

02132 ASSIGNMENT 2 REPORT

HARDWARE IMPLEMENTATION IN CHISEL OF A SMALL CPU RUNNING THE IMAGE EROSION

Group: 22

Mikkel Arn Andersen s224187

Niclas Juul Schæffer s224744

Rasmus Kronborg Finnemann Wiuff s163977

github.com/rwiuff/02132Assignment2 

November 12th

1 WORK DISTRIBUTION

Table 1 shows the work distribution for this project.

Table 1: Work distribution on the project

Name	Development tasks	Report tasks
Mikkel Arn Andersen	Assembler: Memory test, arithmetic test, jump test. Implementation evaluation.	Section 4
Niclas Juul Schæffer	CPU Block design, ALU implementation, CPU Top implementation, Register File implementation, Chisel testers.	Section 3
Rasmus Wiuff	ISA, assembler compilation, machine instruction encoding, CPU Block design, Program Counter implementation, Control Unit implementation.	Sections 2.1 to 2.3

2 DESIGN

The project requirements were open to broad interpretation, so first step of the design was to put on restrictions. The following was stipulated:

1. All blocks in the assignment would be laid out as described in the material.
2. The architecture would be 32-bit.
3. The instruction set architecture should be as simple as possible, but still versatile.

The next step was designing the instruction set architecture and the encoding scheme.

2.1. ISA AND ENCODING SCHEME

The instruction set needs to contain basic arithmetic operations as well as logical ones for computations. Instructions are also needed for creating variables in the registers as well as reading and writing data between registers and memory. A branch (or in this project jump) instruction is needed to navigate between instruction lines as well as at least one conditional branch instruction. Lastly the program needs to terminate. The chosen instructions are listed in Table 2. A total of 14 instructions are settled upon. This requires $\lceil \log_2 14 \rceil = 4$ bits for the opcode part of the instruction. As defined in the assignment, the memory block has an input of 16 bits. This, with the 4 opcode bits, leaves 12 bits for register addresses. Consequently, this limits the chip to 16 registers. The instruction scheme is shown in Fig. 1. Table 3 shows how each instruction can be encoded into 32 bits. Notice how, when writing or reading to/from memory, Rb points to the register with the memory address. This makes the wiring in the CPU block easier, as a wire from the b output of the register file can be connected directly to the address input port in the data memory.

Table 2: Instruction set architecture used in the assignment

Instruction	Syntax	Meaning
Arithmetic instructions		
Addition	ADD Rx, Ry, Rz;	Rx = Ry + Rz
Subtraction	SUB Rx, Ry, Rz;	Rx = Ry - Rz
Immediate addition	ADDI Rx, Ry, z;	Rx = Ry + z
Immediate subtraction	SUBI Rx, Ry, z;	Rx = Ry - z
Immediate multiplication	MULT Rx, Ry, z;	Rx = Ry · z
Increment	INC, Rx	Rx = Rx + 1
Logic instructions		
Bitwise OR	OR Rx, Ry, Rz;	Rx = Ry Rz
Bitwise AND	AND Rx, Ry, Rz;	Rx = Ry & Rz
Memory instructions		
Load immediate	LOADI Rx, y;	Rx = y
Load data	LOAD Rx, Ry;	Rx = memory(Ry)
Store data	STORE Rx, Ry;	memory(Ry) = Rx
Control and flow instructions		
Jump	JMP x	GOTO INST x
Jump if equal	JEQ Rx, Ry, z;	if (Rx > Ry) GOTO INST z
END	END;	Terminate

Figure 1: Instruction layout. Ra and Rb are operands, Rd is the destination register. Remaining bits are used for either memory address or immediate value.

OPCODE (4 bits) 31 ... 28	Rd (4 bits) 27 ... 24	Ra (4 bits) 23 ... 20	Rb (4 bits) 19 ... 16	Value/address (16 bit) 15 ... 0
------------------------------	--------------------------	--------------------------	--------------------------	------------------------------------

Table 3: Encoding instructions under the instruction set and encoding scheme

Instruction	OPCODE	Rd	Ra	Rb	Value/address
ADD R1, R2, R3;	0001	0001	0010	0011	0000 0000 0000 0000
SUB R1, R2, R3;	0010	0001	0010	0011	0000 0000 0000 0000
ADDI R1, R2, 1;	0011	0001	0010	0000	0000 0000 0000 0001
SUBI R1, R2, 1;	0100	0001	0010	0000	0000 0000 0000 0001
MULT R1, R2, 1;	0101	0001	0010	0000	0000 0000 0000 0001
OR R1, R2, R3;	0110	0001	0010	0011	0000 0000 0000 0000
AND R1, R2, R3;	0111	0001	0010	0011	0000 0000 0000 0000
LOADI R1, 1;	1000	0001	0000	0000	0000 0000 0000 0001
LOAD R1, R2;	1001	0001	0000	0010	0000 0000 0000 0000
STORE R1, R2;	1010	0000	0001	0010	0000 0000 0000 0000
INC R1;	1011	0001	0001	0000	0000 0000 0000 0000
JMP 1;	1100	0000	0000	0000	0000 0000 0000 0001
JEQ R1, R2, 1;	1101	0000	0001	0010	0000 0000 0000 0001
END;	1111	0000	0000	0000	0000 0000 0000 0000

2.2. COMPILE AND ENCODE

2.2.1. Compiled to assembler In this project the provided algorithm is used. It is compiled by hand to assembler using the defined instruction set in Table 2. The program does the following:

- Lines 0-4 set initial values.
- Lines 5-6 is the for loop conditional checks.
- Lines 7-9 contains the address for the pixel at $x + y \times 20$.

Listing 1: The program compiled to assembly

```
# Initial values
00. LOADI R0, 0;      # x counter
01. LOADI R1, 0;      # y counter
02. LOADI R2, 19;     # Pixel limit
03. LOADI R3, 0;      # Zero value
04. LOADI R4, 255;    # 255 value

# For loop conditions
05. JEQ R0, R2, 47;    # Check x, GOTO END
06. JEQ R1, R2, 44;    # Check y, GOTO INC X

# Output image address
07. MULT R5, R1, 20;   # y * 20
08. ADD R6, R0, R5;    # x + y * 20
09. ADDI R5, R6, 400;  # Out image address

# Process border pixel
10. JEQ R0, R3, 40;    # If x or y = 0
11. JEQ R1, R3, 40;    #
12. JEQ R0, R4, 40;    # If x or y = 19
13. JEQ R1, R4, 40;    # GOTO erosion

# Process inner pixel
14. MULT R6, R1, 20;   # y * 20
15. ADD R7, R0, R6;    # x + y * 20
16. LOAD R8, R7;       # Get input pixel
17. JEQ R8, R3, 40;    # If 0, GOTO erosion

# Process outer pixels
18. SUBI R12, R0, 1;   # x - 1
19. ADDI R13, R0, 1;   # x + 1
20. SUBI R14, R1, 1;   # y - 1
21. ADDI R15, R1, 1;   # y + 1
22. MULT R6, R1, 20;   # y * 20
23. ADD R7, R6, R12;   # (x - 1) + y * 20

24. LOAD R8, R7;       # Save pixel in R8
25. MULT R6, R1, 20;   # y * 20
26. ADD R7, R6, R13;   # (x + 1) + y * 20
27. LOAD R9, R7;       # Save pixel in R9
28. AND R10, R8, R9;   # AND R8 and R9, save to R10
29. MULT R6, R14, 20;  # (y - 1) * 20
30. ADD R7, R6, R0;    # x + (y - 1) * 20
31. LOAD R8, R7;       # Save pixel in R8
32. AND R9, R8, R10;   # AND R8 and R10, save to R9
33. MULT R6, R15, 20;  # (y + 1) * 20
34. ADD R7, R6, R0;    # x + (y + 1) * 20
35. LOAD R10, R7;      # Save pixel in R10
36. AND R8, R9, R10;   # AND R9 and R10, save to R8
37. JEQ R8, R3, 40;    # If = 0 GOTO Erosion

# No erosion
38. STORE R4, R5;      # Set pixel to 255
39. JMP 42;            # GOTO increment y

# Erosion
40. STORE R3, R5;      # Set Pixel to zero
41. JMP 42;            # GOTO increment y

# Increment y
42. INC R1;            # Increment y
43. JMP 6;             # Continue nested loop

# Increment x
44. INC R0;            # Increment x
45. LOADI R1, 0;       # Zerorise y
46. JMP 5;             # Continue main loop

# Terminate program
47. END;              # Terminate program
```

- Lines 10-13 checks for image border case.
- Lines 14-17 checks the inner pixel.
- Lines 18-37 calculates the addresses for pixels at $x - 1, x + 1, y - 1, y + 1$ and these pixels are loaded and compared with logic AND. The result is the compared to 0.
- Lines 38-39 is the branch if an output pixel is set to 255.
- Lines 40-41 is the branch if an output pixel is set to 0.
- Lines 42-43 increments y and jumps to the nested loop conditional check.
- Lines 44-46 increments x , sets y to zero and jumps to the main loop conditional check.
- Line 47 terminates the program.

The registers are utilised in the following way:

- R0 and R1 contains the current x and y counter.
- R2 contains the picture width in pixels.
- R3 and R4 hold the values 0 and 255 for conditional comparison and for writing either black or white pixels to the output image addresses.
- R5 holds the output image address for the current pixel.
- R6 to R15 are used for storage during calculations.

Listing 2: The encoded program

OP	Rd	Ra	Rb	Value/address	OP	Rd	Ra	Rb	Value/address
00.	1000	0000	0000	0000000000000000	24.	1001	1000	0000	0111 0000000000000000
01.	1000	0001	0000	0000000000000000	25.	0101	0110	0001	0000 00000000010100
02.	1000	0010	0000	0000000000001001	26.	0001	0111	0110	1101 0000000000000000
03.	1000	0011	0000	0000000000000000	27.	1001	1001	0000	0111 0000000000000000
04.	1000	0100	0000	0000000011111111	28.	0111	1010	1000	1001 0000000000000000
05.	1101	0000	0000	0010 00000000010111	29.	0101	0110	1110	0000 00000000010100
06.	1101	0000	0001	0010 00000000010100	30.	0001	0111	0110	0000 0000000000000000
07.	0101	0101	0001	0000 00000000010100	31.	1001	1000	0000	0111 0000000000000000
08.	0001	0110	0000	0101 0000000000000000	32.	0111	1001	1000	1010 0000000000000000
09.	0011	0101	0110	0000 000110010000	33.	0101	0110	1111	0000 00000000010100
10.	1101	0000	0000	0011 00000000010100	34.	0001	0111	0110	0000 0000000000000000
11.	1101	0000	0001	0011 00000000010100	35.	1001	1010	0000	0111 0000000000000000
12.	1101	0000	0000	0100 00000000010100	36.	0111	1000	1001	1010 0000000000000000
13.	1101	0000	0001	0100 00000000010100	37.	1101	0000	1000	0011 00000000010100
14.	0101	0110	0001	0000 00000000010100	38.	1010	0000	0100	0101 0000000000000000
15.	0001	0111	0000	0110 0000000000000000	39.	1100	0000	0000	0000 0000000101010
16.	1001	1000	0000	0111 0000000000000000	40.	1010	0000	0011	0101 0000000000000000
17.	1101	0000	1000	0011 00000000010100	41.	1100	0000	0000	0000 0000000101010
18.	0100	1100	0000	0000 00000000000001	42.	1011	0001	0001	0000 0000000000000000
19.	0011	1101	0000	0000 00000000000001	43.	1100	0000	0000	0000 00000000000110
20.	0100	1110	0001	0000 00000000000001	44.	1011	0000	0000	0000 0000000000000000
21.	0011	1111	0001	0000 00000000000001	45.	1000	0001	0000	0000 0000000000000000
22.	0101	0110	0001	0000 00000000010100	46.	1100	0000	0000	0000 00000000000101
23.	0001	0111	0110	1100 0000000000000000	47.	1111	0000	0000	0000 0000000000000000

2.2.2. Encoding the program Listing 1 is encoded to machine instructions using the instruction scheme in Table 3. Listing 2 shows the encoded program.

2.3. CPU BLOCK

The CPU block is shown in Fig. 2. Blue wires are control bits from the ControlUnit. Along with Fig. 2 this section will explain the connectivity from ProgramCounter(PC) to RegisterFile and ControlUnit. With these anchor points the rest is laid out.

2.3.1. ProgramCounter and RegisterFile The run port drives the PC. PC in turn gives a 16-bit address to the Program-Memory, which in turn outputs a 32-bit instruction. The instruction is split according to the instruction scheme. Rd connects to writeSel on the RegisterFile, Ra to aSel and Rb to bSel. The Value/address goes to the PC in case a jump is performed, and to a mux leading to the ALU in case an immediate value is used as a second operand. The Extender pads the 16-bit value to fit the 32-bit input in the mux. The RegisterFile outputs are connected to DataMemory with a to dataWrite and b to address.

2.3.2. ControlUnit The last instruction piece contains opcode which goes into the ControlUnit. From here Driving bits are wired to various multiplexers and components. writeToRegister drives the writeEnable port and is on if data is to be saved in the RegisterFile. writeToMemory drives the writeEnable port and is on if data is to be saved in the DataMemory. stop goes to the PC and done output port and tells the system to terminate. immediateJump and jump, together with the first bit of the ALU result drives the jump port on the PC, according to Table 4.

Table 4: ControlUnit jump bits

jump	immediateJump	alu.result	ProgramCounter.jump
0	0	X	0
1	0	0	0
1	0	1	1
1	1	X	1

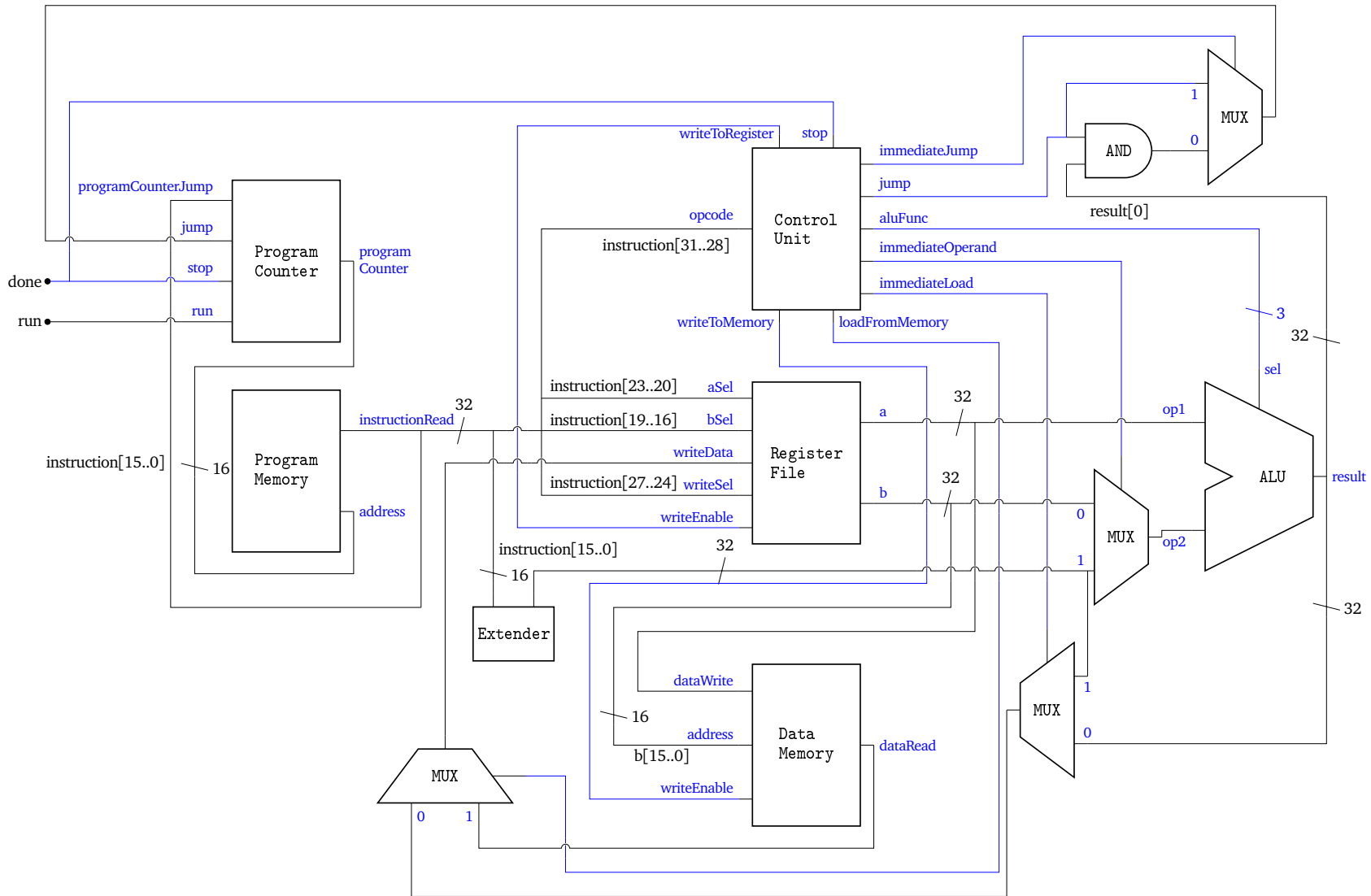
aluFunc tells the ALU which operation to be performed. immediateOperand connects the op2 input on the

ALU with either the padded Value/address instruction or b port on RegisterFile.
immediateLoad either connects the ALU result port or the Value/address instruction to the next mux, which is driven by loadFromMemory. Here either result, Value/address or dataRead on DataMemory is connected to writeData on RegisterFile.
The opcode and corresponding control signals are shown in Table 5.

Table 5: ControlUnit control bits

instruction opcode	ADD 0001	SUB 0010	ADDI 0011	SUBI 0100	MULT 0101	OR 0110	AND 0111	LOADI 1000	LOAD 1001	STORE 1010	INC 1011	JMP 1100	JEQ 1101	END 1111
writeToRegister	1	1	1	1	1	1	1	1	1	0	1	0	0	0
stop	0	0	0	0	0	0	0	0	0	0	0	0	0	1
immediateJump	0	0	0	0	0	0	0	0	0	0	0	1	0	0
jump	0	0	0	0	0	0	0	0	0	0	0	1	1	0
aluFunc	000	001	000	001	101	010	011	000	000	000	110	000	100	000
immediateOperand	0	0	1	1	1	0	0	0	0	0	0	0	0	0
immediateLoad	0	0	0	0	0	0	0	1	0	0	0	0	0	0
loadFromMemory	0	0	0	0	0	0	0	0	1	0	0	0	0	0
writeToMemory	0	0	0	0	0	0	0	0	0	1	0	0	0	0

Figure 2: Block diagram of the CPU architecture. Blue lines are control signals.



3 IMPLEMENTATION

In our chisel code we have designed different components to work together to create and operate as a general programmable CPU, it's made to be able to solve any solvable task, with a balanced distribution of registers and processing units such as the ALU. We have The ALU that makes all the calculations, it uses a switch to determine what mathematical operation is selected, and then checks for overflowing even though for this assignment it is quite overkill, as no value will ever overflow the 16 bits capacity. Then we have our register where we create an array that save it our register we constructed makes it possible to write and read data, and save it to a register if we wanted to. so it has 5 inputs, 2 values, 1 its need to be written, 1 where it needs to be written, and 1 is the specific data that need to be written. "CPUTop" is our main implementation of the CPU, we designed the CPU itself, where we connects every single components that we constructed to communicate together. we designed what bit was reserved to what so that we knew when there was a 32 bit then what component should read each bit from the opcode. Our implementation during emulation communicates with the console through chisel's poke, letting us see the results clearly in the command line in the IDE for easy debugging and use.

4 TEST AND ANALYSIS

Our erosion step has branches that allow it to execute the full process on an image with more or less cycles depending on the original image, therefore each pixel or step takes between 7 cycles and 21 cycles. From our test-runs the full 20×20 image takes about 6815 cycles, leaving the program with an average cycle requirement per pixel with all the processing of: 17.03. That means a 64 MHz CPU would perform a step in 265 nanoseconds, a 500 MHz in 34 ns, 1 GHz in 18 ns, 2 GHz in 9 ns. Although depending on the memory it is reading from, it might not be able to complete the step that fast, due to common RAM taking usually around 20-50ns just to read from.

By counting memory access in an iteration (while ignoring border cases as these decline in number proportional to the total pixel number as the picture grows in size), it can be seen that every black pixel needs two operations, while every white pixels needs six. This means that the memory access, using the above estimate takes for every black pixel takes $50.0 \text{ ns} \times 2 = 100 \text{ ns}$ and every white pixel takes $50.0 \text{ ns} \times 6 = 300 \text{ ns}$. An overview of the memory access times an cycles for the three pictures (Black image, White image, and Cells image) can be seen in Table 6

Table 6: Memory access times and cycles for the three images

	Black image	White image	Cell image
No of black pixels	400	0	341
Black pixel ram access	800	0	682
Memory access time	40,000 ns	0 ns	34,100 ns
No of white pixels	0	400	59
White pixel ram access	0	2400	354
Memory access time	0 ns	120,000 ns	17,700 ns
Total memory access time	40,000 ns	120,000 ns	51,800 ns
Cycles	5635	12115	6815

Using Table 6, the runtimes can be estimated using the following equation:

$$\text{Memory access time} + \frac{\text{Cycles}}{\text{Clockspeed}} \quad (4.1)$$

Table 7 shows the runtimes at various speeds for the three images calculated with Eq. (4.1).

Table 7: Runtimes at various clockspeeds

	64 Mhz	500 Mhz	1 Ghz	2 Ghz
Black image	128,047 ns	51,270 ns	45,635 ns	42,818 ns
White image	309,297 ns	144,230 ns	132,115 ns	126,058 ns
Cells image	158,284 ns	65,430 ns	58,615 ns	55,208 ns

Since our code needs to load from memory and then calculate on the same 6 registers, while using the other registers to reference from, the program would not benefit from a very fast clock cycle, and in the end become very dependent on memory load speed for its total time required to run.

The cells image had a white cell percentage of $\frac{59}{400} \approx 15\%$. The image from assignment one had $950 \times 950 = 902,500$, yielding with the same percentage $902,500 \times 0.15 = 135375$ white pixels and $902,500 \times 0.85 = 767125$ black pixels. If the cycles on that percentage scales linearly the total number of cycles is:

$$\frac{6815}{400} \times 902,500 \approx 15,376,000 \quad (4.2)$$

Table 8: Memory access times and cycles a 950×950 image

No of black pixels	767125
Black pixel ram access	1534250
Memory access time	76,712,500 ns
No of white pixels	135375
White pixel ram access	812250
Memory access time	40,612,500 ns
Total memory access time	117,325,000 ns
Cycles	15,376,000

Table 8 shows the memory access estimate and number of cycles estimate for the picture from assignment 1. On a 1 Ghz chip the runtime would be:

$$117000000 \text{ ns} + \frac{15376000}{1.00 \text{ GHz}} \approx 133 \text{ ms} \quad (4.3)$$

Assignment 1 on the same computer took 201 ms to complete the same task of loading the image (although a bit more complex due to needing to grey scale) and then erode once. Although this is an unfair comparison, due to Assignment 1 ran directly on the Windows machine with an executable file compiled by GCC, which is quite competent in optimising code into assembly, easily beating our own made assembly code.