ECE 327 Design Report

Prepared by Fu, Zuowei (z32fu), Zhang, Jinming (jm3zhang)
Zhao, Youdongchen (y396zhao)

Design Goals and Strategy

Our design goal for dataflow diagram is mainly focused on conducting the correct calculations while minimizing the resources used and maximizing the performance. This means that the major concern for the dataflow diagram at the first stage is that it needs to perform the Kirsch's edge detector's calculations correctly. With that in mind, we conducted our initial design for the dataflow diagram. Despite the fact that this dataflow diagram can provide the correct calculations for the Kirsch's algorithm, this design has serious performance issue and poor operator utilization rate. One of the drawback of our initial design is that it calculates eight directions separately which caused poor operator utilization rate. Another drawback is that since it utilized more operators than it actually needed, this design introduced poor performance.

To optimize the design, our next design goal is to minimize the number of operators and carefully pipeline the dataflow diagram in order to achieve a lower latency. First, we used the equations provided in the manual

$$5a - 3b = 8a - 3(a + b)$$
$$\max(a - c, b - c) = \max(a, b) - c$$
$$\max(a + b, b + c) = b + \max(a, c)$$

to increase parallelism. To achieve parallelism, we adopted the strategy of using multiplexers to select and group the inputs into pairs. At the same time, we pipelined the dataflow diagram in a way such that the number of clock cycles needed on the top paths equals to the clock cycles on the bottom path as well as that each stage takes approximately the same amount of clock cycles to finish. We achieved this by separating the calculations into four stages. First stage takes in convolution table pairs by pairs and compare inputs. Then second stage controls dataflow direction. Third stage adds all the inputs together and compare directions. Finally subtract and output edge and direction. A general idea of that is show below:

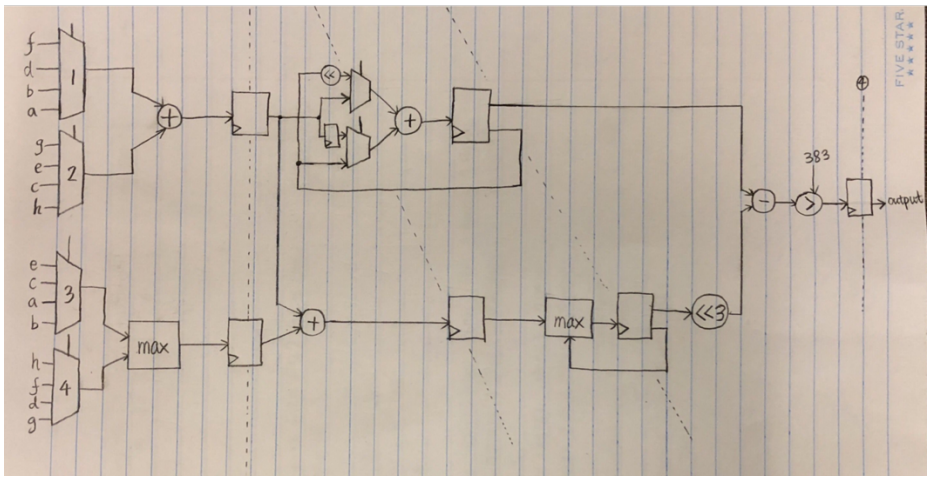

Figure 1. Optimized Data Flow Diagram

With this optimized design, the number of operators, registers, latency, clock periods and the number of stages on our design decreased notably. Our throughput is improved from the optimized design and the total area is improved and evident by the reduction in the total area (Optimized Total area is 266). In the end, our dataflow diagram has a latency value of 8 and a throughput value of 1/4, which appears to be the most optimal value for us. This means that we can receive and process inputs every 4 clock cycles since there is a bubble of 3 cycles.

Performance Results versus Projections

The optimality of our final dataflow diagram design is shown below:
We have a clock frequency of 334MHz and a total area of 266.

$$\text{Optimality} = \text{Functionality} \times \frac{Clock\ frequency}{total\ area} = 1000 \times \frac{334MHZ}{266} = 1256$$

In our actual implementation of the edge detector, the final optimality we got is 1245, with a latency value of 8. The latency value of the actual implementation is as we excepted from our latency calculation based on our dataflow diagram and this match in values is purely resulted from strictly following our designed dataflow diagram during the actual implementation. On the other hand, the optimality value we acquired from the actual implementation is different from our predicted value based on dataflow diagram. An explanation for this difference in the optimality is that during the actual implementations, there are a lot of conditional statements used in the actual implementation for controlling how the algorithm flows. These conditional statements will occupy lots of addition registers (apart from our design) which will increase our total area and hurt our optimality.

In order to achieve our optimality goal for this edge detector implementation, the first design change that we thought would optimized our design is to reduce the number of processes we used during the implementation. By combining stage two and three in our optimized dataflow diagram, we hope that the optimality will be lowered. Since stage two and three have the least amount of computations compared to other stages, it should be safe to combine them. However, the outcome is totally unexpected. By combining stage two and three, not only the optimality decreased, but also the latency increased from 8 to 9. This optimization is unsuccessful and the reason behind it that we can think off is when we combine those two stages, the operation in both stage will be combined. Although those 2 stages have the least amount of operations, when combining those operations, they end up in series which will prolong the time for those computation to complete. Especially for those add operations, they take relatively more time to complete than shifting operations. This will break the symmetries between the stages since this combined stage takes way more times to finish which will undermine the parallelism between all process.

The successful optimizations we used in the implementation is that we have minimized the number of flip-flops that are reset and use the minimum width of data that is necessary in the implementation. By minimizing the number of flip-flops that are reset, our performance is improved since usually there is no need to reset data path registers in the dataflow diagram. At the meantime, use the minimum width of data that is necessary will prevent us from overflow and shorten the computations.

Table 1. Summary of the Design

Area of the Major Component	Maximum Clock Speed	Critical Path	Latency	Throughput
228 cells	284 MHZ	3.51 ns	8	1/4

Verification Plan and Test Cases

The verification plans we have for the implement starts with refactoring the code related to the dataflow diagram. From our concern, the dataflow diagram is where those major computations and data processing happens. This means that there is a bigger likelihood where the dataflow diagram fails its functionally. The way we used for ensuring our dataflow diagram is working properly is to output those related signals on the simulated waveforms. Then by verifying those waveforms' high and low signals, we can make sure that our code is working correctly. If there exist errors in our implementations, it is easier to tell from the simulated waveforms than just looking at the output images.

A common mistake for the implementation is that although the matrix we used for storing and reading is a 256 by 256 matrix, the counter for the convolution table should between the range of 1 to 254 because the convolution table is a 3 by 3 matrix. This means that the actual computation should not start until the 2^{nd} column on the 2^{nd} row was accessed. For our verification plans, by refactoring the code related to the dataflow diagram, it is extremely important to make sure that our row and column counters are functioning correctly. If there is a problem with row or column counters, most likely, there will be pixels slanted on the image all over the places which might be an interesting bug.

After the verification of the row and column counters, we are focusing on the values in the convolution table. In order to get the correct output from our edge detector, we have to make sure the process of reading the values for the pixels from the memory and storing into the convolution table have to be correctly preformed. The easiest and most effective way to achieve this is by checking the values in the convolution table at the output waveforms from the simulations and compared those values with our manually calculated values. It is the most effective way for us to find out the error in our code directly with the tradeoff that this process is very time consuming.

When we finished checking the functionality of our dataflow diagram and refactor it, we moved on to validate the functionality of our state machine used for controlling the flow of the Kirsch's algorithm. Usually there will be a problem with the number of bits we used to initialize the signal. If there are more bits we used then we need, there will be a chance of causing overflow. If there are less bits we used then we need, we will lose data because of that. Known the right number of bits we need is our main emphasis on the verification of our state machine and dataflow diagram. The method we used for the bit verification is to find out how many and what operations that signal had undergo. With the information about the operations of that signal, we can manually calculation the maximum number of bits it needs to finish its computation.

For test cases, the majority cases we used was based on the provided test benches and images since most of the testing and debugging processes we have done is by manually testing on our code.