FPGA Project Report

Prepared for

Dr. Mohsen Raji

Prepared by

Ali Maher, AmirHossein Roodaki

1th of Feb 2024

Introduction

The project is about implementing a computerized digital system in the field of **image coding**. The primary focus is on algorithms that encode images using a segmentation approach to effectively describe the key features of recognizable image sections. One specific algorithm discussed is the **Chain Code**, which utilizes an 8-directional vector to encode the edges of an object in an image. The algorithm starts by identifying the first edge pixel, determining the direction of movement to the next pixel, and iteratively encoding the edges in a clockwise direction.

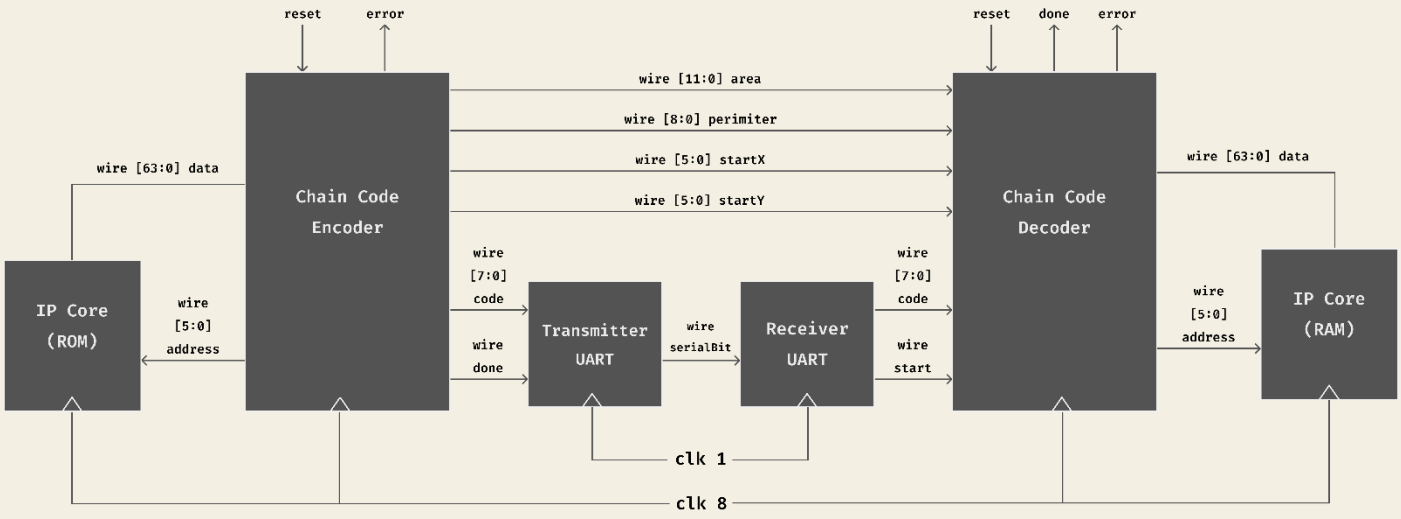
Key points to consider in implementing the algorithm include prioritizing edge coding based on movement direction, assigning codes to edges with higher priority, and handling a 64x64 binary image assumption.

Let's Get Through it!

This project generally contains two parts: the first part is a Python script to generate the binary coded image and the second part is a Verilog code to encode and decode the chain code.

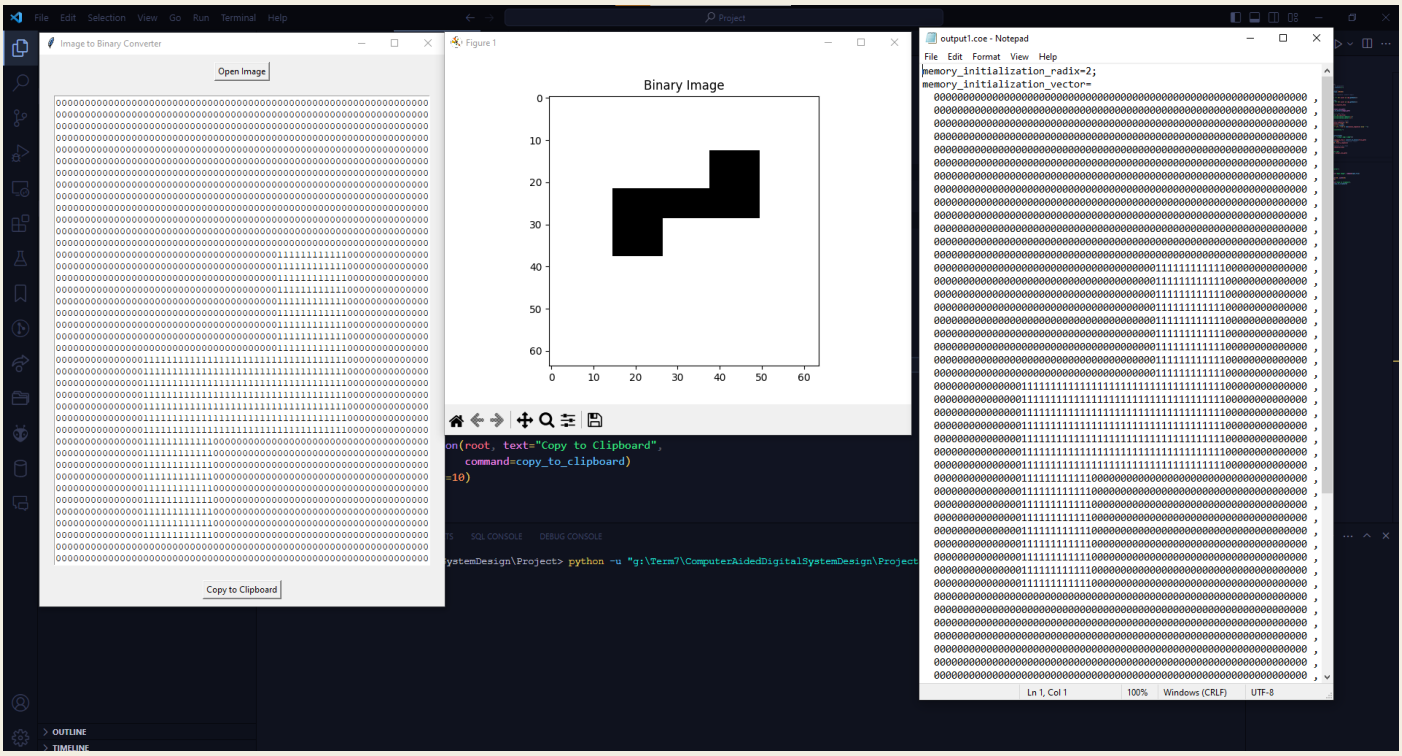
This project includes several parts: we need two RAM IP cores, one for the input binary coded image at the starting module (this can be a ROM) and the other for storing the decoded chain code at the end module. We also need an encoder module to encode the chain code and a decoder module to decode the chain code. And we need *uart_transmitter* and *uart_receiver* modules to transmit and receive the chain code.

The Verilog parts includes 7 modules: *chain_code_encoder*, *chain_code_decoder, uart_transmitter, uart_receiver, ram*, *rom* and a *main* file as the top module.



1. Python Part

This part is implemented for extracting a binary code from a black and white picture. And at the end of it we convert that code into "*.coe*" file format and save that. Because in the following we have to create our IP cores, like RAM and ROM, with the "*.coe*" file.



---

2. Verilog Project

1. ROM IP Core:

We have a ROM IP Core at the beginning of the project which we store the binary code of the picture, which we got from the python part. For creating a ROM IP Core, we should add a module to our project file of IP Core. Then we should select the properties of our IP Core. In one of these steps we should add the "*.coe*" file in order to initialize the IP Core with that. After we make this module with Xilinx panel, we should get an instance from it and then work with that.

2. Encoder:

Inputs: reset, clk, start

Outputs: code, done, error, perimeter, area, startX, startY

In this part we define 9 states for the state machine. Each state has its own logic and transition to the next state.

State 0: INIT ==> Initialize the state machine and reset all variables.

State 1: IDLE ==> Wait for the start signal to start the chain code calculation.

State 2: FIND_START ==> Find the starting pixel. With help of two for loops, we can find the first pixel with value 1.

State 3: CALC_CHAIN ==> Calculate the chain code for the current pixel and store the result in the array.

```
// Chain code calculation logic using the provided conditions
if (loaded_ram[current_i][current_j + 1] == 1'b1 &&  loaded_ram[current_i - 1][current_j + 1] == 1'b0) begin
    current_code <= 3'b000;
    current_j <= current_j + 1;
end
else if (loaded_ram[current_i - 1][current_j + 1] == 1'b1 && loaded_ram[current_i - 1][current_j] == 1'b0) begin
    current_code <= 3'b001;
    current_i <= current_i - 1;
    current_j <= current_j + 1;
end
else if (loaded_ram[current_i - 1][current_j] == 1'b1 && loaded_ram[current_i - 1][current_j - 1] == 1'b0) begin
    current_code <= 3'b010;
    current_i <= current_i - 1;
end
else if (loaded_ram[current_i - 1][current_j - 1] == 1'b1 && loaded_ram[current_i][current_j - 1] == 1'b0) begin
    current_code <= 3'b011;
    current_i <= current_i - 1;
    current_j <= current_j - 1;
end
else if (loaded_ram[current_i][current_j - 1] == 1'b1 && loaded_ram[current_i + 1][current_j - 1] == 1'b0) begin
    current_code <= 3'b100;
    current_j <= current_j - 1;
end
else if (loaded_ram[current_i + 1][current_j - 1] == 1'b1 && loaded_ram[current_i + 1][current_j] == 1'b0) begin
    current_code <= 3'b101;
    current_i <= current_i + 1;
    current_j <= current_j - 1;
end
else if (loaded_ram[current_i + 1][current_j] == 1'b1 && loaded_ram[current_i + 1][current_j + 1] == 1'b0) begin
    current_code <= 3'b110;
    current_i <= current_i + 1;
end
else if (loaded_ram[current_i + 1][current_j + 1] == 1'b1 && loaded_ram[current_i][current_j - 1] == 1'b0) begin
    current_code <= 3'b111;
    current_i <= current_i + 1;
    current_j <= current_j + 1;
end
else begin
    current_code <= 3'b000; // Default case, no move
end
```

Movements are demonstrated with the conditions based on the pixels. And after each iteration we should handle the coordinates updates.

In this part we also calculate the perimeter based on number of iterations (It is not the exact perimeter of the shape, but it is so close). We also have some attempts to get exact perimeter, but we failed.

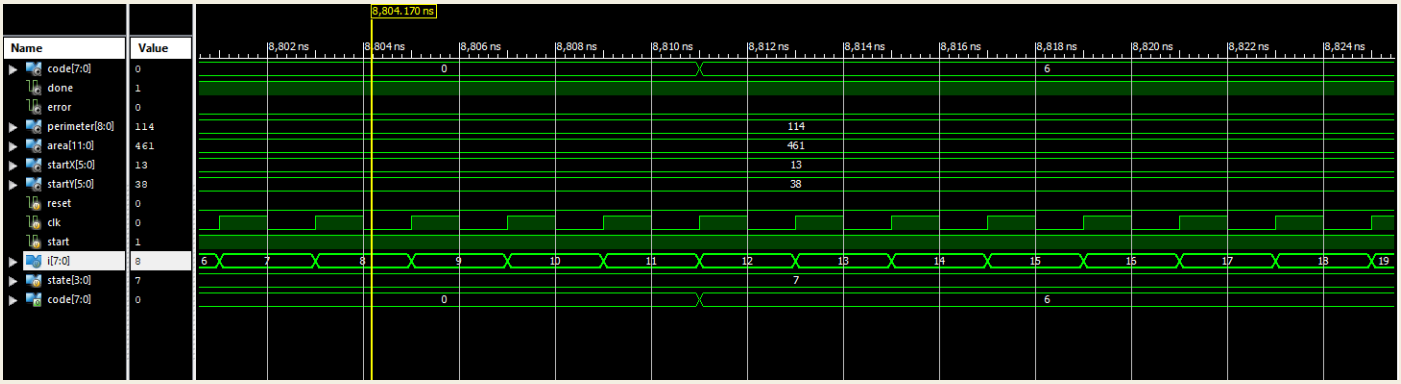
State 4: CHECK_DONE ==> Check if the traversal is done. If not, continue to the next pixel. We check if the current pixel is the same as the starting pixel.

State 5: INIT_PROPERTIES ==> Initialize the properties of the object for area calculation.

State 6: CALC_AREA ==> Calculate the area of the object with iterating through the pixels and counting the number of pixels which are one.

State 7: SEND_RESULTS ==> Send the results to the output. We are preparing 8-bit data contains 3-bit of chain code i and chain code i+1, and two don't care bits, for the parallel input of the *usart_transmiter*.

State 8: TERMINATE ==> Terminate the state machine.



3. UART_Transmitter:

In the project files we named it *sender_uart*.

Inputs: clk, rst, tx_data (8-bit), ready

Outputs: tx_out

In this module we receive the parallel data from last module and try to send it serial as it has happen in UART. We consider one bit for start, 8-bit for data and one bit as stop bit.

Start	Bit-0	Bit-1	Bit-2	Bit-3	Bit-4	Bit-5	Bit-6	Bit-7	Stop

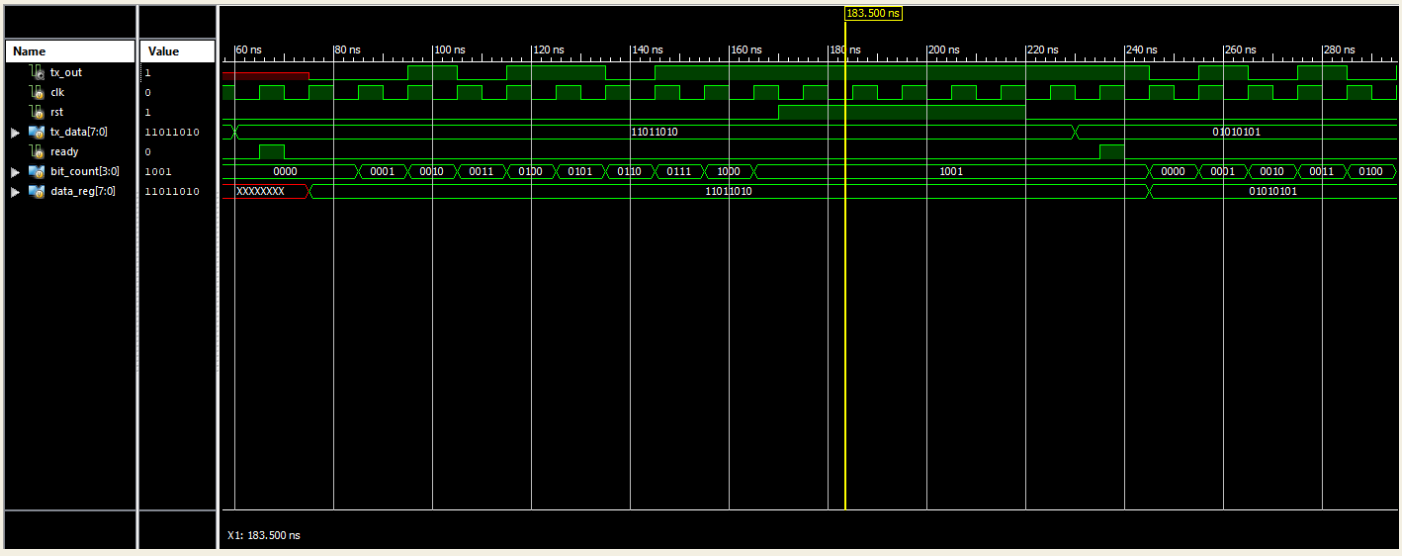
Here we consider three states:

State 0: IDLE ==> In this state, the sender is waiting for the ready signal to be high.

State 1: WAIT_READY ==> In this state, the sender has received the ready signal and is ready to send the data. And also send the start bit.

State 2: SEND_DATA ==> in this state, the sender is sending the data bit by bit (serially)

Simulation: This is the simulation, the test benches are available in the project folder.



4. UART_Receiver:

In the project files we named it *receiver_uart*.

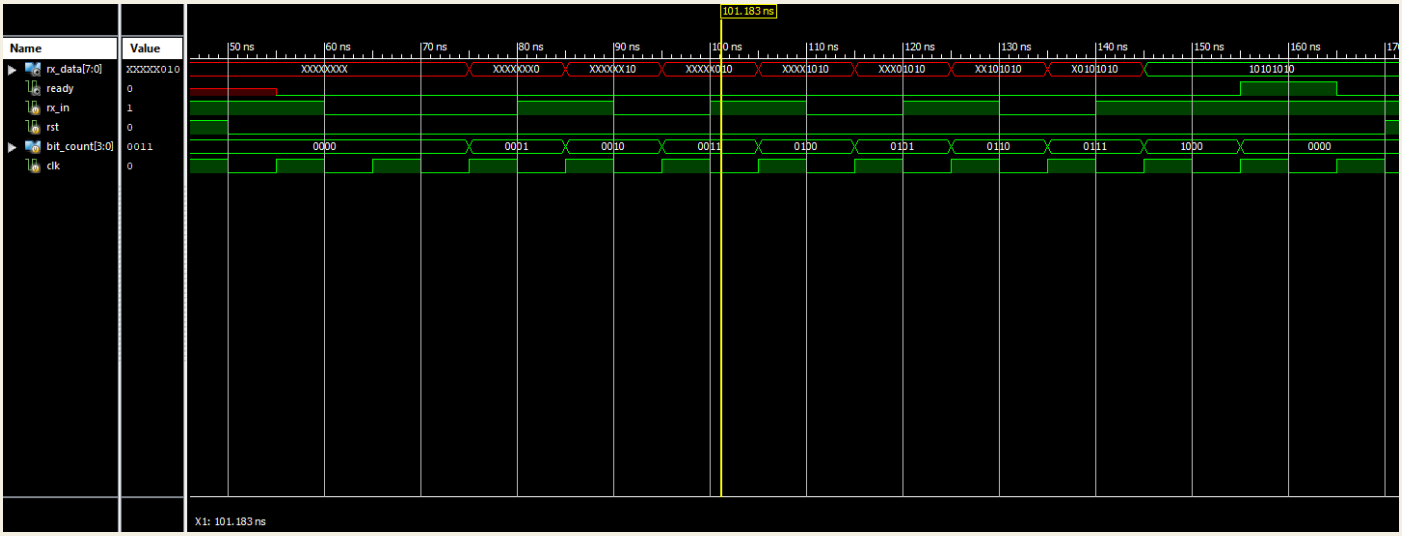
Inputs: clk, rst, rx_in

Outputs: rx_data (8-bit), ready

In this module we receive the serial data from last module and try to send them parallel to decoder module. We have considered two states:

State 0: IDLE ==> Waiting for start bit.

State 1: RECEIVE_DATA ==> Receiving data, 8 bits. Then check for stop bit and set ready.



---

5. Decoder:

Inputs: reset, clk, start, code (8-bit), perimeter, area, startX, startY

Outputs: pixels, done, error

In this part we define 7 states for the state machine. Each state has its own logic and transition to the next state.

State 0: INIT ==> Initialize the RAM and the variables

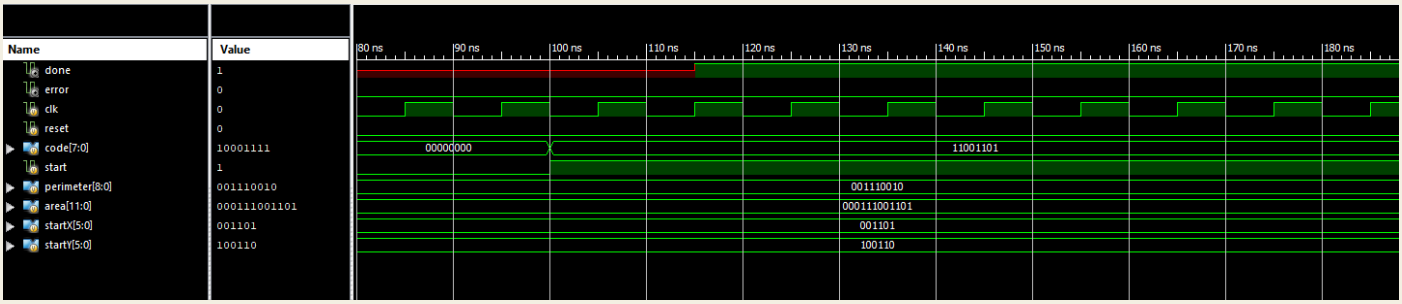
State 1: IDLE ==> Wait for the start signal

State 2: DECODE_IMAGE ==> Decode the image using the chain code

State 3: CHECK_DONE ==> Check if the decoding is done

State 4: CALC_AREA ==> Calculate the area and the perimeter. For calculating this part we should apply an algorithm: we make all the ram spaces 1, and then we go through the memory till we reach the first one pixel based on received data. And make them zero, then we leave the row and go to next row till the end. Then we rotate the ram then we do this again for the next side. Repeat this for the four sides.

State 5: TERMINATE ==> Terminate the state machine



---

6. RAM IP Core:

We need a RAM IP Core at the end of the last module to keep the regenerated (decoded) binary sequence of image. This sequence must be exactly the same with the one which we stored in the ROM, to show that the behavior of the code is correct. For using this module after we create it in the Xilinx panel, we should get an instance from it and then work with that.

---

7. Main as the top module:

The above modules, should be instantiated and used in the "main.v". And the appropriate connections should be set with wires.

```
module main(
    input wire clk_1,
    input wire clk_8,
    input wire reset,
    input wire start
);

    wire [7:0] code___from___chain_code_encoder___to___sender_uart;
    wire done___from___chain_code_encoder___to___sender_uart;
    wire error___from___chain_code_encoder___to___sender_uart___and___receiver_uart;
    wire [8:0] perimeter___from___chain_code_encoder___to___chain_code_decoder;
    wire [11:0] area___from___chain_code_encoder___to___chain_code_decoder;
    wire [5:0] startX___from___chain_code_encoder___to___chain_code_decoder;
    wire [5:0] startY___from___chain_code_encoder___to___chain_code_decoder;
    wire tx_out___from___sender_uart___to___receiver_uart;
    wire [7:0] rx_data___from___receiver_uart___to___chain_code_decoder;
    wire ready___from___receiver_uart___to___chain_code_decoder;

    rom rim_rom (
    .clka(clk_8),
    .addra(addra),
    .douta(douta)
    );

    chain_code_encoder chain_code_encoder (
    .reset(reset),
    .clk(clk_8),
    .start(start),
    .code(code___from___chain_code_encoder___to___sender_uart),
    .done(done___from___chain_code_encoder___to___sender_uart),
    .error(error),
    .perimeter(perimeter___from___chain_code_encoder___to___chain_code_decoder),
    .area(area___from___chain_code_encoder___to___chain_code_decoder),
    .startX(startX___from___chain_code_encoder___to___chain_code_decoder),
    .startY(startY___from___chain_code_encoder___to___chain_code_decoder)
    );

    sender_uart sender_uart (
    .clk(clk_1),
    .rst(reset),
    .tx_data(code___from___chain_code_encoder___to___sender_uart),
    .ready(done___from___chain_code_encoder___to___sender_uart),
    .tx_out(tx_out___from___sender_uart___to___receiver_uart)
    );

    receiver_uart receiver_uart (
    .clk(clk_1),
    .rst(reset),
    .rx_in(tx_out___from___sender_uart___to___receiver_uart),
    .rx_data(rx_data___from___receiver_uart___to___chain_code_decoder),
    .ready(ready___from___receiver_uart___to___chain_code_decoder)
    );

    chain_code_decoder chain_code_decoder (
    .clk(clk_8),
    .reset(reset),
    .code(rx_data___from___receiver_uart___to___chain_code_decoder),
    .start(ready___from___receiver_uart___to___chain_code_decoder),
    .perimeter(perimeter___from___chain_code_encoder___to___chain_code_decoder),
    .area(area___from___chain_code_encoder___to___chain_code_decoder),
    .startX(startX___from___chain_code_encoder___to___chain_code_decoder),
    .startY(startY___from___chain_code_encoder___to___chain_code_decoder)
    );

    ram rim_ram (
    .clka(clk_8),
    .wea(wea),
    .addra(addra),
    .dina(dina),
    .douta(douta)
    );

endmodule
```

- **Other tips:**
  - The UART property mentioned in project documentation, is as below:
    Clock Frequency: **50 Mhz**
    By 50 Mhz, we mean that each bit serial transmission in the UART must be done in T = 1/f = 1/(50*10^6) = **20 ns**
    And in our implementation we get 10-bit (one start bit, 8-bit data, one stop bit) so a packet transmission in UART, takes 200 ns.
    Then we can conclude that: "*one clock in the in the encoder(decoder) equals to 8 clocks in the sender(receiver)*".

The project files with doc are also available on my GitHub.

Thanks for your time