Computer Architecture Report for Lab 1

資科工所 312551006 張勝迪 2024/9/23

1. Introduction

This lab focuses on designing and implementing an iterative multiplier and divider in Verilog. The multiplier supports 32-bit unsigned and signed multiplication, and the divider supports both unsigned and signed division. This report will describe the design choices, testing methodology, and evaluation results for both modules.

2. Design

2-1 Implementation

In this section, I will present the design approach of the multiplier, including the composition of the datapath and control logic, as well as the communication between the two through various signals.

a. Control Logic

The control logic of the divider unit is based on a finite state machine (FSM) with three states: IDLE, COMPUTE, and DONE. I designed the FSM to manage the flow of the multiplication process and ensure proper sequencing of operations.

- **IDLE State:** The system starts in the IDLE state, where it waits for valid input. Once both mulreq_val and mulreq_rdy are high, indicating that the division request is ready and valid, the state transitions to COMPUTE. This ensures that the inputs are correctly loaded into the datapath.
- **COMPUTE State:** In this state, a 5-bit counter is initialized to 31, representing the maximum number of cycles required for a 32-bit division. Each clock cycle, the counter is decremented by 1, progressing the division operation bit by bit. When the counter reaches 0, the computation is complete, and the FSM transitions to the DONE state.
- **DONE State:** In this state, the division result is ready to be returned. The FSM waits for both mulresp_val and mulresp_rdy to be high, signaling that the result has been accepted by the user. Once acknowledged, the state transitions back to IDLE, ready for the next multiplication request.

This FSM design ensures that the division process is well-organized, with clear transitions between states. The use of a counter helps to regulate the number of cycles needed for the computation.

b. Datapath Design

The datapath is designed to efficiently perform the multiplication operation using an iterative approach. I implemented four registers (a_reg, b_reg, acc_reg, and sign_reg) and three multiplexers (a_sel, b_sel, and acc_sel) to manage the data flow.

- **a_reg & b_reg:** These registers store the dividend and divisor, respectively. In the IDLE state, a_reg and b_reg are loaded with the values from mulreq_msg_a and mulreq_msg_b. In the COMPUTE state, a_reg is shifted left by 1 bit each cycle, and b_reg is shifted right by 1 bit. The above data flow is controlled by the multiplexer a_sel and b_sel. This shifting process is essential for the division algorithm, as it progressively aligns the values for subtraction in each cycle.
- **acc_reg:** This register accumulates the partial results of the division. The control signal acc_sel determines whether the value of a_reg should be added to the current accumulator based on the least significant bit of b_reg. If b_reg[0] is 1, the partial product in a_reg is added to the accumulator; otherwise, the accumulator remains unchanged. The above data flow is controlled by the multiplexer acc_sel. This selective addition allows for efficient computation of the quotient.
- **sign_reg:** The sign_reg tracks the sign of the final result, ensuring that the correct sign is applied after the division is complete. This is necessary for handling both signed and unsigned division.

The principles of the divider and multiplier are fundamentally very similar, with the primary difference being the computational algorithms. Therefore, they will not be further elaborated here.

2-2 Design Decisions and Justifications

In this section, I will break down the design philosophy into several points. The first point pertains to the environment setup, such as the declaration of hardware modules. The second point will discuss the connections between specific wires, most of which were insights gained during the debugging process. The third point involves the separate design of two major modules—specifically, the state machine and the computation logic, which should be clearly

delineated.

a. Modulization

For this assignment, I adopted a modular approach to designing components like registers and multiplexers. This design choice brings several advantages. For example, instead of declaring reg to store input values directly, I opted for fully functional registers that can determine whether to update their values based on the "write" signal. Additionally, the inclusion of multiplexers proved very beneficial, allowing me to simplify the datapath without excessive if-else conditions. This way, the registers' input values can be determined purely by the multiplexers' output, making the design cleaner and more manageable.

b. Store the bit at first

In the mutiplier and divider, the timing of signal reception is critical. Take sign_a_bit as an example—it must store the value of divreq_msg_a[31] during the IDLE state to prevent incorrect results if a new request is transmitted prematurely. Therefore, in my design approach, I ensured that values like a_abs, a_abs_unsigned, and final_sign were declared and maintained with stability. This ensures that during computation and when producing the output, these values are unaffected by changes in signals, preserving the integrity of the result.

c. Separation of Modules (Deviations from the prescribed datapath)

In my design, I implemented a clear separation between the datapath and control logic by using distinct modules for each component. This modularization enhances clarity and allows for independent development and debugging of the two areas. The datapath is dedicated to managing data transformations, focusing on how values change through various stages of computation. Meanwhile, the control module concentrates on state transitions, determining how to adjust states based on incoming signals and the changes in the counter. By delineating these functionalities, I ensured that any modifications or troubleshooting can be performed with minimal impact on the overall design. This approach not only promotes code reusability but also simplifies testing, making the system more robust and easier to maintain.

3. Testing Methodology

To ensure the robustness and correctness of the Mul/Div Unit, a comprehensive testing methodology was employed. The test cases cover a wide range of input values, including edge cases such as zero input, maximum possible values, and negative numbers. Each test case includes 64-bit operations, ensuring the full range of bit-width is utilized. This allows for accurate validation of both multiplication and division functionalities. Corner cases, such as division by zero or multiplying by zero, are specifically included to guarantee that the design behaves as expected under these conditions. Additionally, the test bench is automated, and the results are verified using a test sink to check the expected outcomes against the actual results, confirming that the module performs correctly across various inputs.

Additionally, I also added several test cases in the three .t.v files, such as:

- t0.src.m[1] = 67'h0_fffffff8_fffffff8; t0.sink.m[1] = 64'h00000000_00000040;
- t0.src.m[7] = 67'h1_f5fe4fbc_ffffb14a; t0.sink.m[7] = 64'hffffcc8e_0000208b;
- t0.src.m[8] = 67'h2_f2dae217_07f2ca05; t0.sink.m[8] = 64'h04673581_0000001e;

4. Evaluation

Simulation results show that both the multiplier and divider perform as expected:

- 0xbadbeeef * 0x10000000 = 0xfbadbeeef0000000; Cycle Count = 33;
- 0xf5fe4fbc / 0x00004eb6 = 0xffffdf75; Cycle Count = 33;
- 0x08a22334 % 0xfdcba02b = 0x020503b5; Cycle Count = 33;
- 0xf5fe4fbc /u 0x00004eb6 = 0x00032012; Cycle Count = 33;
- 0x0a56adca %u 0xfabc1234 = 0x0a56adca; Cycle Count = 33;

5. Conclusion

In this lab, we successfully implemented and tested an iterative multiplier and divider. The multiplier handles both signed and unsigned 32-bit multiplication, and the divider supports both signed and unsigned division. Testing confirmed the correctness of the design, and simulation results matched the expected cycle counts. This lab provided practical insights into the design and verification of arithmetic units using hardware description languages.