Computer Architecture
Lab 1: Integer Multiply/Divide Unit

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

revision: 9-18-2025

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:** 9/29 (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.gz
% 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`, `imuldiv-iterative-sim`, and `imuldiv-booth-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 second 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.

The third simulator, `imuldiv-booth-sim`, tests the functionality of the Modified Booth Multiplier. You must modify `imuldiv-IntMulBooth.v` and create your own test cases in `imuldiv-IntMulBooth.t.v` as needed. Simulation will allow you to clearly see the differences from the standard iterative version.

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 Section 3.1, you will first design an iterative multiplier unit, then proceed to an iterative divider unit (Section 3.2), and finally combine these two modules into an iterative mul/div unit (Section 3.3). Each design will take two 32-bit operands and produce a 64-bit result. After that, you will improve the efficiency of the multiplier. Since the original shift-and-add iterative algorithm is inefficient, in Section 3.4 you will implement the **Modified Booth Algorithm** for the multiplier. In Section 3.5, you will extend your two-input Booth multiplier from Section 3.4 into a three-input multiplier **by composing two instances of it**, following the Modified Booth model. Before starting the lab, make sure you are familiar with the val/rdy interface described in Section 4 and the Testing Methodology 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 3 of the Appendix. Study the algorithm carefully and make sure you understand how it maps to the hardware datapath shown in Figure 6. The blue signals in Figure 6 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 6 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 4 of the Appendix. Examine the divider datapath in Figure 4 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 adjust the sign of the result at the end. Unsigned operations are more complicated. In this case, operands should not be converted even if the MSB is 1, so that the value is correctly treated as positive and unsigned. The result should also bypass the sign block at the end.

Next, note that the datapath operates on 65-bit values instead of 64-bit values. This extra bit is necessary to properly handle unsigned division by detecting overflow. Consider a simple example with 4-bit operands, where we want to perform unsigned division on 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 each iteration, we shift register A to the left by 1 and subtract register B from it. The overflow issue becomes apparent in the very first iteration:

```
  Reg A = 0000 1111
- Reg B = 1111 0000
--------------------
       1 0001 1111 < correct, the difference is negative
         0001 1111 < 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).

Implement the datapath in Figure ?? by separating it into a datapath module and a control logic module. A single-cycle reference divider is provided in `imuldiv-IntDivIterative.v`; **modify this code to create your iterative divider and add test cases for `divu` and `remu`**.

3.3 Iterative MulDiv Unit

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

After completing the iterative mul/div unit, verify that the cycle counts meet the requirements before proceeding to Section 3.4, since it builds on the implementation from Section 3.3. Submissions that exceed the expected cycle counts may be subject to point deductions.

3.4 Modified Booth Multiplier

Regular iterative muldiv unit results in a relatively high cycle count. The objective of this section is to reduce the number of cycles by implementing the **Modified Booth Algorithm** (also known as the Radix-4 Booth Algorithm).

This optimization applies only to the multiplication operation. The flowchart in Figure 5 illustrates the algorithm. Compared with the standard iterative multiplier, the Radix-4 version examines the last three bits of the multiplier and shifts by two bits per iteration, thereby reducing the cycle count. Since this algorithm is an enhancement of the original iterative multiply algorithm, the datapath can be derived from the original iterative multiplier with only partial modifications.**You are required to implement the Modified Booth multiplier** in `imuldiv-IntMulBooth.v`.

With this algorithm, the required cycle count is approximately half of the original.

3.5 3-input Iterative Multiplier

This part extends the two-input multiplier to a three-input multiplication. Since only a two-input multiplier is available, the operation must be implemented by composing two instances of it.

Only the multiplication operation needs to be supported. The design should cascade two two-input multipliers to realize the three-input functionality, depending on the number of zero bits in the multiplier.

A correct design should work with either the standard iterative multiplier or the Modified Booth version. For this part, however, the implementation must **use two instances of the Modified Booth multiplier unit** in `imuldiv-IntMulThreeInput.v`.

4 The val/rdy Interface

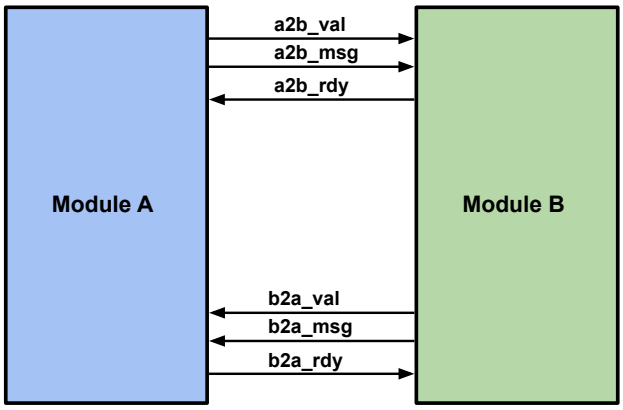

Figure 1: Example of data valid

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 a `val/rdy` interface. The `val/rdy` protocol is latency-insensitive, meaning that data exchange between modules does not depend on precise timing. Instead, a contract governs the transfer of data between modules. This exemplifies the principle of `encapsulation`: implementation details (e.g., `muldiv` unit latency) are hidden from the interface, allowing other modules to interact with the `muldiv` unit without needing to know the number of cycles required to complete an operation and produce a result.

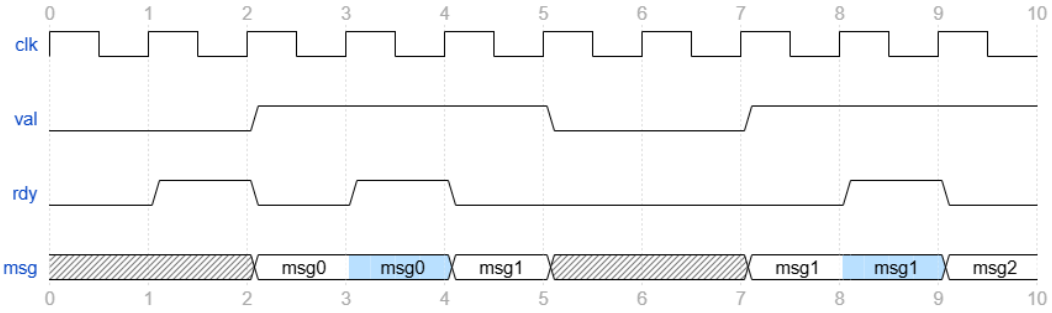

Figure 2: Example of data valid

The `val` signal indicates valid data, and the `rdy` signal indicates readiness to receive. **These signals are independent to avoid combinational loops.** A transfer occurs only when both are high.

In this lab, the `muldiv` units use request and response interfaces based on the `val/rdy` protocol. On the request side, `val` asserts when operands are available and `rdy` asserts when the unit can accept them. On the response side, `val` asserts when the result is valid, and `rdy` asserts when the next module is ready to accept it.

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.

In each case, the operands to be tested are specified directly in the local memory of the `TestSource`, while the expected outputs are stored in the `TestSink`. Underscores are used to separate the fields of the test inputs (e.g., function type, operandA, operandB). The provided examples cover only the `mul`, `div`, and `rem` instructions. **Additional test cases must be created to fully validate the design.** Since the output is 64 bits, `div/rem` and `divu/remu` can be tested simultaneously. It is also recommended to add more tests for `mul`, especially cases that exercise the full 64-bit range, as the single-cycle implementation tests only the 32-bit output.

6 Evaluation

Once the iterative mul/div unit is complete, recompile the simulators `imuldiv-iterative-sim` and `imuldiv-booth-sim` by typing `make` in the build directory. Run `imuldiv-iterative-sim` for each of the following cases and report both the result and the number of cycles. Additionally, use `imuldiv-booth-sim` to report the cycle counts for multiplication.

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

For the iterative mul/div unit, **the design must complete any operation in 33 cycles** (32 iterations for computation plus 1 cycle for the `val/rdy` interface). For the Modified Booth multiplier, **the design must complete any operation in 17 cycles**, since the number of shift operations is reduced by half. For the three-input multiplier, cycle count is not evaluated; instead, all test cases must pass, and the design must strictly follow the specified requirements.

7 Submission

7.1 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**. Penalties will be imposed for exceeding this limit.

7.2 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-IntMulBooth.v
  - imuldiv-IntMulThreeInput.v

- Unit tests
  - imuldiv-IntMulIterative.t.v
  - imuldiv-IntDivIterative.t.v
  - imuldiv-IntMulDivIterative.t.v
  - imuldiv-IntMulBooth.t.v
  - imuldiv-IntMulThreeInput.t.v
- Lab report (including simulated results and cycle counts in the evaluation section)

7.3 Submission Instructions

- You must only submit the files listed in the Deliverables section via e3 (both design files and corresponding test files).
- Before submitting via e3, please use the following commands to package your submission:

```
% cd $LAB1_ROOT/imuldiv
% tar -cvzf student_id-lab1.tar.gz \
  --exclude="imuldiv-IntMulDivSingleCycle*" \
  imuldiv-Int*
```

- Code and lab report should be submitted separately via e3, with the following name: **student_id-lab1-report.pdf**

  For example: 123456789-lab1-report.pdf

8 Grading Rubric

- **Report (30%)**
- **Code (70%)**
  - Iterative Mul/Div Unit (35%)
  - Booth Multiplication Unit (20%)
  - Three Input Multiplication Unit (15%)

.

9 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.
- Develop incrementally; don't try to implement everything at once.
- 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.

10 Acknowledgments

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

11 Appendix

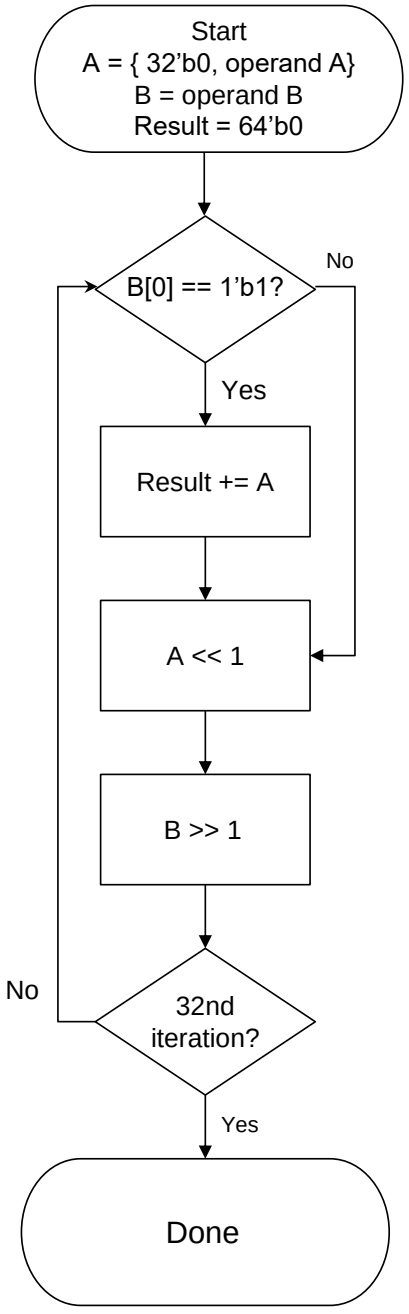

Figure 3: Iterative Multiplication Algorithm

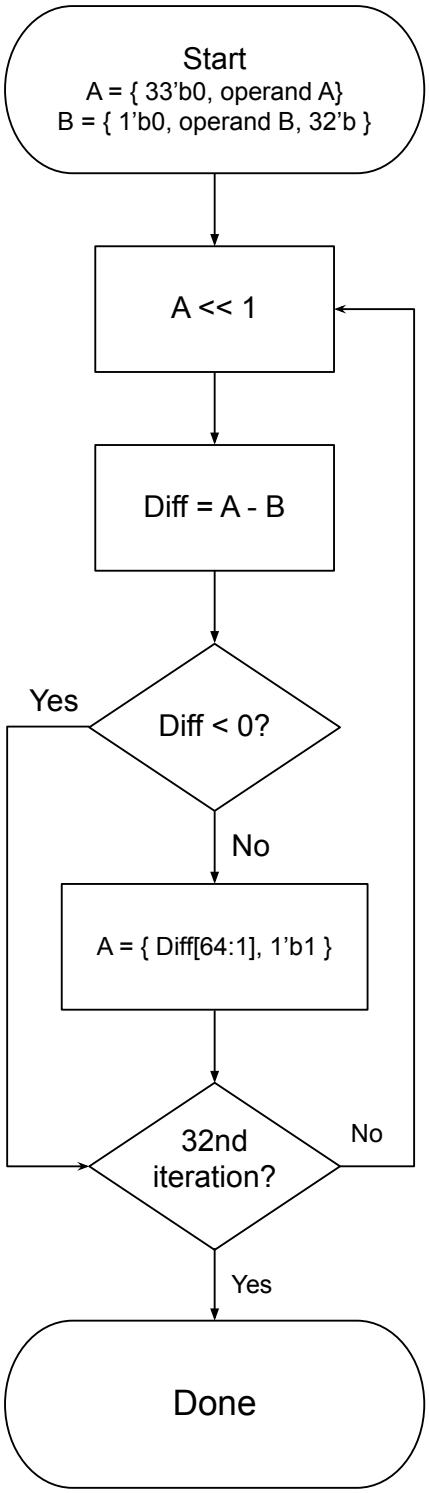

Figure 4: Iterative Division Algorithm

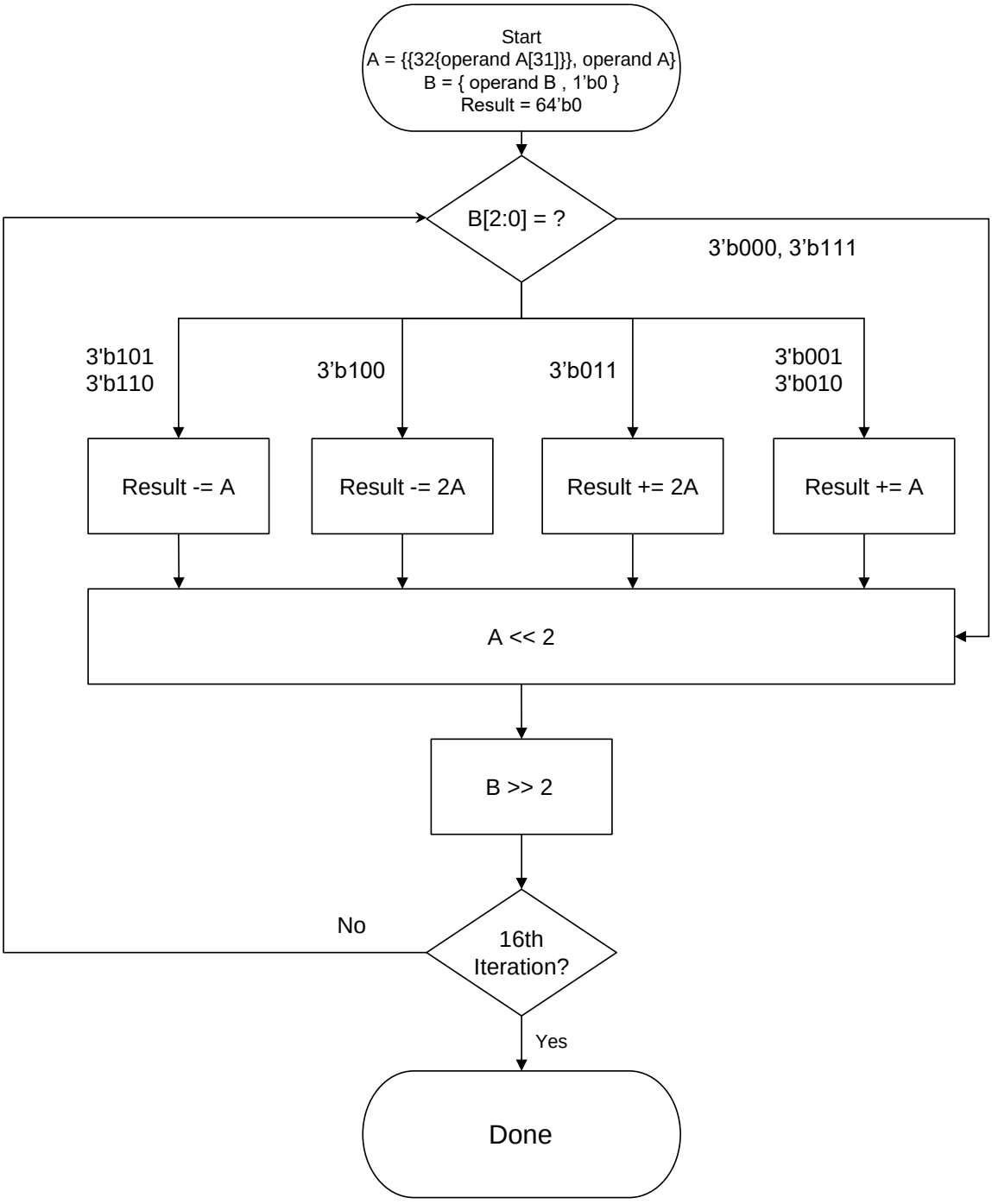

Figure 5: Modified Booth Algorithm

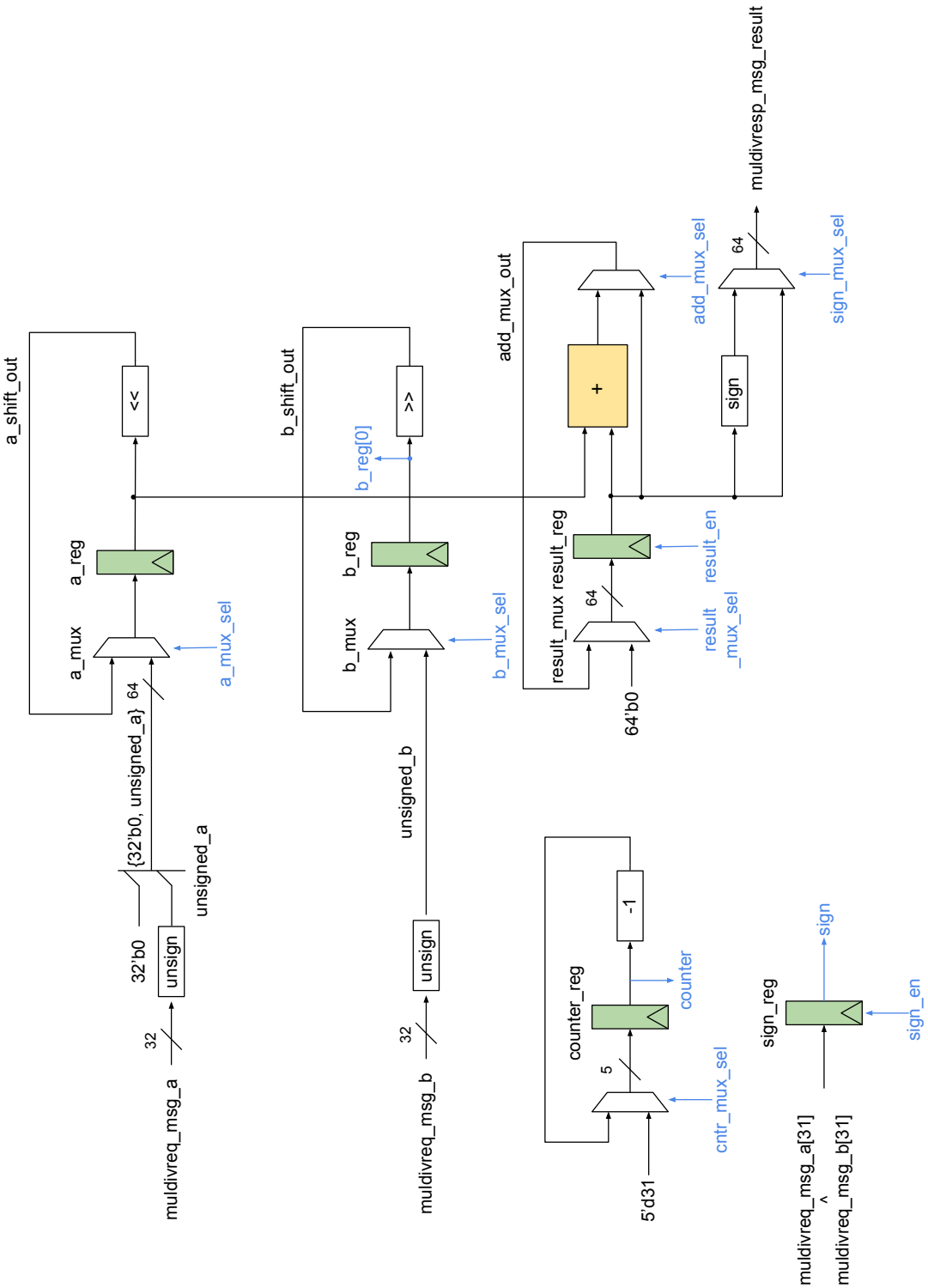

Figure 6: Iterative Multiplier Datapath

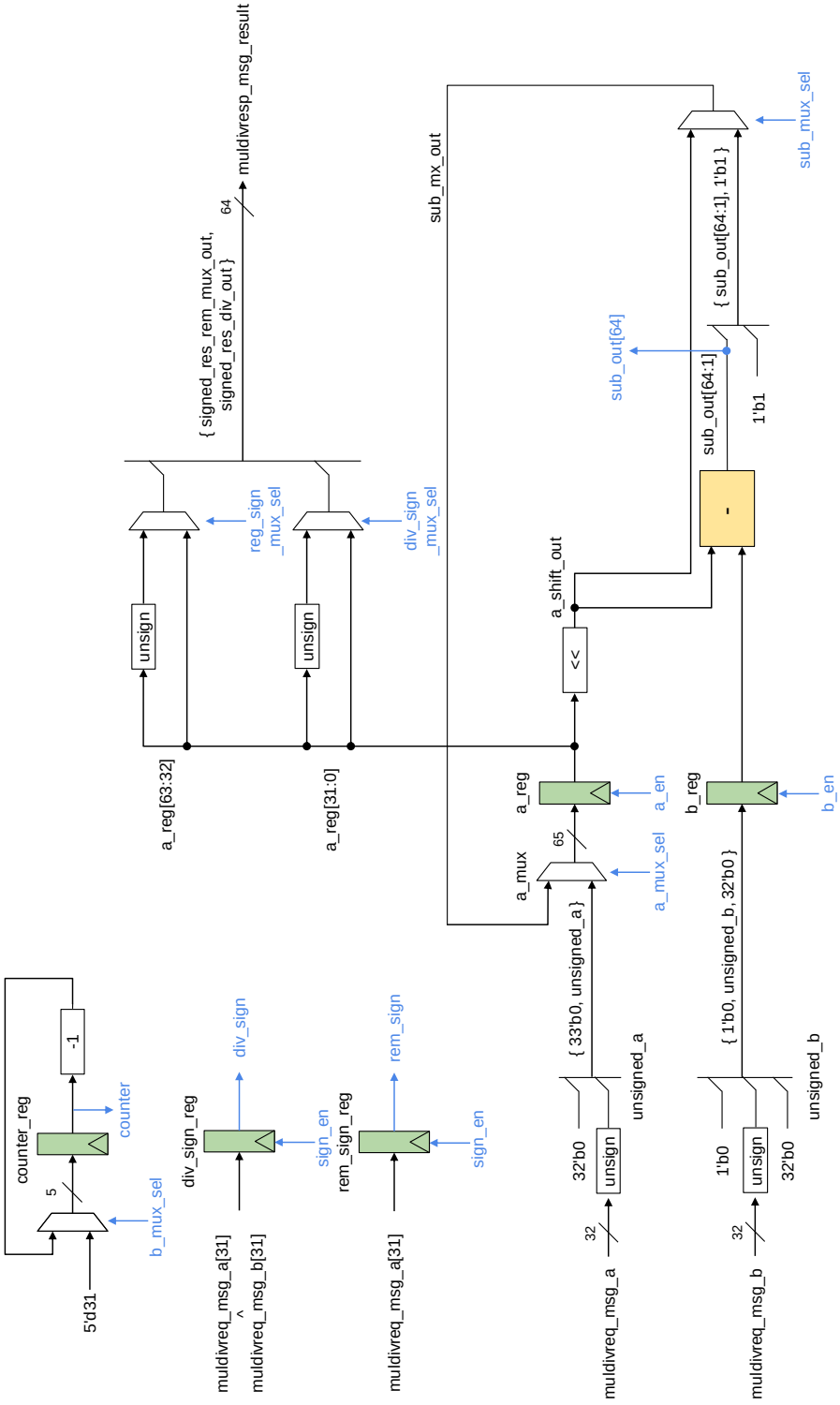

Figure 7: Iterative Divider Datapath