# FPGA QR Code Scanner Final Report

M.Subhi Abo Rdan
*Department of EECS, Department of Physics*
*Massachusetts Institute of Technology*
Cambridge, Massachusetts, U.S.A
msubhi_a@mit.edu

Ayana Alemayehu
*Department of EECS*
*Massachusetts Institute of Technology*
Cambridge, Massachusetts, U.S.A
ayana@mit.edu

***Abstract*—We present a QR Code scanner implemented on an AMD Urbana FPGA, utilizing an OV7670 camera module to identify QR Codes and transmit the decoded information via HDMI to an external monitor. The OV7670 produces a 480x480 image that is subsequently converted to black and white and smoothed through convolutions to reduce graphical noise. Alignment patterns are used to locate the code, determine the module size and generate the downsampled QR code for decoding. Upon decoding, the results are communicated to a computer via Manta. Our scanner can decode QR codes as small as 240x240 pixels in as fast as .02 seconds with possible optimizations to decrease the speed further.**

***Index Terms*—Field Programmable Gate Arrays, QR Code, Digital Systems, Pattern detection, Image Processing.**

## I. Introduction

Quick Response (QR) Codes are visual representations used to encode various data types, such as text, images, and raw bytes. These codes consist of black and white modules, where black represents 1, and white represents 0. QR Codes are typically divided into three regions:

- **Alignment & Metadata Region:** Contains figures and information enabling the determination of alignment, mask, format, and error correction level.
- **Data Region:** Holds information in the format specified by the metadata.
- **Error Correction Region:** Contains error correction codes to recover a set number of lost or misinterpreted modules.

QR Codes are categorized by their Version numbers, where each Version specifies the size of the QR Code. For instance, Version 1 is a 21 x 21 module QR Code, and subsequent versions add four modules on each side, with Version 40 resulting in a QR Code sized at 177 x 177 modules. Our QR Code scanner supports Version 1 QR Codes without error correction capabilities. Higher QR code versions would require a higher resolution camera to decode consistently, and supporting error correction codes was outside of the scope of this project. [1]

## II. Physical Setup

Our QR Code scanner consists of the following components:

- **AMD Urbana FPGA**
- **OV7670 Camera Module** mounted onto a Seeeduino XIAO SAMD21 microcontroller, connected to an adapter.
- **Black and White QR Code** displayed on a white background.
- **Monitor** that displays 720p HDMI output.

Memory contents and key stored values are multiplexed using the FPGA's switches to display on the monitor for debugging purposes. The FPGA's LEDs indicate the current stage and/or relevant numeric values associated with that stage.

## III. Image Processing and Denoising

The camera module generates a continuous stream of image data, initially limited to a resolution of 320x240 due to the large size of a frame (150 kB), consuming a significant portion of the available BRAM memory on the board. To optimize memory usage, we implemented a binarization threshold before saving the image data to the BRAM. Binarizing the photo allowed us to upscale the resolution to 640x480. To conform to the square nature of QR Codes, the final stored image was truncated to 480x480, resulting in a frame occupying approximately 28 kB of memory.

Images can be captured by interrupting this data stream when a user toggles a designated switch on the FPGA. The initial raw image is often speckled with noise. To remove this noise, we employ a simple convolution technique; A kernel is convolved on the stored image, replacing each pixel with the most common value within a 3x3 grid of neighbors. This process effectively reduces the noise in the stored image. The denoising step improves the odds of the scanner both detecting

QR Code alignment patterns and decoding modules correctly.

Both the original image and its denoised counterpart are accessible for viewing through the HDMI output.

Figure 1. Image on the left: Before applying the denoising convolution kernel. Image on the right: After applying the denoising convolution kernel.

## IV. QR Code Detection and Downsampling

### A. Pattern Detection

QR Codes utilize concentric black and white squares located in the top left, top right, and bottom left corners to localize and determine the code's rotation, allowing for the derotation of the scanned code if necessary. In accordance with the QR Code specification, our QR Code scanner identifies these alignment patterns by scanning each line of the inputted image for the occurance of a given ratio; When a ratio of 1:1:3:1:1 of black - white - black - white - black pixels is detected (within a 50% tolerance) the scanner records the horizontal and vertical location of the identified alignment pattern. The horizontal and vertical detection of the alignment pattern is executed separately by the `horizontal_pattern_ratio_finder` and `vertical_pattern_ratio_finder` modules respectively. Each module outputs a 480-bit number, where each bit signifies the detection of the alignment pattern within a row or column. The outputs of these modules are then cleaned by removing lone positive and negative indicators within their output arrays to reduce the number of false alignment patterns identified within the image. Finally, the cleaned output is combined by the `cross_patterns` module to verify the existence of the alignment pattern at the indicated location.

### B. Pattern Verification

Due to the nature of our identification of the alignment patterns within the QR code, at least one of the potential alignment pattern locations (when the horizontal and vertical locations are combined) is incorrect. To address this, our `cross_patterns` module cross-references the potential horizontal and vertical locations of the alignment patterns with the denoised original image.

This verification process begins by segmenting the potential alignment pattern locations into nine zones,

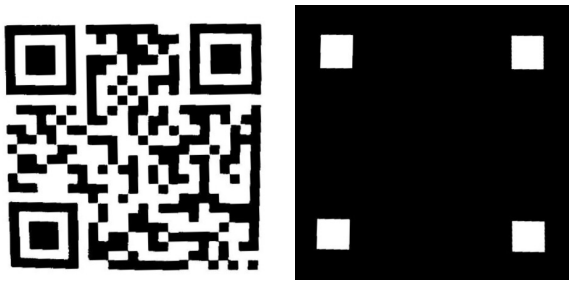

Figure 2. Left: Captured and denoised image of the QR code. Right: The results of `horizontal_pattern_ratio_finder` and `vertical_pattern_ratio_finder`. Despite the absence of an alignment pattern in the bottom-right of the QR code, a white square is present due to the cross of top-right and bottom-left patterns.

each containing at most one potential alignment pattern. The `cross_patterns` module iterates through all the pixels within each zone. Upon encountering a white pixel in the zone, it retrieves the pixel value at that location within the original, denoised image from the BRAM. As it collects the requested pixel values, it tallies the number of black and white pixels it encountered. Upon completing the zone, the tally of black and white pixels is compared to determine if the ratio aligns with that expected for an alignment pattern within a certain tolerance. Since the center of the alignment patters is a 3x3 module black square, we required that at least 87.5% of the requested pixels were black. We used the percentage 87.5% as it could be determined via powers of two, enabling us to calculate this tolerance using bit shifts rather than using a divider module, ensuring the calculation could be done within a clock cycle.

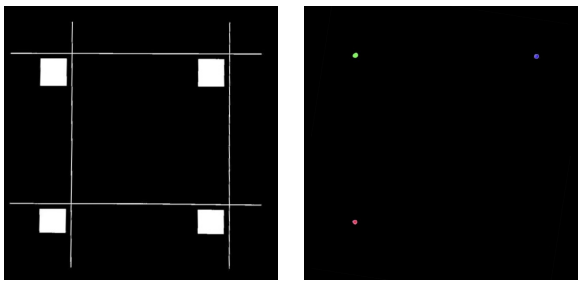

Figure 3. The results of the `bounds` and `cross_patterns` modules: dividing the image into nine zones, finding and verifying potential alignment patterns, and locating their centers. These zones are then visually examined to confirm their representation of alignment patterns, where red, green and blue squares correspond to the centers of alignment patterns in the actual image.

As the `cross_patterns` module identifies valid alignment patterns, it color codes the locations of each alignment pattern according to its position within the QR Code. Assuming the QR code is non rotated when photographed, the `cross_patterns` module would display the bottom left alignment pattern as red, the

upper left pattern as green, and the upper right pattern as blue. We chose to color code the identified positions of the alignment patterns to visually validate the orientation determined by the QR Code scanner. Upon completion, the `cross_patterns` module outputs the horizontal and vertical locations of the three alignment patterns to the downsampling modules.

### C. (Not Implemented) QR Code De-Rotation

Our original project goal was to support the detection and decoding of both translated *and* rotated QR Codes. However, accurate downsampling of the detected QR Code proved to be a difficult task. Any errors in the calculation of the centers of the alignment patterns could throw off our accuracy in downsampling the QR Code image into its pure form; Even errors on the order of one or two pixels could cause the downsample to fail by incorrectly reading modules at the extremities of the QR Code.

Likewise, supporting rotations would have compounded the errors greatly. Just the determination of the QR Code's rotation requires both precise data on the centers of the alignment patterns *and* fine-grained implementations of inverse trigonometric functions to ensure the calculations are accurate enough to avoid destroying the image post-rotation. The subsequent rotation matrix would again rely on the accuracy of the implementations of the trigonometric functions to rotate the image accurately. Thus, supporting rotations would have introduced a fair amount of obstacles that we decided to avoid given our project's time constraint.

### D. Downsampling

Once the locations of the three alignment patterns is determined, the scanner proceeds to convert the 480x480 image of the QR code into its pure 21x21 form. We chose to convert the image such that we can flatten and store the information of the QR Code in a single variable 441 bits long, enabling downstream modules to lookup the QR Code data combinationally (avoiding the 2 cycle delay BRAM's introduce).

This conversion begins by first determining the pixel-to-module ratio, done by the the `find_module_size` module. This module divides the average distance between two adjacent alignment patterns by the number of modules between two alignment patterns in a Version 1 QR Code, that is 15 modules. This division is done sequentially using the provided divider module from previous labs.

Once the module size is determined, our downsampling algorithm begins by splitting the QR Code into three zones, where each zone is defined by an alignment pattern and the modules closest to it. Within the zones, the `downsample` modules combine the pre-determined centers of the alignment patterns with the module size to index into the QR Code image stored in the BRAM and determine the module values within the zone. Finally, the values in these three zones are combined to form the pure, downsampled 21x21 QR Code. We decided to split the downsampling of the QR Code into these three zones to help mitigate the potential errors mentioned previously; By using the closest alignment pattern center to determine the downsampled value of a certain module, we greatly reduce the chances of incorrectly reading modules at the extremities of the QR Code, and we experimentally noticed an increase in reliability and performance in our QR Code scanner after implementing this optimization.

Similar to earlier BRAM contents, the final downsampled QR Code is available for display through HDMI by flipping the relevant switches on the board.

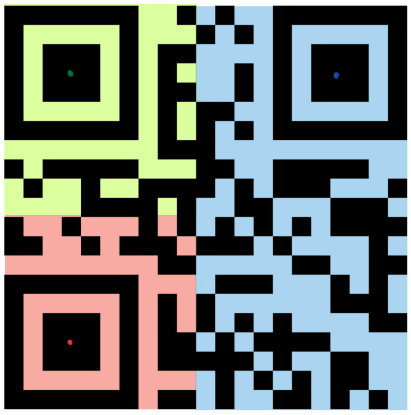

Figure 4. Illustration of module value determination for the downsampled QR Code. The coordinates of each module are derived from the center of the closest alignment pattern. Each alignment pattern center serves as the origin of a coordinate space, and each coordinate space is represented by its highlighted region within the QR Code.

## V. QR Code Decoding

The process of decoding QR code modules involves following a predefined pattern of reads to extract information such as data type, length, mask type, and the actual contents within the QR Code. QR Codes are masked upon creation to minimize the occurrence of special patterns, such as alignment patterns, within the data. This reduces the chance scanners misinterpret parts of the data as a feature. However, our scanner still sometimes misinterprets certain QR Codes due to alignment patterns occuring within the data (they are not always completely removable). In these scenarios, we recreate the QR Code with slightly altered data to remove the unwanted features. Given more time on this project, we could make our pattern detection algorithm

less susceptible to these errors by better specifying the requirements for a valid alignment pattern in our state machines.

As mentioned previously, our scanner does not support error correction capabilities due to the complexity associated with implementing a Reed Solomon decoder module, the time constraints this project entailed, and our lack of access to Xilinx's Reed Solomon Decoder IP. Consequently, the accuracy of the downsampling process is crucial for reliable QR code decoding in this project.

Our decoding process begins by identifying the applied mask type. The mask type is encoded as a 3-bit number that first must be XOR'ed with the number "101" to uncover the actual mask. This results in eight possible mask variations, described in Table I. Once the mask type is determined, our `unmask` module sequentially traverses the downsampled QR Code and XORs each QR Code module according to the mask value at that location. Once finished, the unmasked QR Code is ready for decoding.

The `decode` module combinationally reads the unmasked QR Code to determine the data type, length and the values within each data block. We decided to only support the bytes data type as this is the most common and versatile data type. These values are then packaged into 32 bit chunks and fed into a `Manta` module to be interpreted by a python script and displayed within a terminal.

Table I
MASK PATTERN GENERATION CONDITIONS

| Mask Pattern Reference | Condition |
|---|---|
| 000 | (i+j) mod 2 = 0 |
| 001 | i mod 2 = 0 |
| 010 | j mod 3 = 0 |
| 011 | (i + j) mod 3 = 0 |
| 100 | ((i/2) + (j/3)) mod 2 = 0 |
| 101 | (i j) mod 2 + (i j) mod 3 = 0 |
| 110 | ((i j) mod 2 + (i j) mod 3) mod 2 = 0 |
| 111 | ((i j) mod 3 + (i+j) mod 2) mod 2 = 0 |

## VI. PERFORMANCE

We achieved our project goal, creating a QR Code scanner capable of reading QR codes without rotations. Experimentally, the smallest QR Code our system can read is 240 by 240 pixels, constituting roughly a quarter of our 480 by 480 pixel frame. Our scanner can decode up to 19 bytes worth of information, as that is the most dense form of a Version 1 QR Code.

### A. Timing

Our system runs at 74.25 MHz, constrained by the 720p HDMI output. We have zero negative slack across all paths. The total decoding process takes 8,102,220 cycles or around .1 seconds. Much of this time is spent

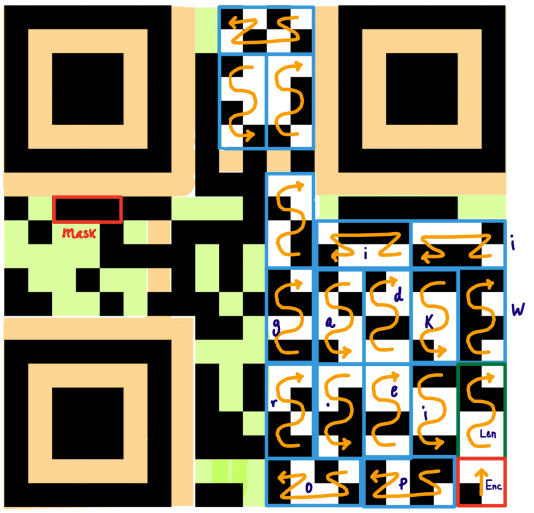


Figure 5. Specifications for QR Code Version 1 data encoding. The highlighted area in gold represents the constant alignment patterns present in every QR code. The four bits at the bottom right, labeled "Enc," indicate the type of encoded data (bytes, numeric, alphanumeric, or kanji). The following byte, labeled "Len," specifies the number of data bytes in the QR code. The three red-surrounded bits indicate the applied mask type on the data. The blue-surrounded bytes, with an arrow indicating the reading direction from the Most Significant Bit (MSB) to the Least Significant Bit (LSB). Green-highlighted blocks are reserved for error correction and are not addressed in this project.

on the denoising convolution. Our scanner can be ran without this convolution applied, and in this configuration the decoding process takes 1,651,020 cycles or around .02 seconds, comparable to a handheld barcode scanner.

The `average` module, responsible for the denoising convolution, accounts for $80\%$ of the decoding process time, making it the most time-consuming module. Following that, the `horizontal_pattern_ratio_finder` and `vertitcal_pattern_ratio_finder` modules each take $8.5\%$ of the computing time when including the `average` module. Excluding the `average` module, they each take $42\%$ of the time.

As many of our algorithms adhere to the QR Code spec, there are not many changes we can make to the scanning and decoding process to substantially decrease runtime. A possible optimization includes improving the *horizontal_patterns* and *vertical_patterns* algorithms to eliminate the need for the *cross_patterns* module. However, this would save at most 260400 cycles, or .003 seconds, but would likely involve other algorithms increasing in complexity which may ruin the potential time save.

Increasing the clock rate, however, serves as an easy way to extract more speed. Our HDMI pixel serializers,

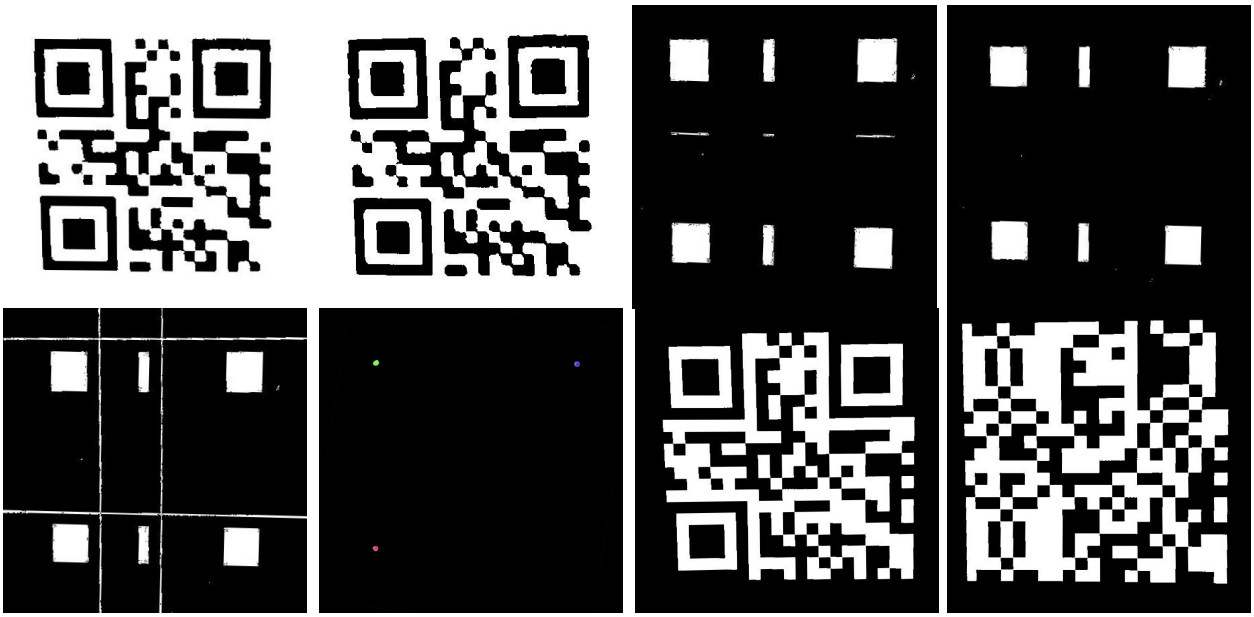

Figure 6. Illustration of the complete QR code decoding process. From left to right: (1) The original captured image of the QR code. (2) A denoised version obtained as explained in Section 3. (3) Visualization of raw horizontal and vertical patterns. (4) Cleaned visualization with error thin lines removed. (5) Division of the image into nine zones, each potentially containing an alignment pattern. (6) Identification of the centers of actual alignment patterns by analyzing each zone from (5). (7) The raw 21 by 21-bit result of the QR code downsampled, representing the same code shown in the image (1). (8) The QR code after the unmasking process, reveals the actual encoded data. Note: The decoded output of this QR code is output through Manta as "wikipedia.org."

used to communicate with an external monitor, constrain our systems speed as they require a clock rate 5x faster than the system's 74.25 MHz. Thus, eliminating the HDMI output completely would free us from this bottleneck and could speed up our system anywhere between 2 - 5 times faster.

### *B. Memory*

We utilize two 480*480 bits-depth BRAMs, each roughly 28 kBs in size, totaling 56 kBs. The first BRAM, (*frame_buffer*), buffers the camera's data stream and stores the raw image of the QR code. The second BRAM, (*BRAM1*), stores the QR code image after the denoising convolution.

## VII. Conclusion

We have successfully implemented an FPGA-based QR code scanner capable of decoding Version 1 QR codes. Our system includes image processing algorithms, pattern detection, downsampling, and QR code decoding. Though we achieved our primary project goals, there are opportunities for future enhancements, such as optimizing algorithms for speed improvements and supporting rotated QR codes.

Given the linear nature of the decoding process and reliance on previous working modules to develop later ones, we developed this project via the peer programming technique. We alternated who coded and who oversaw the coding throughout the development of the project. From their state machines to data structures, all algorithms to port the QR Code decoding process to hardware were developed together.

For the reports, Ayana wrote most of the content and Subhi created all of the visuals, including the block diagrams.

## References


[1] IEC 18004:2000.

[2] Moseley, F.Fischer, (2023). "Manta: An In-Situ Debugging Tool for Programmable Hardware". Massachusetts Institute of Technology. Retrieved from https://dspace.mit.edu/bitstream/handle/1721.1/151223/moseley-fischerm-meng-eecs-2023-thesis.pdf?sequence=1&isAllowed=y.

[3] https://blog.qartis.com/decoding-small-qr-codes-by-hand/


## VIII. Appendix: Block Diagrams

The block diagrams for all stages are provided on the next page.

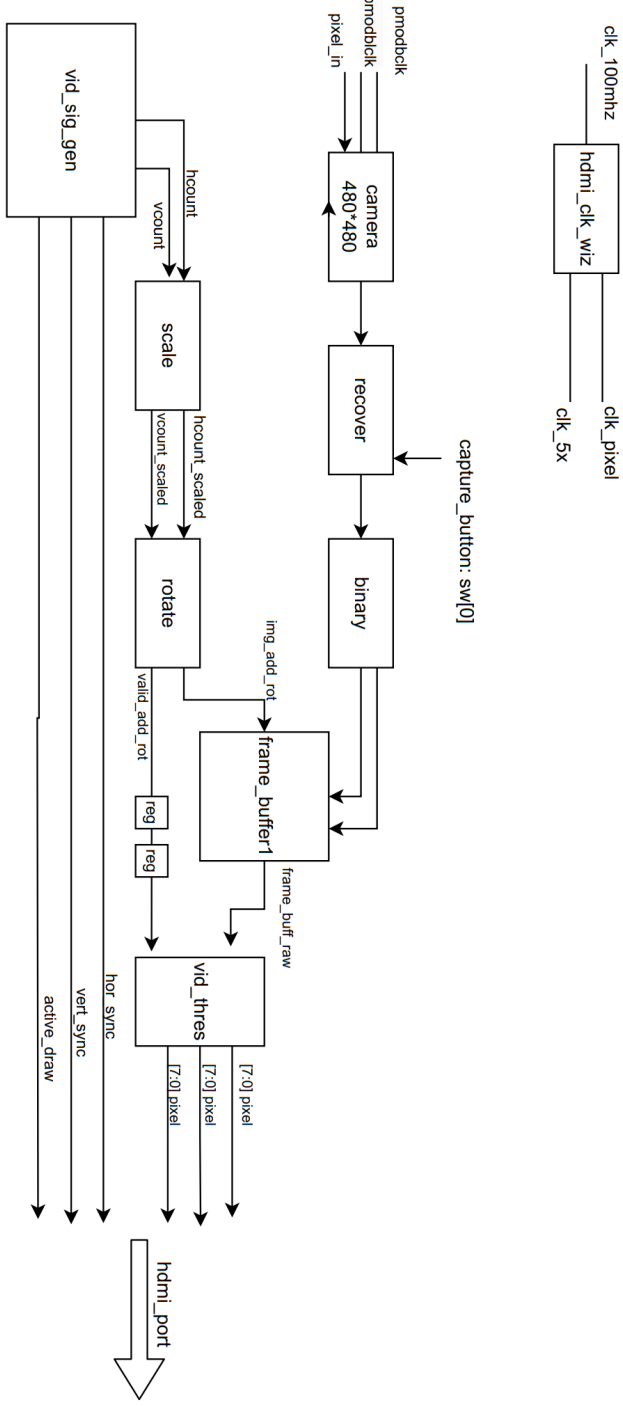


Figure 7. Block Diagram, Stage 1: streaming, capturing, and storing the image.

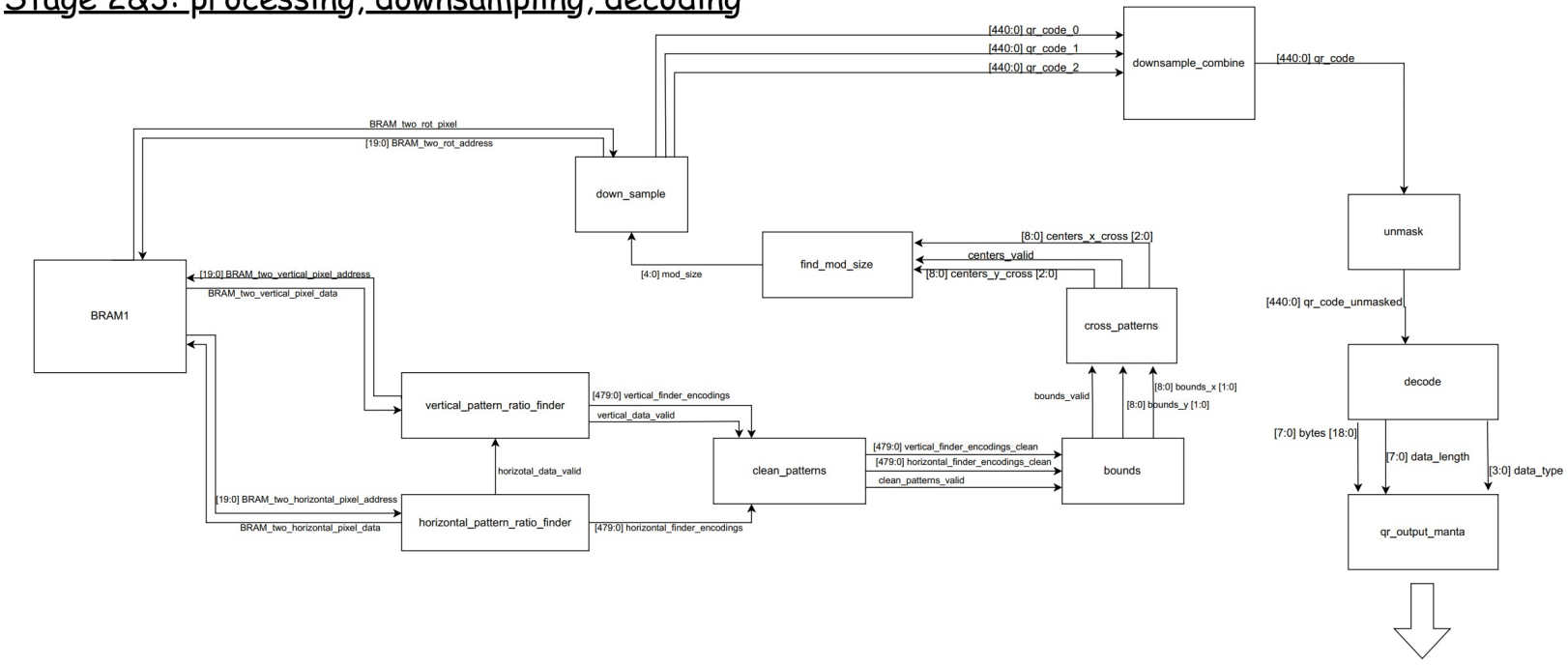


Figure 8. Block Diagram Stages 2 and 3. Image processing, finding the QR Code, and downsampling the data then decoding it.