Computer Architecture
Lab 1: Integer Multiply/Divide Unit

Institute of Computer Science and Engineering
National Yang Ming Chiao Tung University

revision: 9-3-2024

The objective of this lab is to design an iterative integer multiply/divide unit that processes two 32-bit operands to produce a 64-bit result.

This lab will provide you with experience in:

- Verilog hardware description language
- Unit testing concepts
- Val/rdy interfaces
- Design principles, including modularity, hierarchy, and encapsulation

1 Reminder

- **Submission Deadline:** 9/23 (Mon) 11:59 p.m.
- Please refer to Lab 0 for the academic integrity requirements.
- **No sharing or distribution of lab materials is allowed.**

2 Setup

2.1 Getting Started

Once you have the Lab 1 materials, extract them by entering the following commands:

```
% tar -xf lab1.tar
% cd lab1
% export LAB1_ROOT=$PWD
```

Within the lab root directory, you will find three subdirectories, each serving a specific purpose:

- `build`: Contains the Makefile and compiled code.
- `imuldiv`: Houses all Verilog source code.
- `vc`: Contains additional Verilog components.

The `vc` directory is particularly valuable, as it contains numerous parameterized modules that will be instrumental in our labs. Some of the modules included in this directory encompass memories, flip-flops, and multiplexers. Our primary tools for this course will be `iverilog` for compiling and simulating Verilog designs and `gtkwave` for viewing simulation waveforms stored in value change dump (VCD) files.

To maintain consistency throughout the course, we will be utilizing the `bash` shell. Please ensure that you are running `bash`. If you are unsure, you can verify your shell by typing `ps -p $$` at your shell prompt.

2.2 Building the Project

Let's proceed with building the project by following these steps:

```
% cd $LAB1_ROOT/build
% make
```

Upon successful completion, you should observe: `imuldiv-singcyc-sim` and `imuldiv-iterative-sim`. These outputs are simple simulators that can take command line inputs to specify the operation and operands to directly interact with the muldiv unit. It also measures the number of clock cycles that elapse during computation. **It's crucial to note that the simulator exclusively accepts inputs in hexadecimal format.**

You can try running `imuldiv-singcyc-sim` to interact with the single cycle implementation of the muldiv unit. The syntax is as follows:

```
% # ./imuldiv-singcyc-sim +op=TYPE +a=OPERAND_A +b=OPERAND_B
% # where OPERAND is the hexadecimal representation of the input value
% ./imuldiv-singcyc-sim +op=mul +a=fe +b=9
```

You should see the operands and the result of the operation on the output, followed by the number of execution cycles, which should be 1 for all cases.

The other simulator, `imuldiv-iterative-sim`, combines the iterative multiplier and divider modules into a single muldiv unit. The code provided for you in the lab tarball uses single cycle reference versions of the multiplier and divider in `imuldiv-IntMulIterative.v` and `imuldiv-IntDivIterative.v`. **You will need to change these files to implement the iterative versions of the modules for the lab.** You can try running the iterative simulator using the same syntax as above. The cycle count for this simulation should also always be 1 since it implements a single cycle model.

Now, let's proceed by running the unit tests for all the source files:

```
% cd $LAB1_ROOT/build
% make check
```

If the tools are functioning correctly, you should see the source files being compiled, followed by a series of `"Entering Test Suite: module-name"` messages for each module under examination. **Ensure there are no errors before proceeding.**

Finally, let's take a look at the waveform contained in the `imuldiv-IntMulDivSingleCycle.vcd` file that was generated when you executed `"make check"`. Each source in the project will produce its own VCD dump file with the same name but with the .vcd suffix when you run `make check`.

2.3 Viewing Waveforms

To view the waveforms, follow these steps:

```
% cd $LAB1_ROOT/build
% gtkwave imuldiv-IntMulDivSingleCycle.vcd &
```

The trailing ampersand at the end of the command runs the program in the background, allowing you to continue using the terminal.

Once `gtkwave` is loaded, import the reference template by navigating to `File -> Read Save File` and selecting `gtk_template.sav`. This action will load a predefined set of signals specific to the single-cycle

implementation of the mul/div unit. To see relevant information, you may need to zoom out and scroll to where the reset signal transitions low.

Feel free to explore the interface and experiment with adding additional signals from the left-hand panel. The signals are organized under their respective modules, following the hierarchy defined in the source files. You can add signals by simply dragging and dropping them onto the waveform screen. Understanding the basics of navigating `gtkwave` is crucial, as waveforms are invaluable for debugging purposes.

3 Iterative MulDiv Unit

In this lab you will first design an iterative multiplier unit, move onto designing an iterative divider unit, and finally compose these two modules together into an iterative muldiv unit. Each design will take two 32-bit operands and compute a 64-bit result. Before commencing your work on the lab, ensure that you are famaliar with the concepts of the val/rdy interface, as elaborated in 4 of this document. Additionally, familiarize yourself with the Testing Methodology covered in Section 5.

3.1 Iterative Multiply Unit

The iterative multiply unit will support one instruction: the signed multiply, or `mul`. A flowchart of the iterative multiply algorithm that we will be using for this lab is shown in Figure 2 of the Appendix. Study the algorithm carefully and make sure you understand how it maps to the hardware datapath shown in Figure 4. The blue signals in Figure 4 represent control signals that need to be passed between the datapath and the control logic.

The recommended approach for any signed operation is to first unsign the operands if necessary. You can check if the operand is signed by checking if the most significant bit is 1. You can unsign an operand by reversing all the bits and adding 1 to the result. In other words:

```
unsigned_operand =  signed_operand + 1'b1;
```

The 'unsign' block shown in Figure 4 should essentially implement this technique. Of course, you should remember what the sign of the result should be in a special register and sign the result at the end if necessary.

Your task is to implement this datapath in RTL Verilog. Your solution should be separated into a datapath module and a control logic module. The single cycle reference version of the iterative multiplier has been provided in `imuldiv-IntMulIterative.v`. Please modify this code to implement your iterative multiplier.

3.2 Iterative Divide Unit

The iterative divide unit will support four instructions:

- div : signed division
- divu : unsigned division
- rem : signed remainder
- remu : unsigned remainder

Unlike with the multiplier, the divider supports both signed and unsigned instructions. Signed instructions treat the operands as signed values and unsigned instructions treat the operands as unsigned values. The consequence for the latter is that all operands and results can only be positive, but allows for twice the range of positive values. For example, the 4-bit value 4'b1111 will translate to a -1 if treated as a signed value and a +15 if treated as an unsigned value.

The division algorithm we will be using for this lab is described in Figure 3 of the Appendix. Examine the divider datapath in Figure 3 and make sure that you understand the mapping between the algorithm and the hardware.

This algorithm will produce a 64-bit output that has the remainder on the left-half and the quotient on the right-half. As such, we only need to differentiate between signed and unsigned instructions. The iterative divide unit should take an additional 1-bit input that represents the type of instruction.

As before, use the same approach as in the multiplier unit for signed operations: convert any signed operands to unsigned and compensate for the sign of the result at the end. Unsigned operations are a little more complicated. Unsigned operations should not unsign the operands even if the MSB is 1, in order to correctly treat it as a positive, unsigned value. Remember that the result should also bypass the sign block at the end. Next, notice that the datapath operates on 65-bit values instead of 64-bit values. In order to properly handle unsigned division, we need an extra bit to check for overflows. Consider a simple example using 4-bit operands, in which we want to perform unsigned division on the operands 4'b1111 and 4'b1111. According to the algorithm, we initialize the right-half of register A with operand A and the left-half of register B with operand B. At every iteration, we need to shift register A to the left by 1 and subtract register B from it. We can already see the predicament with the overflow in the first iteration:

```
    Reg A = 0000 1111
  - Reg B = 1111 0000
  --------------------
          1 0001 0000 < correct, the difference is negative
            0001 0000 < actual result due to overflow, looks positive
```

The algorithm dictates that if the difference of register A and register B is negative, we need to throw away the result and store the shifted version of register A. Only if the difference is positive, we store the difference with the LSB set to 1. We make the check by examining the MSB of the difference. However, if we only allow for 8-bits, the overflow bit is lost and the difference is interpreted as a positive value, when in fact it was negative, causing us to store the incorrect value. Fortunately, enlarging the datapath by 1 bit does not significantly affect the implementation of the other instructions. Just be sure to check bit 64 of the difference for both signed and unsigned operations and to concatenate the final output of the subtraction unit to 64 bits as shown in the datapath.

The iterative divider must account for the signs of the quotient and the remainder separately. The sign of the quotient is calculated normally. However, the sign of the remainder can be different depending on the application. For this lab, the sign of the remainder equals the sign of the dividend (i.e. operand A).

Your task is to implement the datapath in Figure 5. Your solution should be separated into a datapath module and a control logic module. A single cycle reference version of the iterative divider has been provided in `imuldiv-IntDivIterative.v`. **Please modify this code to implement your iterative divider.**

You will also need to add additional test cases for divu and remu.

3.3 Iterative MulDiv Unit

The final step of the lab is test the wrapper module that combines the multiplier and divider into a single muldiv unit. The code for the combined muldiv unit is in imuld-IntMulDivIterative.v. You will need to add additional test cases for divu and remu.

4 The val/rdy Interface

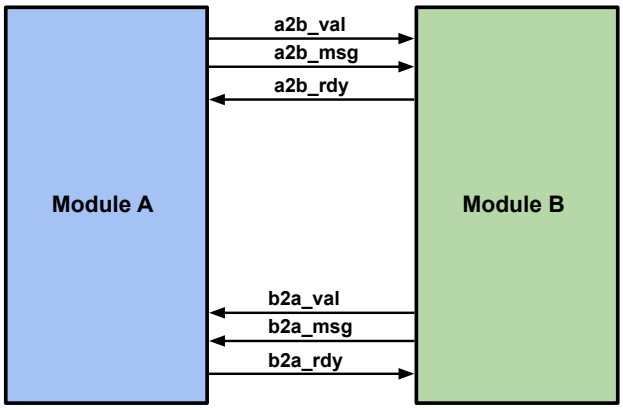

Figure 1: Example val/rdy Interface

Each `val/rdy` bundle is composed of the following:

- `msg`: The data being transferred.
- `val`: Indicates whether the data is valid.
- `rdy`: Indicates whether the module is ready to accept the data.

These signals collectively form what is known as a `val/rdy` interface. The `val/rdy` interface is latency-insensitive, meaning that data exchange between modules does not depend on precise timing. Instead, a contract governs the data transfer between modules using this interface. This exemplifies the `encapsulation` design principle. By using the `val/rdy` protocol, implementation details (e.g., `muldiv` unit latency) are hidden from the interface, allowing other modules to send messages to the `muldiv` unit without concerning themselves with the number of cycles required to complete an operation and return the result.

The `val` signal goes high when a module has valid data to transmit, and the `rdy` signal goes high when a module is ready to receive data. **It is crucial to emphasize that these signals are mutually independent to avoid creating combinational loops.** According to the contract, when both **val** and **rdy** signals are high, a data exchange occurs.

In this lab, the `muldiv` units use a request and response interface, both based on the `val/rdy` framework. The request `val` signal should be set high when new operands are available for the `muldiv` unit, and the request `rdy` signal should be high when the `muldiv` unit is ready to accept new operands. Similarly, on the response side, the response `val` signal should go high when the result is valid and ready, and the response `rdy` signal should be high when the result is ready to be accepted by the next module.

5 Testing Methodology

A unit test framework provides a straightforward method for creating and managing unit tests. While it is possible to write unit tests using ad-hoc code, there are several advantages to utilizing a unit testing framework. A standardized unit test framework simplifies the process of writing tests, enhances the understanding of tests among developers, offers a consistent means to execute and log tests, and reduces the likelihood of false errors arising from incorrect testing code. It is crucial to maintain consistency and simplicity in unit testing since proficient developers will invest almost as much time, if not more, in crafting unit tests as they do in writing the actual code being tested.

Throughout this course, we will employ unit testing to thoroughly evaluate each module in isolation. As previously mentioned, every source file (.v) should always be complemented by a corresponding unit test file (.t.v) responsible for testing the modules within that source.

We provide the framework for each unit test you will need in this lab. Let's briefly look through the unit test for the single cycle muldiv unit in `imuldiv-IntMulDivSingleCycle.t.v`. The first module we define, `imuldiv_IntMulDivSingleCycle`, is a helper module that wraps the `TestSource` and the `TestSink` together.

The `TestSource` contains a local memory that stores the sequence of messages we would like to test our `muldiv` unit with. The `TestSink` contains a local memory that stores the expected responses for each of the corresponding messages. Each time a message is sent from the `TestSource`, its internal index is incremented so that it points to the next message. Similarly, the `TestSink`'s index is incremented each time a response is received. When the `TestSink`'s index points to a location in its local memory that we have not initialized, its done signal is asserted - this is how we know that our test case has finished. At the beginning of our unit tests we must carefully set the memories so that message-response pairs match up, and we must briefly assert the reset signal so that the indexes are reset.

Notice that the width of the entries in `TestSource` must be equal to the width of two operands in addition to the function specifier bits which is 2*32 + 3 = 67 bits total. The output of the `TestSource` is parsed into separate signals by the `imuldiv_MulDivReqMsgFromBits` module. Similarly, the width of the entries in the `TestSink` must be equal to the width of the output, which is 32 bits in this case.

The tester module is where each test case loaded and executed. We need to first invoke the `` `VC_TEST_SUITE_BEGIN `` macro to initialize the unit testing framework located in vc-Test.v. Within the test suite, we can define test cases by invoking the `` `VC_TEST_CASE_BEGIN `` macro with the test number and test name. Note that the test cases in a suite must be numbered sequentially and in ascending order. Each test case should have a line for the operands you would like to test. We can see in the example that we directly set the local memory of the `TestSource` and `TestSink` to the test inputs and expected outputs. The underscores are used to delineate each field of the test inputs (i.e. function type, operand a, operand b). We only provide tests for the mul, div, and rem instructions. **It is your job to create more test cases for the unsigned instructions: divu and remu.** Remember that you can test div/rem and divu/remu simultaneously since the output is 64 bits. The tests you develop should be added to each of the .t.v files in the lab. It would also be wise to add additional tests for mul that test the entire 64-bit range since the single cycle implementation only tests for its 32-bit output.

6 Evaluation

Once the iterative muldiv unit is complete, recompile the iterative muldiv simulator, imuldiv-iterative-sim, by typing `'make'` in the build directory. Run the simulator for each of the following cases and report the result as well as the number of cycles:

```
0xbadbeeef *  0x10000000 = ?
0xf5fe4fbc /  0x00004eb6 = ?
0x08a22334 %  0xfdcba02b = ?
0xf5fe4fbc /u 0x00004eb6 = ? (unsigned)
0x0a56adca %u 0xfabc1234 = ? (unsigned)
```

Your muldiv unit should take the 33-cycles for any operation (32 iterations for computation and 1 cycle for val/rdy interface).

7 Extensions

The following are extension ideas to go beyond the expectations of this lab. Completing extensions is not required. However, completing each extension will earn you an additional 10 points.

- Design a variable-latency iterative multiplier that takes less than 33 cycles
- Build a 3-input iterative muldiv unit by composing two instances of your 2-input muldiv unit depending on the number of zeroes in the multiplier
- Design a parameterized pipelined iterative integer multiplier. You can assume that the number of stages will be either 1, 2, 4, or 8.

Be aware that these extensions are in the order of difficulty.

8 Submission

8.1 Lab Report

In addition to the source code for the lab, you would need to submit a lab report that includes the following sections:

- **Introduction (1 paragraph maximum)**: introductory paragraph summarizing the lab
- **Design**: describe your implementation, justifications for design decisions (if any), deviations from the prescribed datapath, and discussion. **Remember that you must provide a balanced discussion between what you implemented and why you chose that implementation.**
- **Testing Methodology**: describe how you tested the modules and your overall testing strategy (any corner cases?)
- **Evaluation**: report your simulation results and cycle counts
- **Conclusion (1 paragraph maximum)**: a brief qualitative and quantitative overview of the evaluation results (if needed)

Avoid scanning hand-written figures, and **certainly, do not capture hand-written figures with a digital camera**. The lab report holds too much importance to jeopardize its readability with illegible figures. Please ensure that each section is clearly numbered. The lab report should be written in **English** and should not exceed a **maximum of 4 pages**, but if extensions are implemented, please include the details in the method section, and you may extend the report to **5 pages**. Penalties will be imposed for exceeding this limit.

8.2 Deliverables

To summarize, you would need to hand in the following files to meet the expectations of this lab:

- Design files
  - imuldiv-IntMulIterative.v
  - imuldiv-IntDivIterative.v
  - imuldiv-IntMulDivIterative.v
  - imuldiv-IntMulVariable.v (extension)
  - imuldiv-IntMulDivThreeInput.v (extension)
  - imuldiv-IntMulPipelined.v (extension)
- Unit tests
  - imuldiv-IntMulIterative.t.v
  - imuldiv-IntDivIterative.t.v
  - imuldiv-IntMulDivIterative.t.v
  - imuldiv-IntMulVariable.t.v (extension)
  - imuldiv-IntMulDivThreeInput.t.v (extension)
  - imuldiv-IntMulPipelined.t.v (extension)
- Lab report (including simulated results and cycle counts in the evaluation section)

8.3 Submission Instructions

- Please, keep your code in the folder lab1. If you do not put code in a tarball created from this folder, we will unable to grade it.
- Before creating the tarball, please run `make clean` to clean up any generated intermediate files.
- Before submitting via e3, create a tarball with the following name: **student_id-lab1.tar.gz**
- Tarball and lab report should be submitted separately via e3, with the following name: **student_id-lab1-report.pdf**

9 Grading Rubric

- **Report (30%)**
  - Introduction (maximum 1 paragraph)
  - Design
  - Testing Methodology
  - Evaluation
  - Conclusion (maximum 1 paragraph)
- **Code (70%)**
  - Iterative Multiplication Unit
  - Iterative Division Unit
- **Extensions (Bonus, 10% each, up to 30% total)**

  - Variable-Latency Multiplier (fewer than 33 cycles)
  - 3-Input Muldiv Unit
  - Parameterized Pipelined Multiplier (1, 2, 4, or 8 stages)

.

10 Tips

- For more information on the iterative multiply and divide algorithms, take a look at Appendices J.2 and J.6 of the online-only appendices of the Hennessy and Patterson text.
- Utilize `gtkwave` to view waveforms for effective debugging.
- Embrace **incremental development**; avoid attempting to code everything at once with the hope that it will work perfectly from the start.
- If you can't get everything working, be sure to provide a comprehensive explanation of your progress in the lab report, along with a description of your debugging strategy.

11 Acknowledgments

This lab is adapted from ECE 4750 at Cornell University and ECE 475 at Princeton University.

12 Appendix

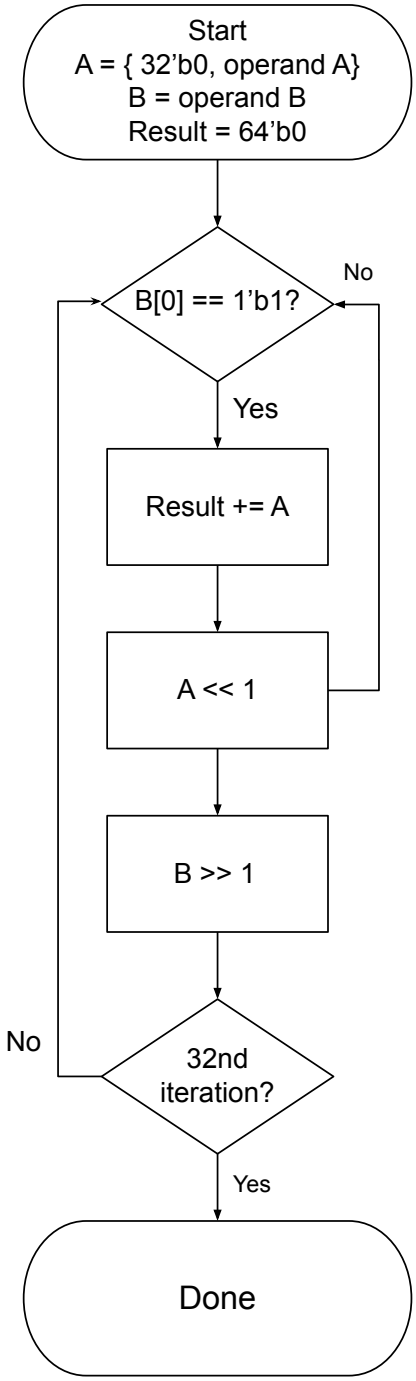

Figure 2: Iterative Multiplication Algorithm

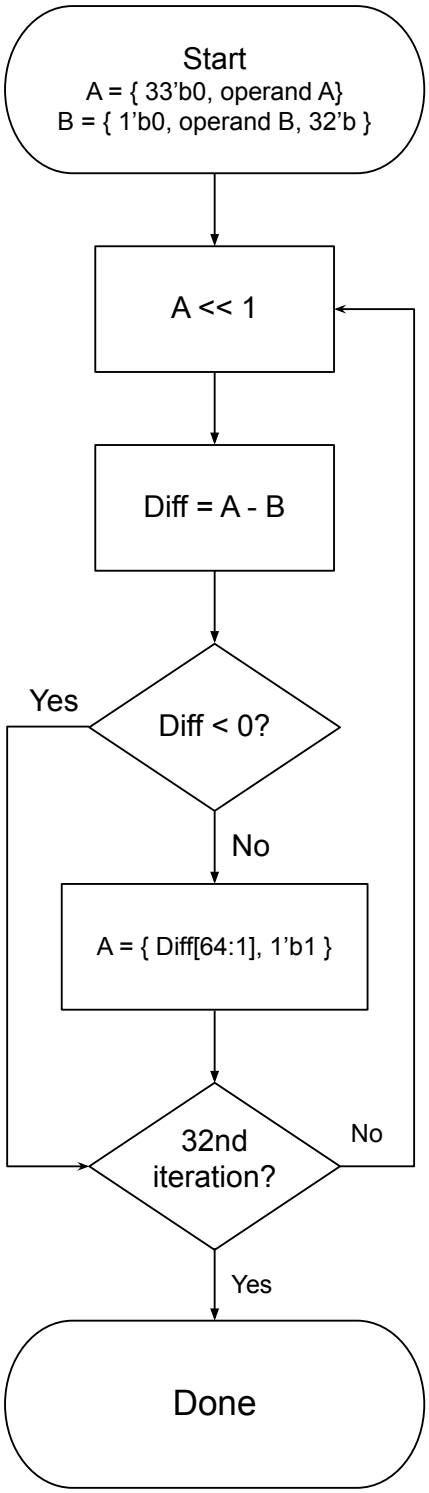

Figure 3: Iterative Division Algorithm

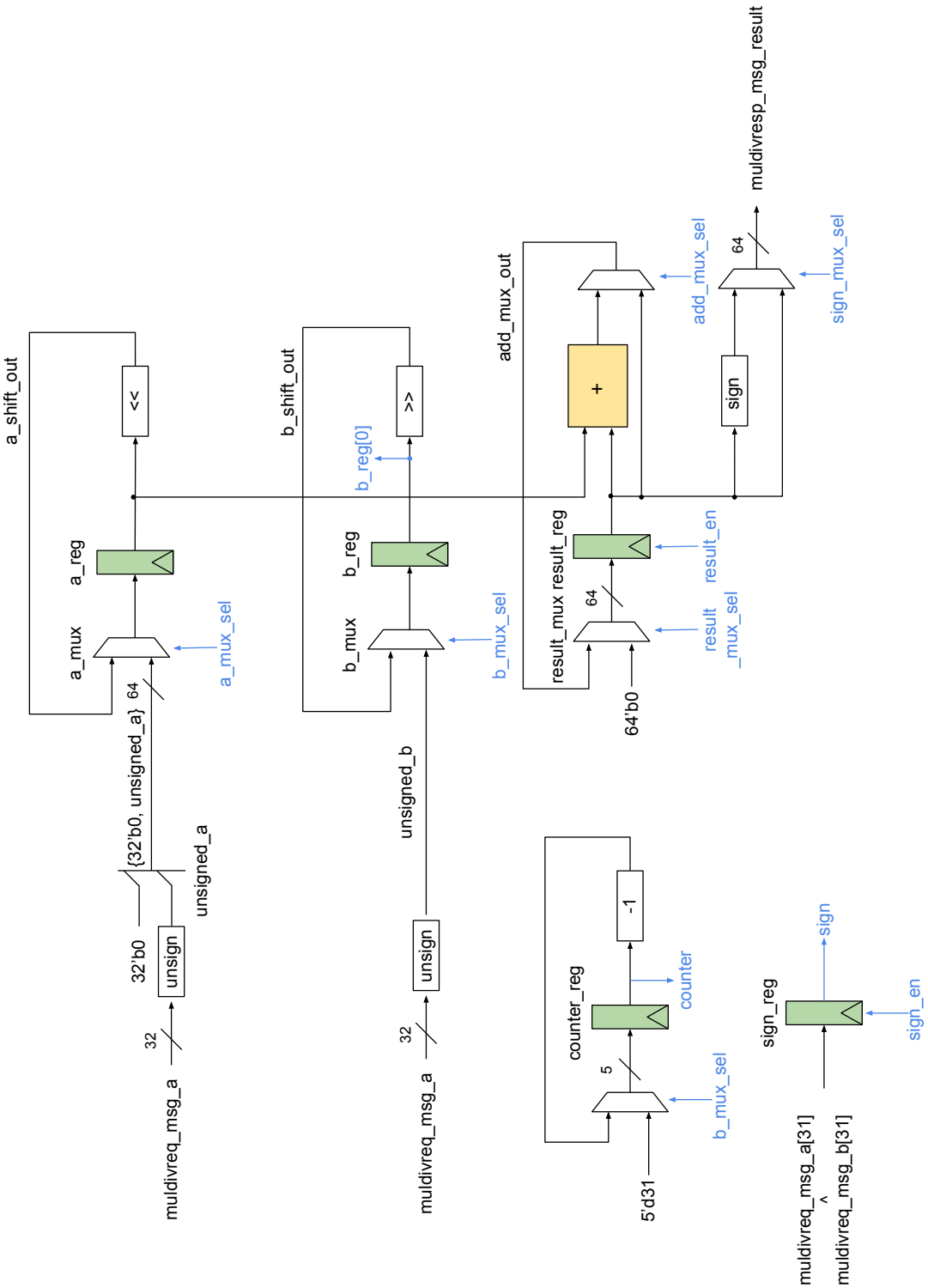

Figure 4: Iterative Multiplier Datapath

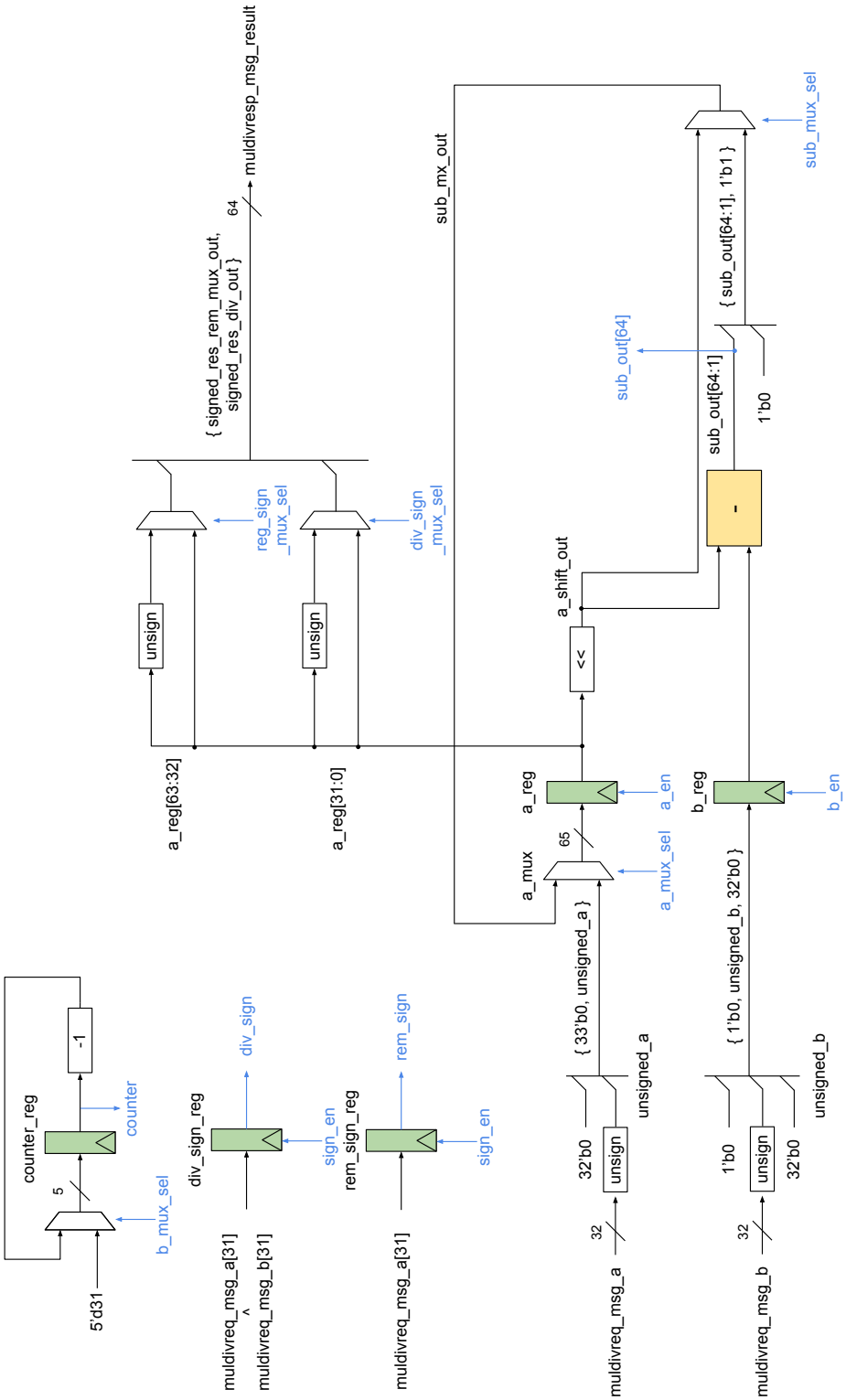

Figure 5: Iterative Divider Datapath