313551137 官劉翔

Introduction: In this lab, the teacher designed an iterative integer multiply/divide unit in Verilog. We set up our workspace with necessary directories and tools like iverilog and gtkwave. The main task was implementing a 32-bit multiplier and a 65-bit divider, both producing 64-bit results. We incorporated signed operations and a val/rdy interface for efficient data transfer. Testing was crucial; we used a provided framework to create comprehensive test cases, including corner cases for unsigned operations. In the end, we evaluated our final muldiv unit to ensure it met the 33-cycle operation requirement.

Design:

I make the architecture base on the Figure 4: Iterative Multiplier Datapath and my personal adjustment.

 # Iterative Multiplier Implementation Explanation

For the datapath

1. Unsigned Conversion:

I chose to convert signed inputs to unsigned at the beginning:

wire [31:0] unsigned_a = mulreq_msg_a[31] ? (~mulreq_msg_a + 1'b1) : mulreq_msg_a;

wire [31:0] unsigned_b = mulreq_msg_b[31] ? (~mulreq_msg_b + 1'b1) : mulreq_msg_b;

This decision simplifies the multiplication process by allowing us to work with unsigned numbers throughout the operation.

3. Sign Handling:

We introduced a `sign_reg` to store the final sign of the result:

sign_reg <= mulreq_msg_a[31] ^ mulreq_msg_b[31];

XOR operation efficiently determines the result's sign based on the inputs' signs.

4. Result Reset Logic:

Since I forget to reset the final_reg at this first time, after debugging, I add this line to make sure every new test will start on initial value.

```
119        result_reg <= ctrl_mux_sel_result ? result_reg : 64'b0;
120        if (ctrl_result_en)
121          result_reg <= result_mux_out;
```

5. Final Result Adjustment:

The final result is adjusted based on the stored sign:

wire [63:0] final_result = sign_reg ? (~result_reg+1) : result_reg;

For the control logic:

The control logic manages the state transitions and control signals for the datapath:

1. State Machine:

We implemented a three-state machine (IDLE, CALC, DONE) to manage the multiplication process:

localparam IDLE = 2'd0, CALC = 2'd1, DONE = 2'd2;

This simple state machine effectively controls the flow of the multiplication operation.

2. Counter:

A 6-bit counter is used to track the number of iterations: This allows us to perform exactly 32 iterations for a complete 32-bit multiplication.

3. Control Signals:

Various control signals (e.g., `ctrl_mux_sel_a`, `ctrl_add_mux_sel`) are managed to control the datapath operations in each state.

4. Val/Rdy signal:

I implemented val/rdy handshaking for both input and output interfaces to ensure proper synchronization with external modules.

 # Iterative divider Implementation Explanation

Many of the concepts of divider is similar to multiplier, I will skip it.

For the datapath:

1. I used two 65-bit registers (a_reg and b_reg) to store the dividend and divisor. The extra bit helps with overflow detection.
2. I implemented the unsign logic to handle both signed and unsigned operations. If the input is signed (divreq_msg_fn is true) and the number is negative, it converts it to its absolute value. Besides, rem_sign is only decided by divreq_msg_a.
3. The subtraction and shifting operations are performed as per the division algorithm we learned.
4. This time I try another way to build counter. It starts from 31 and counts down to 0.
5. For the final result, I made sure to only update it when divresp_val is high, which prevents incorrect outputs during calculation.

For the control logic:

1. I used a simple state machine as same as multiplier with three states: IDLE, CALC, and DONE.
2. The state transitions are based on the input signals and the counter value.
3. I made sure that the subtraction result is only used when it's non-negative (controlled by sub_mux_sel).
4. The sign of the result is determined at the end, considering both the division and remainder signs.

justifications for design decisions

1. Handling of Signed and Unsigned Operations: For the multiplier, I added a sign_reg to store the sign of the result. This allows for correct handling of signed multiplication without significantly altering the main multiplication algorithm. In the divider, I implemented logic to handle both signed and unsigned operations using the

divreq_msg_fn signal. This approach allows for a single hardware unit to handle multiple types of division operations.

2. Use of Extra Bits: In the multiplier, I used a 64-bit a_reg to allow for the full 64-bit result of multiplying two 32-bit numbers. In the divider, I used 65-bit registers for a_reg and b_reg instead of 64-bit. This extra bit helps in detecting overflow and ensures correct operation of the division algorithm.

5. Counter Implementation: In the multiplier, I used a 6-bit counter going from 0 to 31, which allows for 32 iterations. This ensures that we process all bits of the multiplier. For the divider, I want to practice another way, which start the counter from 31. Besides, I also use
6. Val/Rdy Interface: I implemented a val/rdy interface for both units to ensure proper handshaking between the units and the rest of the system. For my understanding,

Signal	IDLE	CALC	DONE
mulreq_rdy	1	0	0
mulresp_val	0	0	1
mulreq_val	1	X	X
mulresp_rdy	X	X	1

Deviations from Prescribed Datapath:

1. In the divider, I added div_sign and rem_sign signals to properly handle sign operations for both quotient and remainder. This wasn't explicitly shown in the original datapath but is necessary for correct operation.
2. In the multiplier, I added the sign handling logic, which wasn't present in the original datapath. This addition allows for correct signed multiplication.
3. The use of a 65-bit register in the divider and a 64-bit register in the multiplier for a_reg are slight deviations from the original 64-bit datapath, but these changes allow for more robust operation.

Testing Methodology:

1. Unit Testing: I used the provided test cases in the imuldiv-IntMulDivIterative.t.v file. These tests cover various scenarios for multiplication, division, and remainder operations, including both signed and unsigned cases. The test suite includes: Multiplication、Division and Remainder、Mixed operations

2. Manual Testing: I used the make command to compile the design, and then ran the iterative simulator with specific inputs. For example: ./imuldiv-iterative-sim +op=div +a=fe +b=9 This allowed me to verify that the division operation was working correctly and that it took exactly 33 cycles to complete, as required by the design specifications.
3. Corner Cases: In the "divu/remu" test case, I included some corner cases:
    - Division by 1 (basic case)
    - Maximum unsigned integer (0xFFFFFFFF) as dividend
    - Division by largest power of 2 (0x80000000)
    - Non-zero remainder case
    - Large number division
    - Maximum value divided by itself
4. Waveform Observation & Error Analysis: I used gtkwave to visualize the waveforms generated during simulation. When tests failed, I carefully analyzed the output, follow the data flow in Figure in lab.pdf and think about the different between the gtkwave wave . This often help me a lot of idea about where the implementation might be incorrect.

Evaluation:

I make sure each case have the correct answer and execute 33 cycels

```
wsl@LAPTOP-6BANPGCT:/mnt/g/我的雲端硬碟/交通大學/1131課程/計算機架構/LAB/Lab1/lab1/build$ ./imuldiv-iterative-sim +op=mul +a=badbeeef
 +b=10000000
VCD info: dumpfile dump.vcd opened for output.
0xbadbeeef * 0x10000000 = 0xfbadbeeef0000000
Cycle Count =         33
wsl@LAPTOP-6BANPGCT:/mnt/g/我的雲端硬碟/交通大學/1131課程/計算機架構/LAB/Lab1/lab1/build$ ./imuldiv-iterative-sim +op=div +a=f5fe4fbc
 +b=00004eb6
VCD info: dumpfile dump.vcd opened for output.
0xf5fe4fbc / 0x00004eb6 = 0xffffdf75
Cycle Count =         33
```

```
wsl@LAPTOP-6BANPGCT:/mnt/g/我的雲端硬碟/交通大學/1131課程/計算機架構/LAB/Lab1/lab1/build$ ./imuldiv-iterative-sim +op=rem +a=08a22334
 +b=fdcba02b
VCD info: dumpfile dump.vcd opened for output.
0x08a22334 % 0xfdcba02b = 0x020503b5
Cycle Count =         33
```

```
wsl@LAPTOP-6BANPGCT:/mnt/g/我的雲端硬碟/交通大學/1131課程/計算機架構/LAB/Lab1/lab1/build$ ./imuldiv-iterative-sim +op=divu +a=f5fe4fb
c +b=00004eb6
VCD info: dumpfile dump.vcd opened for output.
0xf5fe4fbc /u 0x00004eb6 = 0x00032012
Cycle Count =         33
wsl@LAPTOP-6BANPGCT:/mnt/g/我的雲端硬碟/交通大學/1131課程/計算機架構/LAB/Lab1/lab1/build$
```

```
wsl@LAPTOP-6BANPGCT:/mnt/g/我的雲端硬碟/交通大學/1131課程/計算機架構/LAB/Lab1/lab1/build$ ./imuldiv-iterative-sim +op=remu +a=0a56adc
a +b=fabc1234
VCD info: dumpfile dump.vcd opened for output.
0x0a56adca %u 0xfabc1234 = 0x0a56adca
Cycle Count =         33
```

Conclusion:

I am the beginner of the Verilog programming. Through this project, I gained practical skills in digital design principles, Verilog programming, and comprehensive testing methodologies (which make me spend a lot of time to explore XD). The implementation of the val/rdy interface and handling of both signed and unsigned operations. Successfully meeting the 33-cycle operation requirement demonstrated our iterative approach.