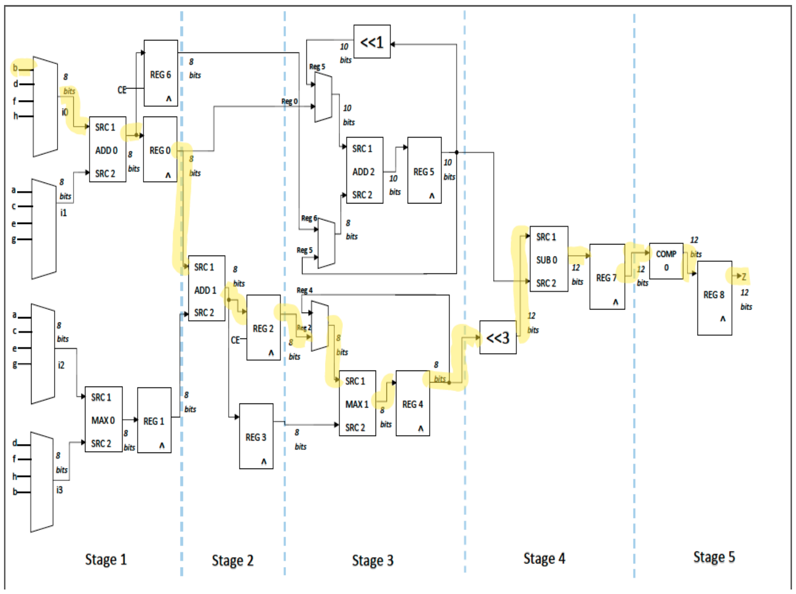
ECE 327 Design Report

Group Members: Ankur Mangal (amangal) Amish Patel (a235pate) Abilas Sathiyanesan (a6sathiy) Nakash Ali (n36ali)

Design goals and strategy



We decided to start with minimizing our clock period as our first major design goal. Our initial design had an extra component, a multiplier, which was needlessly increasing our clock period. We got rid of it by using a shifter to minimize the delay added by the multiplier. This in turn helped us decrease the clock period. We also implemented pipelining to further decrease the clock period. By using resources in parallel, we were able to process more input in a given time, increasing throughput. Using resources in parallel means our clock period will be the result of the max operation between operations in parallel. If these were not pipelined, then we would have to add up every operation in series, significantly increasing our clock period.

Our next major design goal was to minimize area. To obtain minimal area we would have to look at the total number of components, registers and muxes, and try minimizing each of them. For components, we reused our 3 adders and 2 max components in different clock cycles. This saved area, compared to using completely new components for the same functionality. For every component we had a new register, instead of re-using an old register. This may appear to waste space, but by using this approach, we did not have to include more muxes on the old registers. There were certain trade offs between using more muxes or using more registers, they both increased area, however using more registers instead also minimized clock period. This is because muxes include logic, where as for a register we are just storing the value which is not as complex an operation. We also allocated registers in a way that would minimize the number of multiplexers for the component inputs by assigning registers to components. For example, an add component would have its own register it would store to, and we used the value of the register instead of using more multiplexers.

Our circuit had a latency value of 8, which happened to be the most optimal value for us. Trying to reduce the latency any further would have made us increase our area. The latency value also lined up very nicely with our throughput value. We have a throughput of ¼, which means we can receive new input every 4 clock cycles. If our latency was any higher than 8, we would be getting more inputs from the PC on a given latency period. We would then have to process multiple inputs at the same time which would needlessly complicate our code.

This report does not contain a revised dataflow diagram for our design because the only significant changes we made to our original design was the addition of several registers, which pertain to the low-level details of the implementation of our original DFD. Therefore, not only would the revised diagram be too dense with all the new registers, it would also represent the same high-level design. This also explains why our optimality is much higher when calculated using the original DFD, because it doesn't consider all the extra flip-flops and registers that are needed to implement the DFD. Moreover, it also doesn't contain the implementation of the memory, which only outputs one pixel at a time, thus increasing the necessity of more registers.

Performance result and projections

Our first calculated optimality based on our design:

$$Optimality = Functionality * \frac{Clock\ Frequency}{Area}$$

We have a clock frequency of 340Mhz and an area of 99 LUTS based on our DFD calculations.

$$Optimality = 1000 * \frac{340Mhz}{99}$$
$$= 3434.34$$

We obtained a latency of 8. This was an extremely high value for optimality, and we can deduce this to the fact that we are missing a lot of additional circuits.

Our actual optimality based on our implementation (before additional optimizations were made) was calculated to be 694 with a latency of 9. This difference can be explained by a lot of additional circuitry we had to add when trying to implement the logic. Examples of this include conditional statements, additional edge test cases (i.e. negative results in optimized equation, direction prioritization, etc.) and the need for more registers. We only had 8 registers in our design but when implementing we found the need for multiple temporary registers to store and compare values. We added an additional 12, which made a big difference on optimality.

After our final implementation and multiple optimality enhancements we received a final optimality score of 713. We tried several methods to increase optimality, one of the most successful being converting our conditional statements from if-else to when-else statements. We believe this happened because when in a clocked process everything inside will be a register which will increase our area. By using when-else statements, which could be implemented much more efficiently than multiple if-else statements for single assignments, we saved a lot of space.

Another method we used was to maximize concurrency using multiple processes running in parallel. An example is our different stages, which are implemented in separate processes.

One way we thought would increase optimality, but ended up hurting us was using a separate procedure to find max. At first, we were finding the max on each stage manually and we did it twice. We thought using a procedure would end up decreasing the amount of area we used. However, a problem was that, when we were doing prioritization of the signals, and had to define which direction to go to in the case of the inputs being equal, we had to define really complex conditional logic. This was because, if we wanted to use one procedure for both stages that calculated the max (stage 1 and stage 3) we would have to include a bunch of conditionals to check with stage we were on and which equality we were faced with. This was much easier to do as separate pieces of code in each stage. Not only that, but the area was much more with the complex conditional statements in the separate procedure.

Resource	Used	Avail	Utiliziation
IOs	34	-	-
LUTs	296	58080	0.51%

Registers	246	58080	0.42%
Memory Bits	6144	6617088	0.09%

Verification Plan and Test Cases

Our verification plan started with checking to see if we were reading the right inputs and storing them correctly in the different memory arrays. We had to make sure our memory arrays were correct before going on to calculations. We used a similar implementation to that of the one we used in lab 3, however we modified the bounds to account for the fact of it being a 256*256 matrix, and that calculations would not start until the 2nd column on the 2nd row was read. To ensure we had the correct input values, we outputted the results of the memory array into a debug signal so we could see it on the simulation. From there we compared the values to the ones we were getting from i_pixel.

After our input was verified, we tested our convolution table to make sure we were testing the right input. We did this manually again, by outputting values of the inputs used in our calculation as well as the output itself. We manually calculated the value our self to ensure we were getting the right outputs. We did this by tracing the input pixel being calculated, and then finding the other 8 corresponding neighbours of the pixel. Although a lengthy process, we found this the best way to narrow down the specificity of the problem.

Some specific tests that failed, was a few of the pictures had edges we were detecting where there were no edges. We figured out this was because of our input scheduler. Our input scheduler was not changing the last pixel of each row because the address would change to 0 in the clock cycle after the last pixel was passed. Also, we mistakenly made our input scheduler only work for addresses greater than 1. This was because we confused the logic between reading inputs and doing calculations.

Another interesting test case, was how we were doing our directions. The way we set up directions, was that we had a 4 bit variable and a 3 bit variable. In accordance with our dataflow diagram, there were 3 max operations to determine the higher of 4 of the neighbouring pixels. This would immediately tell us which is the higher of (N, NE), (E, SE), (S, SW) and (W, NW). We would output the result, either a 1 or a 0, on to the 3 bit variable. The 4 bit variable would have the value of the next 4 max operations, which would compare the additional add statement results. Once we had the bit position of both variables filled, we would narrow it down to a single direction. This worked well, however a major problem was that before the final value of the first 3 bit variable was ready to use, we would already begin replacing the values with the next input. This was giving us a huge skew in our directions compared to the solution. To counter this, we made 2 additional signals, that were the finalized versions of the 3 and 4 bit variables. These finalized signals, would store the values, so that we could preserve the value while the initial variables would start updating with the next input.