QBART REFERENCE MANUAL

Version 1.0

Alf Martin Eggan
Alice Gudem
Finn Inderhaug Holme
Fredrik Ås
Håkon Flatvål
Karl Andreas Eggan
Kris Monsen
Ole Kristian Pedersen
Oscar Belgau

November 30, 2017

Contents

1 Introduction to QBART

1.1 System Summary

QBART is an accelerator for Quantized Neural Network (QNN) inference computations. It utilizes one or several PYNQ-Z1s with their on-board dual Cortex-A9 CPUs and hardware components synthesized to the on-board FPGA, to accelerate the various matrix operations that take place in the inference process.

The user inputs a valid QNN and a set of one or several images. The system returns the inferred classification as a `results.csv` file.

1.2 Application

QBART can accelerate QNN-related computations on a general basis by using bitstream approaches, including the GEMMBitserial for bitstream matrix computations.

This means that the accelerator is highly general when it comes to QNNs, and has been tested successfully on the MNIST and GTSRB test sets.

A possible use is for image recognition in mobile devices, as QNNs are generally more energy-efficient than standard Artificial Neural Nets. Since QBART is also easily scalable, it can also be used in a server setup to efficiently inference a large set of images, given a trained QNN for the purpose.

1.3 Assumptions

QBART does not train QNNs, but executes already trained QNNs on a set of images in an efficient manner. QBART guarantees correct execution of a given QNN, but the classification accuracy is still entirely dependent on how the given QNN is trained.

Images are assumed to be of the same file type with the same data layout and color channel ordering.

The QNNs that are used in QBART must be constructed by using the included `layers.py` to create a sequence of layer objects that QBART is to execute, and then "pickled" using Python 2's built-in `pickle` module.

2 QBART Setup

3 Setup everything from scratch

In order to run a fully functional version of the system, including development on-board, it is essential to follow these steps.

3.1 Prepare a microSD-card

Download the PYNQ-Z1 v1.4 image from `https://files.digilent.com/Products/PYNQ/pynq_z1_image_2017_02_10.zip` It is important to use the exact version, as later versions have changed some critical functionality that causes system components to break.

Use an image burner (such as Win32 Disk Imager for Windows, or the `dd` utility for Unix-like systems) to clone the image to a microSD-card (minimum 8GB).

Then plug the SD-card into the PYNQ-board, with the board powered off.

3.2 Connect the PYNQ to a network

It is desirable to connect the PYNQ to a network. For one, it enables easy updating of Ubuntu packages, and it is also crucial if one wishes to use the distributed computing opportunities that comes with several PYNQ boards.

The PYNQ-Z1 comes with a 1G-Ethernet connection. How one configures the network will depend on your application.

A general recommendation is to put the QBART PYNQs behind a closed network, as there has been little to no security patching for the board, we cannot guarantee safe operation in an open network.

The tested network configuration was a cheap wireless router with several ethernet ports. The PYNQs were hooked directly up to the router, and the router ran a DHCP service. The PYNQs were given static IPs, for easier usage in distributed computing.

3.3 Starting the board

Plug the PYNQ-board into a wall socket with a power adapter, as USB power will likely not be sufficient to power the board.

Make sure that jumper `JP5` is set to `REG` for power adapter and that `JP4` is set to `SD`, so that we use the SD-card we just prepared.

Next, power up the board by clicking the power switch. The button LEDs should blink after 30-60 seconds. If not, something probably went wrong when writing the image (did you remember to un-mount before removing the SD-card?).

3.4 Connecting to the PYNQ Board

There are several ways to connect to the PYNQ-board, if an Ethernet connection is in place. Either via a direct connection between the PYNQ and your computer, or via a network.

The default username and password is `xilinx` - we'll change this shortly. Root also has `xilinx` as password.

Most of the following is a condensed version of the PYNQ-Z1 documentation [1].

Direct Ethernet Connection

With an ethernet cable connected to the PYNQ and to your device, make sure the board is powered on, open the terminal and run:

```
arp | grep "<iface-name>"
```

Identify the IPv4 address and login:

```
ssh xilinx@{ip}
```

NB! There are cases where the PYNQ only gets assigned an IPv6 address, and you'll have to use other utilities to determine an IPv6 address.

USB Serial Connection

Setup via ethernet can be a bit of a hassle the first time. USB Serial can be a bit more reliable when setting up the PYNQ for QBART.

First, install `minicom`:

```
sudo apt-get install minicom.
```

With the board powered on, connect a cable from the computer to the microusb port labeled `PROG / UART`.

Find the correct serial port for the board, active boards can be found by using

```
dmesg | grep tty
```

It is usually `ttyUSB0` or `ttyUSB1`.

Start `minicom` with

```
minicom /dev/<device-name-from-step-3>
```

Configure `minicom` with Ctrl+A Z, then O for cOnfigure Minicom).

bps must be set to 115200, 8N1, HW and SW Flow control must be off ("No"). Exit setup.

Type something in minicom and press enter, you should see something like this:

```
xilinx@pynq:~\$
```

If not, you are on the wrong serial port.

When you are in the command prompt, use the ifconfig command to see the current network configuration.

Here you should see the ethernet connection if the PYNQ is hooked up to a network, and whether or not you have assigned an IPv4 or IPv6 address.

You can also just continue working on the PYNQ via serial, but ssh is usually less of a hassle in the long run, especially if you configure ssh keys for logging into the device.

Wireless Connection

To ease development, especially for those who lack an ethernet adapter on their development machines, we recommend configuring a wireless router with internet access, to which you can hook up the PYNQ-boards via ethernet.

It is wise to assign static IPs to the PYNQ-board(s).

If you are using Ubuntu or similar, you can edit your `/etc/hosts` file to include this routing information.

An example of the `/etc/hosts` file with IP addresses filled in:

```
127.0.0.1       localhost
127.0.1.1       <your computer hostname here>

192.168.1.2     PYNQ1
192.168.1.4     PYNQ2
192.168.1.5     PYNQ3
192.168.1.7     PYNQ4
```

Now you can simply SSH into, for example PYNQ1, by entering: `ssh xilinx@PYNQ1`

And if you wish to connect to the Jupyter Notebook, go to the following URL in your preferred browser. `http://PYNQ1:9090`

3.5 Ubuntu customization

The PYNQ image uses a custom built variant of Ubuntu 15.10, which has been deprecated for quite some time. This has led to issues when using apt, as the repos has been moved to archive. One possible solution is to change to the repositories of a previous LTS version of the same Ubuntu version.

```
sudo sed -i s/wily/vivid/g /etc/apt/sources.list.d/multistrap-wily.list
```

With proper repositories in place, we do some housecleaning:

```
sudo apt-get update && sudo apt-get upgrade
```

The image already comes with Jupyter Notebook, but it doesn't have the python2 kernel installed by default. This will cause issues if you run QNN tutorial notebooks from github user maltanar

(https://github.com/maltanar/qnn-inference-examples) on the board, as they are written in python 2, and you plan to reuse some of the code in your own Jupyter Notebook. So we'll install the python2 ipython kernel.

First we must install and upgrade pip, as the version included is quite old:

```
sudo apt-get install python-pip
sudo pip install -U pip
```

Then we use pip to install the python2 kernel, and register the python 2 kernelspec. Credit: https://github.com/jupyter/jupyter/issues/71

```
sudo python2 -m pip install --upgrade ipykernel && sudo python2 -m ipykernel install
```

Next, we'd like for security reasons to not use default passwords, as it is always an awful practice not to change them. We need to update both sudo and user, so we run:

```
sudo passwd && passwd
```

There is also an additional default password we need to get rid of: We need to change the password of the Jupyter server, reachable when the PYNQ is running at port 9090, as it is already running out of the box. From the PYNQ-docs: hashed password is saved in the Jupyter Notebook configuration file.

/root/.jupyter/jupyter_notebook_config.py

You can create a hashed password using the function IPython.lib.passwd(). Run these lines either in a python shell, or your preferred IDE, to create a hash:

```
from IPython.lib import passwd
password = passwd("secret") # Secret should be replaced by team password.
print(password)
```

This should print the hash. Copy and paste the hash.

You can then add or modify the relevant line in the jupyter_notebook_config.py file

```
c.NotebookApp.password =u'sha1:<Your hash here>'
```

Lastly, we would most likely want to give each PYNQ a unique hostname, especially if several PYNQs are to be used. Use the provided PYNQ-script:

```
sudo ./scripts/hostname.sh <NEW HOSTNAME>
```

Remember to reboot the board afterwards for the changes to take effect.

In addition, several other packages must be installed for the system to work:

```
sudo pip install --no-cache numpy pillow matplotlib cffi
```

3.6 Get the project on the PYNQ

Clone the github repository for this project locally to your computer:

```
git clone https://github.com/DMProAccelerator/qbart
```

First, copy the folder named `qbart_main` in the repository files. We want it in the `jupyter_notebooks` folder on the PYNQ.

```
scp -r path/to/local/copy/of/qbart_main xilinx@<pynq_ip>:jupyter_notebooks/
```

With the development files in place, setup a sudo crontab job for server.py by doing the following:

```
sudo crontab -e
```

And in the crontab file, add the following line:

```
@reboot python2 ~/path/to/server.py/in/qbart/folder >> qbart_server_crontab.log &
```

Then reboot. If any errors occur, it will be printed out to `qbart_server_crontab.log` in user xilinx's home folder.

Now, AT EVERY REBOOT, you have to run (in their respective directories):

```
sudo ./setclk.sh 50
python2 cffi_build.py
sudo ./load_bitfile.sh
```

setclk.sh is located in qbart_main/ cffi_build.py is located in qbart_main/rosetta/rosetta load_bitfile.sh is located in qbart_main/rosetta/rosetta

Setclk and cffi_build can be placed as crontab jobs as well, but we had difficulties executing load_bitfile.sh correctly.

3.7 Image cloning

It is a lot of work to repeat this process for every single board. In order to reduce the amount of work, one would ideally set up one board in the above manner, then clone the SD-card to an image file, and burn the image file to other PYNQ SD-cards. It is important to change the hostname after flashing to other boards, as having the same hostname on all boards can easily lead to confusion.

4 System operation

4.1 The jupyter notebook

The user interface of QBART is centered around a jupyter notebook (qbart_demo).

PYNQ is running a jupyter notebook server out of the box, and is easiliy reachable in its network via http://<PYNQ-IP>:9090.

The notebook is designed to be as compact as possible, with calls to several abstracting functions. These should reside in the same folder as the notebook, and are thoroughly commented. As long as you configure everything correctly and follow our set assumptions, everything should be running fine.

4.1.1 Notebook configuration

The user is expected to configure the clearly marked configuration section before running the system for the first time. This includes the following fields:

qnn_path
This is the filepath (either absolute or relative to where the notebook resides) to the file with the pickled QNN. How this is pickled is described later.

This should be provided as a string.

image_dir
The image directory where all the images resides, as a string of a relative or an absolute filepath.

We assume that the image folder only contains image formats that are supported by PILLOW. No other file types must reside in this folder. QBART has been tested with .jpg and .ppm .

We assume the images to be of the same file extension, and have the same channel ordering and data layout.

image_limit
For testing purposes, or for debugging purposes, executing the entirety of some problems such as the GTSRB image test set will take a mighty long time. Therefore, you can set a limit for how many images QBART should look at during this run.

This must either be an integer, or float("inf") to perform inference on all of them

image_channels
This is a string specifying the color order of the input images. For an RGB image, this would therefore be "RBG".

image_data_layout
A string that specifies the data layout of the image, where C is for Channel, r is for row, and c is for column. So if the image is saved in column major, we would get "Crc".

qbart_data_layout
String that specifies the data layout that the QNN expects. This is for the image loader, so that it can rearrange the data layout of the image if the two don't match.

qnn_trained_channels
String that specifies the channel order that the QNN expects. The image loader uses this to rearrange the channels if the two channel orders don't match.

qnn_trained_imsize_col
Integer that specifies the QNN expected column size of the input images. This is used by the image loader to reshape the image to the correct width.

qnn_trained_imsize_row
Integer that specifies the QNN expected row size of the input images. This is used by the image loader to reshape the image to the right height.

server_list
This is a list of one or several tuples. The tuples consist of two elements: an ip adress as a string, and a port number as an int.

By default, the qbart processing servers run on port 64646, while the IP addresses for each PYNQ is determined by how you configure the network. To use the PYNQ that the notebook is running, use "localhost" as its IP-adress.

image_classes
When QBART has classified all the images, we can translate the final classification from a class id number to a more human readable format.

It must be either a list of all potential image classes, or specified to None.

If it is specified to None, we don't convert from numbers to a classification class name.

4.2 User input files

The user needs to input a QNN and a set of image files to QBART. We will discuss both here.

4.2.1 The pickled QNN

The QNN that we use are identical to those used in maltanars inference examples[1]. The QNN training is not discussed here, but the resulting layer operations, weights, and vital metadata (expected image size, bit depth, etc) should be translated to python objects with layers.py. Then, it should be pickled with the python2 pickle library, or cPickle and stored to a file.

This file can then be used for QBART. Place it in an appropiate location in the filesystem, and add the absolute or relative filepath to the notebook configuration.

[1] `https://github.com/maltanar/qnn-inference-examples`

4.2.2 The image set

The image set should be an unzipped folder consisting of image files that have identical structure (file extension, channel ordering, and data layout).

Remember that total file size can be an issue when working with very large image sets. If it doesn't fit on the SD-card, try scaling down the image size.

If the images are not .ppm or .jpg, you have to specify image channel order and image data layout in the notebook configuration. Elsewise, the builtin image loader won't known how to properly reshape the images.

4.3 Execution

Execution of QBART is pretty straight forward once all the files are in place, and the notebook is configured. Start at the top of the notebook, and simply press shift+enter to execute each code block.

On the prebuilt image, we have stored a QNN for MNIST and GTSRB, with their corresponding image sets, ready for use.

All of these have successfully been tested on QBART. The repo includes test images and QNNs under qbart_main/qbart_user_files .

5 System Description

5.1 Xilinx PYNQ-Z1

The Xilinx PYNQ-Z1 SoC is the main computing unit of the QBART system, which has two ARM Cortex A9 cores an FPGA, an Ethernet port, DRAM, and an SD-card, among other things.

All QBART-related hardware components are synthesized to the FPGA.

There is also a custom shield, which displays important information to the user during runtime.

Important limitations to the board include the fact that the DRAM only have four read/write port, which limits simultaneous execution of several PYNQ components at the same time, and the fact that the fast FPGA Block RAM is a very scarce resource.

5.2 QBART System

QBART can consist of one or several PYNQs, which are connected together via Ethernet. They are organized by a master-slave configuration. The master receives commands from the user, and is responsible for delegating work and receiving results.

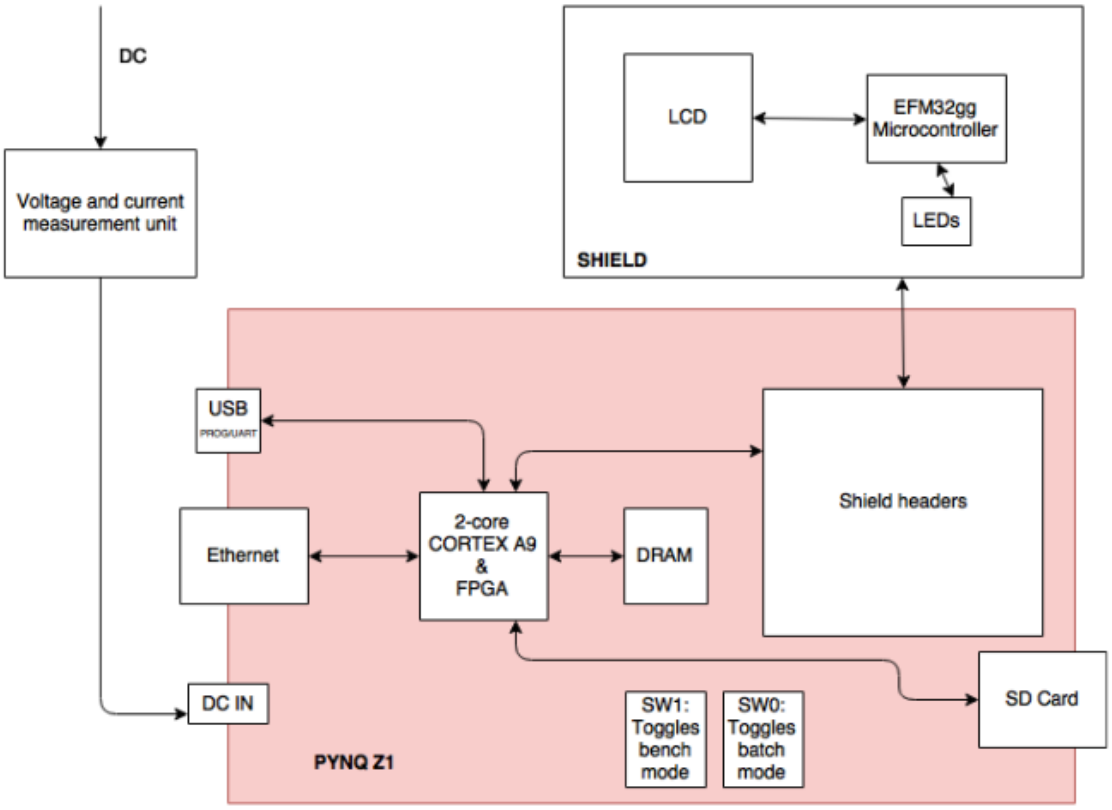


Figure 5.1: PYNQ Diagram

The master PYNQ evenly distributes all of the received images to QBARTs PYNQs, which individually executes the QNN on its images. Part of the QNN execution occur on the cortex A9s, and part on the on-board FPGA.

5.3 QNN Execution

For a given QNN and image set, a PYNQ running QBART will work independently on classifying the images. The Jupyter Notebook runs on the Cortex A9s, and "schedules" layers. As illustrated in 5.2 the Jupyter Notebook can choose to execute either a Python implementation of a given layer, or use the Python API to execute the layer on the FPGA.

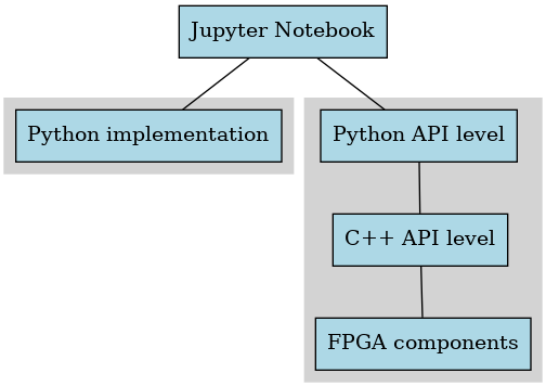


Figure 5.2: A very high-level overview of the system architecture. The Jupyter Notebook executes either the Python implementation of a given layer in the QNN, or uses the FPGA components (if available) by calling the Python API.

In order to execute on the FPGA, the data has to pass through two API levels: The Python API level and the C++ API level. The Python API level accepts N-dimensional NumPy arrays (where N varies, but is always less than or equal to three). This layer shapes data and packs data to the data format needed by the C++ API level. The C++ API level is called by using C's Foreign Function Interface (FFI), by using the Python 2 package `cffi`[2].

The C++ API level aims to be minimal, and only do the necessary operations to start the FPGA. It (mostly) expects the data to be pre-formatted to the correct format, and it also provides some convenience functions in order to efficiently allow the Python API level to pack data, allocate DRAM etc. After the computation is done, a pointer to the result is propagated upwards in the API level.

Both the Python API level and the C++ API level can be used stand-alone.

[2] `http://cffi.readthedocs.io/en/latest/using.html`

5.4 System limitations

As previously mentioned, 4 ports on the DRAM greatly inhibits the possibility of scheduling several FPGA components as one. Several components use 2-3 ports, so they have to be run one at a time, leaving the inactive components unused.

Different units have to use different data layouts (how columns, rows and channels in the activation matrix is organized in memory),

5.5 Potential improvements

A data reshape component on the FPGA to reshape the data as needed by some of the components in order to enable generalized, bitstream execution.

6 Printed Circuit Board

The PCB is used as a daugtherboard for the PYNQ. Its purpose is to provide information from the PYNQ to the user. To use it, put it on the arduino header on the PYNQ.

6.1 PCB Design

The MCU is an EFM32GG990F1024 and is communicating with the PYNQ through UART via the FPGA. See section 6.3.1. Its main task is to read data from the PYNQ and push it to the display.

The display is a Sharp LS013B7DH03. It is connected to the board through a FFC/FPC-connector. Communication with the MCU is done through SPI. See section 6.1.2.

The leds and buttons are controlled directly by the PYNQ through GPIO pins. The pins are used for debugging, and there are no specific uses in the final product.

6.1.1 Display

The display has two states, default state and running state. The default state will display the logo. The running state will display the active PYNQs in a list with progress in percentage.

6.1.2 Update protocol

In order to update the daughterboard, a byte is sent through UART. Given a high reset bit the local tables on the MCU is reset to the default state. When the reset bit is low the MCU will update its tables with the rest of the byte. The PYNQ id will indicate which PYNQ to update and the progress bits is a number from 0 to 15 where 15 is 100%. If a PYNQ doesn't have an entry it will be added and put last in the list.

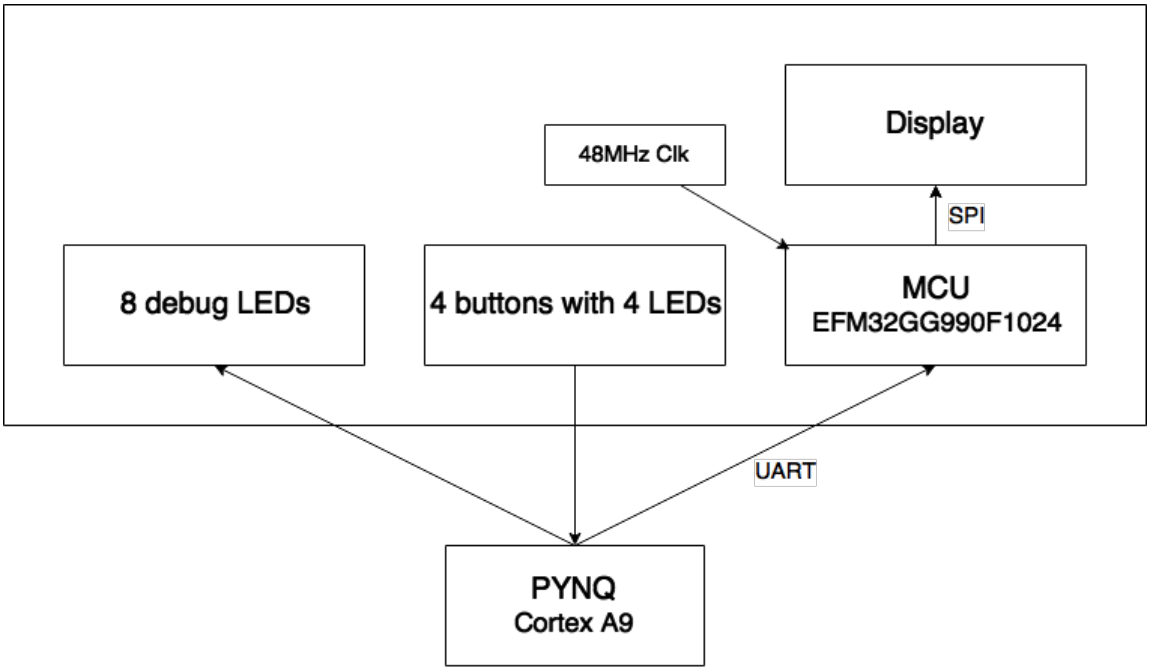


Figure 6.3: PCB Overview

Bits	0	1-3	4-7
Function	Reset	PYNQ id	Progress(%)

Table 6.1: Update protocol

6.2 PCB Errata

In the schematics the analog voltage and ground of the FFC/FPC connector are erroneously marked as not connected. This was solved by soldering them to the digital voltage and ground respectively. The schematics also suggest I2C as the communication protocol between the PYNQ and the MCU, but our solution uses UART on pin IO8.

6.3 MCU

6.3.1 RS 232 communication with the MCU

Communication with the microcontroller happens over RS 232. A tiny FPGA implementation of RS 232 transmit is adapted from Martin Schoeberls uart module [3]. The RS 232 module has no buffering, and it writes directly to the pins exposed by rosetta. When valid is set it will put the

[3] `https://github.com/schoeberl/chisel-examples/blob/master/examples/src/main/scala/uart/Uart.scala`

character that is on data on transmit with one start-bit (low), and finish with two stop-bits(high). By repeating the process one can send multiple characters. Provided is a program that sends one 8-bit character at the time.

6.4 What needs to be soldered

In order to flash the device, the MCU, clock and probably decoupling need to be soldered. It may also be convenient to solder the debug header. For normal operation the display connector and the arduino pins has to be soldered as well. For debugging possibilities the leds and buttons may be soldered.

7 FPGA Components

All the FPGA components are written in Chisel[4], specifically Chisel2, an open-source hardware construction language developed at UC Berkeley, embedded in the Scala programming language. The components are contained in Rosetta[5], which is a project template to rapidly deploy Chisel accelerators on the Xilinx PYNQ platform together with the Chisel library fpga-tidbits[6], both created by Yaman Umuroglu.

7.1 BitserialGEMM - Matrix multiplication

BitserialGEMM is a general, fully runtime-configurable matrix multiplication unit. It is used by fully connected layers as a matrix-vector multiplier, and by convolutional layers as a matrix-matrix multiplier. It implements the bitserial algorithm[2] given in figure 7.4 , with a slight twist. Instead of iterating through bit-planes in the outer for-loops, it iterates through the bit-planes in the innermost for-loops. This is so that we don't need to store any intermediate accumulation results, and can start writing the result matrix immidiately.

Input matrices are read from DRAM, and result matrix is written back to DRAM. It needs the base address in DRAM of the two matrices (left and right hand side so that the order of operands is conserved), as well as the base address of where to store the result matrix. Byte-counts and strides for reading and writing are all computed internally. BitserialGEMM also needs the input dimensions, signedness and bit-depths.

It expects a bitserial-packed format, channels/bits/rows/columns. Look at our C++ code[7] for specific details on how a matrix can be converted to this format.

Note that for bipolar networks (all weights are -1,+1), two-bit signed integers must be used. Right-hand side must be transposed. This is because DRAM reads are faster in row-major. Result is also produced transposed. Most of the computation is carried out in the DotProduct module.

[4] `https://chisel.eecs.berkeley.edu/`
[5] `https://github.com/DMProAccelerator/qbart/tree/master/rosetta`
[6] `https://github.com/maltanar/fpga-tidbits/`
[7] `https://github.com/DMProAccelerator/qbart/blob/master/rosetta/src/main/cpp/app/matrix_convert.cpp`

function BINARYGEMM(W, A, res, α)
  for $r \leftarrow 0 \ldots rows - 1$ **do**
    for $c \leftarrow 0 \ldots cols - 1$ **do**
      for $d \leftarrow 0 \ldots \lceil depth/wordsize \rceil - 1$ **do**
        $res[r][c]$ += $\alpha \cdot$ POPCOUNT($W(r, d)$ & $A(c, d)$)

Algorithm 1: W^1A^1 GEMM using AND-popcount.

function BITSERIALGEMM(W, A, res)
  for $i \leftarrow 0 \ldots w - 1$ **do**
    for $j \leftarrow 0 \ldots a - 1$ **do**
      sgnW $\leftarrow (i == w - 1 \ ? \ -1 : 1)$
      sgnA $\leftarrow (j == a - 1 \ ? \ -1 : 1)$
      BINARYGEMM($W[i]$, $A[j]$, res, sgnW $\cdot$ sgnA $\cdot$ 2^{i+j})

Algorithm 2: Signed W^wA^a GEMM using BINARYGEMM.

Figure 7.4: BitserialGEMM algorithm.

7.1.1 DotProduct

This is a binary multiplication on two bitserial-packed words. It is done by computing bitwise AND, and then counting the number of set bits (population count).

7.2 Fully Connected Layer

There is no actual Fully Connected FPGA component - any FC layer simply uses BitserialGEMM to do matrix-vector multiplication.

7.3 Convolution - Sliding Window

The sliding window unit lays out the convolution operation as a matrix-matrix multiplication. It uses BitserialGEMM for the matrix-matrix multiplication.

The sliding window module extracts windows of pixels from a multi-channel image and put them consecutively in memory. The result is to be interpreted as a column-major matrix.

It reads image from DRAM, and writes the result back to DRAM, and can cache input into BRAM to avoid redundant DRAM access. By always having *windowsize* relevant rows in BRAM, we will avoid reading these repeatedly from DRAM, and can slide the window through these.

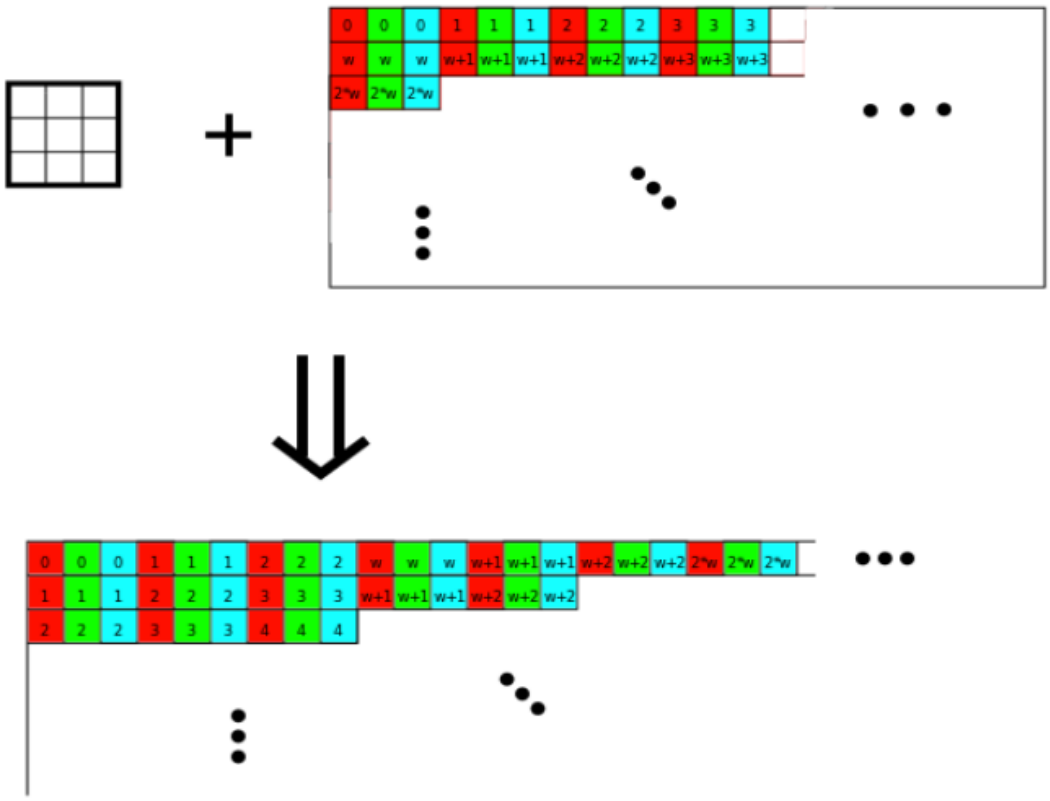


Figure 7.5: How sliding window works

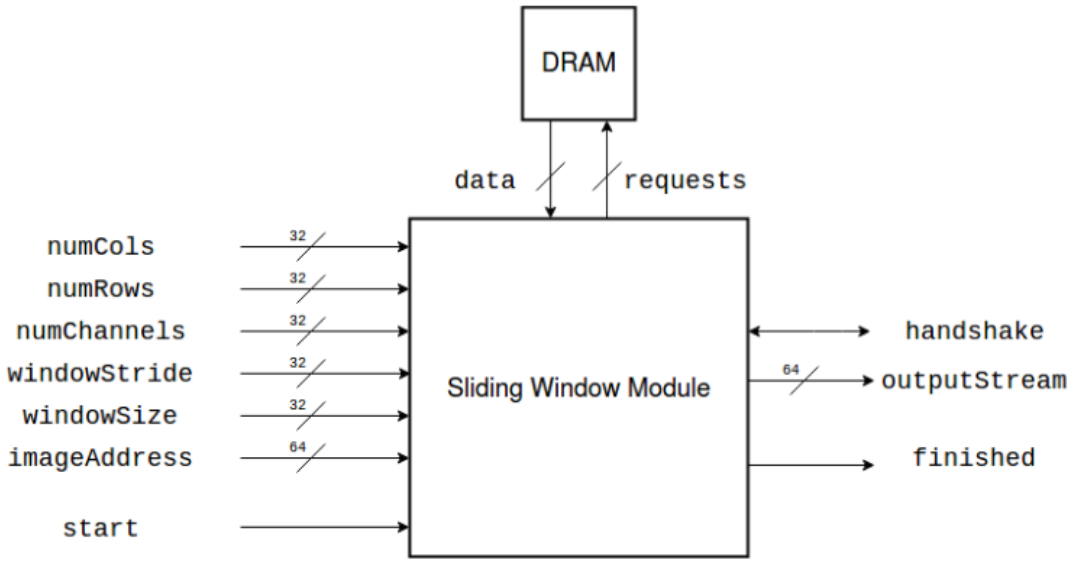


Figure 7.6: The sliding window component: black box view

Remark: The sliding window component takes in a `stride_exponent` parameter, which is $log_2(stride)$. Thus, the component only accepts strides that are powers of two.

7.4 Thresholding Layer

7.4.1 Introduction

Thresholding is done by comparing a matrix element $x_{ij} \in \mathbf{M}$ with each element of a threshold vector $\mathbf{t}$ using the hard-step function as in equation (7.2). QBART supports up to 8-bit thresholding, which means the size of $\mathbf{t}$ must always be within the range $[2^1 - 1, 2^8 - 1] = [1, 255]$. The outputs of the hard-step function are accumulated and the resulting value is the threshold value of the matrix element.

$$\mathbf{M} = \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix} \tag{7.1}$$

$$f(x_{ij}, \mathbf{t}_k) = \begin{cases} 1 & x_{ij} \geq \mathbf{t}_k \\ 0 & otherwise \end{cases} \tag{7.2}$$

The output matrix $\mathbf{O}$, which is the result of the threshold vector being applied to all elements of the matrix, is shown below.

$$\mathbf{O} = \begin{bmatrix} \sum_k f(x_{11}, \mathbf{t}_k) & \cdots & \sum_k f(x_{1n}, \mathbf{t}_k) \\ \vdots & \ddots & \vdots \\ \sum_k f(x_{m1}, \mathbf{t}_k) & \cdots & \sum_k f(x_{mn}, \mathbf{t}_k) \end{bmatrix} \tag{7.3}$$

A special case is when the threshold vector is of size one, i.e. when applying bipolar thresholding. Then the hard-step function becomes:

$$f(x_{ij}, \mathbf{t}_k) = \begin{cases} 1 & x_{ij} \geq \mathbf{t}_k \\ -1 & otherwise \end{cases} \tag{7.4}$$

7.4.2 The DMA Handler

The DMA handler is the top-level entry point for the thresholding unit and has two main objectives:

- It schedules reads and writes from/to DRAM.
- It starts and stops the thresholding unit appropriately.

An overview of the DMA handler interface is show in table 7.2 and the corresponding unit model in figure 7.7.

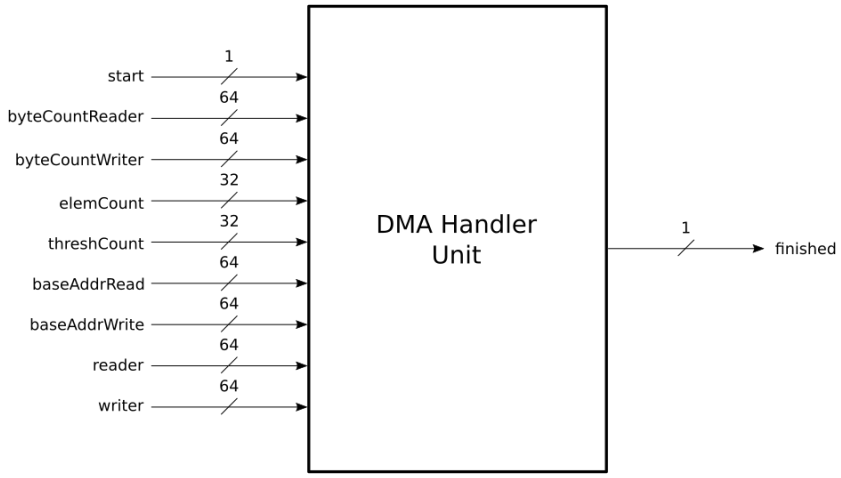


Figure 7.7: DMA handler unit.

Table 7.2: DMA Handler interface.

Port	Type	Direction
`start`	`Bool`	Input
`byteCountReader`	`UInt`	Input
`byteCountWriter`	`UInt`	Input
`elemCount`	`UInt`	Input
`threshCount`	`UInt`	Input
`baseAddrRead`	`UInt`	Input
`baseAddrWrite`	`UInt`	Input
`finished`	`Bool`	Output
`reader`	`StreamReaderIF`	Interface
`writer`	`StreamWriterIF`	Interface

Before the `start` signal can be set, the `reader` and `writer` interfaces must be provided and the other signals set appropriately. When the start signal is set, the DMA handler tells the reader to start processing thresholds from DRAM at the address specified by `baseAddrRead`; the threshold values are set as input signals on the corresponding ports of the thresholding unit. When all threshold values have been read, the DMA handler starts reading matrix elements. For each matrix element, the thresholding unit is started, and the result is written back to DRAM at the location specified by the `baseAddrWrite` address.

See figure 7.8 and 7.9 for a high-level overview of the memory access pattern and state model of the DMA handler.

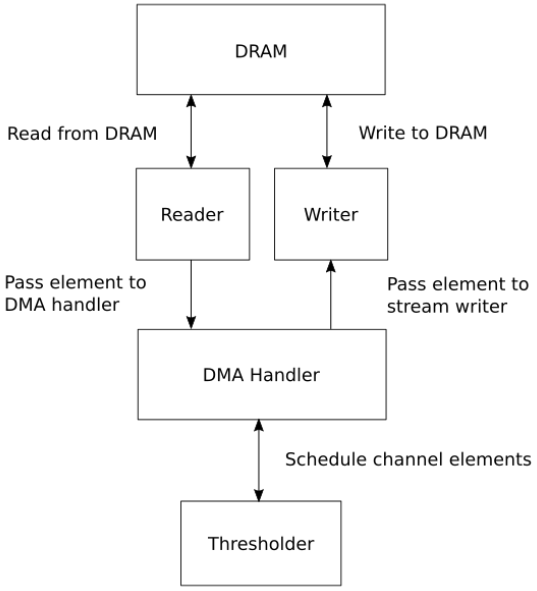


Figure 7.8: DMA handler memory model.

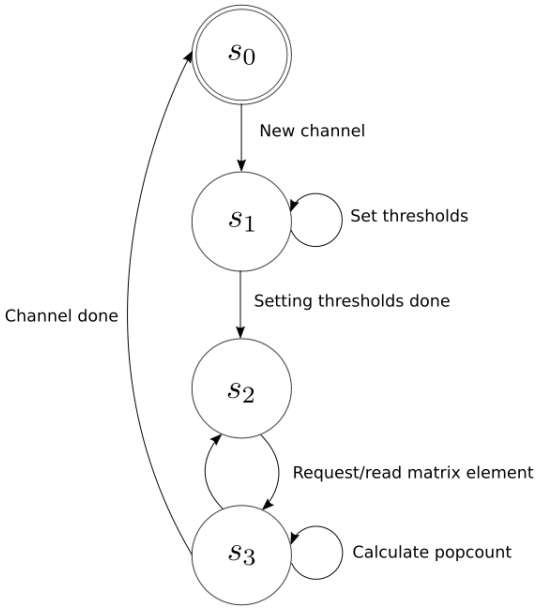


Figure 7.9: DMA handler state model.

7.4.3 Thresholding with several comparisons each cycle

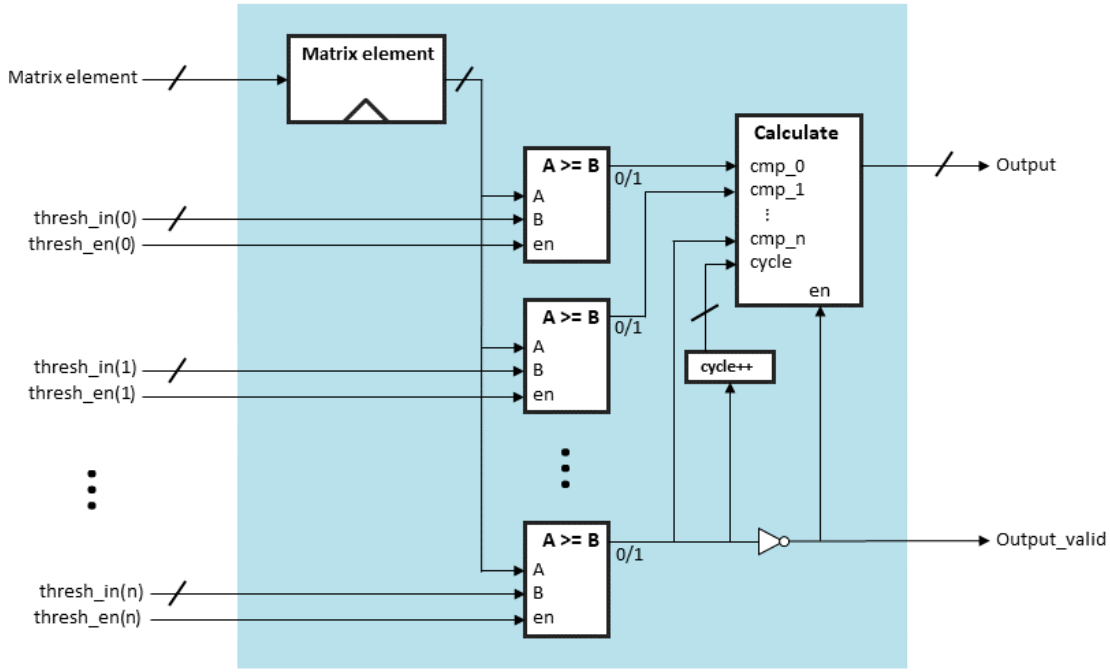


Figure 7.10: Detailed view of the threshold component. Control signals are omitted for simplicity.

As seen in figure 7.10, the threshold unit can do several comparisons during each cycle. Unfortunately, this version has some errors and did not work as expected. The unit expects thresholds to be sorted in ascending order and its interface is shown in 7.11.

When `matrix_element` is provided and the `start` signal set, the unit will compare `matrix_element` towards `n` thresholds during each clock cycle, namely `threshold_in(0 ... n)`. Whenever the `threshold_valid` signal is set, the threshold inputs must be valid, and the comparison will be calculated in that same cycle. If the last comparing unit outputs zero, the threshold has been found, and the result will be valid on the `output` line in the next cycle. If the last comparing unit outputs one, `output_valid` is false, the next `n` thresholds must be provided and it will need to run an additional cycle.

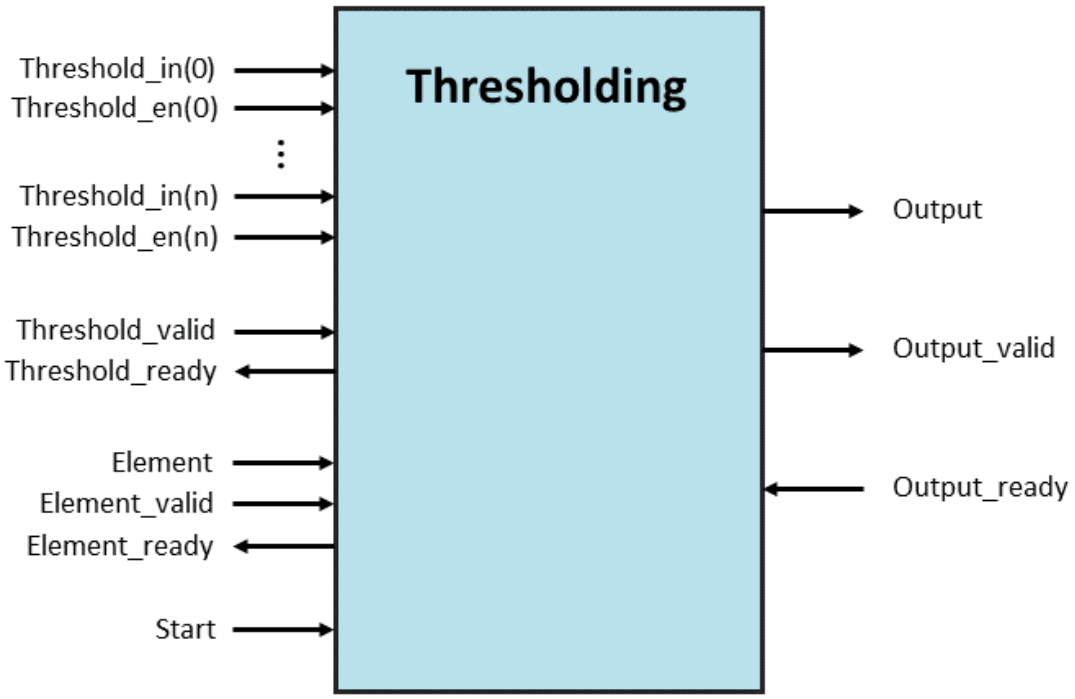


Figure 7.11: The interface of the threshold unit that does several comparisons.

Each threshold input also has an enable signal, to signal that that particular threshold input is invalid. If, for example, there are 8 comparing units, and 7 thresholds total, `threshold_en(7)` must be zero for correct output. The corresponding comparing unit will output zero when it is disabled.

The resulting output will be calculated by the following formula:

$$output = cycle * n + PopCount(cmp\_0 + cmp\_1 + ... + cmp\_n)$$

7.4.4 DMA Handler Performance Model

The DMA components used by the DMA handler requires 8-byte word alignment, which imposes the restriction that elements used by the DMA handler must be 64-bit integers. The DMA handler reads and writes elements from DRAM word for word, making the performance model quite simple, as shown in equation (7.5). Uniform cycle rate per word processed is assumed to simplify the model. Reading elements to/from DRAM takes $N(M+T)CC_{word}$ cycles; applying thresholding on each matrix element takes $NM\frac{T}{n}C_{word}$ cycles. Since the thresholding unit can deploy multiple compare units, the iteration of threshold elements are paralleized to by a factor of n. From the performance model it is also clear that increasing the clock cycle f_{CC} will give some speedup.

$$t_{threshold} = N\left(M + T + M\frac{T}{n}\right)\frac{CC_{word}}{f_{CC}} \tag{7.5}$$

where

CC_{word}	Clock cycles per 8-byte word.
f_{CC}	Clock frequency.
N	Number of channels.
M	Number of matrix elements per channel.
T	Number of thresholds per channel.
n	Number of compare units.

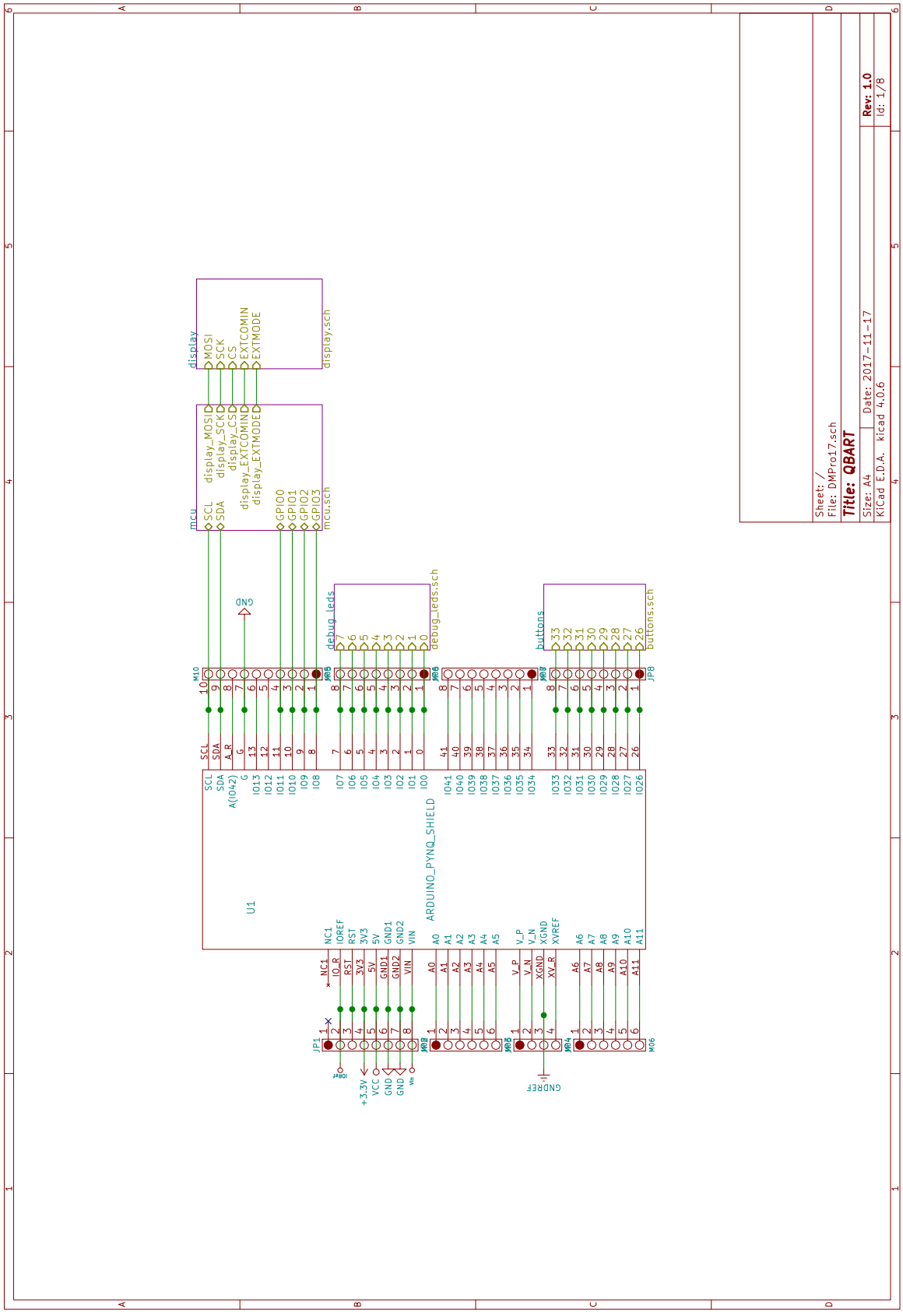
References

[1] Digilent inc. *PYNQ-Z1 Reference Manual*. URL: https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/reference-manuall.

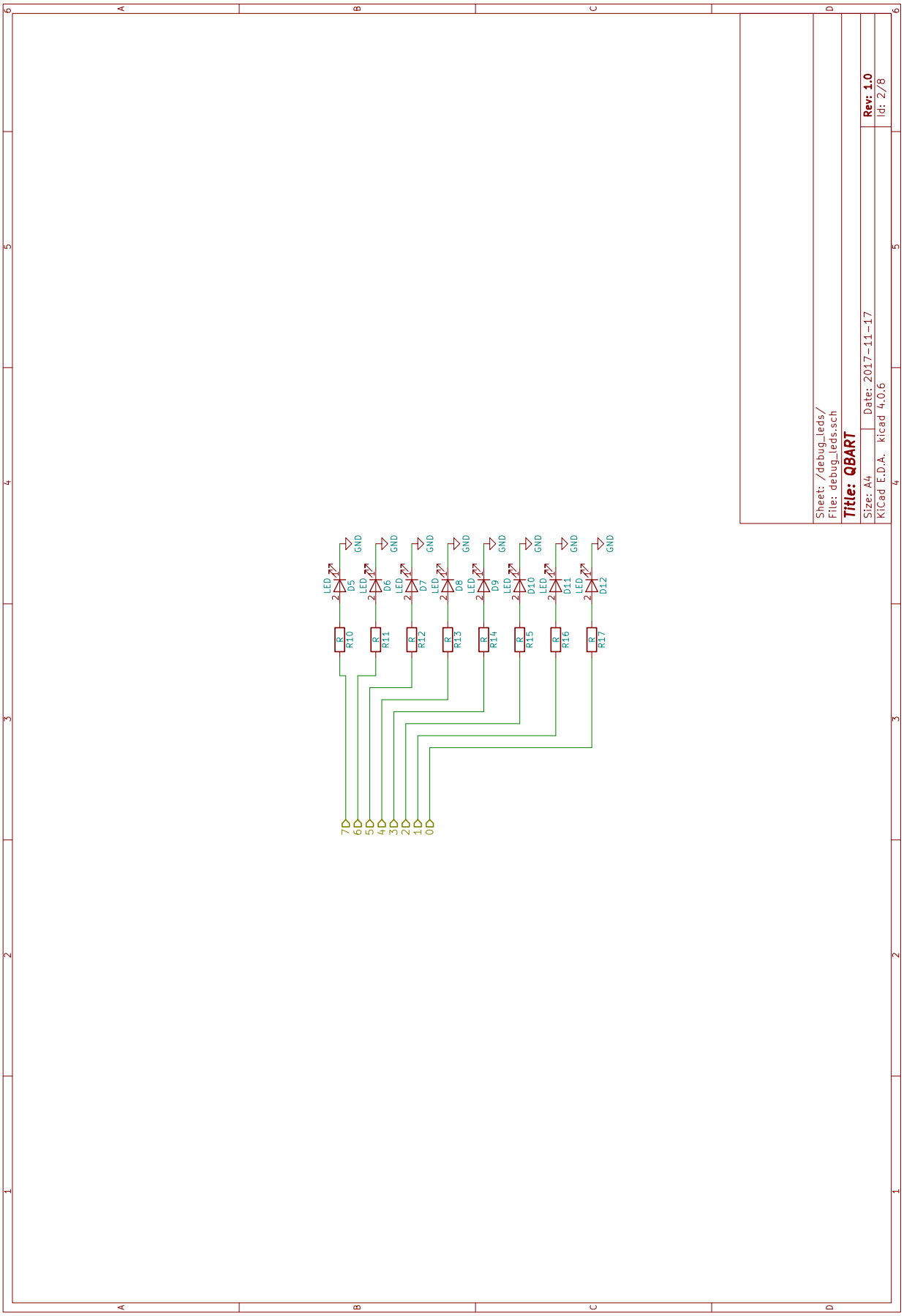
[2] Yaman Umuroglu & Magnus Jahre. *Streamlined Deployment for Quantized Neural Networks*. 2017. URL: https://arxiv.org/pdf/1709.04060.pdf.

Appendix A PCB Schematics

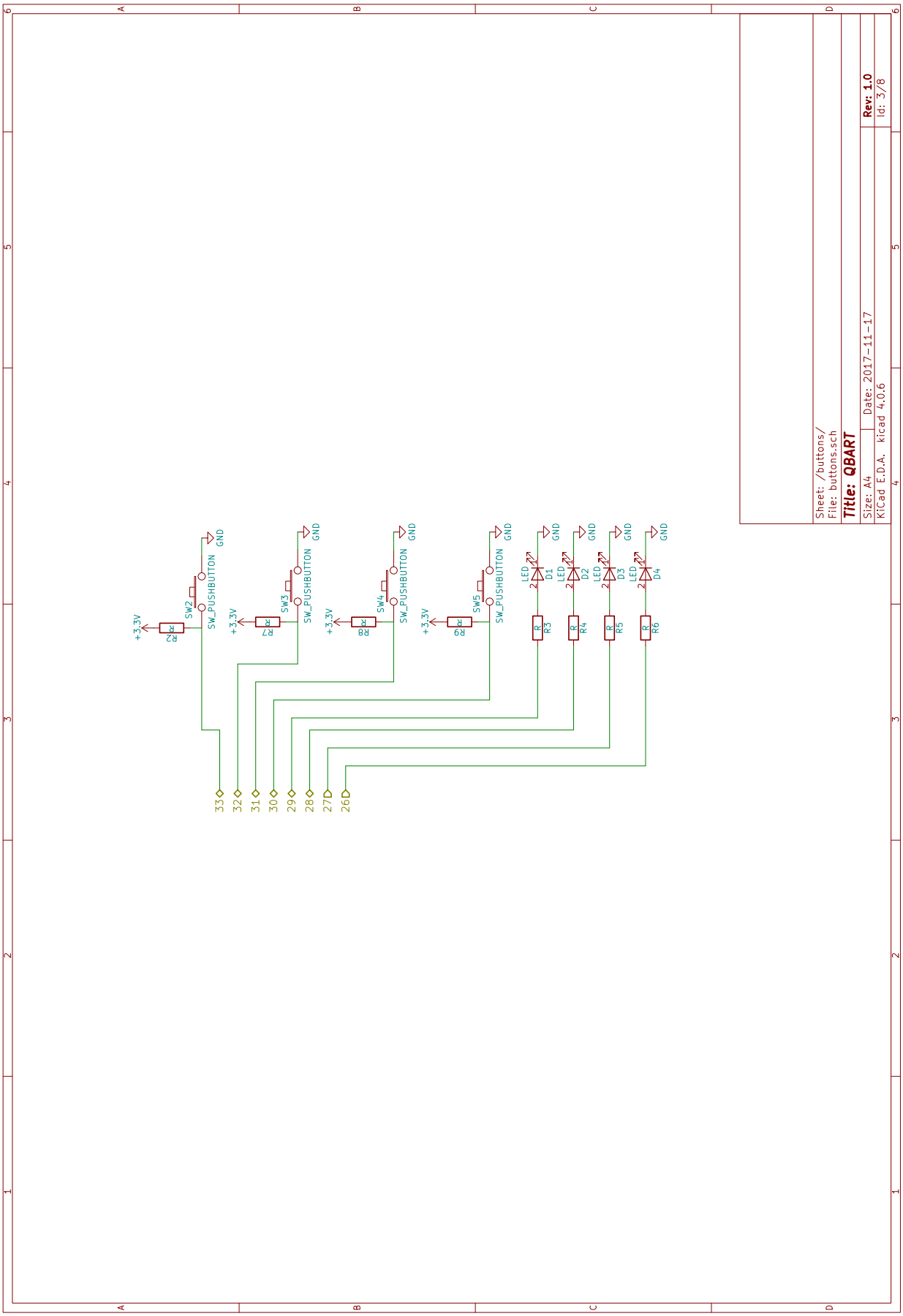
A.1 Overview


display
MOSI
SCK
CS
EXTCOMIN
EXTMODE
display.sch
mcu
SCL
SDA
display_MOSI
display_SCK
display_CS
display_EXTCOMIN
display_EXTMODE
GPIO0
GPIO1
GPIO2
GPIO3
mcu.sch
GND
debug_leds
debug_leds.sch
buttons
buttons.sch
U1
ARDUINO_PYNQ_SHIELD
JP1
JP2
JP3
JP4
JP5
JP6
JP7
JP8
GNDREF
+3.3V
Sheet: /
File: DMProI7.sch
Title: QBART
Size: A4
Date: 2017-11-17
Rev: 1.0
KiCad E.D.A. kicad 4.0.6
Id: 1/8

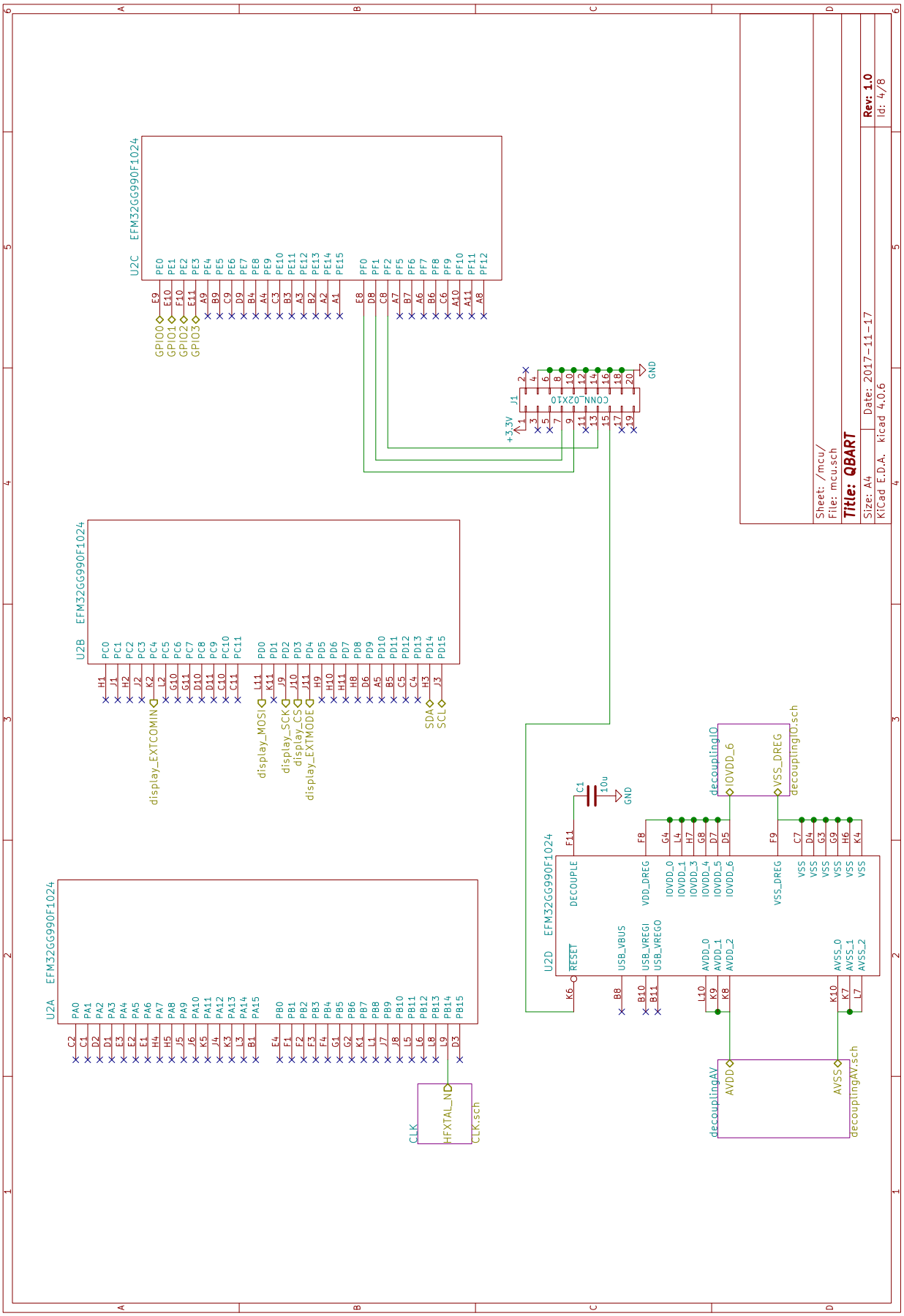
A.2 Debug LEDs



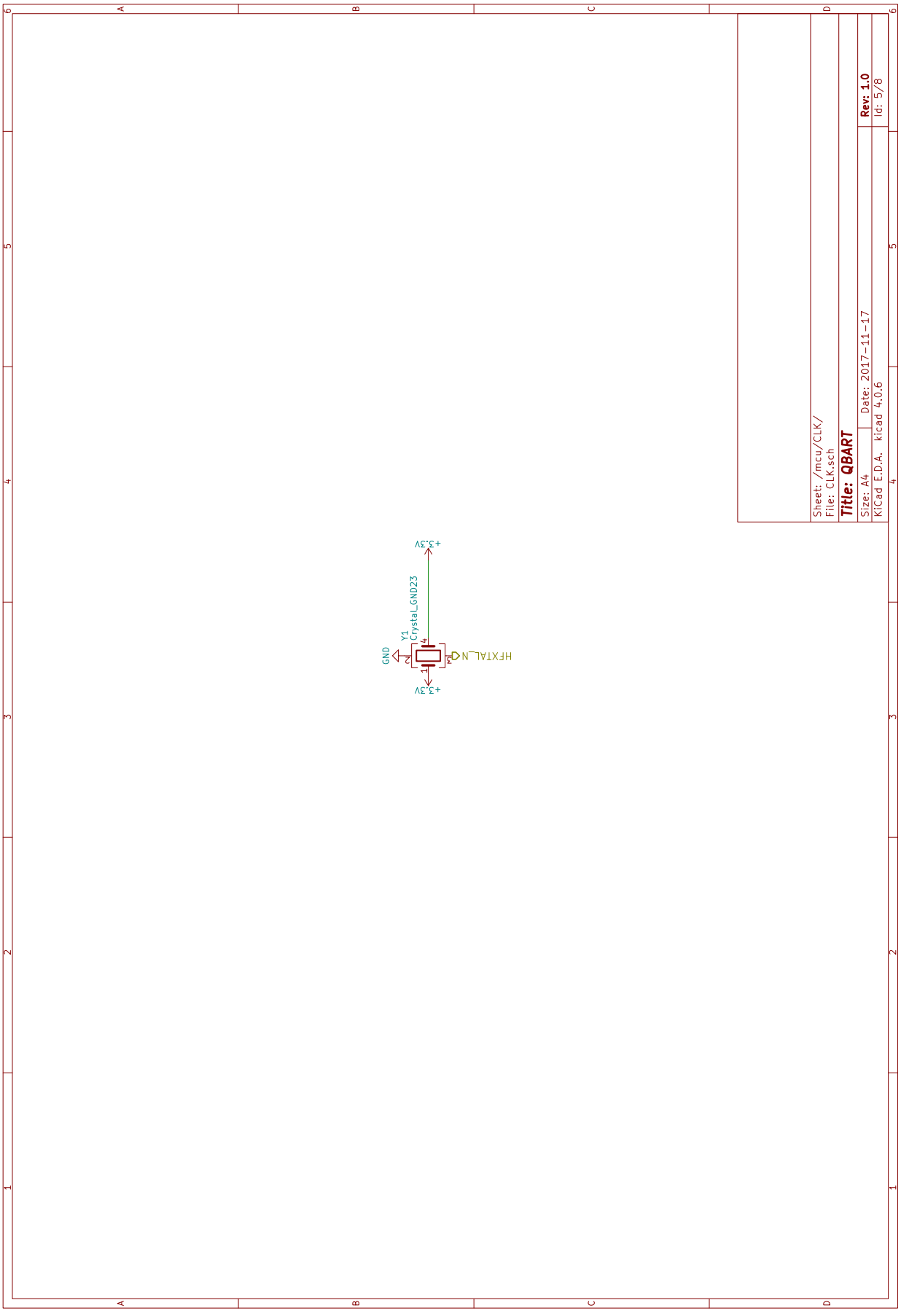
A.3 Buttons


Sheet: /buttons/
File: buttons.sch
Title: QBART
Size: A4
Date: 2017-11-17
KiCad E.D.A. kicad 4.0.6
Rev: 1.0
Id: 3/8
+3.3V
SW2
SW_PUSHBUTTON
GND
SW3
SW4
SW5
LED
D1
D2
D3
D4
R
R2
R3
R4
R5
R6
33
32
31
30
29
28
27
26

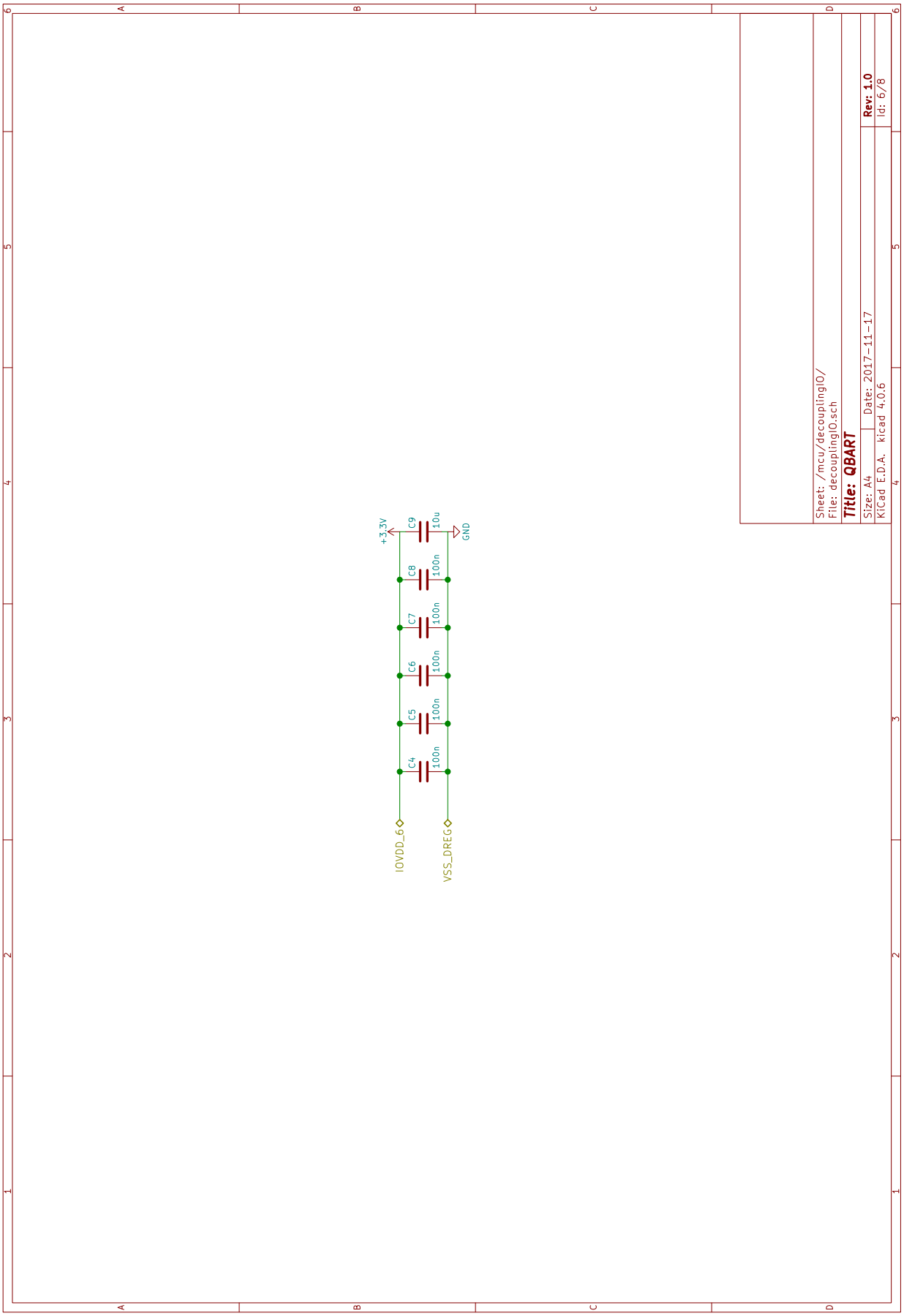
A.4 MCU


U2A EFM32GG990F1024
U2B EFM32GG990F1024
U2C EFM32GG990F1024
U2D EFM32GG990F1024
J1
CONN_02X10
+3.3V
GND
C1
10u
display_EXTCOMIN
display_MOSI
display_SCK
display_CS
display_EXTMODE
SDA
SCL
GPIO0
GPIO1
GPIO2
GPIO3
CLK
HFXTAL_ND
CLK.sch
decouplingIO
decouplingIO.sch
decouplingAV
decouplingAV.sch
Sheet: /mcu/
File: mcu.sch
Title: QBART
Size: A4
Date: 2017-11-17
KiCad E.D.A. kicad 4.0.6
Rev: 1.0
Id: 4/8

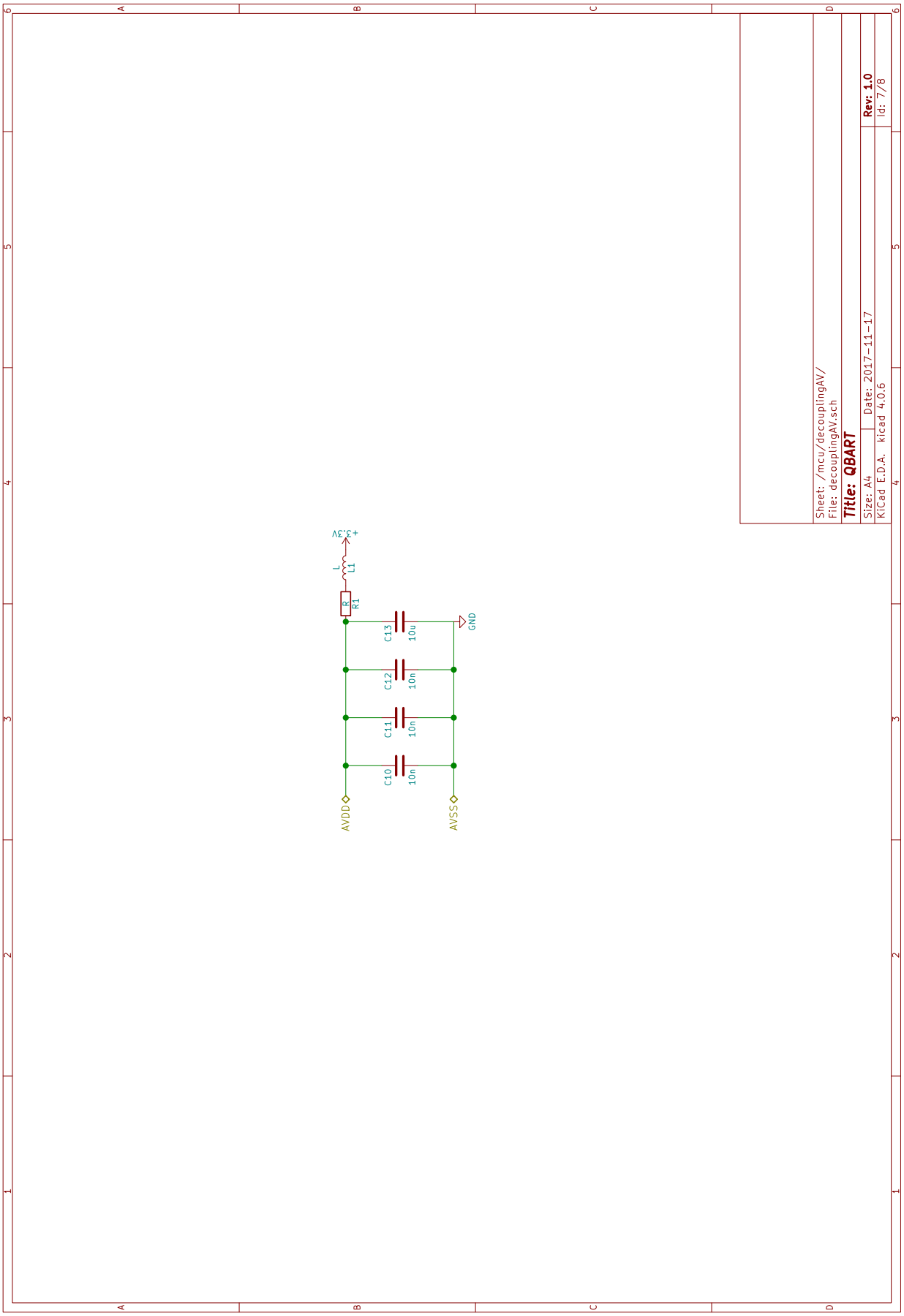
A.5 Clock


Sheet: /mcu/CLK/
File: CLK.sch
Title: QBART
Size: A4
Date: 2017-11-17
KiCad E.D.A. kicad 4.0.6
Rev: 1.0
Id: 5/8
+3.3V
Y1
CrystaL_GND23
GND
HFXTAL_N
+3.3V

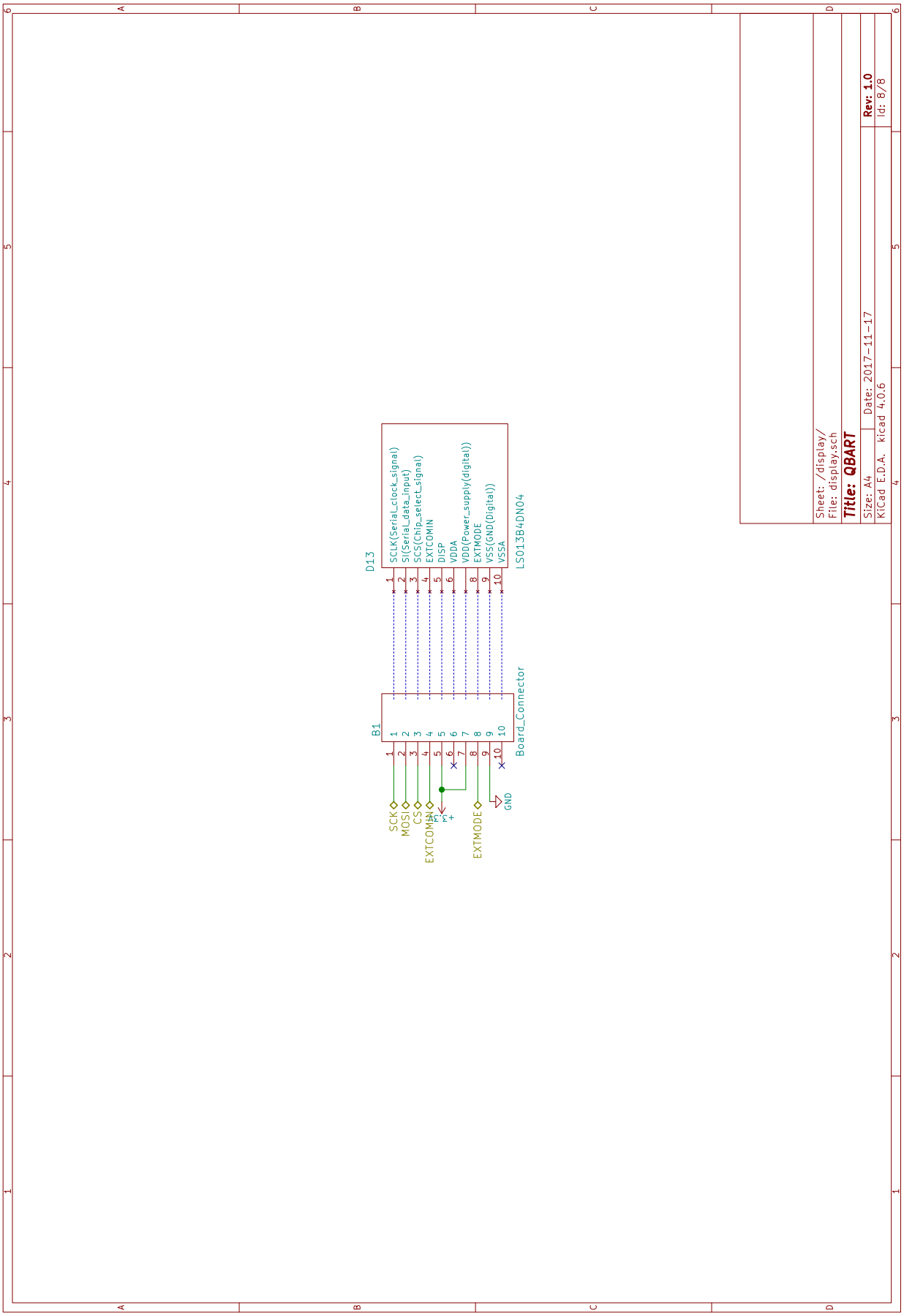
A.6 Decoupling IO



A.7 Decoupling AV



A.8 Display


D13
SCLK(Serial_clock_signal)
SI(Serial_data_input)
SCS(Chip_select_signal)
EXTCOMIN
DISP
VDDA
VDD(Power_supply(digital))
EXTMODE
VSS(GND(Digital))
VSSA
LS013B4DN04
B1
Board_Connector
SCK
MOSI
CS
EXTCOMIN
EXTMODE
GND
Sheet: /display/
File: display.sch
Title: QBART
Size: A4
Date: 2017-11-17
Rev: 1.0
KiCad E.D.A. kicad 4.0.6
Id: 8/8

Appendix B Software and toolchain versions

B.1 Software

- Chisel 2.+
- Vivado 2016.4 WebPack
- SBT 0.13.13
- Python 2.7
- Xilinx PYNQ Image 1.4
- GCC 7.+ with C++14
- CFFI 1.11

B.2 EFM32GG

- arm-none-eabi toolchain 5.4
- SEGGER JLink v6.20