Dhruv Chopra,Shail Choksi, Sachet Vora, Parth Gajaria

Kirsch Edge Detection Project Design Report

Design Goals and Strategy

The initial goal of our design was to implement the Kirsch Edge Detector algorithm with a latency of 8 while maximizing clock speed. Although minimizing area was not a primary goal, we did attempt to account for it while staying true to our primary goals. To maximize clock speed we avoided having any dependencies within a single clock cycle. For example, we ensured that the result of a mux did not depend on the result of an addition operation within the same cycle. In order to calculate direction, the maximum value of 4 different equations was needed. For edge detection, the same maximum value was needed and the sum of all 8 numbers in the convolution table was needed. To speed up the calculations of the 4 equations, we computed two at a time in parallel within a clock cycle. Thus, with 1 adder and a mux we were able to design our dataflow to compute the max of all 4 equations in 7 clock cycles. Meanwhile, on the side as the sum of two individual numbers from the convolution table became available, we added those two results so that we can get the sum of all 8 numbers by the 7^{th} cycle as well. With these results we were able to get the direction and edge in 9 cycles. Although we did not meet the goal of 8 cycles, we were satisfied. We could have added another mux and an adder to reach a latency of 8. However we did not want our area to get too big either. Hence we traded off an extra cycle to avoid increasing area too much. Additionally, adding too many computations within a clock cycle could also negatively impact our clock speed. Based on our design we could see the clock cycles being small since there were never too many computations in a single cycle. The only “bottle-neck” our dataflow had was when computing the max value between two large numbers and using the subtraction operator for 13 bit registers. In hindsight this was not a very optimal solution. At the time of drawing our first dataflow we did not consider pipelining our design as we were unaware of its many benefits.

Upon finishing the code for our design and attaining 100% functionality, we were not satisfied with our results. Our clock speed was only around 100MHz and our area was about 340 cells. Moreover, the design was not very optimal. Thus, we decided to redesign our algorithm and change our design goals so that we can attain a more optimal solution. With the new design, our aim was to attain a clock speed of 200 MHz and have an area of 300 cells. To compensate for any issues, we would aim to keep the latency under 10. A dataflow diagram of the new design is shown in **Figure 1** below. This diagram has overlapping pipeline stages which help increase our overall clock speed. The underlying thinking behind each stage is to avoid dependencies. If the dependencies for a certain group of calculations are ready then we can compute that group of calculations in a parallel stage to the current one. This way we can compute our results faster. The optimized equation to determine whether an edge exists, 8a-3b (where a is the largest sum of any three edge pixels and b is the sum of all edge pixels in the convolution table), can benefit from this strategy. If the right combination of pixels is selected, then the direction can be calculated while calculating EdgeMax as well. Each direction is represented by 3 pixels and any two adjacent directions have 2 common pixels. Hence, instead of adding all possible directions and comparing which one is highest to find a possible edge, we will first compare two adjacent directions and see which one is more dominant. Since these two directions differ by one pixel, we can do a

comparison of which pixel is bigger, and only perform the EdgeMax calculation for that one direction (we will have implicitly compared the two sets of pixels without actually computing their sum).

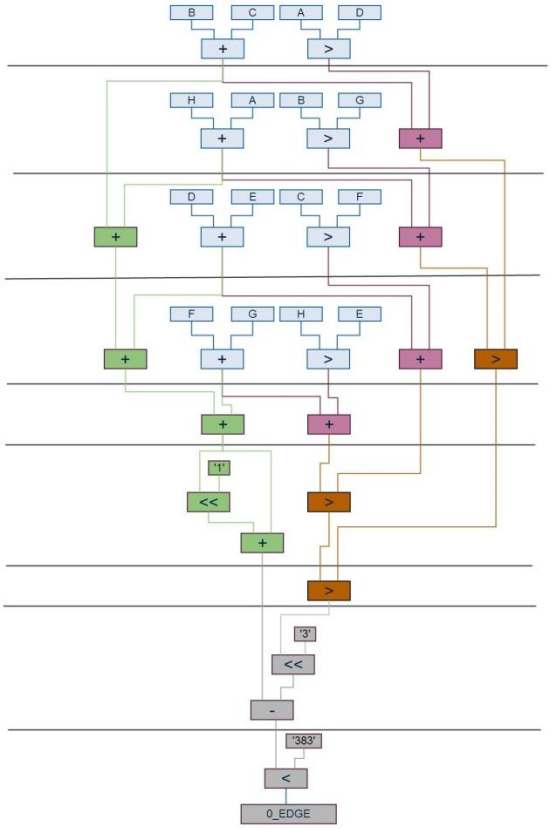

Figure 1 Optimized Dataflow Diagram. Blue=Stage1, Pink=Stage2, Green=Stage3, Orange=Stage4, Grey=Stage5

The diagram on the left illustrates our dataflow diagram. Stage 1 (blue) computes the maximum between all of the distinct pixels and sum of all the pairs of pixels that are common to two directions. Stage two (pink) computes the sum of all the equations needed to calculate direction (a+h+Max(g,b), b+c+Max(d,a), d+e+Max(c,f), g+f+Max(h,e)). Stage 4 (Orange) finds the maximum (or derivative) of these equations to find the final direction of the possible edge. Stage 3 (green) will calculate the sum of all 8 pixels and stage 5 will compute the derivatives and write to o_edge and if needed o_dir. To determine if an edge exists, we apply the equation 8a-3b where a represents the output of the sum of the maximum direction and b represents the sum of all the pixels. To multiply by 8 and 3, we do a wired bit shift (accessible instantaneously) by 3 and 1 respectively. This is beneficial in terms of optimization as we avoid a multiplication operation and do not have to wait an extra clock cycle to access the results.

Performance Results vs. Projections

Following the redesign of our dataflow diagram, we decided to go for a conservative estimate of 200 cells and a clock speed of 200MHz. We wanted to ensure that we meet our initial goal of 200 MHz for clock speed, and so were prepared to have a higher area as a trade-off. Taking into account the overall implementation, we predicted that the number of cells could be 50% higher for a total of 300. After implementing several optimizations and readjustments to the logic, the performance results were close to the projections. The clock speed was 200 MHz (5ns), area was 306 logical elements, latency was 9 clock cycles and raw optimality score was 653 (readjusted to 621 with latency penalty). Our throughput was 1/4, as recommended in the specifications. Our critical path exists in Stage 4 (Orange) where the maximum sum of all the directions is computed (4.77ns). This is expected because the sizes of the registers that are being compared are relatively large (13 bits in length). Due to time constraints and scheduling, we did not have time to optimize for this case.

To reduce the area down to our desired range we made a control table to minimize the number of registers in use. This approach helped us cut down the total number of registers from 340 to 306 which improved our optimality significantly. After this approach, we focused on cutting down the widths of the

registers to the lowest possible size. Initially we set the width of all the registers to 13 in order to account for the worst case scenario. Once we had functional code, we cut down the width of all the registers as much as possible. Hence, a register is only as wide as it needs to be.

Another optimization was to cut down on the number of adders and muxes by running them in parallel through different pipeline stages. This allowed the adders and muxes to be reused in the next clock cycle. In total, we only used 3 adders and 2 muxes throughout the edge detection program.

Unfortunately, some optimizations did not give us a boost in optimality as we expected. Based on critical path analysis, we tried to optimize how values in registers are compared. For large registers, we tried looping from the most significant bit to the least significant bit and comparing them. Based on our thought process, this would increase clock speed since not all bits in the registers are compared. However, this optimization yielded negative results. In some cases our optimality went down because the loop significantly increased our area due to the number of extra muxes required (we had muxes inside the for-loop). Another optimization we tried was to reduce the latency by combining the last two clock cycles in stage 5. Instead of subtracting and then comparing in the next cycle (to 383), we tried to compute and compare the result (8a-3b) to 383 within the mux condition/input. This actually reduced our optimality by 100 points. The main reason why is because all of that calculation in one clock cycle reduced the clock speed significantly.

Verification Plan and Test Cases

In order to verify our results, we made heavy use of the ModelSim software. Whenever we had an unexpected output, we would narrow down to the pixel where we had an issue (using diff_ted). Then, we would use the appropriate test.txt file to find the convolution table. Then we used the refmodel.xls file in the project folder to find the expected direction and derivative. Finally, we used ModelSim and verified that at every cycle, each register had the value we would expect it to have. Several times we discovered that there was an overflow issue with the registers. Generally, the issue was to find which register was overflowing. We found this approach to be very slow, but effective. Every time we used this approach, we were able to narrow down the exact problem without much frustration.

One interesting bug we ran into was that our image yielded perfect match in simulation, but had a 40% error yield on the Altera DE2 board. It was difficult to debug this because we could not use our usual approach of comparing values in ModelSim. After exhausting all other possibilities, we tried to simulate our results with bubbles set to 4 (initially it was 3). Using this we were able to exactly replicate our problem in simulation. This was a strange bug because generally if the code works for 3 bubbles, then it should be able to handle 4 as well. After consulting with the TAs and discussing among the group, we decided that we should re-design our dataflow diagram because we were not satisfied with our optimality score (it was at 350 at that time). We also improved our thin-line implementation to ensure it is optimal and can handle varying number of bubbles at any time.