02132 Assignment 3 report

Implementation of an FSMD-based hardware accelerator for the image erosion in Chisel

Group: 22

Niclas Juul Schæffer **s224744**
Rasmus Kronborg Finnemann Wiuff **s163977**
github.com/rwiuff/02132Assignment3

December 1st

1 Work distribution

Table 1 shows the work distribution for this project.

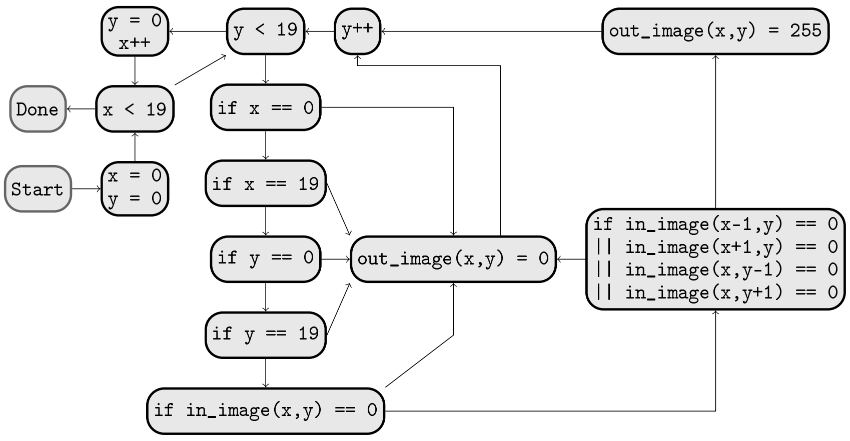
Table 1: *Work distribution on the project*

Name	Development tasks	Report tasks
Niclas Juul Schæffer	Implementation	Implementation, Evaluation
Rasmus Wiuff	Design	Design, Test

2 Design

First a CFG was generated from the pseudocode in the assignment material, yielding the graph in Fig. 1.

Figure 1: *Control flow graph of erosion algorithm*



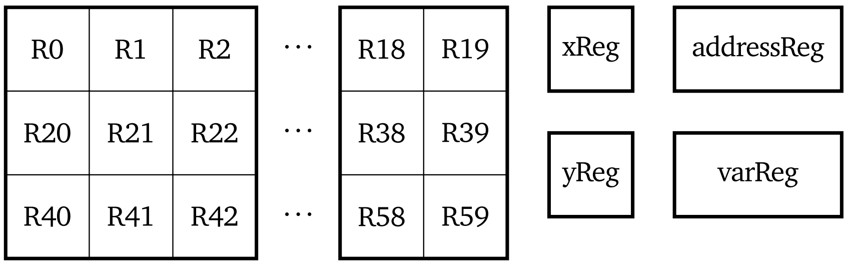
The diagram was then optimised:

1. All the logical comparisons regarding the inner pixel and neighbours can be merged into a single state.
2. Incrementors and loop conditions can be merged into a single state.
3. Border check can be merged into the loop condition state.
4. Write operations can be merged into a single state.

Next step focused on registers. The assignment allows O(x_size) registers, and thus registers for three image rows are chosen. Once the rows are stored in the registers, a row can be processed with neighbouring pixels without reading from memory, limiting memory access. Furthermore, registers for `x`, `y`, the memory address and a variable was decided upon. The variable register frees up tampering with `x` or `y` throughout the erosion (other than incrementing). The register layout is shown in Fig. 2.

This gives two development tasks:

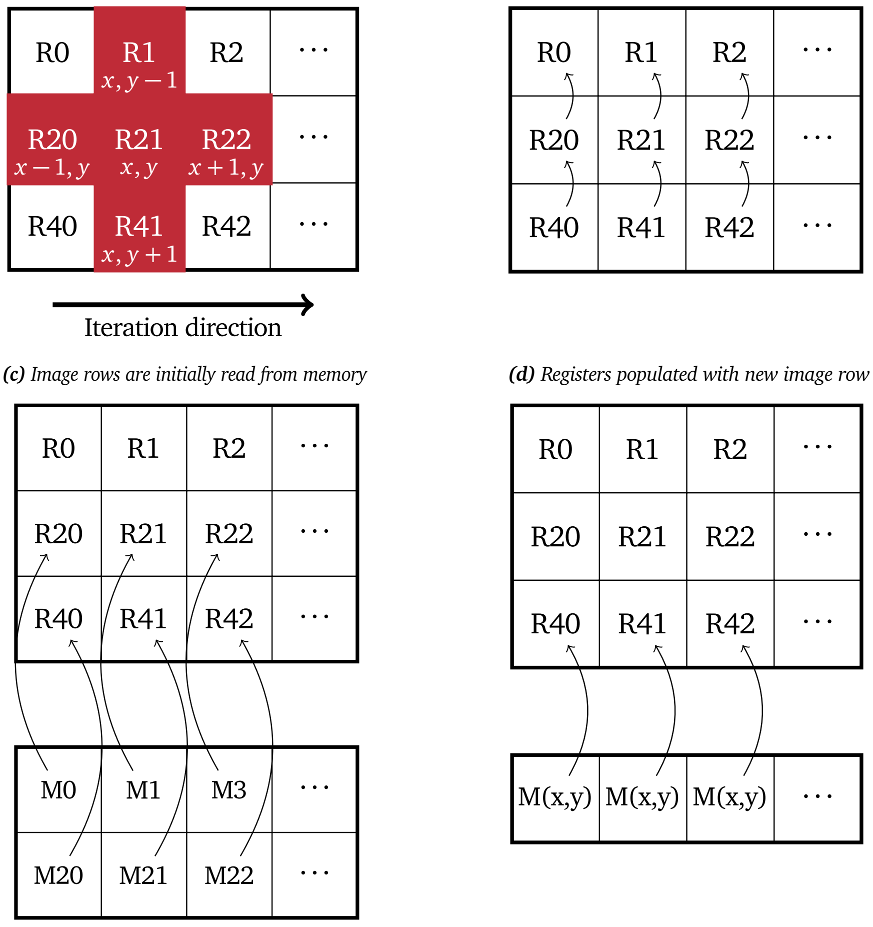
Figure 2: *The registers in the accelerator*



1. Registers need to be initialised.
2. Registers need to be updated on changing rows.

Each task is solved by a separate state. An initialisation state will store pixels in the registers R20 to R59 as shown in Fig. 3c. On every row only the registers are used as shown in Fig. 3a. Once the registers needs updating the register at position x is updated with the data in $x + 20$ for the first 40 registers, as shown in Fig. 3b. Once done the last 20 registers are updated with the next row from memory as shown in Fig. 3d.

Figure 3: *Data register handling*



(a) *A row is analysed using the registers only*

(b) *Image rows moves up in registers only*

(c) *Image rows are initially read from memory*

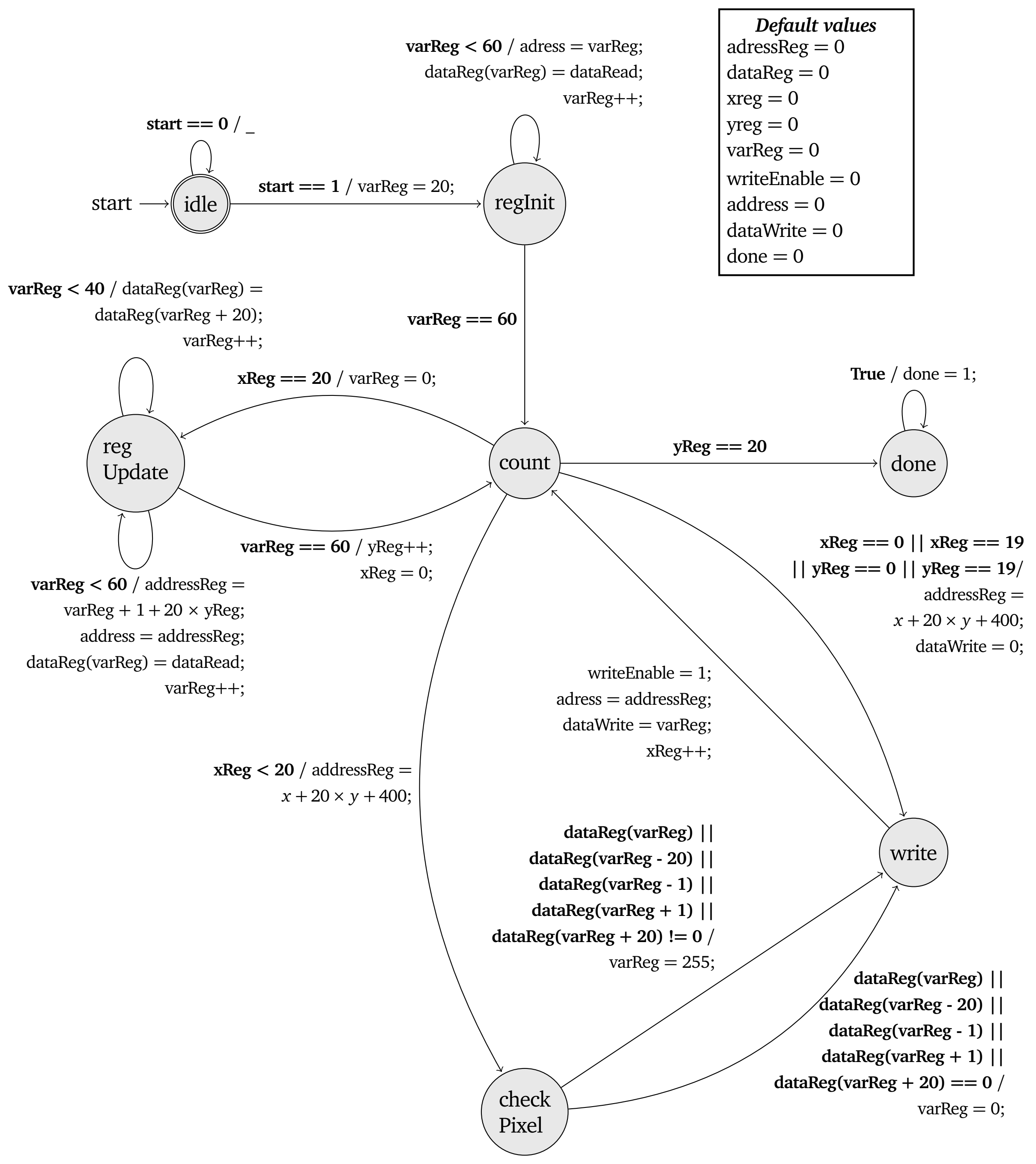
(d) *Registers populated with new image row*

To recapitulate: The following states are needed

- A state to initialise the registers with image rows.
- A state to update the image rows when a row have been analysed and eroded.
- A state to check if a pixel should be eroded.
- A state to write the output image to memory.
- A state to control incrementors.

The resulting FSMD is shown on the state diagram in Fig. 4.

Figure 4: *State diagram of the erosion accelerator FSMD*


Default values
adressReg = 0
dataReg = 0
xreg = 0
yreg = 0
varReg = 0
writeEnable = 0
address = 0
dataWrite = 0
done = 0
start
idle
start == 0 / _
start == 1 / varReg = 20;
regInit
varReg < 60 / adress = varReg; dataReg(varReg) = dataRead; varReg++;
varReg == 60
count
reg Update
varReg < 40 / dataReg(varReg) = dataReg(varReg + 20); varReg++;
xReg == 20 / varReg = 0;
varReg == 60 / yReg++; xReg = 0;
varReg < 60 / addressReg = varReg + 1 + 20 × yReg; address = addressReg; dataReg(varReg) = dataRead; varReg++;
yReg == 20
done
True / done = 1;
xReg == 0 || xReg == 19 || yReg == 0 || yReg == 19/ addressReg = x + 20 × y + 400; dataWrite = 0;
writeEnable = 1; adress = addressReg; dataWrite = varReg; xReg++;
write
xReg < 20 / addressReg = x + 20 × y + 400;
check Pixel
dataReg(varReg) || dataReg(varReg - 20) || dataReg(varReg - 1) || dataReg(varReg + 1) || dataReg(varReg + 20) != 0 / varReg = 255;
dataReg(varReg) || dataReg(varReg - 20) || dataReg(varReg - 1) || dataReg(varReg + 1) || dataReg(varReg + 20) == 0 / varReg = 0;

3 Implementation

The Chisel implementation is quite straight forward. Support registers are initialised, and a state register is used to enumerate states. A switch statement is run, and setting a new state in the `stateReg` means the FSMD changes states. A thing to notice is the usage of a register vector to store the image row data, as shown in Listing 1.

Listing 1: *Image data register defined in Chisel*

```
val dataReg = Reg(Vec(60, UInt(32.W)))
```

Otherwise, the FSMD is laid out in Chisel as described in the diagram in Fig. 4.

4 Test And Evaluation

4.1. Testing

The FSMD was first tested using white box testing for individual states. Afterwards a 4 by 4 picture was used to test the FSMD in hand. No problems were identified, and the Chisel implementation was carried out. The next tests were purely based on looking into waveforms and prodding the system. The first test (simulation) showed a shift on the x-axis of all output pixels. By investigating the waveforms we realised that the clock cycles are skewed as opposed to the last assignment. This test meant the FSMD and implementation needed to be altered to accommodate the memory chip, by adding a 1 to the read address. This can be seen in Listing 2.

Listing 2: *Alteration to fix clock cycle latency*

```
addressReg := varReg + 1.U + 20.U * (yReg)
```

4.2. Execution time and speed-up

In assignment 2 the clock cycles for the cell image of 20×20 pixels was 6815. The FSMD based hardware accelerator is running at 2406 cycles for the same picture. So we can clearly see that we have got a significant boost in performance. Eq. (4.1) shows the relative increase.

$$\frac{6815}{2406} \times 100 = 283.25\% \tag{4.1}$$

From this we can see that the program is 283.25 % faster, that way it is a significant performance boost. The increase in performance is calculated in Eq. (4.2).

$$1 - \frac{2406}{6815} \times 100 = 64.7\% \tag{4.2}$$

This means the accelerator gives an increased performance of 64.47 %.

4.3. Utilisation of resources

The number of needed functional units are derived from the state diagram in Fig. 4. Table 2 depicts units used by various states.

Table 2: *Number of functional units used in states*

State	$=$	$<$	$\|\|$	$+$	$-$	$\times$
idle	1					
done						
regInit	1	1		1		
regUpdate	1	2		3		1
count	6	1	3	2		1
checkPixel	4		4	2	2	
write				1		

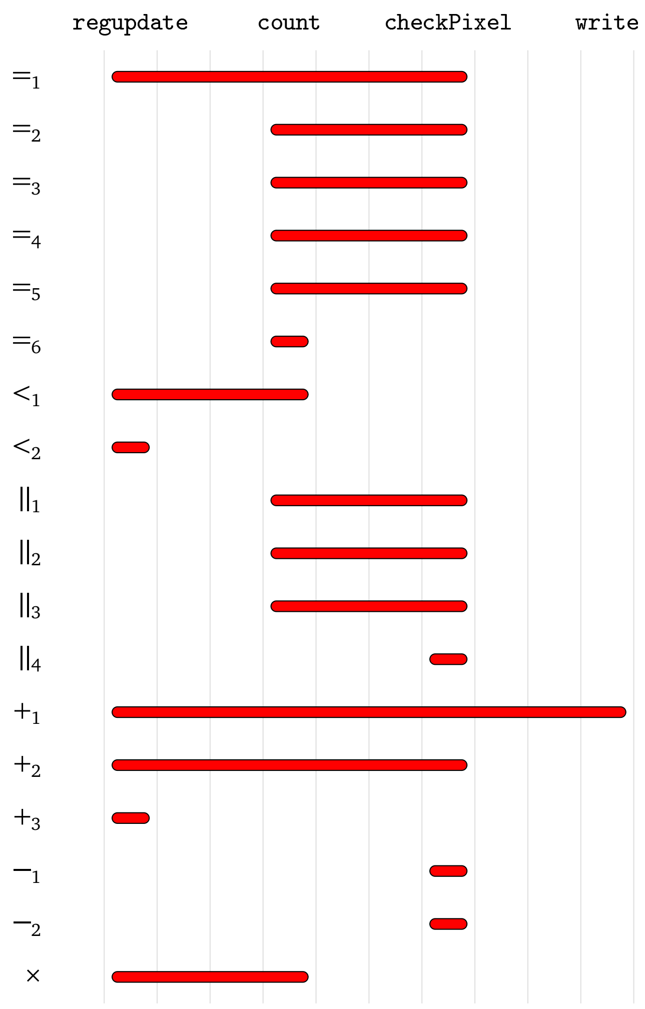
The highest number of units are summarised in Table 3.

Table 3: *Total number of functional units in the FSMD*

$=$	$<$	$\|\|$	$+$	$-$	$\times$
6	2	4	3	2	1

The longest path begins when the pixel registers are updated and during analysis throughout the row. The path is defined as `regUpdate` $\rightarrow$ `count` $\rightarrow$ `checkPixel` $\rightarrow$ `write` $\rightarrow$ `count`. The utilised resources in this path are shown in Fig. 5.

Figure 5: *Schedule of functional units in the longest FSMD path*



Using the utilisation formula for the functional units,

$$\frac{\text{\# of states when its used}}{\text{\# of states when it is used} + \text{\# of states when it is not used}} \tag{4.3}$$

the utilisation of the functional units is determined. For the path in Fig. 5 the utilisation is given in Table 4.

Table 4: *Utilisation of functional units in the longest FSMD path*

Functional unit	$=_1$	$=_2$	$=_3$	$=_4$	$=_5$	$=_6$	$<_1$	$<_2$	$\|\|_1$
Utilisation	0.75	0.5	0.5	0.5	0.5	0.25	0.5	0.25	0.5
Functional unit	$+_1$	$+_2$	$+_3$	$-_1$	$-_2$	$\times$	$\|\|_2$	$\|\|_3$	$\|\|_4$
Utilisation	1	0.75	0.25	0.25	0.25	0.5	0.5	0.5	0.25

Most functional units are fairly shared between states, however few are only needed for the `checkPixel` state, as this state makes a bit more logical comparisons and arithemics. This state is the only one using subtraction. The conclusion is that some functional units can not be removed and as such some units have limited utilisation.

4.4. Hardware size

All operations need to deal with numbers no larger than 255, which means an 8 bit size is suitable. However, the memory addresses needs to go to 799, meaning the memory address register needs to be of size 16. There are two options. Either the functional units are of size 16 or a padding circuit is introduced before each memory address is computed. We chose the first option. A circuit overview is seen in Table 5.

Table 5: *Overview of circuits in the accelerator*

Circuit	Number	Bit size
Register	64	16
Comparator (=)	6	16
Comparator (<)	2	16
Logic OR	4	16
Adder	3	16
Subtractor	2	16
Multiplicator	1	16