Implementation of a Fibonacci Calculator with User Interface on Board

Wang Zhuoyang[1]

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

Abstract

Lazy. No abstract.

Keywords: FSMD; Fibonacci; VHDL

Contents

Digital System Designing Laboratory Reports (2023)

Corresponding author
Wang Zhuoyang

1 Introduction

In this lab a FSMD (Finite State Machine with Datapath) will be implemented to calculate the n-th term of the Fibonacci series (Formula 1).

$$fib(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ fib(n-1) + fib(n-2), & n > 1 \end{cases} \tag{1}$$

The basic requirement is to process the inputs up to 6 bits, i.e., the 63rd term of the series. Note that $fib(63)$ is 6557470319842, which exceeds the range of $integer$.

2 Designing the Fibonacci Calculator

2.1 Step 1: Defining the input and output signals

In step one, the input and output ports are supposed to be defined (Code 1).

```
entity fibonacci is
    port (
        clk, rst, en: in std_logic;
        start: in std_logic;
        n_in: in int;
        ready: out std_logic;
        f_out: out int
    );
end entity;
```

Listing 1. The declaration of the module fibonacci.

clk is the input signal for the system clock, which will be connected to pin $E3$, the 100 MHz clock provider. rst is the asynchronous reset signal (active high). en is the input enable signal generally generated by the frequency divider (pulse generator).

$start$ is the synchronous start signal, driving the FSMD to work when asserted at a enabled clock rising edge.

n_in is the input signal, representing the index of the term to calculate. The type int is defined as *unsigned(INT_RADIX - 1 downto 0)*, where $unsigned$ is from the package $IEEE.std\_logic\_arith$, i.e., is *array (natural range <>) of std_logic*.

$ready$ is the output signal, indicating if the FSMD is idle.

f_out is the output signal for the result.

2.2 Step 2: Converting the algorithm to an ASM chart

The second step requires abstracting the algorithm into an ASM (Algorithm State Machine) chart.

Since writing this report has taken a lot of time, only the ASM chart optimized using fast matrix exponentiation will be included (Figure 1). The method for cumulative calculation is relatively simple and straightforward, so it is omitted here.

The basic idea of this algorithm is to represent the Fibonacci sequence as a matrix and use matrix exponentiation to compute the n-th term of the sequence. From the definition of fibonacci series, simply we have

$$f_n \triangleq \begin{bmatrix} fib(n-1) \\ fib(n) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} fib(n-2) \\ fib(n-1) \end{bmatrix} \triangleq A f_{n-1} \tag{2}$$

Let $f_1 = \begin{bmatrix} 0 & 1 \end{bmatrix}^T$. So when we need to calculate the n-th term of the series, we can calculate

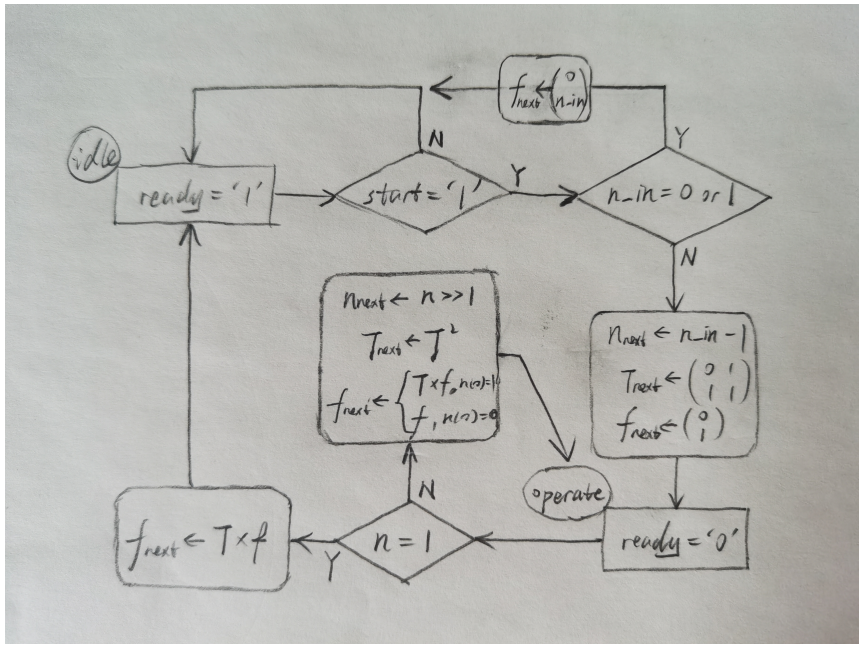

Figure 1. The ASM chart for the fibonacci calculation using matrix exponentiation.

$$f_n = A^{n-1} f_1 \tag{3}$$

By using matrix exponentiation, we can efficiently compute the matrix A raised to the power of $n-1$ and obtain the n-th term of the Fibonacci sequence. This approach is faster than the traditional recursive or iterative approaches for computing the Fibonacci sequence, as it has a time complexity of $O(log\, n)$ instead of $O(n)$.

2.3 Step 3: Constructing the FSMD

In step three we also have four steps. First, we list all possible paths for state transitions, which is useful to write the vhdl code:

1. IDLE $\rightarrow$ IDLE, when $start \neq 1$.
2. IDLE $\rightarrow$ IDLE, when $start = 1$ and $n\_in = 0/1$.
3. IDLE $\rightarrow$ OPERATE, when $start = 1$ and $n\_in \neq 0/1$.
4. OPERATE $\rightarrow$ OPERATE, when $n \neq 1$ and $n(0) = 1$.
5. OPERATE $\rightarrow$ OPERATE, when $n \neq 1$ and $n(0) = 0$.
6. OPERATE $\rightarrow$ IDLE, when $n = 1$.

From it we can list out all possible RT operations for the three registers n, T and f in the ASM chart, which is useful to draw the conceptual diagram:

1. With n:
   - $n \leftarrow n$ (Path 1, 2, 6).
   - $n \leftarrow n\_in - 1$ (Path 3).
   - $n \leftarrow n >> 1$ (Path 4, 5).
2. With T:
   - $T \leftarrow T$ (Path 1, 2, 6).
   - $T \leftarrow A$ (Path 3).
   - $T \leftarrow T^2$ (Path 4, 5).
3. With f:

- $f \leftarrow f$ (Path 1, 5).
- $f \leftarrow [0 \quad n\_in]^T$ (Path 2).
- $f \leftarrow [0 \quad 1]^T$ (Path 3).
- $f \leftarrow Tf$ (Path 4, 6).

According to the list we can obtain the circuit for each group RT operations, and finally we can draw the entire conceptual diagram (Figure 2).

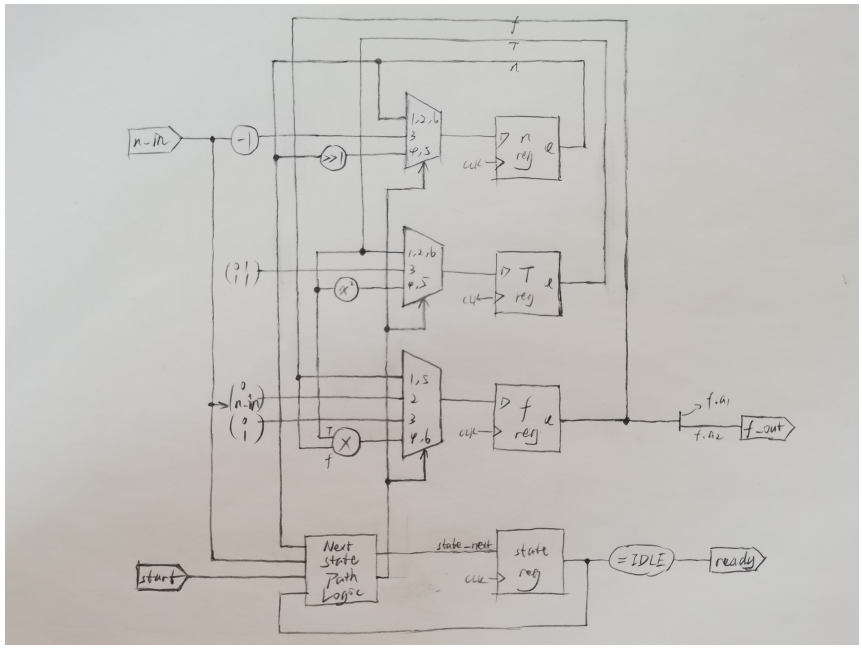

Figure 2. The conceptual diagram for the fibonacci calculation module.

2.4 Step 4: VHDL descriptions of FSMD

With the above steps, we can now write the VHDL code for the state machine (2). However, in fact, it's not necessary to draw the conceptual diagram since it doesn't directly help with programming. We only need to draw the ASM chart - but it seems to be a requirement for the report.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

use work.types.all;


entity fibonacci is
    port (
        clk, rst, en: in std_logic;
        start: in std_logic;
        n_in: in int;
        ready: out std_logic;
        f_out: out int
    );
end entity;


architecture Behavioral of fibonacci is
    type state_t is (IDLE, OPERATE);

```

```
    signal state, state_next: state_t;
    signal n, n_next: int;
    signal T, T_next: mat2_t;
    signal f, f_next: vec2_t;
begin
    process (clk, rst, en) is
    begin
        if rst = '1' then
            state <= IDLE;
            n <= to_int(0);
            T <= (others => to_int(0));
            f <= (others => to_int(0));
        elsif rising_edge(clk) and en = '1' then
            state <= state_next;
            n <= n_next;
            T <= T_next;
            f <= f_next;
        end if;
    end process;

    process (start, n_in, state, n, T, f) is
    begin
        state_next <= state;
        n_next <= n;
        T_next <= T;
        f_next <= f;

        ready <= '0';
        f_out <= f.a2;

        case state is
            when IDLE =>
                if start = '1' then
                    if n_in = 0 or n_in = 1 then
                        state_next <= IDLE;
                        f_next <= (to_int(0), n_in);
                    else
                        state_next <= OPERATE;
                        n_next <= n_in - 1;
                        T_next <= (to_int(0), to_int(1), to_int(1), to_int
    (1));
                        f_next <= (to_int(0), to_int(1));
                    end if;
                else
                    state_next <= IDLE;
                end if;

                ready <= '1';
            when OPERATE =>
                if n = 1 then
                    state_next <= IDLE;
                    f_next <= T * f;
                else
                    state_next <= OPERATE;
                    n_next <= shift_r(n, to_int(1));
                    T_next <= T * T;
                    if n(0) = '1' then
                        f_next <= T * f;
                    else
                        f_next <= f;
                    end if;
                end if;
```

```
                ready <= '0';
            end case;
        end process;
end architecture;
```

Listing 2. The VHDL code for the fibonacci calculating state machine.

Part of the definitions in the package $types$ are shown below (Code 3).

```
...
    constant INT_RADIX: natural := 70;
    subtype int is unsigned(INT_RADIX - 1 downto 0);
    function to_int(x: integer) return int;
    ...

    type mat2_t is record
        a11, a12: int;
        a21, a22: int;
    end record;
    function "*"(A: mat2_t; B: mat2_t) return mat2_t;

    type vec2_t is record
        a1: int;
        a2: int;
    end record;
    function "*"(A: mat2_t; B: vec2_t) return vec2_t;
...
```

Listing 3. Part of the package types.

3 Optimizations

3.1 Time Complexity Optimization

As mentioned earlier, the original idea was to perform $n-1$ recursive computations to obtain the n-th term of the sequence. With the addition of an $IDLE$ state to the state machine, it would take approximately n clock cycles to calculate the result.

The first optimization that comes to mind is to optimize the time complexity of the algorithm by using the matrix fast power algorithm. This can reduce the required number of clock cycles to the level of $log(n)$. This method is what we employed in the previous sections.

Of course, this method also has a drawback, which is the consumption of more hardware resources to support more complex operations such as matrix multiplication instead of simple addition.

On this basis, further optimization can be achieved. By still using the same idea of fast exponentiation, after mathematical derivation, it can be found that there is a recursive relationship between the sum of squares of two adjacent Fibonacci sequence terms and the square of the term at twice the index position. This can reduce the number of large integer computations and save hardware resources, but the details will not be discussed here.

3.2 Chip Resource Utilization Optimization

Another optimization direction is to reduce the utilization of hardware resources. There is no perfect solution to optimize resource usage, and it often comes at the expense of time. One method to reduce hardware resource usage is to elevate the conditional output box to a state, by adding intermediate states to decrease the complexity of the state transition logic. Sometimes this approach can also allow certain operation circuits to be shared among different scenarios - however, this is not significant in this case and will not be further elaborated.

4 Comparison and Analysis

4.1 Comparisons Between the Plain Method and the Matrix Fast Powering Method

We will summarize the analysis of the two optimization directions proposed in the previous section:

1. Time Complexity Optimization Approach:
   - Lower tick consumption.
   - Longer combinational logic delay (which limits the maximum clock frequency, and brings timing issues).
   - Higher bit-width demanding.
2. Chip Resource Utilization Optimization Approach:
   - Higher tick consumption.
   - Lower combinational logic delay.

4.2 Simulation

Considering that I am quite lazy, I did not write code for the naive method, and only performed post-implementation timing simulation for the matrix fast power method. The results are shown in Figure 3.

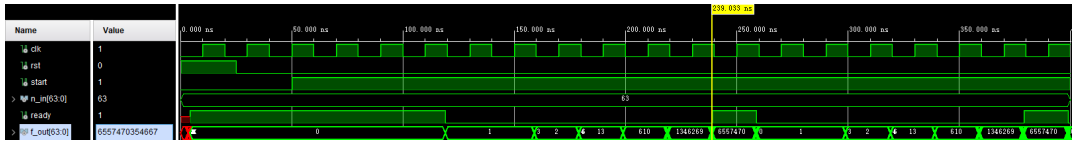

Figure 3. The result of the post-implementation timing simulation of the matrix exponentiation method. (The position of the cursor on the graph is misplaced, as it is placed in the part where the output result is not yet stable. In fact, the actual simulation result is the correct value of 6557470319842.)

5 Program and Run

5.1 Function Descriptions and I/O Ports

Finally, we implemented the above algorithm on the development board and designed a user interface to check the correctness of the results.

We designed the following operation mode. The reset terminal rst is connected to the rightmost switch, and after resetting, the system enters the input mode.

In the input mode, the seven-segment display shows the decimal digits of the number of the item to be solved, and the decimal point on the display represents the cursor. Pressing the left and right buttons can move the cursor, and pressing the up and down buttons can change the selected digit. Due to the consideration of numerical size, the input only accepts two-digit numbers.

After confirming the input item number, press the middle confirm button to enter the calculation mode. In the calculation mode, the LED in the lower right corner lights up until the calculation is completed, and the system automatically enters the output mode after completion.

In the output mode, the seven-segment display shows eight decimal digits of the result. Pressing the left and right buttons can move the digits to view higher digits, and up to twenty digits can be displayed. After viewing is completed, press the confirm button to return to the input mode and modify the input value.

5.2 Module Implementations

Button Debouncing

In practical implementation, button actions may have mechanical jitter, causing problems such as key bouncing. Therefore, a specialized button debounce module needs to be developed.

The main principle of the button debounce module is to insert a piece of logic between input and output. Only when the input has been stable for a fixed time, the output will toggle. This is one of the debounce ideas, and others are not elaborated here.

The VHDL code is as follows (Code 4).

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;


entity button_state is
    generic (
        PERIOD: integer := 1000 -- 10 us
    );
    port (
        clk, rst: in std_logic;
        btn_in: in std_logic;
        btn_out: out std_logic
    );
end entity;


architecture Behavioral of button_state is
    signal cnt, cnt_next: integer;
    signal btn, btn_next: std_logic;
begin
    process (clk, rst) is
    begin
        if rst = '1' then
            cnt <= 0;
            btn <= '0';
        elsif rising_edge(clk) then
            cnt <= cnt_next;
            btn <= btn_next;
        end if;
    end process;
    cnt_next <= cnt + 1 when btn /= btn_in and cnt < PERIOD - 1 else 0;
    btn_next <= not btn when cnt = PERIOD - 1 else btn;

    btn_out <= btn;
end architecture;
```

Listing 4. The VHDL code for the button debouncing module.

Its RTL diagram is as follows (Figure 4).

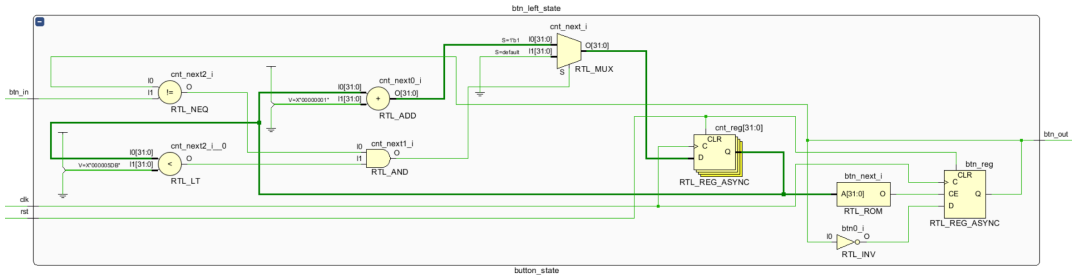

Figure 4. The RTL diagram for the module $button\_state$.

The behavioral simulation results are as follows (Figure 6).

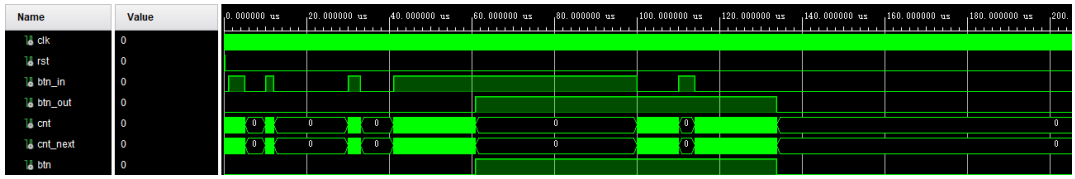

Figure 5. The behavioral simulation results for the module $button\_state$.

Frequency Divider

The frequency divider here is used to obtain a pulse signal with a frequency lower than the system clock frequency, which is used to drive the UI state machine, Fibonacci algorithm state machine, and seven-segment display scanning, etc.

The specific implementation has been explained in the previous report and will not be repeated here.

Seven Segment Display Driver

The seven-segment display driver has also been implemented in the previous report, and its RTL diagram is shown in Figure 6, which will not be elaborated on further.

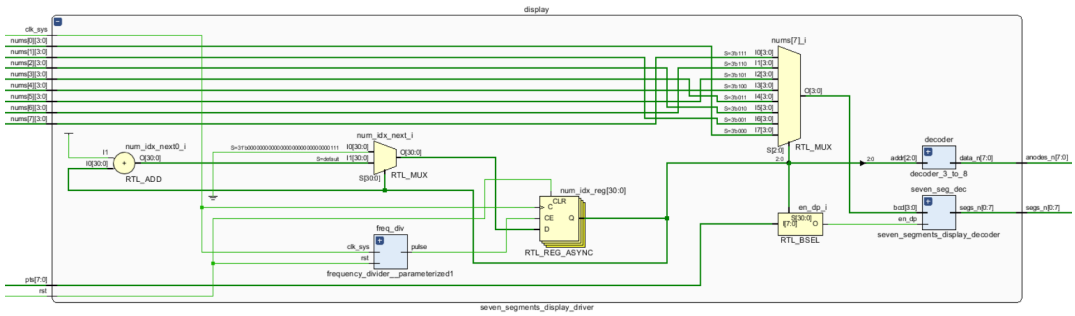

Figure 6. The RTL diagram for the module $seven\_segments\_display\_driver$.

Fibonacci Calculation

The Fibonacci sequence calculation module is described in the previous sections.

Top Module State Machine

The design of the top-level state machine is not elaborated on here, please refer to the attached code.

BCD-std_logic_vector Convertors

The digit representation converter is used to convert the input decimal number into binary for the calculation module to calculate, and to convert the binary calculation result into decimal output.

There is no good optimization method found, and the implementation method is relatively brute-force and resource-intensive, so it is omitted here.

The Overall Design and Resource Utilization

The RTL diagram with comments for the overall design is shown in Figure 7.

And the resources utilization is as below (Figure **??**).

As can be seen, the DSP usage is quite high because there are many high-precision multiplication operations involved.

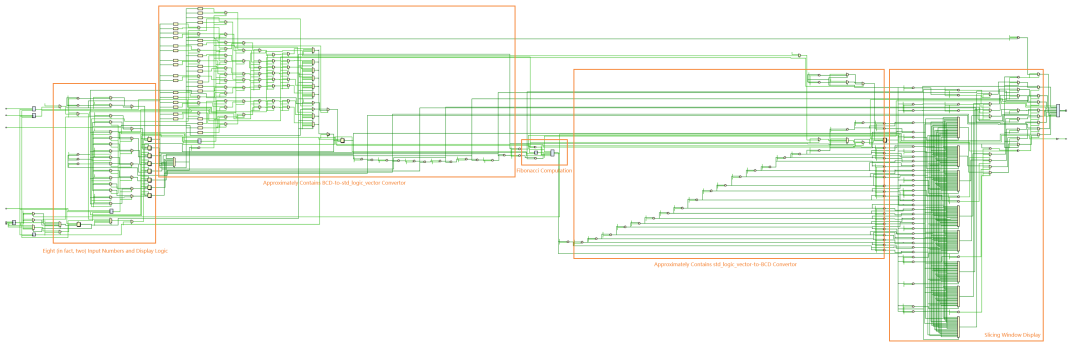

Figure 7. The RTL diagram for the overall design.

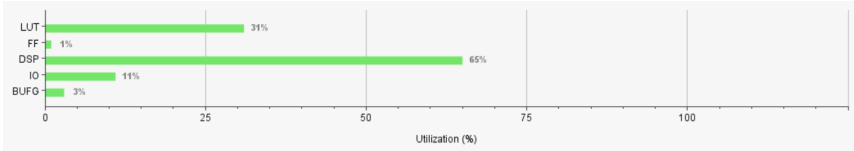

Figure 8. The resources utilization of the overall design.

5.3 Demonstration

Reseting

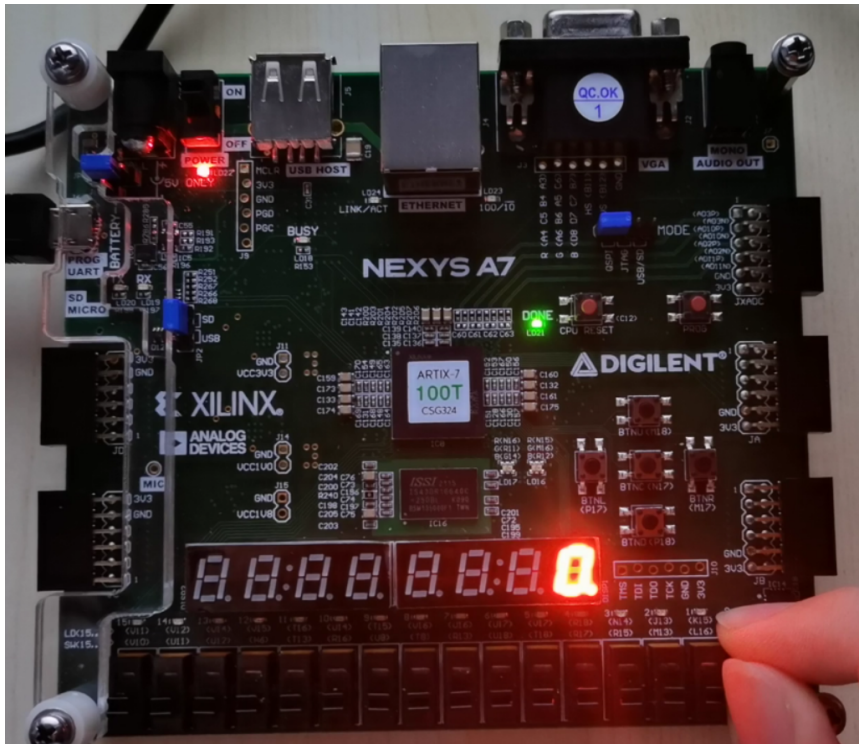

Figure 9. rst asserted.

Input Mode

Switching cursor and modifying numbers in input mode (Figure 10).

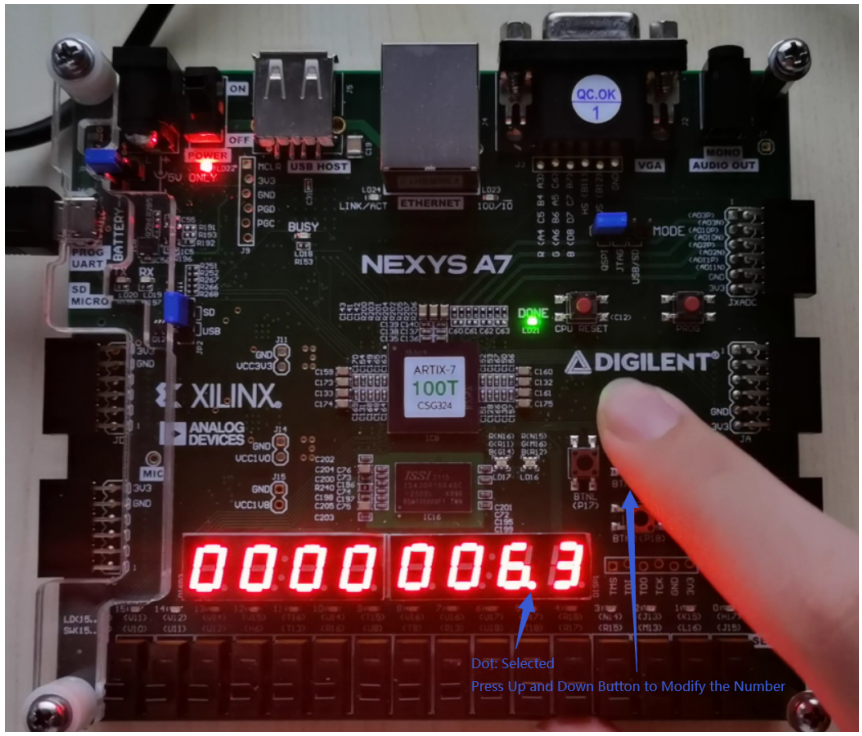

Figure 10. Input mode.

Compute Mode

As can be seen, the LED in the lower right corner lights up for a moment and then enters the output mode (Figure 11).

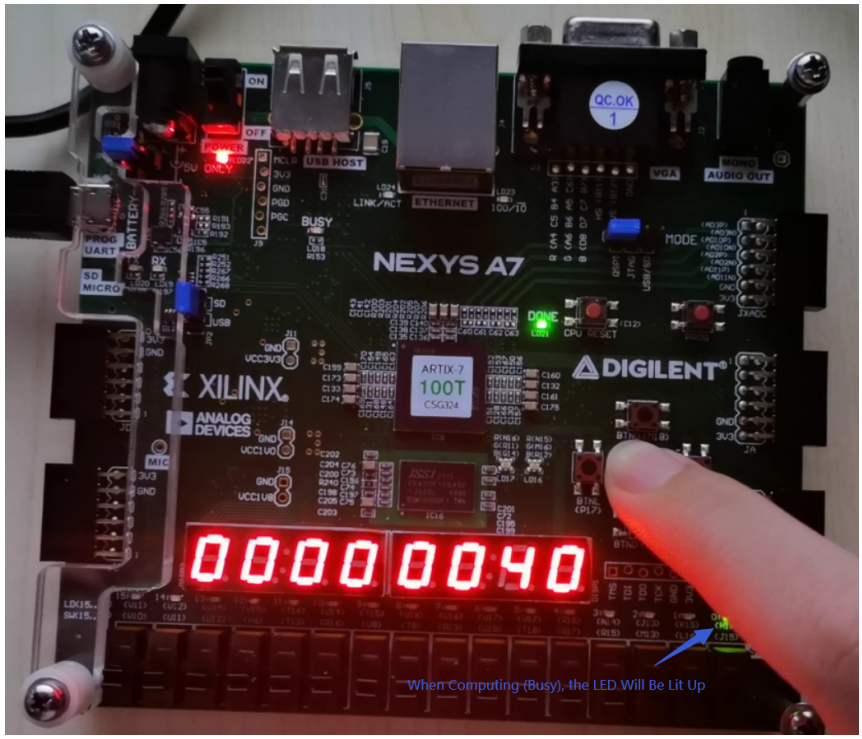

Figure 11. Compute mode.

Output Mode

In output mode, use the left and right keys to view the result, and it is found that the correct result 6557470319842 is given (Figure 13, 12).

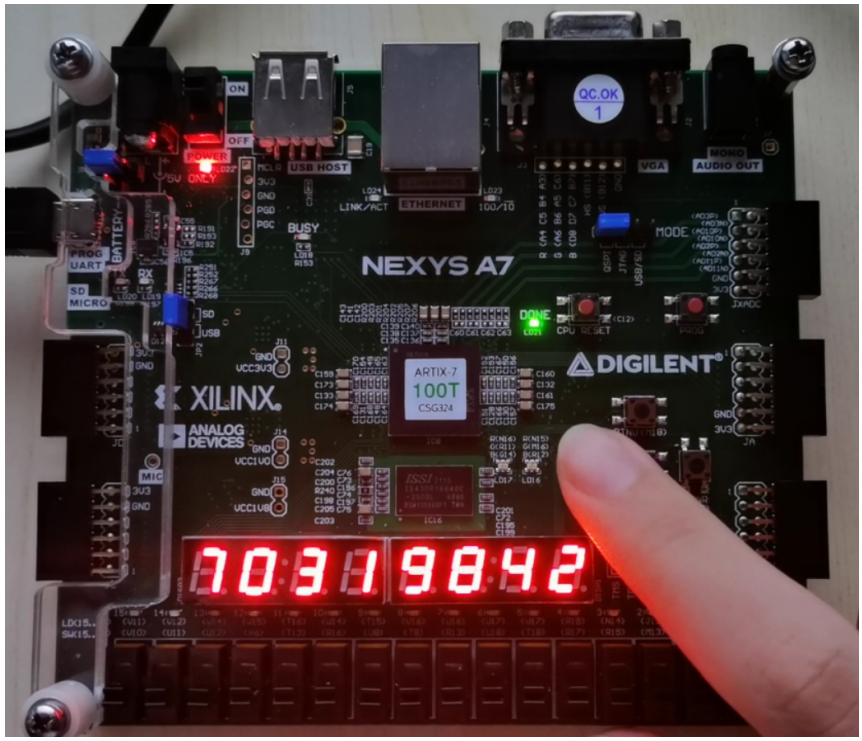

Figure 12. Output mode (low bits).

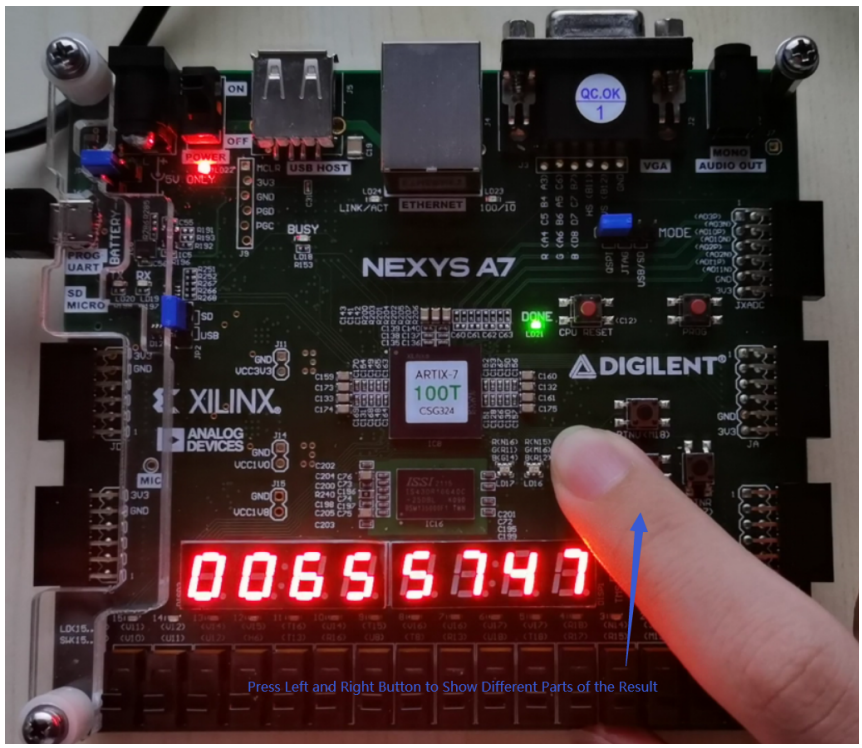

Figure 13. Compute mode (high bits).

In fact, due to the limitation of the bit width, the correct result is only up to the 78th term. However, the solvable range can be expanded by increasing the bit width and other methods. Since synthesis is time-consuming, this was not attempted here.

6 Conclusion

In summary, this experiment achieved the implementation of the algorithm state machine for solving the Fibonacci sequence. During the design process, there were several thoughts:

1. The design of the state machine requires an accurate state diagram as the foundation, and errors in the state diagram can easily lead to problems in the code writing process;
2. There seems to be no example with a conditional output box in the class;
3. Is the drawing of the conceptual diagram lacking in necessity? Having the state diagram is enough to write the code;
4. Surprise! std_logic_arith has no division, and multiple conversions using $numeric\_std$ were needed to achieve division by 10 (the self-made implementation had precision issues, and the precision was high but the resources were insufficient, so I will check IEEE's implementation later);
5. I haven't thought of anything else for now, so that's all.