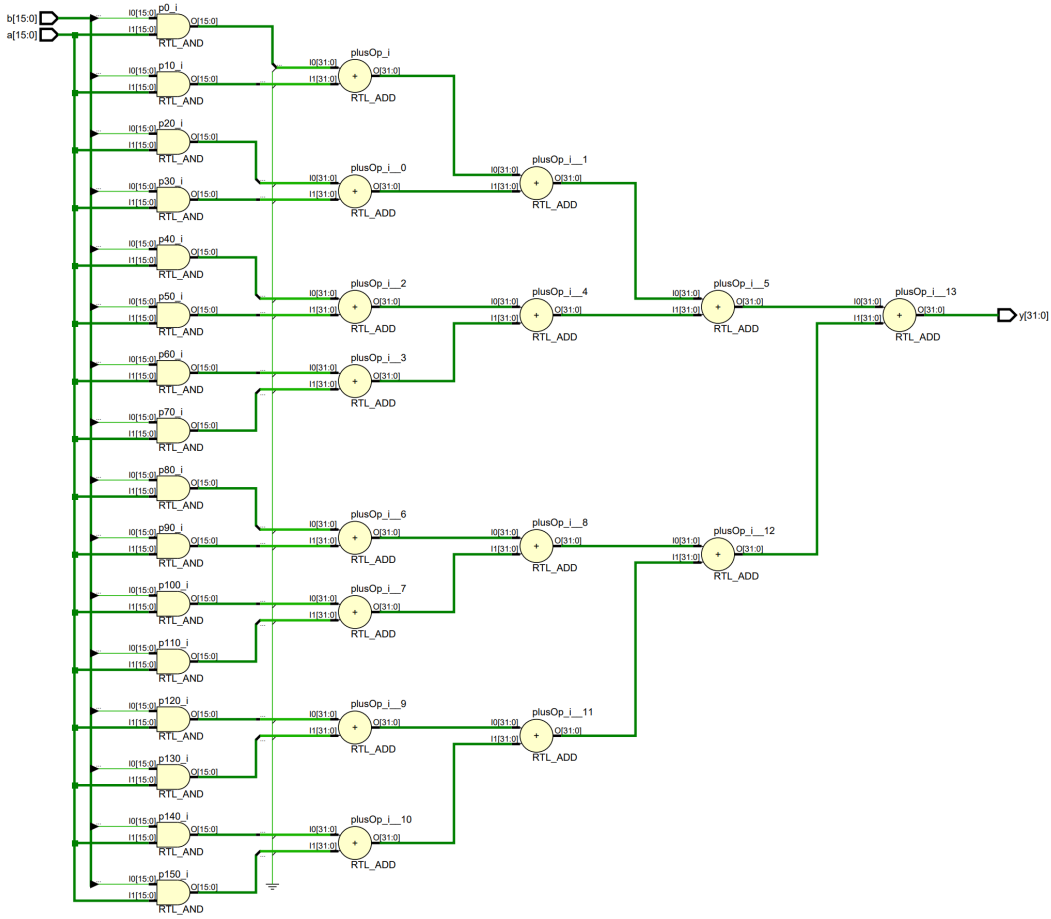

Figure 2. The RTL diagram for the first combinational design.

Implementation

The code is shown in Code 2.

```
architecture arch_2 of combinational_multiplier_16 is
    signal au, bv0, bv1, bv2, bv3, bv4, bv5, bv6, bv7, bv8, bv9, bv10, bv11,
     bv12, bv13, bv14, bv15: std_logic_vector(15 downto 0);
    signal pp0, pp1, pp2, pp3, pp4, pp5, pp6, pp7, pp8, pp9, pp10, pp11,
    pp12, pp13, pp14, pp15: std_logic_vector(16 downto 0);
    signal prod: std_logic_vector(31 downto 0);
begin
    au <= a;
    bv0 <= (others => b(0));
    bv1 <= (others => b(1));
    bv2 <= (others => b(2));
    bv3 <= (others => b(3));
    bv4 <= (others => b(4));
    bv5 <= (others => b(5));
    bv6 <= (others => b(6));
    bv7 <= (others => b(7));
    bv8 <= (others => b(8));
    bv9 <= (others => b(9));
    bv10 <= (others => b(10));
    bv11 <= (others => b(11));
    bv12 <= (others => b(12));
    bv13 <= (others => b(13));
    bv14 <= (others => b(14));
```

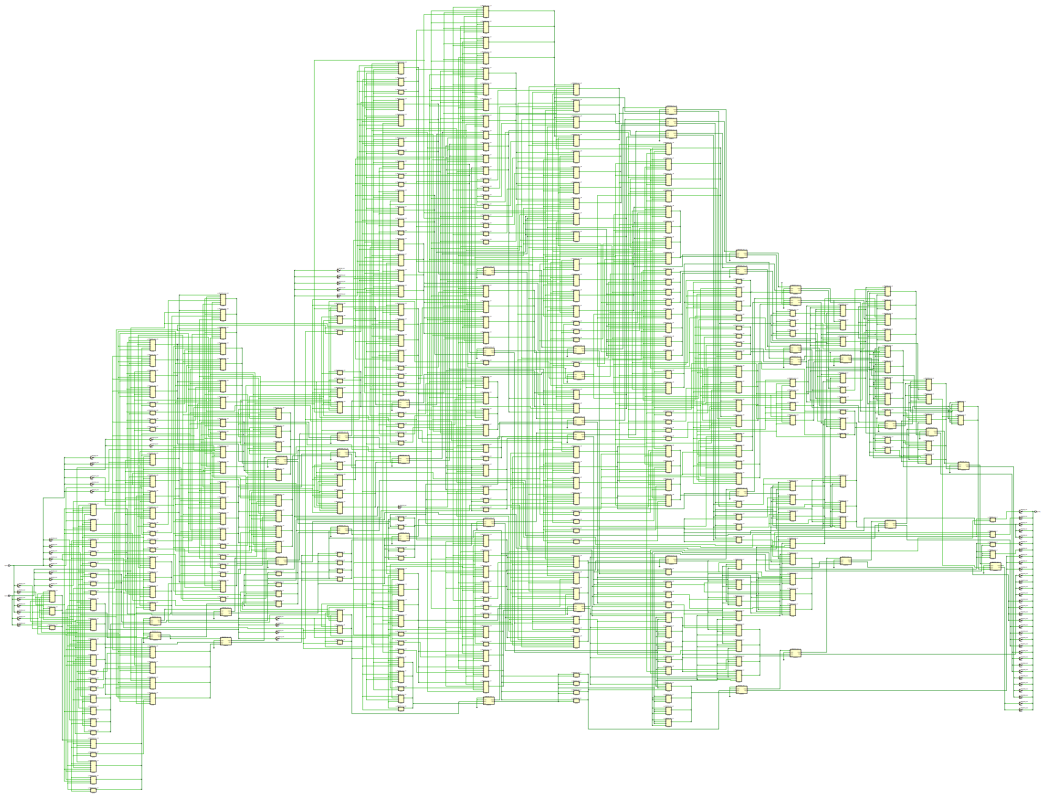

Figure 3. The synthesized schematic for the first combinational design.

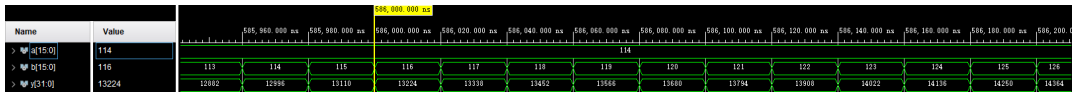

Figure 4. The result of the behavioral simulation for the first combinational design.

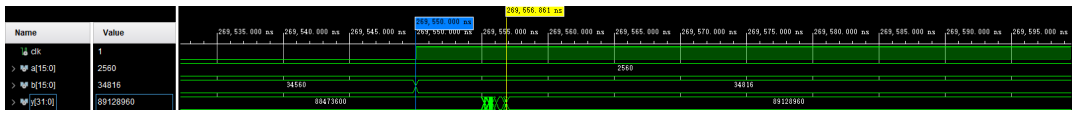

Figure 5. The result of the post-synthesized simulation for the first combinational design.

Utilization

Post-Synthesis | Post-Implementation

Graph | Table

Resource	Estimation	Available	Utilization %
LUT	294	63400	0.46
IO	64	210	30.48

Figure 6. The device utilization for the first combinational design.

```
22      bv15 <= (others => b(15));
23      pp0 <= '0' & (bv0 and au);
24      pp1 <= ('0' & pp0(16 downto 1)) + ('0' & (bv1 and au));
25      pp2 <= ('0' & pp1(16 downto 1)) + ('0' & (bv2 and au));
26      pp3 <= ('0' & pp2(16 downto 1)) + ('0' & (bv3 and au));
27      pp4 <= ('0' & pp3(16 downto 1)) + ('0' & (bv4 and au));
28      pp5 <= ('0' & pp4(16 downto 1)) + ('0' & (bv5 and au));
29      pp6 <= ('0' & pp5(16 downto 1)) + ('0' & (bv6 and au));
```

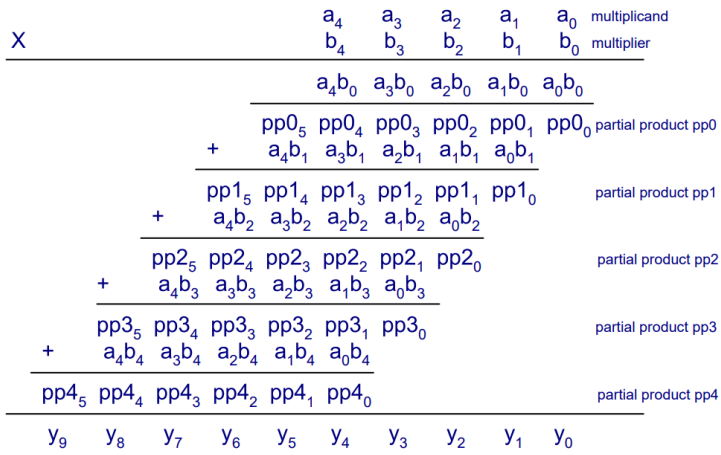

Figure 7. Optimize the device utilization by separating the partial products.

```
    pp7 <= ('0' & pp6(16 downto 1)) + ('0' & (bv7 and au));
    pp8 <= ('0' & pp7(16 downto 1)) + ('0' & (bv8 and au));
    pp9 <= ('0' & pp8(16 downto 1)) + ('0' & (bv9 and au));
    pp10 <= ('0' & pp9(16 downto 1)) + ('0' & (bv10 and au));
    pp11 <= ('0' & pp10(16 downto 1)) + ('0' & (bv11 and au));
    pp12 <= ('0' & pp11(16 downto 1)) + ('0' & (bv12 and au));
    pp13 <= ('0' & pp12(16 downto 1)) + ('0' & (bv13 and au));
    pp14 <= ('0' & pp13(16 downto 1)) + ('0' & (bv14 and au));
    pp15 <= ('0' & pp14(16 downto 1)) + ('0' & (bv15 and au));
    prod <= pp15 & pp14(0) & pp13(0) & pp12(0) & pp11(0) & pp10(0) & pp9(0)
    & pp8(0) & pp7(0) & pp6(0) & pp5(0) & pp4(0) & pp3(0) & pp2(0) & pp1(0)
    & pp0(0);
    y <= prod;
end architecture;
```

Listing 2. Code for combinational design (architecture 2).

RTL Diagram and Synthesized Schematic

The RTL diagram is shown in Figure 8 and the synthesized schematic is shown in 9.

Comparing the results here with the original design, it is clear that the new design has a cascading structure in its RTL diagram, which results in a longer critical path than the original design. This is also clearly reflected in the synthesized schematic, which shows a longer and flatter structure, indicating longer signal paths.

Simulation

An example for the result of the behavioral simulation is shown in Figure 10 and the one for post-synthesized simulation is shown in Figure 11.

After verifying the correctness of the results from the simulation, we can see that the delay of the operation at the same time point (269550 ns) with the original scheme is approximately 11.416 ns, which is significantly longer than the delay of the original scheme. Due to time constraints, we did not conduct a general statistical analysis on other cases.

Specifications

The device utilization is shown in Figure 12

The resource utilization of 241 LUTs for the second design is less than the original design's 294 LUTs. However, the actual optimization level is not as high as expected due to the characteristics of

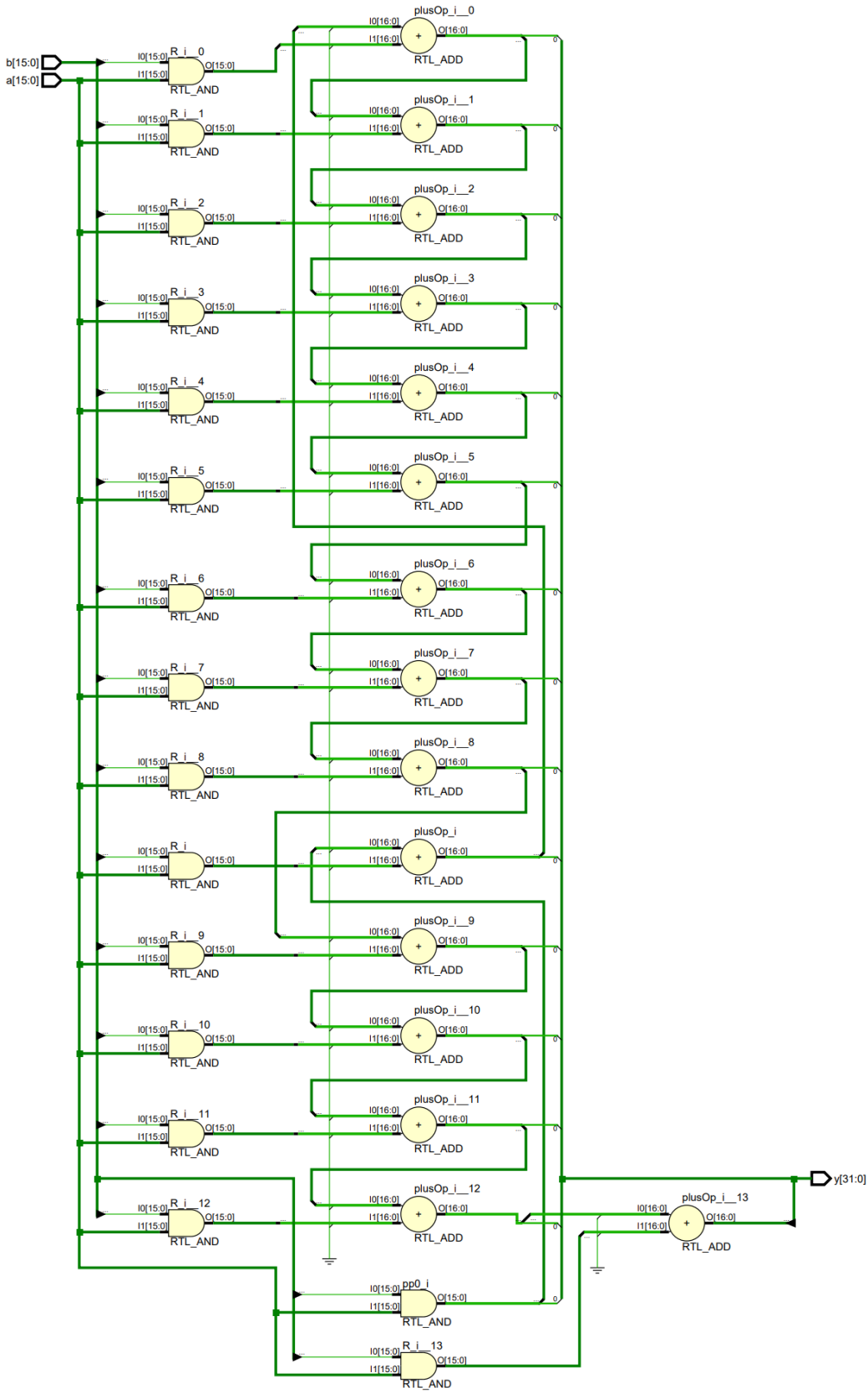

Figure 8. The RTL diagram for the second combinational design.

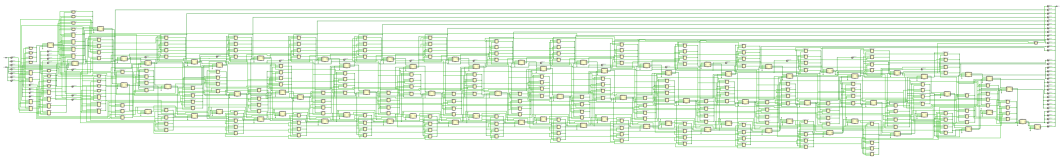

Figure 9. The synthesized schematic for the second combinational design.

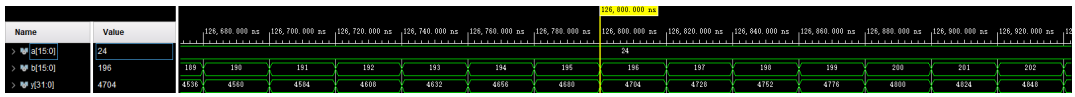

Figure 10. The result of the behavorial simulation for the second combinational design.

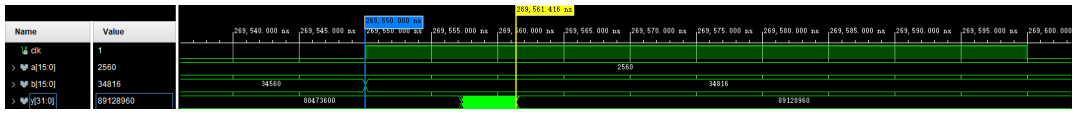

Figure 11. The result of the post-synthesized simulation for the second combinational design.

Utilization — Post-Synthesis | Post-Implementation

Graph | Table

Resource	Estimation	Available	Utilization %
LUT	241	63400	0.38
IO	64	210	30.48

Figure 12. The device utilization for the second combinational design.

the synthesizer. It can be considered in situations where resources are extremely limited.

The delay is revealed in the simulation results.

2.3 Sequential Repetitive-addition Design

Principle

This scheme adopts the idea of FSMD and implements multiplication in an accumulative way. The specific principle can be seen in the ASM chart in Figure 13.

The advantage of this method is that it is based on synchronous circuits and can be applied in situations where the clock cycle is too short to support large-scale combinational logic operations. However, the disadvantage is that the required clock cycle is too long, and it depends on the decimal magnitude of one operand, resulting in very high delay.

Implementation

The code is shown in Code 3.

```
...

entity repetitive_addition_multiplier_16 is
    port (
        clk, reset, start: in std_logic;
        a_in, b_in: in std_logic_vector(15 downto 0);
        ready: out std_logic;
        r: out std_logic_vector(31 downto 0)
    );
end entity;

architecture Behavioral of repetitive_addition_multiplier_16 is
```

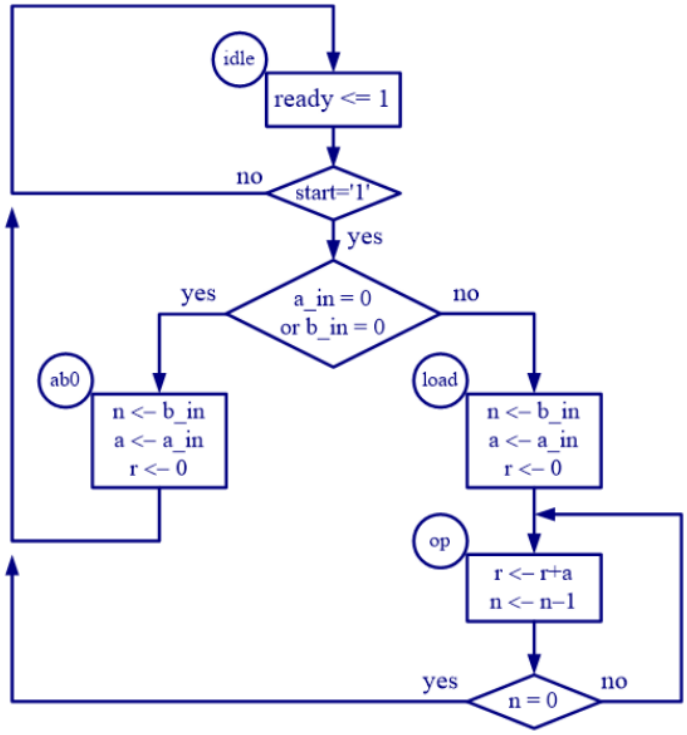

Figure 13. The ASM chart for the repetitive-addition multiplier.

```
    type state_type is (idle, ab0, load, op);

    signal state_reg, state_next: state_type;
    signal a_reg, a_next, n_reg, n_next: std_logic_vector(15 downto 0);
    signal r_reg, r_next: std_logic_vector(31 downto 0);
begin
    process (clk, reset) is
    begin
        if reset = '1' then
            state_reg <= idle;
            a_reg <= (others => '0');
            n_reg <= (others => '0');
            r_reg <= (others => '0');
        elsif CLK'event and CLK = '1' then
            state_reg <= state_next;
            a_reg <= a_next;
            n_reg <= n_next;
            r_reg <= r_next;
        end if;
    end process;

    process (start, state_reg, a_reg, n_reg, r_reg, a_in, b_in) is
    begin
        a_next <= a_reg;
        n_next <= n_reg;
        r_next <= r_reg;

        ready <= '0';

        case state_reg is
            when idle =>
```

```
                if start = '1' then
                    if (a_in = 0 or b_in = 0) then
                        state_next <= ab0;
                    else
                        state_next <= load;
                    end if;
                else
                    state_next <= idle;
                end if;

                ready <= '1';
            when ab0 =>
                a_next <= a_in;
                n_next <= b_in;
                r_next <= (others => '0');
                state_next <= idle;
            when load =>
                a_next <= a_in;
                n_next <= b_in;
                r_next <= (others => '0');
                state_next <= op;
            when op =>
                n_next <= n_reg - 1;
                r_next <= ("0000000000000000" & a_reg) + r_reg;
                if (n_reg = 1) then
                    state_next <= idle;
                else
                    state_next <= op;
                end if;
        end case;
    end process;
    r <= r_reg;
end architecture;
```

Listing 3. Code for sequential repetitive-addition multiplier.

RTL Diagram and Synthesized Schematic

The RTL diagram is shown in Figure 14 and the synthesized schematic is shown in 15.

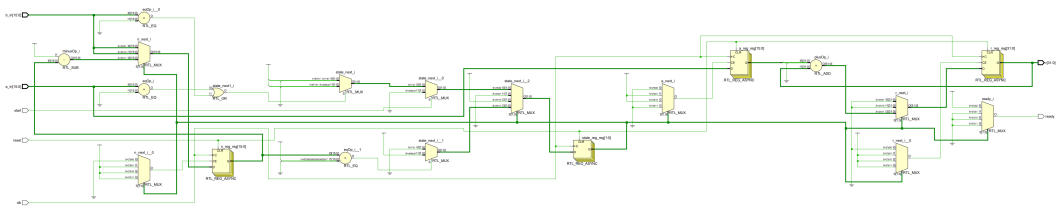

Figure 14. The RTL diagram for the repetitive-addition design.

The block diagram and synthesized schematic of this scheme are the simplest-looking ones. In fact, it is indeed the scheme with the smallest resource utilization.

Simulation

An example for the result of the behavioral simulation is shown in Figure 16 and the one for post-synthesized simulation is shown in Figure 17.

The result is correct, and similarly, it takes a large number of clock cycles for **ready** to be asserted once, and the delay will also change according to the input value.

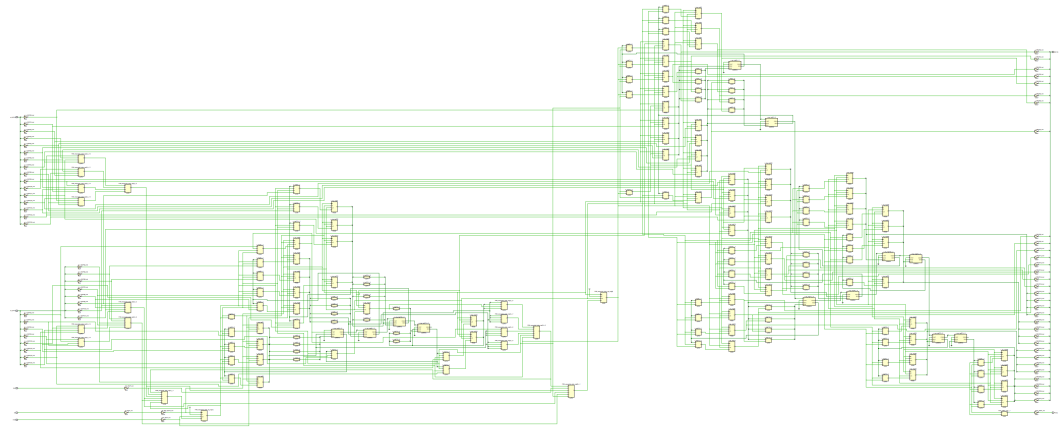

Figure 15. The synthesized schematic for the repetitive-addition design.

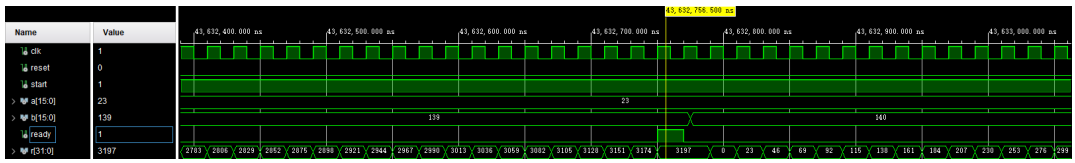

Figure 16. The result of the behavioral simulation for the repetitive-addition design.

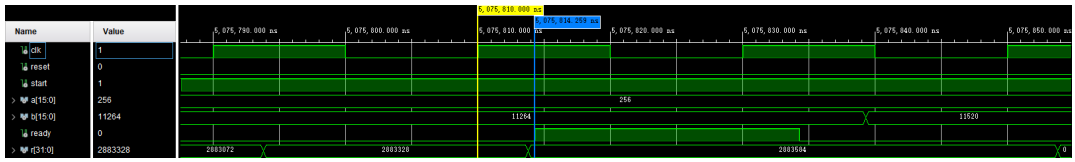

Figure 17. The result of the post-synthesized simulation for the repetitive-addition design.

Specifications

The device utilization is shown in Figure 18

Utilization — Post-Synthesis | Post-Implementation

Graph | Table

Resource	Estimation	Available	Utilization %
LUT	74	63400	0.12
FF	66	126800	0.05
IO	68	210	32.38
BUFG	1	32	3.13

Figure 18. The device utilization for the repetitive-addition design.

It can be seen that this approach has the smallest device utilization rate.

The delay depends on the input, and can be simpled represented as $b \times T$ where T is the period of a clock period.

2.4 Pipelined Design

Principle

Pipeline design refers to breaking down a large combinational logic circuit into multiple stages with similar volumes of tasks, inserting registers between them to enable pipelined execution of tasks (Figure 19), performing multiple operations simultaneously, and generating one result per

clock cycle after stabilization.

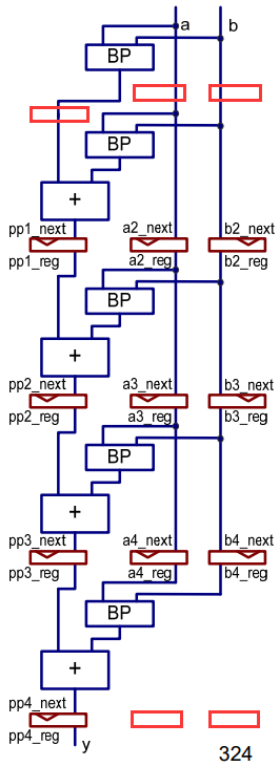

Figure 19. The diagram for the pipelined design. The red rectangles represents the registers we add to the reference design in the slides, aiming to simplify the code.

Implementation

The code is shown in the following section. And in this section the component refers to a 16-bit instance of the generic n-bit pipelined multiplier in the next section.

RTL Diagram and Synthesized Schematic

The RTL diagram is shown in Figure 20 and the synthesized schematic is shown in 21.

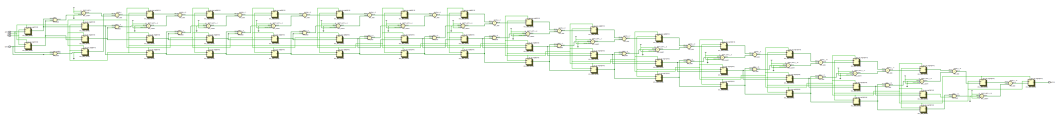

Figure 20. The RTL diagram for the pipelined design.

Simulation

An example for the result of the behavioral simulation is shown in Figure 22 and the one for post-synthesized simulation is shown in Figure 23.

The simulation results speak for themselves, as we can see the correct result appears after sixteen cycles.

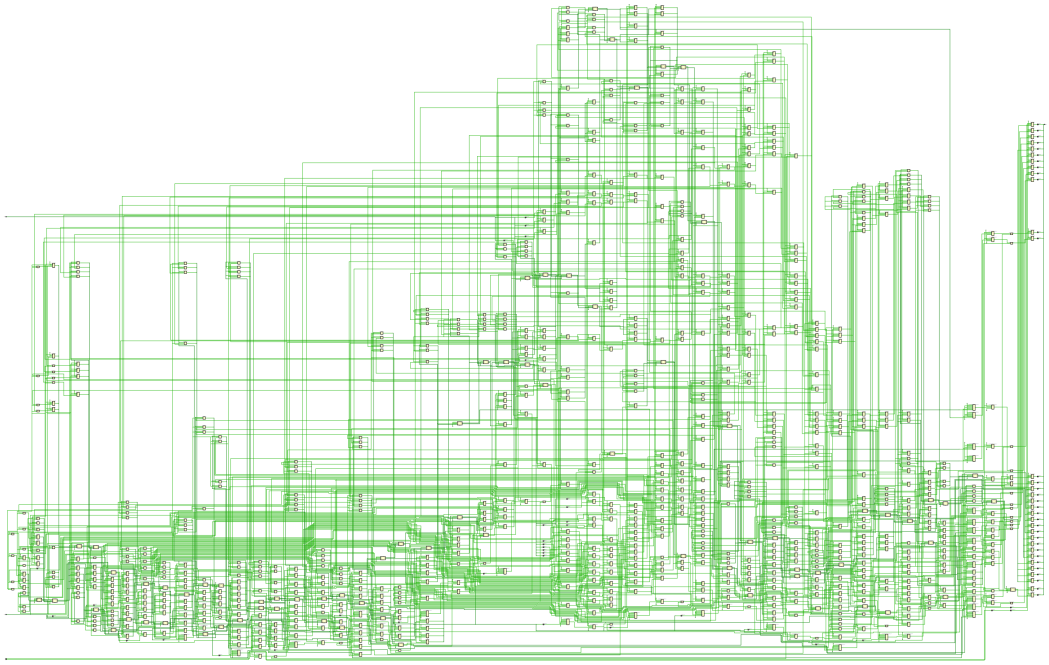

Figure 21. The synthesized schematic for the pipelined design.

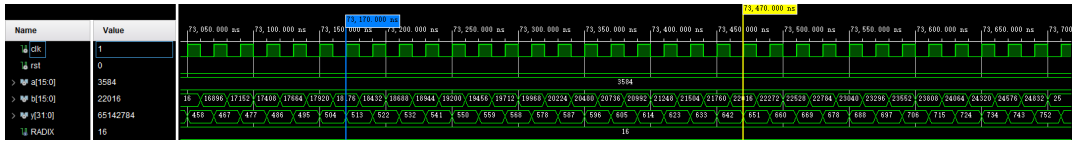

Figure 22. The result of the behavioral simulation for the pipelined design.

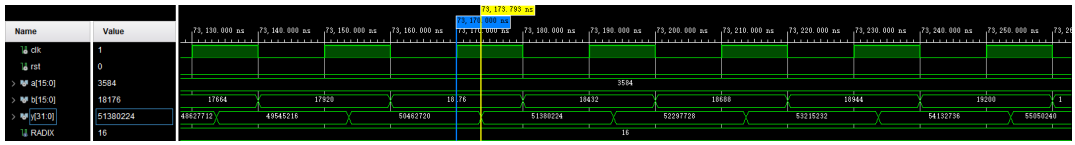

Figure 23. The result of the post-synthesized simulation for the pipelined design.

Specifications

The device utilization is shown in Figure 24

Utilization — Post-Synthesis | Post-Implementation

Graph | Table

Resource	Estimation	Available	Utilization %
LUT	284	63400	0.45
LUTRAM	24	19000	0.13
FF	585	126800	0.46
IO	66	210	31.43
BUFG	1	32	3.13

Figure 24. The device utilization for the pipelined design.

The delay is about 16 clock periods for the 16-bit pipelined multiplier.
And the throughput is about $\frac{1}{T_c}\frac{k}{15+k}$ where k is the number of the data to process.

3 Generic N-bit Pipelined Multiplier

3.1 Implementation

The code is shown in Code 4.

```
...

entity pipelined_multiplier_generic is
    generic (
        radix: natural := 16
    );
    port (
        clk, rst: in std_logic;
        a, b: in std_logic_vector(radix - 1 downto 0);
        y: out std_logic_vector(radix * 2 - 1 downto 0)
    );
end entity;

architecture Behavioral of pipelined_multiplier_generic is
    type ppn_t is array(0 to radix - 1) of std_logic_vector(2 * radix - 1
    downto 0);
    type abn_t is array(1 to radix) of std_logic_vector(radix - 1 downto 0);
    type bvn_t is array(0 to radix - 1) of std_logic_vector(radix - 1 downto
     0);

    signal ppn_reg, ppn_next: ppn_t;
    signal an_reg, an_next, bn_reg, bn_next: abn_t;
    signal bvn: bvn_t;
begin
    process (clk, rst) is
    begin
        if rst = '1' then
            ppn_reg <= (others => (others => '0'));
            an_reg <= (others => (others => '0'));
            bn_reg <= (others => (others => '0'));
        elsif rising_edge(clk) then
            ppn_reg <= ppn_next;
            an_reg <= an_next;
            bn_reg <= bn_next;
        end if;
    end process;

    -- Stage 0
    bvn(0) <= (others => b(0));
    ppn_next(0) <= std_logic_vector(resize(unsigned(a and bvn(0)), radix *
    2));
    an_next(1) <= a;
    bn_next(1) <= b;

    -- Stage 1 ~ radix - 1
    stages: for i in 1 to radix - 1 generate
        bvn(i) <= (others => bn_reg(i)(i));
        ppn_next(i) <= ppn_reg(i - 1) + std_logic_vector(shift_left(resize(
    unsigned(an_reg(i) and bvn(i)), radix * 2), i));
        an_next(i + 1) <= an_reg(i);
        bn_next(i + 1) <= bn_reg(i);
    end generate;

    -- Output
    y <= std_logic_vector(ppn_reg(radix - 1));
end architecture;
```

Listing 4. Code for generic n-bit pipelined multiplier.

3.2 Specifications

The device utilization depends on the number of stages, i.e., the width of the operands.

The delay for this n-bit pipelined multiplier is $n \times T$ where T is the clock period.

Specially, for the pipelined design we focus on another specification called throughput, that is, the average number of operations completed per second. In this general case it can be write as

$$throughput = \frac{1}{T_c} \times \frac{k}{n+k-1} \rightarrow \frac{1}{T_c}(\text{when k is large enough}) \tag{1}$$

4 Conclusion

In summary, this experiment simulated, synthesized, and compared four architectures of multipliers in terms of delay, resource utilization, and throughput.

In terms of results, the multiplier implemented with combinatorial logic has a high delay in a single clock cycle. Between the two combinatorial implementations, the former has lower time delay but higher resource utilization, while the latter has larger time delay but smaller resource utilization.

The multiplier implemented with the FSMD method has a high and uncertain delay but is suitable for situations with a short clock cycle and has very little resource utilization.

The multiplier designed with a pipeline architecture has a delay of a determined length and throughput depending on the clock cycle, making it suitable for processing a large amount of data continuously. For example, it can be used in a digital mixer to process a continuous stream of input data.