# E154 Lab 4

Design

Now that you are starting this lab, don't refer to the textbook multiplier chapter or to the main branch of Wally that already contains a working multiplier.

Make sure you have the latest version of the cvw repository:

```
cd $WALLY
git pull upstream main
```

Run regression to make sure your repository passes lint and tests.

```
lint-wally
regression-wally
```

Now run a script to strip out some of the multiplier from ieu/controller.sv and mdu/mul.sv (don't look at this script).

```
$WALLY/examples/exercises/16p1/antimul.sh
```

Consider P = A x B, where A and B are N-bit numbers and P is a 2N-bit product.

We can express A = {Am, A'}, where Am is the most significant bit and A' are the other N-1 least significant bits. If A is unsigned, its value is A' + Am x $2^{N-1}$. If A is signed (2's complement), its value is A' - Am x $2^{N-1}$.

We can do the same for B, and define products of the terms:

P' = A' x B'
PA = Bm x A'
PB = Am x B'
Pm = Am x Bm

We now find expressions for the product of various unsigned and signed multiplications:

| **A** | **B** | **P** |
|---|---|---|
| Unsigned | Unsigned | $P' + (PA + PB)2^{N-1} + Pm2^{2N-2}$ |
| Signed | Signed | $P' + (-PA + -PB)2^{N-1} + Pm2^{2N-2}$ |
| Signed | Unsigned | $P' + (PA + -PB)2^{N-1} - Pm2^{2N-2}$ |

To handle negative terms, shift them left to the appropriate column (by N-1 or 2N-2) and zero-extend to 2N bits. Then take the two's complement by inverting all of the bits including the zeros at the start and end, and adding one to the lsb. Now we are left with some 1s that we can pre-add to simplify. Work this out for yourself to see the math.

We can now express P = P1 + (P2 + P3) << N-1 + P4. The signed x signed case is color-coded to track the example below. The signed x unsigned case is left for you.

| **A** | **B** | **P1** | **P2** | **P3** | **P4** |
|---|---|---|---|---|---|
| Unsigned | Unsigned | P' | PA | PB | Pm << 2N-2 |
| Signed | Signed | P' | ~PA | ~PB | Pm << 2N-2 + 1 << 2N-1 + 1 << N |
| Signed | Unsigned | | | | |

For example, consider A = 1111 and B = 1110.
Am = 1 A' = 111
Bm = 1 B' = 110
P' = 111 x 110 = 101010
PA = 1 x 111 = 111 -> ~PA = 000
PB = 1 x 110 = 110 -> ~PB = 001
Pm = 1 x 1 = 1

For signed x signed multiplication, the bits are added up on the grid below

| P1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| P2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P3 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| P4 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| P | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

In summary, A = -1, B = -2, and P = (-1)x(-2) = 2, so the result is correct.

Now it is your turn to work out signed x unsigned multiplication by computing P1, P2, P3, and P4. Your answers should be similar to the other cases, and P4 should contain

a one in the most significant column and a 1 in another column (not column N). Try doing a 4-bit example such as -1 x 14 to help figure this out.

## SystemVerilog

Modify the SystemVerilog to add support for all multiplication instructions (MUL, MULH, MULHU, MULHSU, and, for RV64M, MULW).

In ieu/controller.sv, add a few lines to the case statement to decode multiply and MULW type instructions. You'll need to consult the RISC-V specification for the machine language encodings. You should only need to consider OpD and Funct7D, not Funct3D. Work out the values of the control signals for the new cases. All M-extension signals should assert the MulDiv output (named "MDUE") to say that a multiplication or division is occurring, and should set ResultSrc to $011_2$ so the datapath will select MulDivResultW.

In mdu/muldiv.sv, read over what is already there, but you won't need to change anything. This module handles selecting between multiplication and division, and handles sign extension for W-type instructions. In mdu/mul.sv, write the logic to compute the four PP*E signals (called P1-P4 in the table above). You will rely on the equations in the table above and the new ones you derived for signed x unsigned. The file has pipeline registers to advance these signals to the Memory stage and add them there to form ProdM.

Run `lint-wally` and fix any lint errors. To test your changes, you can simulate multiply tests using:

```
wsim rv64gc arch64m
```

Hopefully you will have some mistakes and get experience debugging. Pick a failing case from the test vectors that you can readily analyze by hand. Run your simulation with the GUI by adding `--gui`. Predict what all of the multiplier signals should be, and compare them with the simulation on that test case. With discipline, you can isolate your bugs quickly. If you shoot at random or don't have a test case where you know the expected results, you could spend a long time on this.

Once you've gotten the 64-bit tests running, run `regression-wally` to do the 32-bit tests as well, and to make sure you didn't break anything else.

## What to Turn In

1) [4] P1, P2, P3, and P4 for signed x unsigned multiplication.
2) [3] The controller lines you have changed.
3) [3] The mul unit, with your changes highlighted.
4) [5] Do arch64m tests pass for MUL? MULH? MULHU? MULHSU? MULW?
5) [1] Does `lint-wally` pass?
6) [1] Does `regression-wally` pass overall?
7) [1] Please report how long you spent on this lab.