Lab1 Report

111550076 楊子睜

September 23, 2024

1 Introduction

In this lab, I implemented a 32-bit integer multiply/divide unit using the val/rdy interface, designed with a finite state machine (FSM) that operates through three states: idle, calculate, and done. This FSM allows for iterative processing of multiplication and division, including support for unsigned division and remainder operations. Specific test cases were developed to validate the correctness of these functions. Additionally, I optimized the multiplier by evaluating two bits per iteration, reducing the cycle count and enhancing performance. As an extension, I also developed a 3-input multiply/divide unit, which added complexity and demonstrated advanced capabilities in the design.

2 Design

2.1 Multiply unit

The main logic is a three-state finite state machine (FSM), where state transitions control the "control signals" that are sent to the datapath module to maintain the iterative algorithm. The FSM operates with three states: idle, calculate, and done. Transitions between these states are based on external signals and the internal counter, determining the progression of the iteration. Below is the FSM code in Verilog:

```
always @(posedge clk) begin
  if (reset) S <= S_IDLE;
  else S <= S_next;
end

always @(*) begin
  S_next = S;
  case (S)
    S_IDLE:  if (mulreq_val) S_next <= S_CAL;
    S_CAL: if (counter == 0 && mulresp_rdy) S_next <= S_DONE;
    S_DONE: S_next <= S_IDLE;
  endcase
end
```

All control signals depend on the current state. When the state is idle, the request-ready signal is asserted (true), indicating that the unit is waiting for the request-valid signal. Once the valid signal is received, the FSM transitions to the calculation state.

In the calculation state, a counter is initialized to 31 and decrements by 1 with each clock cycle. When the counter reaches 0 and the response is ready, the FSM transitions to the done state. In the done state, it waits for one additional cycle to determine the sign of the result before finally returning to the idle state, ready to process the next request. The whole process takes 33 cycles.

When the state is idle, `request_ready` is set to true. In the calculate state, all related signals such as `a_mux_sel`, `b_mux_sel`, `result_mux_sel`, `add_mux_sel`, and `result_en` are activated according to the algorithm and datapath outlined in the homework specifications. Finally, when the state reaches done, `response_valid` is set to true, indicating that the output data is correct. Additionally, `sign_en` is true in both the idle and done states, as the sign needs to be stored at the beginning and determined at the end of the process.

```
reg [4:0] counter_reg;
  always @(posedge clk) begin
    if (reset) counter_reg <= 5'd31;
    else counter_reg<= (b_mux_sel) ? counter_reg - 1 : counter_reg;
  end

assign mulreq_rdy = (S == S_IDLE);
assign sign_en = (S == S_IDLE || S == S_DONE);
assign a_mux_sel = (S == S_CAL);
assign b_mux_sel = (S == S_CAL);
assign result_mux_sel = (S == S_CAL);
assign add_mux_sel = (S == S_CAL);
assign result_en = (S == S_CAL);
assign mulresp_val = (S == S_DONE);
```

The data path follows the algorithm in the homework file. The sign signal is determined when `sign_en` is true. During the process of handling the `a` and `b` data, when `a_mux_sel` and `b_mux_sel` are false (i.e., the state is not in the calculate state), `a_reg` and `b_reg` are assigned the input data. When the state enters the calculate state, `a_reg` shifts left on each clock cycle, and `b_reg` shifts right.

For `result_reg`, when the state is in the calculate state and `result_mux_sel` is true, on each cycle, if `b_reg[0]` equals 1, `a_reg` is added to the result. The process continues by shifting `b_reg` to the right and `a_reg` to the left. After 32 iterations, the correct 32-bit multiplication result is produced. Finally, the result is output with `response_valid` set to true. This basic algorithm is covered in the computer organization course. Below is the primary algorithm in Verilog:

```
 always @(posedge clk) begin
  if (reset) sign_reg <= 0;
  else sign_reg = (sign_en) ? mulreq_msg_a[31] ^ mulreq_msg_b[31] : sign_reg;
  end

 always @(posedge clk) begin
    if (reset) a_reg <= 64'b0;
    else a_reg <= (a_mux_sel) ? a_reg << 1 : unsigned_a;
  end

  always @(posedge clk) begin
    if (reset) b_reg <= 64'b0;
    else b_reg <= (b_mux_sel) ? b_reg >> 1 : unsigned_b;
  end

  always @(posedge clk) begin
    if (reset) result_reg <= 64'b0;
    else result_reg = (result_mux_sel) ? (b_reg[0]) ? result_reg + a_reg : result_reg
      : 64'b0;
  end
```

2.2 Divide unit

The main logic is also a three-state finite state machine, and the transition logic is the same as the multiply unit, so it will not be elaborated further here.

The counter logic follows the same approach. The counter register is initialized to 31 and decrements by 1 with each clock cycle. When it reaches 0 and the response is ready, the state transitions to the done state. Thus, the entire process takes 33 cycles.

However, the datapath algorithm logic is more complex due to the nature of the algorithm. When the state is in the calculate state (i.e., when the counter is not zero), the process checks whether the `Diff` register is positive. If it is, `a_reg` is assigned to `Diff` shifted left and a `1'b1` is added, indicating that at this bit, a subtraction can occur, setting the quotient for that bit to 1. After 32 iterations,

the left 32-bit portion of `a_reg` becomes the remainder, while the right 32-bit portion becomes the quotient. This is achieved by repeatedly subtracting `b` and checking if the difference is positive, which is also a fundamental algorithm taught in computer organization (CO) class. However, the calculation requires some shift operations during the process.

The assignment specifically mentions handling unsigned values, but the process is actually quite simple. When dealing with the sign, if the `msg_fn` indicates unsigned, the sign can be ignored, and the calculation is done using the original values. Additionally, both `a_reg` and `b_reg` have an extra bit reserved for overflow handling, but for unsigned values, the extra bit is simply disregarded.

Below is the primary algorithm in Verilog:

```
wire [64:0] diff_nxt = {Diff[64:1], 1'b1};
 always @(posedge clk)begin
    if(reset) begin
      a_reg <= 65'b0;
      fn_reg <= 1'b0;
    end else begin
      if(a_mux_sel)begin
        if(!Diff[64])a_reg <= (!counter) ? diff_nxt : diff_nxt << 1 ;
        else a_reg <= (!counter) ? a_reg  : a_reg << 1;
      end
      else begin
        a_reg <= unsigned_a << 1'b1;
        fn_reg <= divreq_msg_fn;
      end
    end
  end

  always @(posedge clk) begin
    if (reset) Diff <= 65'b0;
    else begin
      if (a_mux_sel & counter != 0)
        Diff <= (!Diff[64]) ? ((diff_nxt << 1) ) - (b_reg)
                            : (a_reg << 1'b1) - (b_reg);
      else Diff  <= (unsigned_a << 1'b1) - (unsigned_b);
    end
  end
```

2.3 Design Concept

I simplified several signals marked in the assignment instructions during my datapath design. Since many signals were simply passed from the datapath to the control unit and then returned with a different name, I decided to handle them directly within the datapath to streamline the design. The control signals are managed using a finite state machine (FSM), with values assigned via assign statements. This approach makes it easy to control the operation logic and the number of cycles, offering a straightforward and clear design method to implement the algorithm.

3 Testing Methodology

In the testing methodology, I wrote some C++ code to implement the test cases. The code randomly generates two numbers and performs either a multiplication or division operation. It then converts the values and results into hexadecimal format and outputs the test case as a string, which is subsequently pasted into the unit test space. I also added several test cases for each unit, with a particular focus on testing unsigned division in the divide unit.

The divide test case generating code in c++:

```
void divide(uint32_t a, uint32_t b) {
    uint32_t quotient = a / b;
```

```
    uint32_t remainder = a % b;
    std::cout << std::setfill('0') << std::setw(8) << std::hex << remainder
              <<"_" << std::setfill('0') << std::setw(8) << quotient;
}
int main() {
    std::srand(static_cast<unsigned int>(std::time(0)));
    for (int i = 0; i < 10; ++i) {
        uint32_t num1 = static_cast<uint32_t>(std::rand());
        uint32_t num2 = static_cast<uint32_t>(std::rand());
        cout<<"t0.src.m[" << i << "] = 65'h0_" << std::hex << num1
        << "_" << num2 <<";t0.sink.m[" << i << "]=64'h";
        << num2 << std::endl;
        divide(num1, num2);
        cout<<";\n";
    }
}
```

4 Evaluation

each takes 33 cycles.

```
0xbadbeeef *  0x10000000 = 0xfbadbeeef0000000
0xf5fe4fbc /  0x00004eb6 = 0xffffdf75
0x08a22334 %  0xfdcba02b = 0x020503b5
0xf5fe4fbc /u 0x00004eb6 = 0x00032012
0x0a54adca %u 0xfabc1234 = 0x0a56adca
```

```
ldiv-iterative-sim +op=mul +a=badbeeef +b=10000000
VCD info: dumpfile dump.vcd opened for output.
0xbadbeeef * 0x10000000 = 0xfbadbeeef0000000
Cycle Count =         33
ldiv-iterative-sim +op=div +a=f5fe4fbc +b=00004eb6
VCD info: dumpfile dump.vcd opened for output.
0xf5fe4fbc / 0x00004eb6 = 0xffffdf75
Cycle Count =         33
ldiv-iterative-sim +op=rem +a=08a22334 +b=fdcba02b
VCD info: dumpfile dump.vcd opened for output.
0x08a22334 % 0xfdcba02b = 0x020503b5
Cycle Count =         33
ldiv-iterative-sim +op=divu +a=f5fe4fbc +b=00004eb6
VCD info: dumpfile dump.vcd opened for output.
0xf5fe4fbc /u 0x00004eb6 = 0x00032012
Cycle Count =         33
ldiv-iterative-sim +op=remu +a=0a56adca +b=fabc1234
VCD info: dumpfile dump.vcd opened for output.
0x0a56adca %u 0xfabc1234 = 0x0a56adca
Cycle Count =         33
```

Figure 1: Evaluation result

5 Extensions

5.1 variable-latency iterative multiplier

I simplified the design by reducing the counter to 16 and shifting `a_reg` and `b_reg` by 2 bits each cycle. In the `result_reg` calculation, if `b_reg[0]` is positive, we add `a_reg` directly. When `b_reg[1]` is positive, we add `a_reg` shifted left by one, which effectively multiplies it by 2. This approach yields the correct answer and reduces the cycle count to 17 cycles. The rest of the multiplier design remains unchanged. The main modifications in Verilog are as follows:

```
always @(posedge clk) begin
    if (reset) begin result_reg <= 64'b0;
    else begin
      if(result_mux_sel)begin
        if(b_reg[0])result_reg = result_reg + a_reg;
        if(b_reg[1])result_reg = result_reg + {a_reg[62:0], 1'b0};
        else result_reg = result_reg;
      end
      else result_reg = 64'b0;
    end
  end
```

5.2 3-input iterative muldiv unit

This problem is actually quite challenging; I spent two days completing it. Unfortunately, I was only able to implement a simplified version. When performing division, it can only process $a/b/c$ when $b \times c$ does not exceed 32 bits, as I only implemented a 32-bit division after calculating $b \times c$, so I generated some test case that ensure b * c will not exceed 32 bits.

Regardless of whether the function is multiplication or division, I first calculate $b \times c$ using my custom `IntMulIterative` unit. If the function is multiplication, I perform a 64-bit $\times$ 32-bit multiplication and output the result. The most difficult part of this problem is handling the `val/rdy` platform. The request signal of the first multiplier is connected to the ready signal of the three-input module. Thus, when the first multiplier is ready to respond, it starts calculating $b \times c$. Once this multiplication is complete and the response valid signal is true, the request valid signal for the second multiplier is set to true. The second multiplier then computes $a\times$ the intermediate result. When this second multiplication finishes and the response valid signal is positive, the output response valid signal is set to true.

If the function is division, the 32-bit result of $b \times c$ and a are fed into my `IntDivIterative` unit, which performs a 32-bit division and outputs the result. The result is then formatted as a 48-bit remainder and quotient before being output. The entire process takes around 66 cycles. This is somewhat similar to pipelined logic: once the first multiplier is ready, it calculates $b \times c$ first, passes it to the second stage, and continues calculating the next $b \times c$ until completion. I spent considerable time maintaining the values in the pipeline since some data may not synchronize properly during calculations.

6 Conclusion

In this lab, I successfully implemented a 32-bit iterative multiply/divide unit using the val/rdy interface and further extended the design with a variable-latency iterative multiplier, reducing the cycle count for multiplication. Through the development process, I encountered and solved several complex challenges, including the integration of a multi-stage pipeline and the handling of synchronization issues. Although some limitations remain in the division operation, particularly with multi-input division, the overall design is functional and provides a solid foundation for future improvements. The experience gained from this lab significantly deepened my understanding of hardware design and finite state machine logic, particularly in optimizing iterative algorithms for performance.