314513064-lab1-report

1. Introduction

We implemented iterative multiplier/divider units, a radix-4 Booth multiplier, and a three-input multiplier. The project provided hands-on experience with Verilog, val/rdy interfaces, datapath/control separation, and systematic testing.

2. Design

2.1 Iterative Multiplier

The iterative multiplier implements the classic shift-and-add algorithm with a **clear datapath/control split**. We made this separation not only for modularity, but also because it allowed us to reuse the same datapath when experimenting with different control strategies during debugging. This proved helpful in waveforms: registers always updated in a predictable manner, regardless of FSM timing.

Instead of recalculating the result's sign at the very end, we chose to **capture it at request acceptance** and store it in sign_reg. This avoided ambiguity if inputs changed mid-operation, and simplified downstream logic. While sign handling could have been deferred, early capture provides robustness in pipelined environments.

We used a 6-bit counter to comfortably cover the full 32 iterations plus one interface cycle. This added a small area overhead, but greatly simplified termination detection. Our FSM retained four states (IDLE–LOAD–RUN–RESP) even though LOAD could theoretically be merged into IDLE. The extra explicit state improved readability for us and made waveform traces easier to interpret when validating handshakes.

2.2 Iterative Divider

The divider follows the restoring algorithm but with **strict constant latency of 33 cycles**. A natural idea would have been to terminate earlier on divide-by-zero or overflow. We explicitly avoided this to ensure **deterministic timing**, which is crucial for the val/rdy protocol: downstream logic never needs to guess when a result might appear. This decision traded some performance in edge cases for

simplicity and protocol compliance.
We also decided to apply **sign correction in the same cycle as result packing**. Splitting this into another pipeline stage could have reduced combinational delay, but would have added an extra cycle and a more complicated FSM. Keeping correction inline kept the control logic small and easy to verify.
Although the datapath required 65-bit subtraction to detect overflow, our choice was to check diff[64] combinationally rather than add another register stage. This made timing closure slightly tighter but guaranteed cycle-accurate behavior. The FSM remained a concise three states (IDLE–CALC–DONE), which we preferred over more elaborate breakdowns, since cycle count was fixed by design anyway.

2.3 Iterative MulDiv Unit

The combined MulDiv wrapper acts as a **routing layer only**, ensuring both multiplier and divider share a single external interface. A key choice was to **require both submodules to be ready before asserting top-level ready**. This prevents one submodule from being starved or overloaded if a request arrives while the other is still processing. Although this slightly restricts concurrency, it eliminates subtle deadlock scenarios.
We also decided not to include arbitration FIFOs or internal buffering. Since the lab environment guarantees that only one operation arrives at a time, the wrapper can stay purely combinational on routing logic. This reduced hardware overhead and preserved constant cycle counts across all operations.
Importantly, this design let us **reuse the same testbenches** for all units, since the external behavior is identical regardless of the underlying arithmetic module.

2.4 Modified Booth Multiplier

The radix-4 Booth multiplier halves iteration count by recoding three bits at a time, but our design emphasized a few deliberate decisions:

- We chose to **latch the result on the last calculation cycle** rather than introduce an extra RESP cycle. This avoided wasting a clock and made cycle count exactly 17, aligning cleanly with the specification.
- The multiplier operand B was updated using **arithmetic right shifts**, not logical, so that negative multipliers retained their sign bit correctly. This

was a conscious choice: while a logical shift is simpler, it would mis-handle two's-complement cases.

- Booth decoding was moved to the control logic instead of embedding it in the datapath. This improved **observability in waveforms**, as control outputs now directly reflect decoding outcomes, and kept datapath arithmetic straightforward.
  These choices increased the complexity of the FSM slightly, but provided a cleaner modular structure and predictable latency, which was easier to verify against both random and corner-case test vectors.

2.5 Three-Input Multiplier

The three-input multiplier cascades two Booth units to produce a 96-bit result. While one option was to implement iterative multiply-accumulate directly, we instead **reused verified Booth modules**. This minimized new logic, increased confidence in correctness, and aligned with the lab's requirement to build upon Section 3.4.

A subtle challenge was handling the L*C correction when L is negative in two's-complement form. Instead of introducing another multiplier, we applied a **shift-and-add fix term (C << 32)**. This was a deliberate trade-off: it slightly complicated accumulation logic but saved significant hardware resources.

Our FSM was structured into phases: Phase 1 computes AB, Phase 2 concurrently launches L*C and H*C, Phase 3 accumulates results, and finally RESP delivers the output. By parallelizing Phase 2 instead of serializing all three products, we achieved lower total response latency. This required additional sent/received flags for synchronization, but we judged the overhead acceptable for the gain in throughput.

Ultimately, this design demonstrated how **reuse, correction logic, and partial parallelism** can be combined to meet correctness requirements without deviating from cycle constraints.

3. Testing Methodology

In addition to the provided test cases, we extended the testbenches with new vectors to cover corner cases. For the divider, explicit divu/remu cases were

added, including 0 ÷ 1, maximum ÷ 2, a < b, and a = b, along with the two unsigned examples specified in the evaluation section. For the three-input multiplier, we created a diverse set of vectors stressing negative products (e.g., $(-1)^3$), extreme values such as INT_MIN combined with small constants, powers of two to check carry propagation, and random patterns. These additional tests confirmed correctness across sign boundaries, high-bit carries, and special conditions.

4. Evaluation

We evaluated the design using both functional correctness and cycle counts. The following test cases were run as required by the lab specification:

```
allenlee@LAPTOP-2M4NVB8L:~/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
../imuldiv/imuldiv-iterative-sim.v:130: $finish called at 345 (1s)
allenlee@LAPTOP-2M4NVB8L:~/lab1/build$ ./imuldiv-iterative-sim +op=div  +a=f5fe4fbc +b=00004eb6
VCD info: dumpfile dump.vcd opened for output.
0xf5fe4fbc / 0x00004eb6 = 0xffffdf75
Cycle Count =         33
../imuldiv/imuldiv-iterative-sim.v:130: $finish called at 345 (1s)
allenlee@LAPTOP-2M4NVB8L:~/lab1/build$ ./imuldiv-iterative-sim +op=rem  +a=08a22334 +b=fdcba02b
VCD info: dumpfile dump.vcd opened for output.
0x08a22334 % 0xfdcba02b = 0x020503b5
Cycle Count =         33
../imuldiv/imuldiv-iterative-sim.v:130: $finish called at 345 (1s)
allenlee@LAPTOP-2M4NVB8L:~/lab1/build$ ./imuldiv-iterative-sim +op=divu +a=f5fe4fbc +b=00004eb6
VCD info: dumpfile dump.vcd opened for output.
0xf5fe4fbc /u 0x00004eb6 = 0x00032012
Cycle Count =         33
../imuldiv/imuldiv-iterative-sim.v:130: $finish called at 345 (1s)
allenlee@LAPTOP-2M4NVB8L:~/lab1/build$ ./imuldiv-iterative-sim +op=remu +a=0a56adca +b=fabc1234
VCD info: dumpfile dump.vcd opened for output.
0x0a56adca %u 0xfabc1234 = 0x0a56adca
Cycle Count =         33
../imuldiv/imuldiv-iterative-sim.v:130: $finish called at 345 (1s)
```

The iterative design met the 33-cycle requirement, and the Booth multiplier successfully reduced multiplication to 17 cycles. The three-input multiplier passed all functional tests.

5. Conclusion

This lab demonstrated iterative and optimized multiply/divide units, with Booth multiplication reducing latency to 17 cycles and the three-input multiplier delivering correct 96-bit outputs. The work reinforced modular datapath/control design, val/rdy usage, and systematic testing.