Imperial College London

Department of Electrical Engineering

ECG project – Group 13

Supervisors:
Dr. Christos Papavassiliou
Zifan Wang

Group members:

Alan Yilun Yuan	CID:01333818
Edward Turner	CID:01364481
Edoardo Santi	CID:01373388
Xuying Zhao	CID:01319778
Yinzi Zhang	CID:01368804
Grigoryan Rudik	CID:01348311

3^{rd} *year group project*

June 26, 2020

Contents

1 Introduction

Our project briefing given by the client was to make an affordable device for measuring electrocardiogram (ECG) signals to enable medical professionals to get an elaborate insight in patients' health outside of contact hours. The goal of this project was to provide a way to process ECG data with a focus on a hardware implementation. Advantages of a hardware implementation are reduced power consumption, latency and increased throughput. The main disadvantage is a much greater development time compared to software implementations.

FPGAs (Field Programmable Gate Array) devices were chosen for this task as they allow for a fast and efficient implementation without the initial design costs of developing an ASIC. Additionally, the possibility of easily changing the hardware design by simply reconfiguring the FPGA makes them suitable for prototyping. The main focus of this project is to provide a proof of concept of using FPGAs to classify ECG data that could be later embedded in a portable device designed for continuous ECG monitoring of patients at risk of heart disease and that warns the user about potential heart disease. In this case, the main constraints are the price and the portability of the device. However, our design could be optimised for other tasks. For example, an implementation with a favourable throughput to power consumption ratio could be useful as a hardware accelerator in a server that processes extended amounts of data.

Our work mainly focused on the classification of heart conditions (e.g. arrhythmia, left ventricular hypertrophy, ...) using ECG signals. Each condition, could be detected with custom algorithms or using a data-driven approach. The initial plan was to use custom algorithms for each disease. One of the parts that this involved was the Pan-Tompkins algorithm [1] for beat detection. This was first done in Matlab and was going to be implemented on FPGA using a finite state machine approach[2]. However, using a custom algorithm for each disease would significantly increase the complexity of the project, as the processing required for each condition can vary significantly. Thus the data driven approach was chosen due to its flexibility and the ability of using one model to detect all the diseases.

It was chosen to use neural networks due to their performance and flexibility. The model implemented in hardware is a fully connected network which is fed as input one cycle of a 2-lead electrocardiogram with normalised values and outputs whether the patient is healthy or not. This design achieved an accuracy of 93% and an AUC[1] of 0.94.

Our work was conducted using the DE1-SoC board from Altera, which contains a Cyclone V FPGA from Intel. This was chosen mainly due to our familiarity with it and with the software used to configure it, which is Quartus Prime from Intel. The DE1-SoC is also relatively cheap, retailing at $175 for academic customers, demonstrating the affordability of the project (further detail in the 'Cost of the project' section).

The latency of our network is 8 clock cycles at a clock frequency of 50MHz, although the limit frequency was not experimented as this would require overclocking the DE1-SoC board. The power consumption of the FPGA using our design was calculated to be around 0.5W. For more detail, visit the 'Hardware' section of this report.

When considering single predictions there is still a significant amount of error. If the outputs of our network relative to the same patient at different times are independent (this assumption would require investigation), the probability of error could be reduced to an acceptable level by taking several predictions into consideration. This methodology would need further consideration

[1] area under the receiver operating characteristic curve, which measures performance of a classifier by varying the decision threshold.

before a device using our network is taken into production.

Our design can be optimised to fit different specifications. Pipelining could be implemented to increase the throughput, while a folded implementation of the network would allow to configure large networks on boards with limited resources (ALMs, DSP blocks), such as the ResNet discussed in the 'Software' section. The FPGA design could be modified for working in real-time and interfacing with a hard or a soft processor could be explored.

2 Documentation

2.1 Requirements gathering

The goal of this project was to achieve fast and affordable hardware implementation of ECG signal classification. The total cost of the product should be as low as possible. Low power consumption and low hardware resource occupation should also be included as one of the criteria. In addition, the product should be extendable to adding other features and well documented for future use. The requirements mentioned above gave the team guidelines on how the product should be designed.

To ensure the team stayed on target throughout the project we used a Gantt chart and held regular meetings every Tuesday (with the client present) and Thursday. All relevant documents and code were placed on either GitHub or Microsoft teams. We also split the team into hardware and software teams. The software team focussed on the software neural network model and the interface and the hardware team worked on the Verilog and Quartus design. One member of the team also worked on the bridge between the hardware and the software components. The GitHub repository can be accessed by: `https://github.com/alanyuan97/ARM_ECG`.

2.2 Gantt chart

The Gantt chart keeps a record of the work done and what still needs to be done by what date. The chart is shown in figure 1.

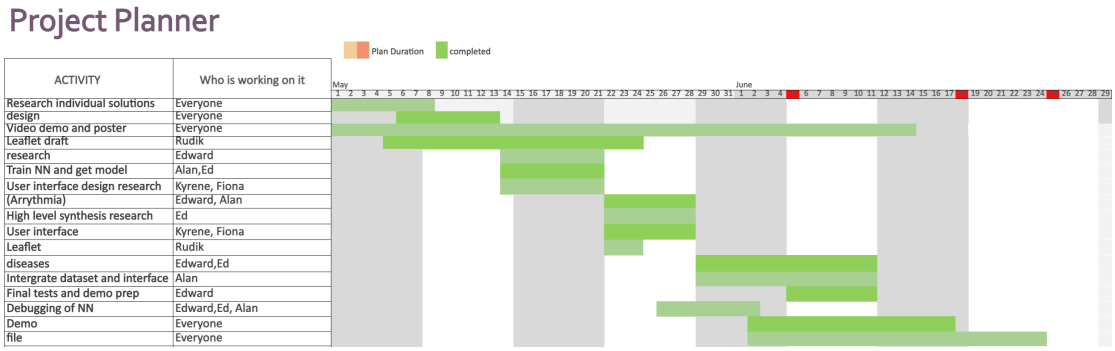

Figure 1: Gantt chart

2.3 Meeting minutes

After every Thursday meeting we took minutes to make note of what we discussed and outline the next steps we will take over the coming week. Figure 2 shows the log of all the decisions made at each meeting. This is where we discussed the allocation of work. In between meetings we corresponded using a WhatsApp group chat, the meeting minutes only the show the direction of the project not the specifics.

May 7th	• Research individual solutions • Make MATLAB simulations of solutions (HLS deep learning, traditional, deep learning fully connected, machine learning) • Agree to focus on arrythmia detection as our disease
May 14th	• Agreed on implementing neural network on FPGA • Agreed on using a website interface • Allocated hardware and software teams • Start work on making software neural network model • Start work on how to make neuron on hardware
May 21st	• Allocated work on leaflet • Check viability of using high level synthesis for deep learning on hardware research • Have website complete by next week • Have a working neuron on hardware
May 28th	• Difficulty with fitting NN model on the FPGA, make a smaller implementation • HLS is not feasible, so we only focus on the fully connected • Website is complete, now software focus on connecting interface and automated framework to build Verilog files • Discussed redesign of the leaflet
June 4th	• Leaflet shown as completed • Automated Verilog script is made • Hardware team have working simulations, need to place on the board • Software team work on QSYS to connect FPGA to interface
June 11th	• QSYS proves more difficult than expected, software team now help with demo prep • Hardware have a working implementation on the FPGA and now need to test accuracy and prepare a demo • Presentation slides allocated • Agreed on structure of the video to show result on hardware and then use interface as a place to store datasets which are on the FPGA and give insight into the signals • Agreed on who will represent and record video
June 18th	• Allocated part of design history file/final report

Figure 2: Meeting minutes

3 Software

Before implementing the neural networks on hardware, different models were experimented with in software, due to faster development time and easier experimentation. This was done using the Deep Learning API Keras in Python. The networks were trained on Google Colab to allow the networks to train in a much faster time. Google Colab provides free/cheap access to very powerful GPUs (e.g. Nvidia Tesla P100), which are orders of magnitude faster than our own devices. This led to the development of two main models:

- A fully connected neural network which was developed by prioritising compactness and ease of development on hardware. This network is the one used in the final implementation of our project and it takes in input pre-processed ECG data[2] (the input signal needs to be 'cut' to one heart beat cycle and normalised).
- A ResNet convolutional neural network which has greater capability and is state-of-the-art in the classification of raw ECG signals (doesn't require pre-processing of the input ECG signals).

3.1 Fully connected model

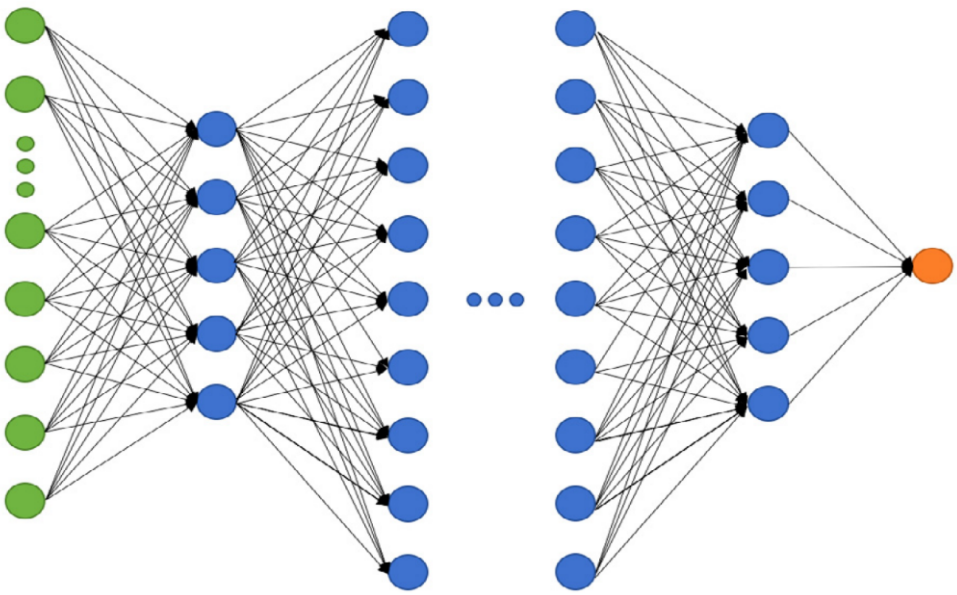

Figure 3: Fully connected model

Fully connected Neural Networks consist of fully connected (FC) neurons only. As denoted in Figure 3, several layers are linked to back-propagate the training loss and to calculate the final set

[2]The pre-processing was not implemented due to the limited time duration of the project. The implementation of this network can be considered as a starting point for implementing a different network that doesn't require preprocessing.

of weights. The weights are a set of matrices, where each subset can be interpreted as the weights between successive layers. The calculation steps involved in a FC model can be summarized into matrix multiplication and addition. Novel research has been made on several tasks i.e. MNIST Handwritten digits classification and regression[3]. The main reason of using a FC model as the first design choice was that only multiplication and addition operations are required. Figure 3 indicates the initial structure of the FC model[4].

The hidden layers consist of 5, 10, 30, 50, 30, 50, 10 and 5 neurons respectively [4]. An activation function is required after each neuron to introduce non-linearity. Our network uses the ReLU function in all layers, which sets negative values to 0. In the final layer, the activation function is a sigmoid function which maps all real numbers to the range between 0 and 1[3]. This allows to set a threshold between 0 and 1 for the output, above which the output is considered a positive and below which it is a negative[4]. The total number of training parameters is around 4.5k. The initial model had a high testing accuracy of more than 95%. However, the model did not fit on the FPGA using our unfolded implementation (see 'Hardware' section), thus we reduced the number of neurons to make it smaller. This slightly reduced the test accuracy to 93%. This model reported an AUC of 0.94.

3.2 ResNet model

Residual convolutional neural networks were shown to achieve the best performance in classification of raw ECG signals, comparable to adopting custom algorithms for each disease [5], while not needing any pre-processing (filtering, normalisation, etc...) and thus simplifying the design. Convolutional neural networks are particularly suited to tasks in which features within an input must be identified but they do not always occur in the same position. An example of this is image classification, as certain features, such as the presence of a horse or a bird must be identified, wherever they are located within the image. Similarly, when feeding raw ECG data in a network, features showing disease could appear anywhere within the data. The chosen model is an adapted version of the ResNet model from:
`https://github.com/antonior92/automatic-ecg-diagnosis\/blob/master/model.py`,
which implements the network described in A. Ribeiro's paper [6].

The code of the model can be accessed using the link: `https://colab.research.google.com/drive/1Wlu85NdX7ubAeod_t48yV_FFPObENAKo?usp=sharing`.

The network was modified in different versions in order to work with inputs of different sizes and to classify disease at different levels (superclass of the disease, class of the disease or the specific disease). The performance obtained in the classification of the 4 diagnostic super classes of the MIT-XL dataset [5] with different versions of this network is shown in Table 1.

The best performance was obtained with the original model, having a 12-lead ECG input of 4096 samples (equivalent to 10 seconds of signal at 409.6Hz). The network was also tested for predicting diagnostic classes and subclasses (different levels of specificity in identifying the diseases), obtaining similar AUC per class, although in these cases the test set is too small to give a reasonable estimate of the performance due to increased number of outputs.
Despite achieving a good level of accuracy and having the capability of diagnosing several diseases, this network was not implemented in hardware. It was decided that due to time constraints, it

[3]The code can be seen at `https://colab.research.google.com/drive/1u7e2QiSwjzpV1alrc3lRCp2dx56YR3IZ?usp=sharing`

[4]Setting the threshold requires estimating the probability of false negatives and false positives, assigning a cost to these two types of error and setting the threshold in order to minimise the total cost

	Accuracy	Average AUC per superclass
2-lead, 1000 input samples	87.1%	0.811
2-lead, 4096 input samples	86.5%	0.819
12-lead, 1000 input samples	87.6%	0.892
12-lead, 4096 input samples	90.1%	0.906

Table 1: Performance of different versions of the ResNet model.

would be better to first focus on the fully connected neural network first, due to its simplicity and limited size and that the acceleration of this residual neural network on FPGA could be done later as an extension of the project.

3.3 Quantisation of the network

As explained in the hardware section, the final hardware model adopts a fixed-point format, however, the weights of a Keras model are normally 32-bit floating point numbers. In order to convert them to fixed-point, the weights of the network can either be simply rounded to the closest number representable with the fixed-point format or the networks could be retrained using quantisation aware training. In the final implementation it was opted for the second option as it allows for a lower loss of accuracy relative to the non-quantised network. This was done using 'QKeras', a 'Tensorflow Keras' library which provides quantised network layers. The following shows the difference of the code between the non-quantised and the quantised network:

```
Dense(10)(x)
Activation('relu')(x)
```

quantise

```
x=QDense(10,kernel_quantizer=quantized_bits(bits=8, integer=2,symmetric=0,
    keep_negative=1), bias_quantizer=quantized_bits(bits=8, integer=2, symmetric
    =0, keep_negative=1))(x)
x=Lambda(clipit)(x)
x=QActivation("quantized_bits(bits=8, integer=2, symmetric=0, keep_negative=1)",
    name='L2')(x)
x=QActivation('relu')(x)
```

There are three main modifications:

- Keras "Dense" layers are substituted with QKeras "QDense" layers which introduce weights and biases of the chosen fixed-point format.
- A clipping layer is added after "Dense" layer ("Lambda" layer calling "clipit" function) in order to force its outputs to be within the range of values representable by the chosen fixed-point format. The clipping layer was also added in the hardware implementation and it serves the purpose of ensuring that the software and the hardware implementations deal with number outside the range in the same manner.
- A "quantized bits" layer is added in order to quantise the output to follow the chosen fixed-point format. This is not required but it makes the software implementation behave identically to the hardware implementation, making debugging of the latter easier.

Making the FPGA and the Keras implementations identical also serves the purpose of being able to test the classification performance of the hardware implementation using the software one.

3.4 Python scripts automating the Verilog

The software tool that was used for creating, compiling and synthesising Verilog code for the DE1-SoC is Quartus Prime Lite 18.1. Coding the fully connected network using Verilog is a repetitive task that would be tedious if done manually. It is well suited to being automated, thus we developed a Python script that automatically generates the Verilog files, given in input the dimensions of the networks and the values of its parameters. The files are generated by executing the genall_win.sh file (unfolded version, use genlight.sh for the reduced area version), which calls other modules that create the Verilog files of the nodes, the layers and the top-level entity and places them in directories organised by layer. The code also generates .mif files to initialise the memory of the ROM blocks in the design. The weights of the network are input using files with .npy extension that were obtained from the quantised software model. The Verilog code is printed by the Python script and is captured via the Linux terminal's pipelining feature, also noted as the '>' command. The code can be seen on our Github repository in the Hardware folder(`https://github.com/alanyuan97/ARM_ECG/tree/fully_automated_verilog_only`). The original plan was to retrieve the input to the network data from the user interface. However, this was shown to be complicated and will be thoroughly explained in the 'Bridge' section below. The alternative approach which was used to handle data was to save all test data in a ROM file.

4 Interface

4.1 Webpage display

To better improve users' experience and visualisation qualities, a frontend webpage interface has been implemented in HTML as well as a Python backend webserver. On the webpage, users can input a number corresponding to the index of an ECG signal from the test dataset. As shown in figure 4, the user interface will display the signal and the outcome of its classification performed on the FPGA.

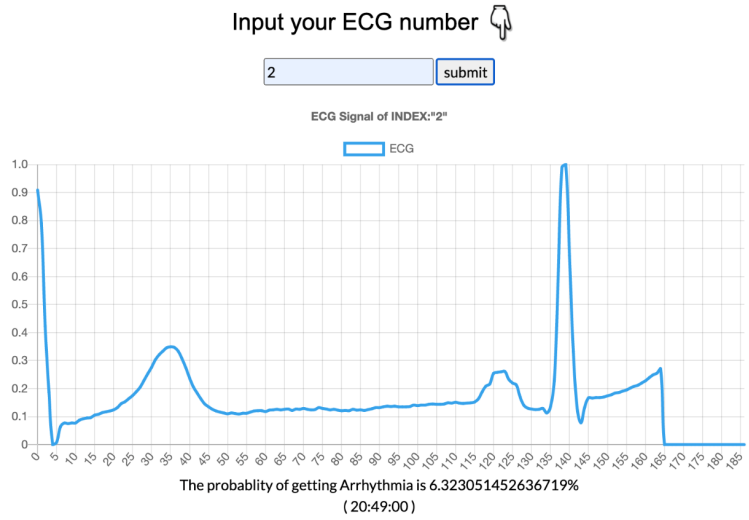

Figure 4: Web display

Once the user clicks on 'submit', a function named **get_ecg** is called and the number in the form is assigned to the variable **id**.

```
<form name="ecg_form" onsubmit="return get_ecg()">
          <input type="text" name="id"/>
          <input type="submit" value="submit">
          <p id="ecg_form_error"></p>
        </form>
```

On the client side of the webpage, function **get_ecg** sends a request with uniform resource locator(url) /**api**/**id**, where id is the input number. It receives information from the server and displays it on the webpage when **readystate** is 4 and **status** is 200, which means the request of connecting the server side has been accepted, **responseText** is available to use and the page is ready to be displayed. The function uses Asynchronous Javascript and XML (AJAX) to update the graph and the probability of having arrhythmia without refreshing the entire page. The function is shown below.

```
        <script>
          function get_ecg(){
            id=document.forms["ecg_form"]["id"].value

            request=new XMLHttpRequest();
            request.onreadystatechange = function() {
              if (this.readyState == 4){
                error_msg=""

                if (this.status == 200){
                  // Typical action to be performed when the document is ready:
                  var myObj = JSON.parse(this.responseText);
                  document.getElementById("heart_result").innerHTML ="The
    probablity of getting " +myObj.name+ " is "+ myObj.probability + "%" +"<br/> (
     "+myObj.time+" ) ";
                  var datastring = JSON.stringify(myObj.voltage),
                    array = datastring.match(/\d+(?:\.\d+)?/g).map(Number)
```

```
                    var list = [];
                      for (var i = 0; i < array.length; i++) {
                          list.push(i);
                      }
                    var data={
                      labels: list,
                      datasets: [{
                        label: 'ECG',
                        backgroundColor: 'transparent',
                        borderColor: "#36A2EB",
                        pointBackgroundColor: "#36A2EB",
                        pointBorderColor: "#36A2EB",
                        pointRadius:1,
                        data: array
                      }]
                    };
                    console.log(list)
                    var ctx = document.getElementById("myChart").getContext('2d');
                    var myLineChart = new Chart(ctx, {
                        type: 'line',
                        data: data,
                        options: {
                            title: {
                                display: true,
                                text: "ECG Signal of INDEX:" + JSON.stringify(id)

                            }
                        }
                    });
                  }else{
                    error_msg="Unable to retrive data from server, get "+this.status
                    document.getElementById("heart_result").innerHTML ='<img src="
heart_beat.jpg" class="w3-round w3-margin-bottom" alt="Random Name" style="
width:60%">'
                  }

                  document.getElementById("ecg_form_error").innerHTML=error_msg
                }


              }
              request.open("GET", "/api/"+id)
              request.send();

              return false; // prevent default form submission
            }
          </script>


```

Class **MyHandler** is a child class of **SimpleHTTPRequestHandler** and has all its properties. When the /**api** is detected on the server side, function **do_api** is called. **erasima_id** is the number inputted on the server side, which is then passed into the FPGA by recalling the **get_erasmia** function of class **HeartDiagnosis**, which casts the string of the number to integer. The function also returns the needed information from FPGA which is then passed into variable **data**. If there is no error, the probability of getting arrhythmia is represented as 32 bits, which is then casted to float and stored into the variable **wfile** in the form of json.

```
class MyHandler(SimpleHTTPRequestHandler):

    def do_GET(self):
```

```
        """Serve a GET request."""

        ####################################
        # ROUTER
        ####################################
        if self.path.split('/')[1] == 'api':
            # api stuff
            return self.do_api()
        else:
            # static stuff (default stuff)
            return SimpleHTTPRequestHandler.do_GET(self)


    def do_api(self):
        # current only disease is aresmia, no need furether routing
        try:
            erasmia_id = self.path.split('/')[-1]
            data = heartDiagnosis.get_erasmia(erasmia_id)
            print(erasmia_id)

        except Exception as e:
            # unable to retrive data
            print(e)
            self.send_error(500, "Unable to retrive data from diagnosis")

        else:
            # retrived data without expection
            self.send_response(200)
            self.send_header("Content-type", "application/json")
            self.end_headers()

            self.wfile.write(json.dumps(data).encode('utf-8'))


if __name__ == '__main__':
    # parser argument
    parser = argparse.ArgumentParser()
    parser.add_argument('port', action='store',
                        default=8000, type=int,
                        nargs='?',
                        help='Specify alternate port [default: 8000]')
    parser.add_argument('--bind', '-b', default='', metavar='ADDRESS',
                        help='Specify alternate bind address '
                             '[default: all interfaces]')
    args = parser.parse_args()

    # init handlers
    heartDiagnosis=HeartDiagnosis()

    httpHandler = MyHandler
    httpHandler.protocol_version="HTTP/1.0"

    # init http demon
    httpd = HTTPServer((args.bind, args.port), httpHandler)

    # http demon status
    sa = httpd.socket.getsockname()
    print("Serving HTTP on", sa[0], "port", sa[1], "...")

    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
```

```
66          print("\nKeyboard interrupt received, exiting.")
67          httpd.server_close()
68          sys.exit(0)
```

4.2 Connection to FPGA

The DE1-SoC board used for hardware development in the project has an ARM processor which can access the FPGA by using the Lightweight Hardware Processor System(HPS)-to-FPGA bridge, which can be mapped to regions in the ARM memory space. After connecting the FPGA-side component to the bridge, memory-mapped registers within the memory region can be read or written by the processor [5].

In the code section below, /**dev**/**mem** is the system memory device file, which represents the physical memory of the computer system. The physical memory can be accessed at a offset address. The **mmap**(memory map) function maps the /**dev**/**mem** file into virtual memory, which means that physical addresses can be mapped to virtual addresses and programs are now able to access physical addresses. The two functions *write* and *read* defined in class **axi** will be called to inform the FPGA which sample is selected from the database and a 32 bit register value will be read back to terminal from the FPGA fabric.

```
1  class axi:
2      def __init__(self, addr, size):
3          self.addr = addr
4          self.size = size
5
6          self.mem = open('/dev/mem', 'r+b')
7          self.map = mmap.mmap(self.mem.fileno(), self.size, offset = self.addr)
8
9      def __del__(self):
10         self.map.close()
11         self.mem.close()
12
13     def read(self, addr):
14         'Read 4 bytes from register at addr'
15         self.map.seek(addr)
16         return struct.unpack('<L', self.map.read(4))[0]
17
18     def write(self, addr, data):
19         'Write data as 4 bytes to register at addr'
20         self.map.seek(addr)
21         self.map.write(struct.pack('<L', data))
```

5 Hardware

5.1 Hardware introduction

To implement a neural network on an FPGA we first needed to design the lower level of a neural network which is a single neuron in Verilog. A neuron can be mathematically represented as a sum of weighted inputs which then goes through an activation function. The activation function in our model is a ReLU activation function, which outputs the input itself if this is positive

[5] ftp://ftp.intel.com/pub/fpgaup/pub/Intel_Material/16.1/Tutorials

and outputs 0 otherwise. The Verilog implementation of summing and multiplying by weights is shown in the code section below. The parameters W0X-W4X represent the weights which the inputs to the neuron are multiplied by, these values lie within -2 and 1.984375, which is the range of values representable with our chosen fixed-point format, which is Q2.6[6].

The wires sum0x-sum4x are assigned the product of each input and its respective weight. The factors have Q2.6 format, thus the products have Q4.12 format (16 bits). This operation was originally done by declaring the factors as 'signed' in Verilog and using the built-in DSP units for signed multiplication. However, this required too many resources on the board. Currently, the multiplication is carried out by sign-extending the 8-bit inputs to 16-bits and selecting the 16-LSB of the 32-bit product as the result. This way a signed multiplication can be done using unsigned arithmetic and requiring fewer resources. Although this solution performs the operation correctly, it might be suboptimal from a resource usage point of view, due to the use of extra bits, especially in the case that all operations could be performed in DSP blocks if there are enough of these. A more economical multiplication method could be further investigated.

sumout stores the value of the all inputs multiplied by their weights summed together (plus the bias). It uses the smallest number of bits that allows for no overflow to possibly happen in the biggest layer of the network, given the size and the quantity of the operands of the addition. In this network, the layer with the most inputs has 187 inputs. This means that in that layer, **sumout** is the sum of 187 products of 8-bit numbers and one 8-bit bias (converted to Q4.12), meaning that no overflow can occur if using at least 23 bits. The **sumout** of each layer uses the number of bits required by the biggest layer. This is slightly inefficient although of negligible effect. On line 26 you can see a reset is implemented to reset the inputs.

```
1  module node_2_1(clk,reset,N1x,A0x,A1x,A2x,A3x,A4x);
2    input clk;
3    input reset;
4    input [7:0] A0x, A1x, A2x, A3x, A4x;
5    reg [7:0] A0x_c, A1x_c, A2x_c, A3x_c, A4x_c;
6    wire [15:0] sum0x, sum1x, sum2x, sum3x, sum4x;
7    output reg [7:0] N1x;
8    reg [22:0] sumout;
9
10   parameter [7:0] W0x=8'd62;
11   parameter [7:0] W1x=-8'd36;
12   parameter [7:0] W2x=-8'd44;
13   parameter [7:0] W3x=8'd40;
14   parameter [7:0] W4x=8'd0;
15   parameter [15:0] B0x=16'd512;
16
17
18   assign sum0x = {A0x_c[7],A0x_c[7],A0x_c[7],A0x_c[7],A0x_c[7],A0x_c[7],A0x_c[7],
       A0x_c[7],A0x_c}*{W0x[7],W0x[7],W0x[7],W0x[7],W0x[7],W0x[7],W0x[7],W0x[7],W0x};
19   assign sum1x = {A1x_c[7],A1x_c[7],A1x_c[7],A1x_c[7],A1x_c[7],A1x_c[7],A1x_c[7],
       A1x_c[7],A1x_c}*{W1x[7],W1x[7],W1x[7],W1x[7],W1x[7],W1x[7],W1x[7],W1x[7],W1x};
20   assign sum2x = {A2x_c[7],A2x_c[7],A2x_c[7],A2x_c[7],A2x_c[7],A2x_c[7],A2x_c[7],
       A2x_c[7],A2x_c}*{W2x[7],W2x[7],W2x[7],W2x[7],W2x[7],W2x[7],W2x[7],W2x[7],W2x};
21   assign sum3x = {A3x_c[7],A3x_c[7],A3x_c[7],A3x_c[7],A3x_c[7],A3x_c[7],A3x_c[7],
       A3x_c[7],A3x_c}*{W3x[7],W3x[7],W3x[7],W3x[7],W3x[7],W3x[7],W3x[7],W3x[7],W3x};
22   assign sum4x = {A4x_c[7],A4x_c[7],A4x_c[7],A4x_c[7],A4x_c[7],A4x_c[7],A4x_c[7],
       A4x_c[7],A4x_c}*{W4x[7],W4x[7],W4x[7],W4x[7],W4x[7],W4x[7],W4x[7],W4x[7],W4x};
23
24   always@(posedge clk) begin
25
```

[6]Qx.y format refers to a fixed point format with x integer bits and y fractional bits (e.g. 00.000000 has Q2.6 format).

```
26      if(reset)
27        begin
28        N1x<=8'd0;
29        sumout<=16'd0;
30        A0x_c  <= 8'd0;
31        A1x_c  <= 8'd0;
32        A2x_c  <= 8'd0;
33        A3x_c  <= 8'd0;
34        A4x_c  <= 8'd0;
35        end
36      else
37        begin
38        A0x_c  <= A0x;
39        A1x_c  <= A1x;
40        A2x_c  <= A2x;
41        A3x_c  <= A3x;
42        A4x_c  <= A4x;
43        sumout<={sum0x[15],sum0x[15],sum0x[15],sum0x[15],sum0x[15],sum0x[15],sum0x
        [15],sum0x}+{sum1x[15],sum1x[15],sum1x[15],sum1x[15],sum1x[15],sum1x[15],sum1x
        [15],sum1x}+{sum2x[15],sum2x[15],sum2x[15],sum2x[15],sum2x[15],sum2x[15],sum2x
        [15],sum2x}+{sum3x[15],sum3x[15],sum3x[15],sum3x[15],sum3x[15],sum3x[15],sum3x
        [15],sum3x}+{sum4x[15],sum4x[15],sum4x[15],sum4x[15],sum4x[15],sum4x[15],sum4x
        [15],sum4x}+{B0x[15],B0x[15],B0x[15],B0x[15],B0x[15],B0x[15],B0x[15],B0x};
44
45        if(sumout[22]==0)
46          begin
47          if(sumout[21:13]!=9'b0)
48            N1x<=8'd127;
49          else
50            begin
51            if(sumout[5]==1)
52              N1x<=sumout[13:6]+8'd1;
53            else
54              N1x<=sumout[13:6];
55            end
56          end
57        else
58          N1x<=8'd0;
59        end
60      end
61  endmodule
```

Listing 1: Verilog implementation of a neuron.

The output of this multiplication and summing is then passed through the ReLU activation function which is shown the code section above at code line 45-61. This code sets negative values to output binary all 0. It also prevents overflow by setting any values too large to lie inside of the fixed-point format (sets them to the maximum=1.984375). This clipping feature was added to both the software and hardware implementations to prevent them from handling overflows differently and being able to assume the equivalence of the two.

sumout has in this case format Q11.12, thus we take the bits [13:6] as the output of the neuron to be consistent with the Q2.6 binary format that we are using. When two 8-bit Q2.6 numbers are multiplied the results will need to be truncated correctly to keep the correct decimal format at the output. In order to round to the closest number, when sumout[5] is 1, the output adds one to the bits [13:6].

The neurons make up neuron layers. The number of layers and nodes within the layer also depends on the specific model being recreated on the hardware, this means another automated

script was required to generate the Verilog code for each layer.

```
module layer_2(reset,clk,N1x,N2x,N3x,N4x,N5x,N6x,N7x,N8x,N9x,N10x,R0x,R1x,R2x,R3x,
    R4x);
  input reset, clk;
  output [7:0] N1x,N2x,N3x,N4x,N5x,N6x,N7x,N8x,N9x,N10x;
  input [7:0] R0x,R1x,R2x,R3x,R4x;

  node_2_1 node_2_1(
    .A0x(R0x),
    .A1x(R1x),
    .A2x(R2x),
    .A3x(R3x),
    .A4x(R4x),
    .clk(clk),
    .reset(reset),
    .N1x(N1x)
  );
  ...
endmodule
```

Listing 2: Verilog implementation of a fully connected layer.

The top-level entity of our design instatiates the different layers and declares the inputs and outputs of the model:

```
module top(clk, reset, out0, addr);
  input clk, reset;
  output [7:0] out0;
  input [5:0] addr;

  wire [1496:0] l1;
  wire [7:0] l2_0, l2_1, l2_2, l2_3, l2_4;
  ...

  ROM rom_in(.address(addr),.clock(clk),.q(l1));
  layer_1 layer1(reset, clk, l2_0, l2_1, l2_2, l2_3, l2_4, l1[1495:1488], l1
    [1487:1480], l1[1479:1472], ..., l1[7:0]);
  layer_2 layer2(reset, clk, l3_0, l3_1, l3_2, l3_3, l3_4, l3_5, l3_6, l3_7, l3_8,
     l3_9, l2_0, l2_1, l2_2, l2_3, l2_4);
  ...
endmodule
```

Listing 3: Top-level entity.

5.2 Choosing a design

Network considered	Problem
32-bit floating point	Too large
16-bit floating point	Too large
32-bit fixed point	Too large
24-bit fixed point	Too large
24-bit fixed point downsampled input signal	Too large and the accuracy drops off too much
8-bit fixed point	No problem
8-bit fixed point reusable blocks	Uses fewer resources but speed is reduced significantly

Table 2: Problems encountered with different number formats

There are multiple ways to implement a neural network on an FPGA. All the types we explored are 32-bit floating-point IEEE-754 standard, 16-bit floating point, 16-bit fixed point with Q3.13 format and 8-bit fixed point Q2.6 format.

The floating-point implementations were considerably more demanding in terms of space on the board as they use more ALUs (Arithmetic Logic Units) to perform the floating-point addition and multiplication. This meant that the 32-bit floating point implementation used over double the amount of ALUs we had on the board. The 16-bit implementation was also too large for the board. Both implementations worked in simulation but were not realisable.

The 16-bit implementation stored all the inputs and weights as 16-bit numbers in the Q3.13 format but the arithmetic used less space than the floating-point implementations. This 16-bit model proved to also be too large for the board. The only option was to use the 8-bit implementation which uses 80% of the ALUs on the board. We discovered that the loss in accuracy in the 8-bit is negligible compared to the 16-bit method. This was shown by the fact that the 8-bit model has an accuracy of 93% which is very close to the accuracy of the non-quantised CPU model.

An additional trade off was discovered near the end of our project. This is the trade-off between space on the board and the speed of the network. We developed reusable neuron blocks which would reduce the amount of ALU the network uses, we found that we could reduce the space on the board from 80% to 18%.

The code below shows the modifications applied to the layer module in order to use reusable neuron blocks. In this implementation, only one neuron module is instantiated in each layer and the weights of the different neurons of the layer are loaded sequentially into the module from a ROM, thus the outputs of the different neurons are obtained sequentially. While this implementation uses a smaller area, it is a lot slower due to obtaining the outputs of neurons sequentially and because of loading weights from the ROM, requiring a variable 'count' to switch between neurons every a fixed amount of time (the code is not optimised for the smallest possible time interval). A better trade-off of area and speed could be to implement reusable layer blocks instead of reusable neuron blocks, which is described in the 'Future development' section.

```
    initial address = 4'b0;
  initial count = 3'b0;

  ROM_params_2 rom_params(.address(address),.clock(clk),.q(w));
  node_2 node_in(clk,reset,tmpout,in0,in1,in2,in3,in4,w[47:40],w[39:32],w[31:24],w
    [23:16],w[15:8],w[7:0]);

  always @(posedge clk) begin
    count <= count + 3'b001;
    if (count == 3'b000) begin
      if (address == 4'd9)
        address <= 4'd0;
      else
        address <= address + 4'd1;
      case (address)
        4'd0  :  out0<=tmpout;
        4'd1  :  out1<=tmpout;
        4'd2  :  out2<=tmpout;
        4'd3  :  out3<=tmpout;
        4'd4  :  out4<=tmpout;
        4'd5  :  out5<=tmpout;
        4'd6  :  out6<=tmpout;
        4'd7  :  out7<=tmpout;
        4'd8  :  out8<=tmpout;
```

```
24          4'd9   :   out9<=tmpout;
25          default :   out0<=tmpout;
26        endcase
27      end
28    end
```

Listing 4: Modifications applied to the layer module in order to use reusable neuron blocks

5.3 Simulations

Many simulations were recorded of different implementations. To test if the network was working, we would test an individual neuron with random numbers and check the output is as expected. Once we know a neuron is working, we then test a whole layer. The testing was done on the Altera University Program Simulator. Take note only the final tests show the final implementation, all the simulations are proof that other bit implementations worked in simulation.

5.3.1 32-bit floating point adder and subtractor

For our first implementation, which turned out to be too large for the FPGA board, we tested the floating-point adders and subtractors. I1x and I2x are the inputs and 03x is the output.

-2+4=2 (testing case when I1x is negative and I2x is positive):

Figure 5: Positive test case

-1.234-8.5= -9.73399925232 (Both negative case):

Figure 6: Both negative case)

100-100= 3.81469726562e-06 (second input is negative case):

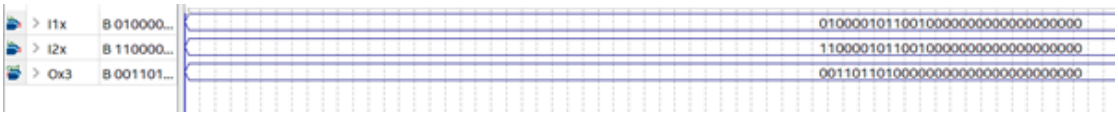

Figure 7: Second input is negative

This shows that our floating-point addition and subtraction worked.

5.3.2 32-bit floating-point neuron

This tests a whole neuron. The table next to the block diagram shows the weights within the neuron.

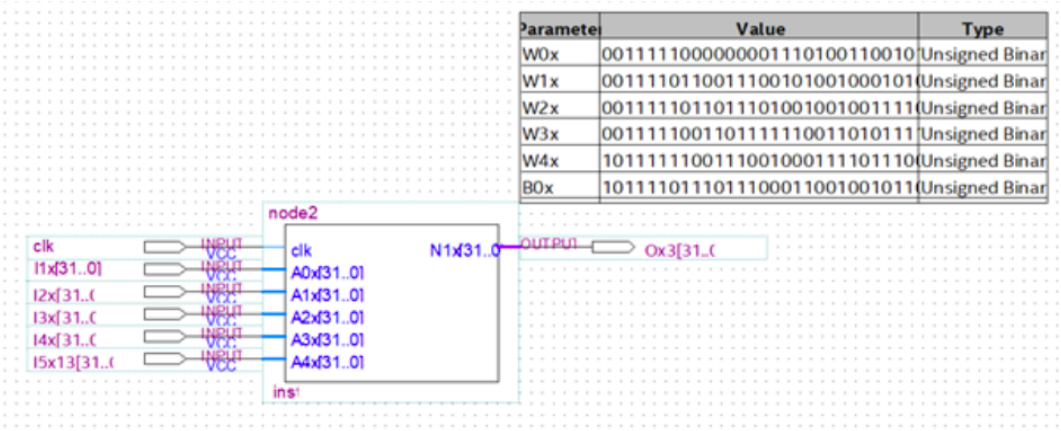

Figure 8: Block diagram to test the whole neuron

Input	Expected output before ReLU (0 after, if negative)
0.4 / 00111110110011001100110011001101 0.25 / 00111110100000000000000000000000 100 / 01000010110010000000000000000000 3 / 01000000010000000000000000000000 1000 / 01000100011110100000000000000000	-679.27 / 11000100001010011101000101010111

Table 3: Negative output test case

The waveform outputs Ox3 shows an output of zero which is as expected.

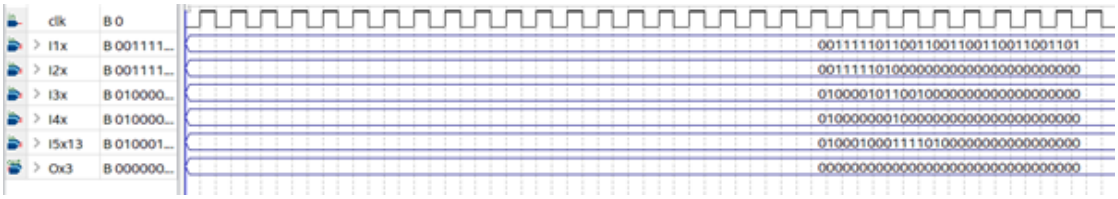

Figure 9: Negative output waveform for a 32-bit neuron

Here is a test case with a non-zero output.

The output waveform is very close to the expected output. There is a difference in the order of magnitude of 10^{-3}. This is potentially caused by overflow errors. Due to the small difference in the outputs it makes no impact on the accuracy on the final output of the whole system.

Input	Expected output before ReLU (0 after, if negative)
0.5 / 00111111000000000000000000000000 10 / 01000001001000000000000000000000 -5 1.234 -100	71.09 / 01000010100011100010110000010100

Table 4: Test case with a non-zero output

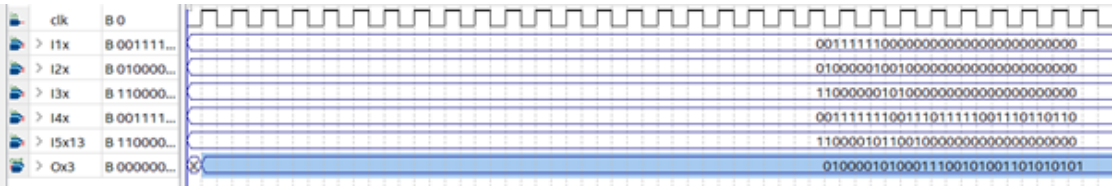

Figure 10: Positive output waveform for a 32-bit neuron

5.3.3 Layer 1 test with Q3.5 format(000.00000) and 8-bit input/16 bit overflow

Expected outputs

```
Node_1:-0.4065424180;          Q3.5 format binary:1111111111110011

Node_2:0.1691306406;           Q3.5 format binary:0000000000000101

Node_3:0.7398626498;           Q3.5 format binary:0000000000010111

Node_4:-0.2039555064;          Q3.5 format binary:1111111111111010

Node_5:0.0506941965;           Q3.5 format binary:0000000000000001
```

Quartus outputs

Node_1= 0 as expected for a negative number

Node_2=0.09375

Node_3=0.625

Node_4= 0

Node_5= 0.0625

Figure 11: 8-bit Q3.5 format layer expected outputs

The problem we encountered here was the fact that we were not using enough overflow bits and did not have a special case to set the output to the max value in the case of an overflow, in our final implementation we had the intermediary steps as 24 bit to prevent overflow and had the special case. We also found that using the q format Q3.5 was unnecessary because most values lie within –1 and 1 so we need more decimal bits. This because at this point we were not using quantisation aware training so we had to choose the format depending on the range of values of the weights of the network. This was not an issue in the final implementation as when using quantisation aware training, the weights are a consequence of the chosen format.

5.3.4 Layer 1 with Q6.26 with 16 bit input

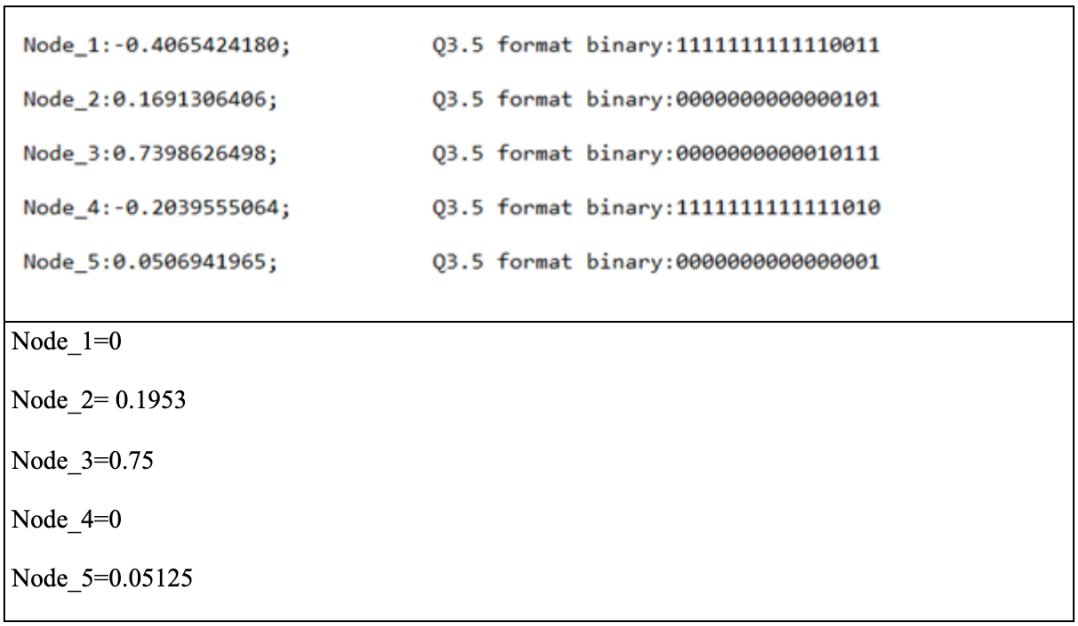

Figure 12: Output waveform for q6.26 with 16 bit input

This 16-bit implementation would have worked but like the floating-point implementations we discovered it would not fit on the board.

5.4 Testing the final implementation

Figure 13 shows the results of the final 8-bit implementation which was placed on the board. The results are taken from the output of the board rather than Quartus simulations as shown for the implementations which could not fit on the FPGA board. 50 healthy datasets and 50 unhealthy datasets were placed on the board for testing. Out of the 100 datasets, the neural network only output 7 incorrect values placing the accuracy at 93%. This agrees with the software model which had an accuracy of 94% when tested with thousands of datasets. The software model is made to behave in exactly the same way as this hardware implementation, thus it is safe to assume if tested further the two accuracies would be identical. The output in figure 13 are on a 7-segment display. Figure 13.1 shows the output of a dataset which is healthy, and the output is '0' as expected. Figure 13.2 shows the output of an ECG signal which has arrhythmia and the output is '1' as expected. The switches below the 7-segment display refer to which ECG signal in the ROM is being accessed.

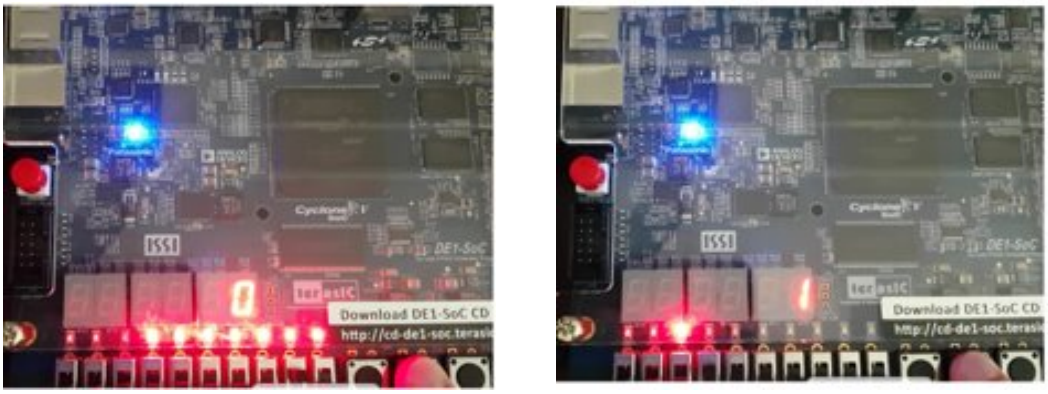

Figure 13: Implementation on the board

5.5 Power consumption of the hardware

The power consumption of the hardware was estimated using a PowerPlay Early Power Estimator from Altera. This tool takes in input the type of device being used and the number of resources used by the design, which can be found by reading the compilation report of the design. PowerPlay estimates that our current design has the FPGA consuming 0.507W while the whole SoC consumes approximately 0.861W. This is low considering the low latency and potentially high throughput of the design. By comparison, the GPU used on Colab (Nvidia Tesla P100), performed inference of our network with an average latency of around $200\mu s$[7] (our FPGA design achieves a $0.16\mu s$) and it is a very big and expensive GPU with a maximum rated power of 250W.

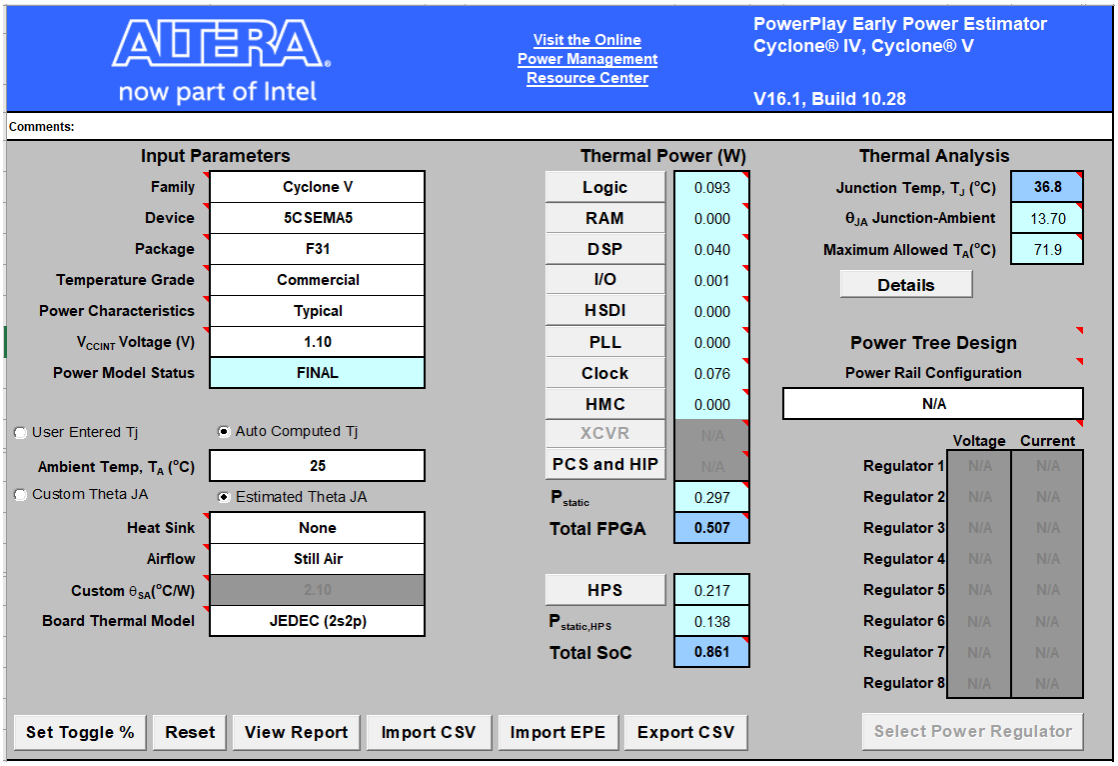

Figure 14: Screenshot of the main screen of Altera PowerPlay Early Power Estimator.

[7] this value might be strongly overestimated as it was calculated as the time taken to do inference of a whole test dataset, including function calls, etc... and by then dividing by the size of the dataset.

5.6 Speed of the hardware

The hardware has a maximum clocking speed of 50MHz, and our model only required 8 clock cycles to complete a classification which means our hardware network takes 160ns. Compared to our software model, which took roughly $200\mu s$ on GPU, as described in section above, this is a great improvement in the latency of the network. Furthermore, our hardware is not limited to the 50MHz speed. A potential future development could be to overclock the hardware to get it to operate closer to its timing constraints which would increase the speed of the network further. Pipelining could also be implemented as described in the 'Future development' section, achieving the extremely high throughput of one result per clock cycle.

5.7 Cost of the project

Prototyping cost:

- DE1-SoC board: $175 (price for academic istitutions).

Production cost:

- A single Cyclone V FPGA costs £40 on RS-components. This price may be lower when buying in bulk.
- If a portable device using our design were to be produced, other material costs would incur, such as for batteries, displays, etc... However, this is outside of the scope of the project.

Regulation cost:

- Hiring a company to deal with regulations (see ethical consequences section for more detail): Outsource cost variable but one off payment, updating the model or adding more features would require us to be in line with regulation again and incur extra costs.

5.8 Bridge

The Cyclone V DE1-SoC FPGA has both the FPGA fabric and a Hardware Processor System (HPS) onboard. The FPGA fabric can be programmed by the Raw Binary File (extension name rbf) by Quartus Prime. Fig 15 below indicates the device peripherals and the how they are connected.

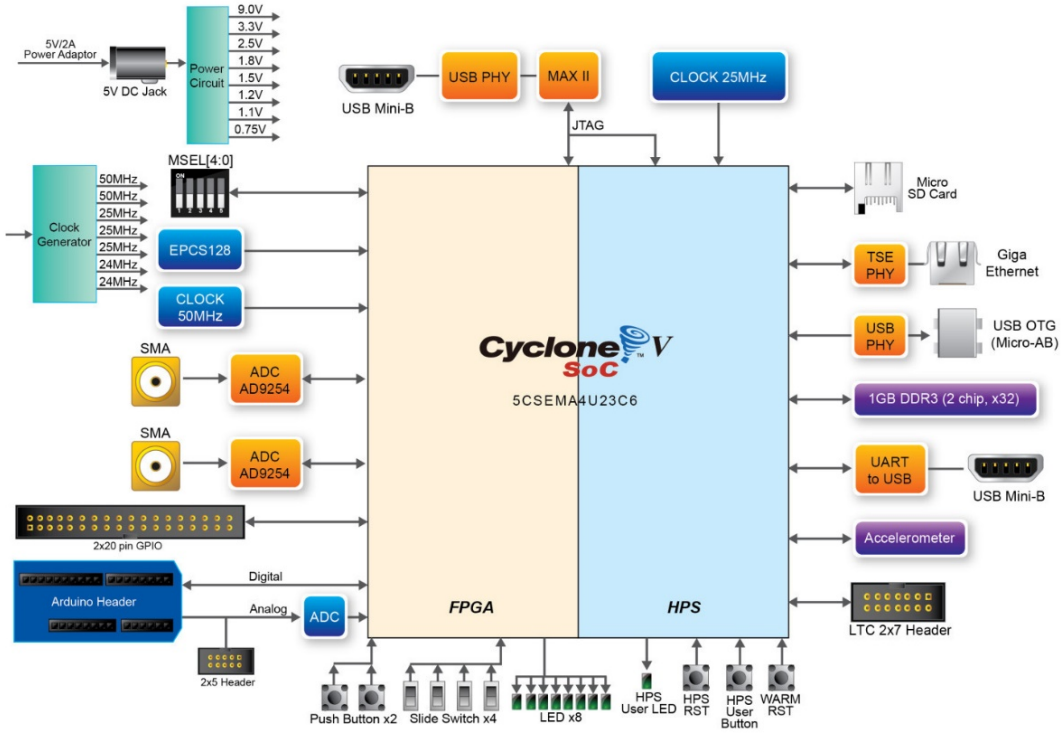

Figure 15: Connection between the device peripherals

The FPGA device has a CPU onboard which supports Linux Kernel. The initial approach was to boot Linux from an SD card containing the operating system image. This fully functional kernel supports ethernet connection which gives us the chance of running a backend server programme on the device. The server should be capable of handling HTTP web requests with its corresponding JSON carriage. As shown in the interface section, the test case index can be transferred from the user to the FPGA fabric. The results from the FPGA would then be retrieved back and displayed to the user by a JSON text dump. This work was not fully implemented due to the limited available time. However, it was shown that connecting the FPGA fabric with the CPU can be done by setting up the memory addresses via the Platform Designer in Quartus Prime. Hardware synthesis involves many low-level functional blocks that are not obvious to programmers. Hence, a Golden Reference Design file [8] ghrd_top.v is used to ease the development process. A typical design file (extension name qsys) is shown in Fig 16. A master-slave relationship can be seen between the HPS and the FPGA. Thus, FPGA memory blocks with the specified memory addresses can be read/written via the Low-Weight HPS2FPGA bridge. Specifically, the physical memory address of this bridge is defined as 0xFF20_0000. In order to access the registers that are relative to this address, an offset is required by the user. Detailed information can be referenced in the interface section, where control functions and tools used are explained.

[8]The GSRD file can be referenced by the following link: `https://rocketboards.org/foswiki/Documentation/GSRD`

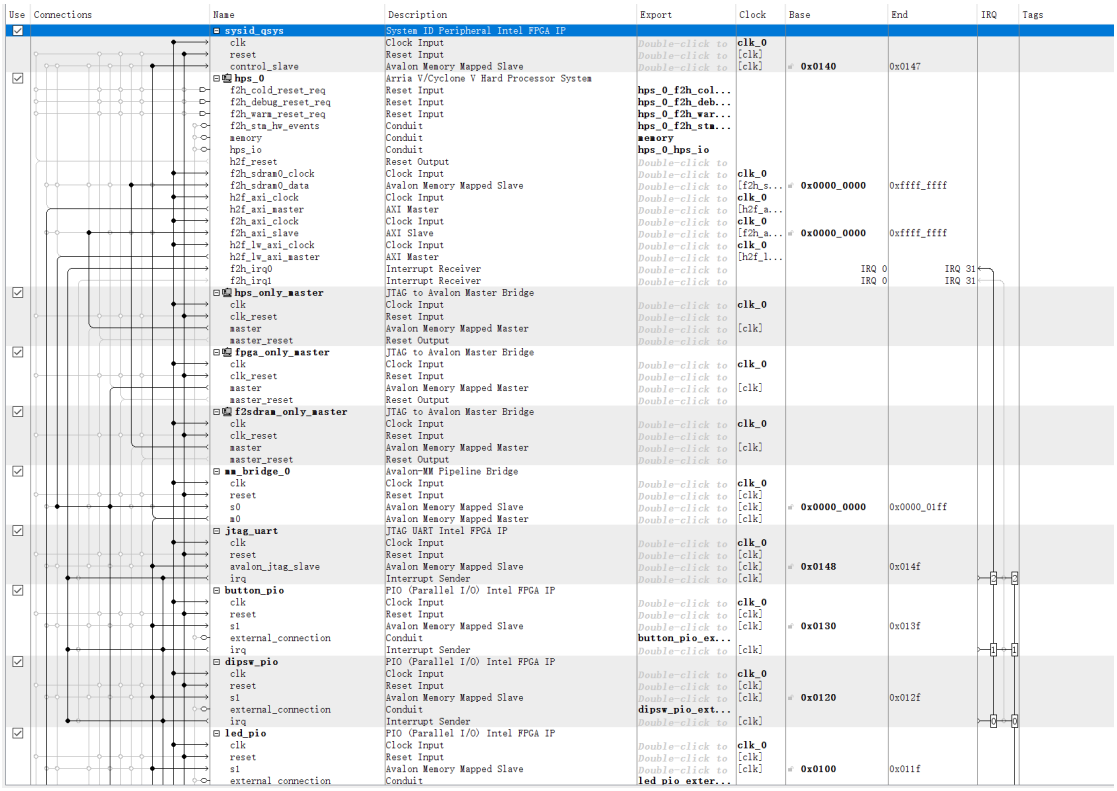

Figure 16: Golden Reference Design

6 Ethical consequences

Selling a medical device in the UK requires a certificate from the Medicines and Healthcare products Regulatory Authority (MHRA). The MHRA enforces criteria known as Medical Device Regulations. Before the device is deployed, trials will have to be conducted to prove that the device is of a high enough standard to be sold as a medical device[7]. Our device would be considered a Class I device due to being non-invasive. The only risk comes in the form of misdiagnosis thus it will be important to stress that the neural network does not have a 100% accuracy and is meant to assist medical professionals rather than replacing them. Improving the accuracy is important for future development. Because our device is Class I we would have to contact the relevant authority within the MHRA and follow their specific assessment procedure. After contacting a clinical scientist, he gave us some guidance on how we would get a license to sell to the NHS. Any software or hardware which stores or deals with patient data is classified as a medical device according to the MD Regulation 2017/745 Article 2. We would need to contact two companies, one to get our CE marking (New MDR changes state even software needs CE certification) and another to make a Quality Management System which records all changes made to the device throughout its lifecycle. A user manual would need to be developed in order to get our CE mark and QMS. Due to Brexit this regulation is also in a state of flux.

Due to the nature of the project there are many ethical considerations to be made. The main ethical issue we faced during our project was the fact that we were not authorized to use our

device as a medical device, so we used pre-existing datasets to test our device rather than taking ECG signals from real patients. As the accuracy of our network is high but not high enough to make a sure diagnosis it would be unethical to test it on real people as a misdiagnosis could cause a panic or reinforce the patient's anosognosia. By referring people to a professional this risk can be avoided.

7 Sustainability

The project is more sustainable than other machine learning implementations because it requires less power to run. A FPGA can be reprogrammed which means the product can be updated for new models, this will enhance the longevity of the device as a new device will not need to be developed for every new addition to the new neural network model. We would only outsource our manufacturing to companies which rank highly in the sustainability in the sustainability industry report[8]. The device will be reusable and recyclable, the ability to update the model makes this viable.

The lifecycle of the product is the main sustainability consideration. There are many different approaches to life cycles. We have been operating on an agile (minimum viable product) model as our device is a proof of concept. This model would not be acceptable to keep our device on the market for long periods of time. We would need to plan future updates and improve the model on the device through time to ensure it does not become obsolete and serves its purpose for the longest period. It is hard to give an estimate of the lifespan of the device, but a fair guess would be in the order of 10 years or longer with updates, the health service want to make investments, they are not as interested in short term solutions.

Another sustainability consideration is the effect of reducing the burden on healthcare professionals. Healthcare is stretched thin in the UK, so any device which can free up time for diagnosis will allow workers to focus on other issues. This relates to sustainability because the more stressed the NHS is, the more funding the government will need to funnel into it, which can reduce funding on other environmental departments within the government. If healthcare professionals can use this device to monitor patient's health from a distance it will prevent unnecessary travel improving the patients' wellbeing.

8 Future development

Our network's design on FPGA could be optimised in different ways depending on which aspect has the priority (latency, throughput, power consumption, area).

One possible improvement is the implementation of pipelining in the design. Pipelining operates on the principle that each layer has as inputs the outputs of the previous layer, meaning that once the outputs of layer_n have been fed into the next layer_n+1, layer_n can start processing another input without affecting the results of the previous one. Each layer's processing takes one clock cycle, thus pipelining could achieve a very high throughput of one output every clock cycle. Chip area utilisation reduction is possible by implementing more general network layers in a folded design. As shown is in Figure 17, only one layer would need to be configured in the board. The outputs would feed back as inputs a number of times depending on the number of layers, changing the layer weights at each iteration to represent the different layers of the network. Such an implementation would increase latency and would make pipelining impossible, reducing

throughput. However, the reduced area utilisation would make it possible to implement bigger, more capable and accurate networks such as the ResNet described in the 'Software' section. This would allow to update our design to detect more diseases.

It could be argued that in a portable device, latency and throughput are not important since data is acquired at a very slow rate and there is no strict requirement for latency. However, a low latency combined with a high throughput rate could be used to extend battery life. The FPGA could be momentarily turned on to quickly process data in batch at regular intervals, instead of always being on. When an FPGA turns on there is a rush of current to establish bias conditions, meaning that the power consumption is momentarily higher, therefore further investigation would be required to establish if this approach would be feasible and what kind of interval would be required between turning off and on.

Other improvements, include making our design work with data collected in real time rather than with dataset, as the goal of this project is to provide a base for the development of an instrument to perform real-time automatic ECG monitoring.

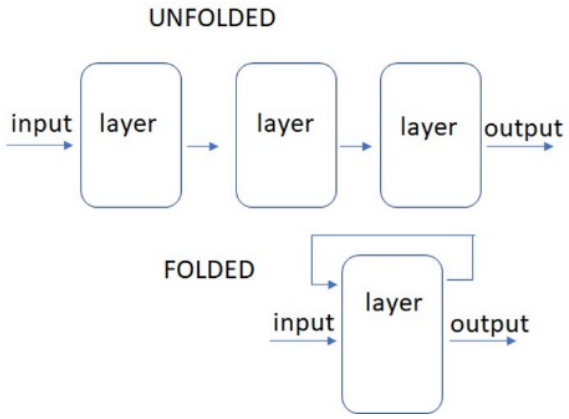

Figure 17: Block diagram of folded and unfolded designs.

9 Conclusion

The product fits the brief well. The client criteria were to make an affordable and portable heart disease detection device to aid medical professionals outside of working hours. Our device has been proven to be affordable and portable. Several steps would have to be made to take the device into production, these have been outlined in this report. The future developments section gives a good review of the steps which can be taken to make our proof of concept into a better product. The addition of convolutional layers, real time detection, increased accuracy and more disease detection (more than just arrhythmia) would be the priority if taking this product to market. As discussed in the ethical considerations section, contact would also need to be made with the MHRA to understand the process of acquiring a license to sell a medical related device. Our contact working within the NHS as a clinical scientist outlined the steps, we would need to take sell our product as a medical device.

Our project is innovative because we have developed the first framework to build fully connected neural networks on hardware in Verilog code. The flexibility of the framework is its

strength. There is no open source framework which builds neural networks on hardware from the ground up, most are a form of High-Level Synthesis which operate at a much higher level of abstraction and don't allow for as much control over the design of the specific network. The scope of our project is not limited to ECG disease detection. If anyone wishes to take the project further, the GitHub repository is well documented.

References

[1] J. Pan and W. J. Tompkins, "A real-time qrs detection algorithm," *IEEE Transactions on Biomedical Engineering*, vol. BME-32, no. 3, pp. 230–236, 1985.

[2] D. Alhela, K. A. I. Aboalayon, M. Daneshzand, and M. Faezipour, "Fpga-based denoising and beat detection of the ecg signal," *IEEE*, 2015.

[3] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[4] G. Sannino and G. D. Pietro], "A deep learning approach for ecg-based heartbeat classification for arrhythmia detection," *Future Generation Computer Systems*, vol. 86, pp. 446 – 455, 2018.

[5] P. Wagner, N. Strodthoff, R.-D. Bousseljot, D. Kreiseler, F. I. Lunze, W. Samek, and T. Schaeffter, "Ptb-xl, a large publicly available electrocardiography dataset," *Scientific Data*, vol. 7, p. 154, May 2020.

[6] A. H. Ribeiro, M. H. Ribeiro, G. M. M. PaixÃ£o, D. M. Oliveira, P. R. Gomes, J. A. Canazart, M. P. S. Ferreira, C. R. Andersson, P. W. Macfarlane, W. Meira Jr., T. B. SchÃ¶n, and A. L. P. Ribeiro, "Automatic diagnosis of the 12-lead ecg using a deep neural network," *Nature Communications*, vol. 11, p. 1760, Apr 2020.

[7] "An overview of medical device regulations in the uk,"

[8] "Key sustainability issues in the electronics industry: Sustainability industry report,"