Danmarks Tekniske Universitet

Computer Systems 02132

Assignment 3

Bjarke Søderhamn Petersen
s235240

William Pii Jæger
s235245

Hugo Bede Johnson
s242071

February 9, 2025

Work distribution

We split the work evenly between group members and everyone had a input in all parts of our project both for the code and the report.

1 Design

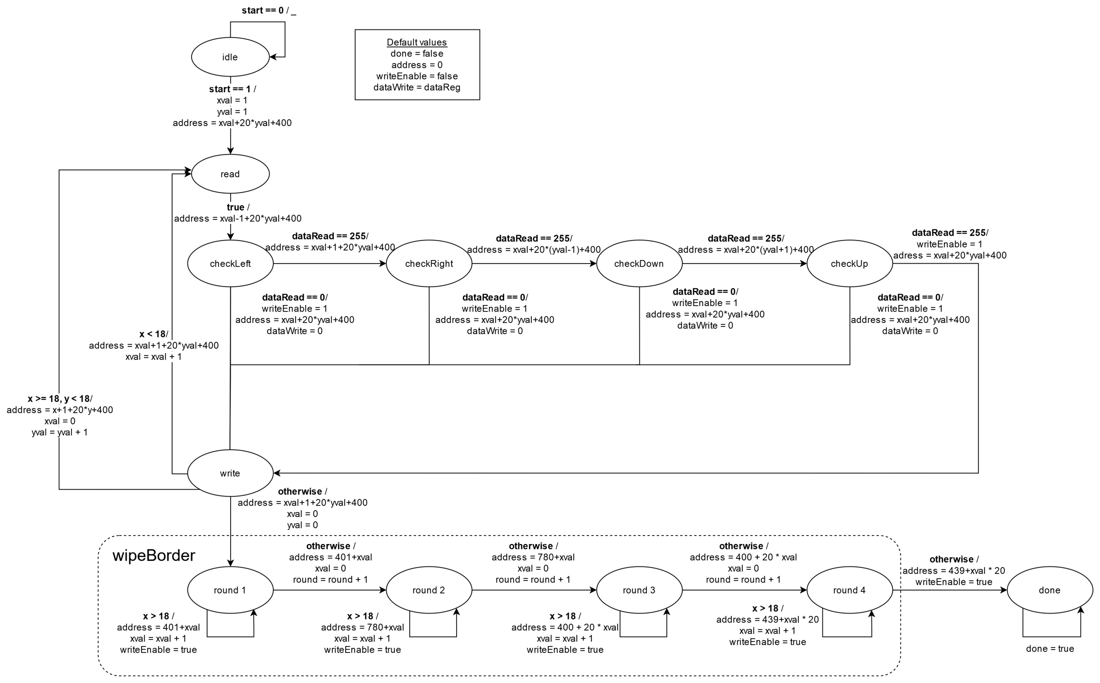

Figure 1: Diagram of FSMD

The formula below is used to calculate the 4 offsets left, right, down and up.

$$x + \text{x\_offset} + (20 * (y + \text{y\_offset}))$$

Note that x_offset and y_offset are variables in the range $\{-1, 0, 1\}$ which can be adjust to target the neighboring pixels. For example, to check the right pixel, x_offset = 1, y_offset = 0.

In order to minimise the amount of states that we have, we made the decision to not check the pixel to be eroded itself, and instead check its neighbours first. If a black pixel is surrounded by white pixels, it will pass all four direction checks. However we write the value of the center pixel to the new image. In other words, a black pixel surrounded by white pixels will still come out as black in the new image. Because we are working with cells, our assumption is that white pixels are always clumped together. The scenario of multiple occurrences of black pixels surrounded by white pixels is therefore less likely, and the amount of checks we perform is lowered since we don't need an initial state we load the value of the address but instead skip directly to checking the surroundings.

The state wipeBorder has logic which iterates the round register. The rounds are not states themselves. They only look like states in figure 1 to make it more understandable what it is we are doing. The address calculation pull from the vectors borderOffsets and borderRowlens. See section 2.2 for further explanation of the implementation of this design choice.

2 Implementation

The Chisel implementation was written to mirror the FSMD diagram as closely as possible. Each state was set up as an Enum to give the states names corresponding to the diagram.

The dataReg register stores data from the last cycle due to a latency of one cycle needed to read the pixel value from data memory. Each state sets up the address to read from for the next state. For example the read state sets the address to be skewed to the left side of the center pixel and set the state to checkLeft. In the next cycle, checkLeft is first setting up the address for the next cycle. Then it checks the variable in the input dataRead. If it is 0 it will override the address to be the center pixel address and set the state to write.

In this way each subsequent state is preparing the address for the next state.

2.1 Function usage

Each state checkLeft, checkRight, checkDown and checkUp initially prepares the next neighbor to check. However the logic for seeing if the neighboring pixels is 0 or 255, and what happens depending on the branch, is the same. That is why we have extracted that logic to the function checkDirectionSetState. The four states then provide an Enum (represented as aUInt), which is the state they want to set in case the neighboring pixel that is checked is 255. This makes the code more maintainable as we know for a fact that all four states perform the same logic and still work - identical to how we designed our FSMD.

2.2 Optimization

To reduce the amount of cycles spent, the act of iterating both x and y are performed within the writing cycle. That counts for both the write state and the wipeBorder state.

To wipe the border, the outline of the cell image, we have stored the variables necessary to calculate where in the array to wipe the cells. The border wiping was split into four rounds: up, bottom, left and right. The default values needed for each state are stored in borderOffsets and borderRowlens, which are lists of four registers and are initiated such that borderOffsets$_n$ represent the correct border offset for round n (and the same for borderRowlens). To keep track of the round we are on we store it in the register `round`.

When calculating the position to write 0 to when wiping the border, we utilize the round register to access that specific round's border-offset and border-row length. It is calculated with:

$$\text{borderOffsets}_n + (x * \text{boderRowlens}_n)$$

where n is the round we are on and borderOffsets and borderRowlens vectors.

By performing this optimization we avoid having to read or check if we are at a border. The avoidance of having to read reduces the amount of cycles. If we did not perform this optimization we would have had to check if we were on the border for each cell we iterated through for the entire image. This way we theoretically save on performance. This is also why we initiate both x and y at the value 1 and iterate them until they are below 18 when doing the erosion step.

3 Test and Evaluation

The FSMD diagram was first tested manually with a pencil and paper on a much smaller (4x4) image. The test image, as well as the expected output, is shown in Figure 2. In the eroded image only two pixels remain white, which can be seen in lines 7 and 23 of Table 1.

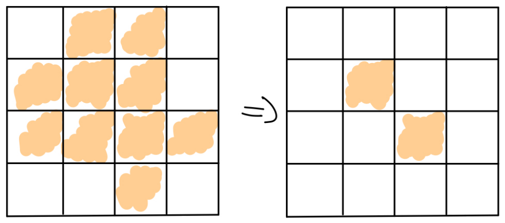

Figure 2: Test image used in the pencil and paper calculations

It was next implemented in Chisel and tested on the cells image. Lastly, we wanted to make sure that our program was actually writing black to the borders of the image (instead of leaving them unwritten). To test this we modified our accelerator to write a white border instead, and changed it back to black after ensuring this was the case.

Step	Xval	Yval	Addr	WriteEnable	DataWrite	DataRead
idle	0	0	0	0	0	0
read	1	1	5	0	0	0
checkLeft	1	1	4	0	0	255
checkRight	1	1	6	0	0	255
up	1	1	1	0	0	255
down	1	1	9	0	0	255
write	1	1	5	1	255	255
read	2	1	6	0	0	255
checkLeft	2	1	5	0	0	255
checkRight	2	1	7	0	0	0
write	2	1	6	1	0	0
read	1	2	9	0	0	255
checkLeft	1	2	8	0	0	255
checkRight	1	2	10	0	0	255
checkUp	1	2	5	0	0	255
checkDown	1	2	13	0	0	0
write	1	2	9	1	0	0
read	2	2	10	0	0	255
checkLeft	2	2	9	0	0	255
checkRight	2	2	11	0	0	255
checkUp	2	2	6	0	0	255
checkDown	2	2	14	0	0	255
write	2	2	10	1	255	255
round 1	0	0	0	1	0	0
round 1	1	0	1	1	0	1
round 1	2	0	2	1	0	1
round 1	3	0	3	1	0	0
round 2	0	0	12	1	0	0
round 2	1	0	13	1	0	0
round 2	2	0	14	1	0	1
round 2	3	0	15	1	0	0
round 3	0	0	0	1	0	0
round 3	1	0	4	1	0	1
round 3	2	0	8	1	0	1
round 3	3	0	12	1	0	0
round 4	0	0	3	1	0	0
round 4	1	0	7	1	0	0
round 4	2	0	11	1	0	1
round 4	3	0	15	1	0	0

Table 1: Step-by-step state transitions

3.1 Evaluations

We can calculate our speedup based on the programs we wrote in assignment 1 and 2. We ran assignment 1 on a 2.6 GHz computer. The execution time was 1.142 milliseconds. Given that:

$$Cycles = ExecutionTime * ClockFrequency$$

Then assignment 1 took:

$$1.142 * 10^{-3} s * 2.6 * 10^{9} Hz = 2969200 cycles$$

We ran assignment 1 on a picture of the scale 950 by 950. We downsize it to compare to our accelerator:

$$(2969200 * 20 * 20)/(950 * 950) = 1315.989 cycles$$

Assignment 1 therefore took approximately 1316 cycles.

From this we can now calculate the speed up between assignment 1 and 3 and assignment 2 and 3. For these calculations we assume that all 3 programs run on the same clock frequency. The speed up from assignment 1 is therefore:

$$1316/1161 = 1.13 \text{ times faster}$$

And the speed up from assignment 2 is:

$$7159/1161 = 6.17 \text{ times faster.}$$

State	Adders	Subtractors	Multipliers	Comparators
idle	1	0	1	0
read	1	1	1	0
checkLeft	3	0	2	1
checkRight	2	1	2	1
checkDown	3	0	2	1
checkUp	2	0	2	1
write	4	0	2	2
wipeBorder	3	0	1	2
done	0	0	0	0

Table 2: Maximum functional Units Used in Each State

In table 2 you can see how many of each functional units are used in the different states. We assume that if a adder is used in one state it can be reused in another state. The limiting factor is therefore that some states use multiple of the same functional unit in one clock cycle and it is therefore not possible to reuse them in the same state. We can see that we at least need 4 adders, 1 subtractor, 2 multipliers and 2 comparators.

We could have utilized signed integers to perform our calculations. If we did this, we could have used the general formula mentioned in section 1. However our current implementation uses Unsigned integers which does not allow for doing addition of negative integers and an unsigned integer cannot be negative. Given we had more time we could refactor this to increase the amount of states we could share.

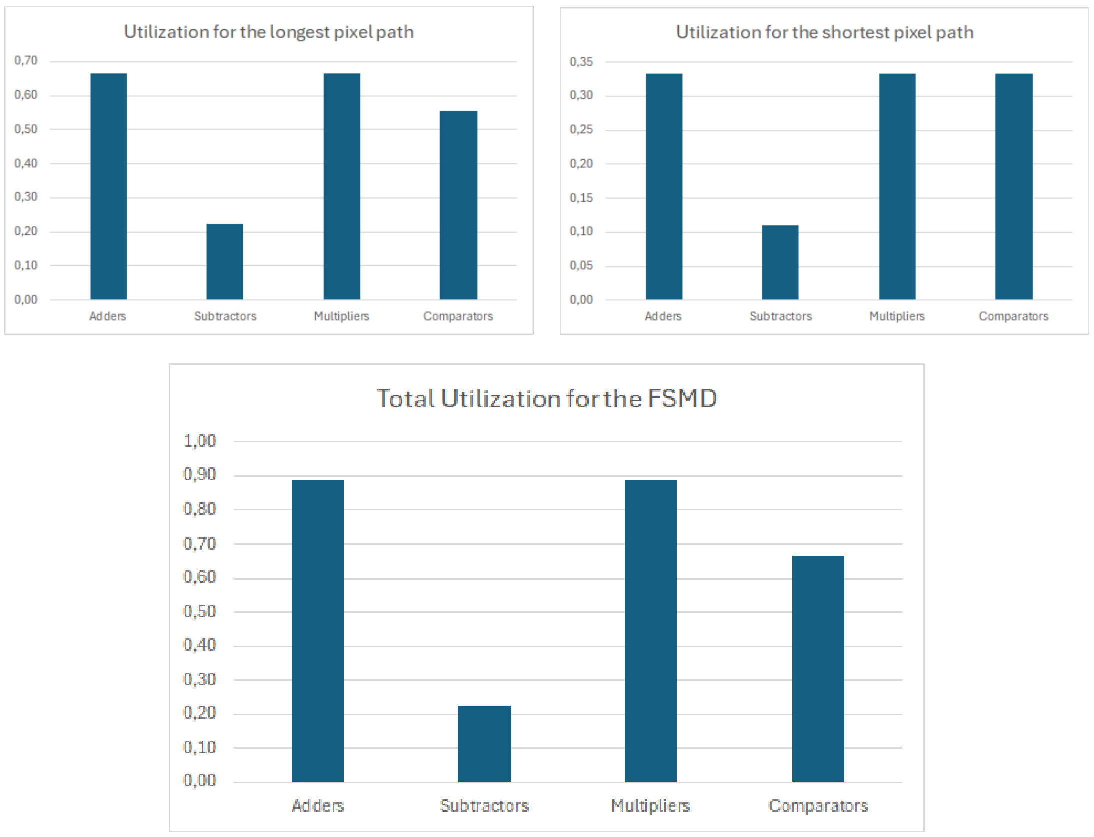

Figure 3: Utilization

In figure 3 you can see three histograms on utilization for different paths. The first plot shows the utilization for the longest cycle a pixel can take in our FSMD. The path taken is as follows: $read \rightarrow checkLeft \rightarrow checkRight \rightarrow checkDown \rightarrow checkUp \rightarrow write \rightarrow read$. The second path shows the shortest cycle for a pixel which is: $read \rightarrow checkLeft \rightarrow write \rightarrow read$. And the final plot shows the utilization for the total FSMD. The total states used for the calculations is not the total amount of states for the cycles but instead for the entire FSMD.

3.2 Size

Assuming the functional units in the accelerator can be shared, we can deduce from Table 2 that a maximum of 4 32-bit-adders, one 32-bit-subtractor, two 32-bit-multipliers and two 32-bit-comparators are needed in the CPU. Furthermore, a total of 13 32-bit-registers are used. 8 of these (borderOffsets and borderRowLens) are static registers, and the other five keep track of the accelerator's state (i.e. stateReg, xval, yval, round, and dataReg).