# 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<br>Niclas Juul Schæffer<br>Rasmus Wiuff | ISA, assembler compilation, machine instruction encoding, CPU Block design, Program Counter 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 report jump) instruction is needed to navigate instructions and 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.

***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.*

***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` |

***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 $x + y \times 20$.
- Lines 10-13 checks for image border case.
- Lines 14-17 checks the inner pixel.

***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 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.

**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.

***Listing 2:*** *The encoded program*

```
    OP   Rd   Ra   Rb   Value/address
00. 1000 0000 0000 0000 0000000000000000
01. 1000 0001 0000 0000 0000000000000000
02. 1000 0010 0000 0000 0000000000010011
03. 1000 0011 0000 0000 0000000000000000
04. 1000 0100 0000 0000 0000000011111111
05. 1101 0000 0000 0010 0000000000101111
06. 1101 0000 0001 0010 0000000000101100
07. 0101 0101 0001 0000 0000000000010100
08. 0001 0110 0000 0101 0000000000000000
09. 0011 0101 0110 0000 0000000110010000
10. 1101 0000 0000 0011 0000000000101000
11. 1101 0000 0001 0011 0000000000101000
12. 1101 0000 0000 0100 0000000000101000
13. 1101 0000 0001 0100 0000000000101000
14. 0101 0110 0001 0000 0000000000010100
15. 0001 0111 0000 0110 0000000000000000
16. 1001 1000 0000 0111 0000000000000000
17. 1101 0000 1000 0011 0000000000101000
18. 0100 1100 0000 0000 0000000000000001
19. 0011 1101 0000 0000 0000000000000001
20. 0100 1110 0001 0000 0000000000000001
21. 0011 1111 0001 0000 0000000000000001
22. 0101 0110 0001 0000 0000000000010100
23. 0001 0111 0110 1100 0000000000000000
24. 1001 1000 0000 0111 0000000000000000
25. 0101 0110 0001 0000 0000000000010100
26. 0001 0111 0110 1101 0000000000000000
27. 1001 1001 0000 0111 0000000000000000
28. 0111 1010 1000 1001 0000000000000000
29. 0101 0110 1110 0000 0000000000010100
30. 0001 0111 0110 0000 0000000000000000
31. 1001 1000 0000 0111 0000000000000000
32. 0111 1001 1000 1010 0000000000000000
33. 0101 0110 1111 0000 0000000000010100
34. 0001 0111 0110 0000 0000000000000000
35. 1001 1010 0000 0111 0000000000000000
36. 0111 1000 1001 1010 0000000000000000
37. 1101 0000 1000 0011 0000000000101000
38. 1010 0000 0100 0101 0000000000000000
39. 1100 0000 0000 0000 0000000000101010
40. 1010 0000 0011 0101 0000000000000000
41. 1100 0000 0000 0000 0000000000101010
42. 1011 0001 0001 0000 0000000000000000
43. 1100 0000 0000 0000 0000000000000110
44. 1011 0000 0000 0000 0000000000000000
45. 1000 0001 0000 0000 0000000000000000
46. 1100 0000 0000 0000 0000000000000101
47. 1111 0000 0000 0000 0000000000000000
```

## 2.3. CPU BLOCK

The CPU block is shown in Fig. 2. Blue wires are control bits from the `ControlUnit`. Alond with Fig. 2 this section will explain the connectivity from `ProgramCounter(PC)` to `RegisterFile` and `ControlUnit`. With these anchorpoints 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 `ProgramMemory`, 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 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 `operand2` input on the `ALU` with either the padded `Value/address` instruction or `b` port on the `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 the `DataMemory` is connected to `writeData` in the `RegisterFile`.
The opcode and corresponding control signals are shown in Table 5.

***Table 5:*** *ControlUnit control bits*

| instruction | ADD | SUB | ADDI | SUBI | MULT | OR | AND | LOADI | LOAD | STORE | INC | JMP | JEQ | END |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opcode | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 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 |

Mikkel Arn Andersen **s224187**
Niclas Juul Schæffer **s224744**
Rasmus Kronborg Finnemann Wiuff **s163977**

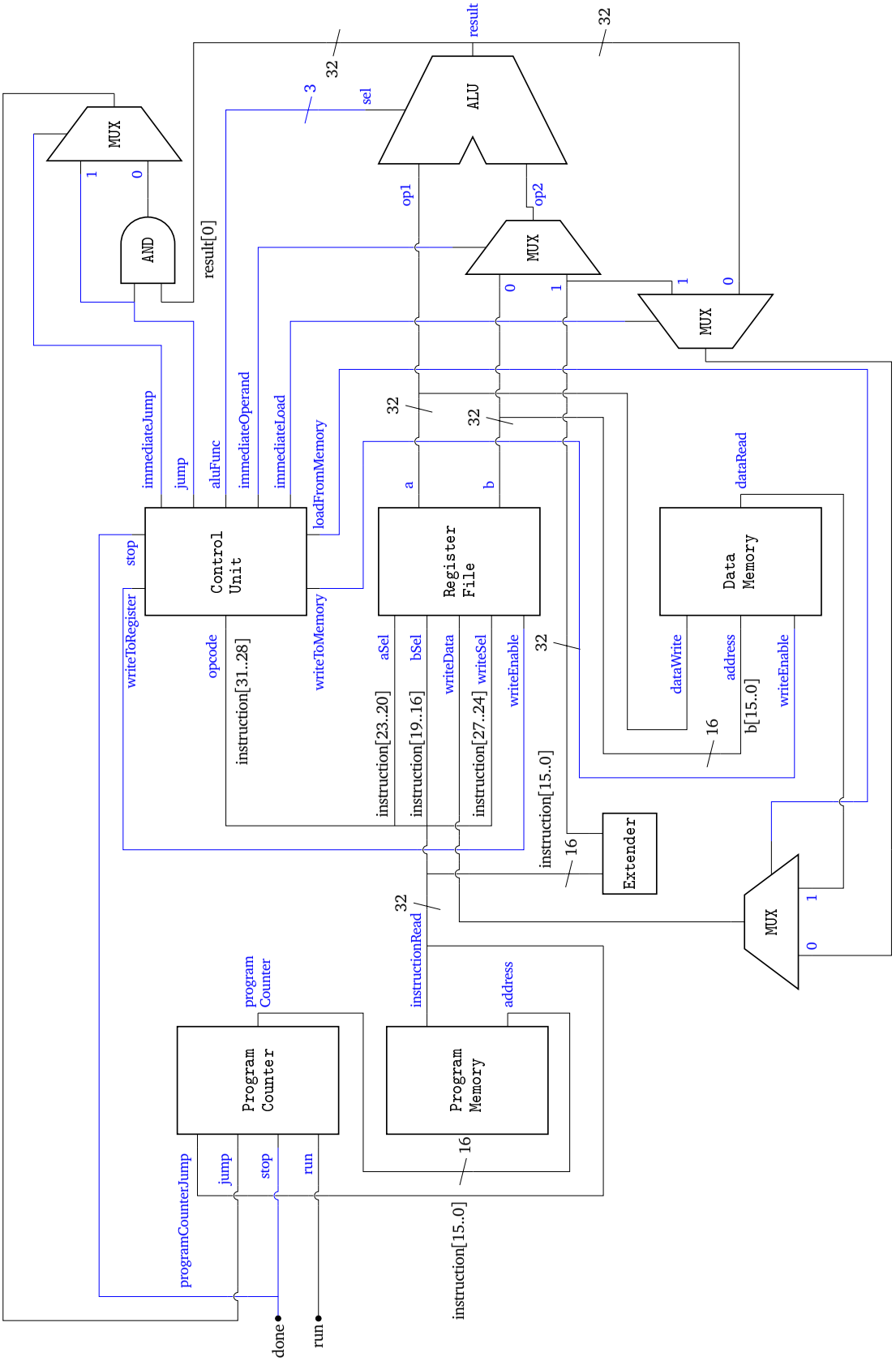


***Figure 2:*** *Block diagram of the CPU architecture. Blue lines are control signals.*

## 3 Implementation

*Briefly discuss the implementation in Chisel of your design. You can include some code snippets if these are relevant to explain certain aspects of the implementation. In other words, try to answer the question "What does a reader need to know about your Chisel implementation?"* in our chisel code we have designed different components to work together to create and operate as a CPU. We have The ALU that makes the calculations. The way we structured is that, it has 3 inputs and 1 output. 1 input is for the function and 2 input it's for the values, then we have an output that are for the result of the calculation. and i used a switch to determent what function is selected, and we have written some tests that test different scenarios about this and even check for overflowing even tho, we will not overflow this 32 bit system. 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 need to be written, and 1 is the specific data that need to be written.

## 4 Test and analysis

*Report here the results from the test you have carried out. Present the test you have developed (if any). Remember to discuss the results and the test you have carried out, do not just present them, but explain and argue their meaning. Address the design evaluation questions listed in Task 11 in the Assignment 2 document.*
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 20x20 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 64MHz CPU would perform a step in 265 nanoseconds, a 500MHz in 34 nanoseconds, 1GHz in 18ns, 2GHz in 9ns. 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. Since our code needs to load from ram 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 dependant on memory load speed for its total time required to run. Scaling this concept up, on the emulated CPU which ran at 2761Hz it completed 400 pixels in 2.9 seconds and printed it out, for a 950x950 image like in assignment 1, it is reasonable to assume that would take 87,7 minutes with the same CPU setup. Assignment 1 on the same computer took 201ms 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. Thusly it was able to load and calculate the pixels at full speed and a higher Hz than the emulated CPU was capable of.