Mirror Me
6.205 Final Project - Final Report

Janette (Jan) Park
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
janp@mit.edu

Thienan Nguyen
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
thienann@mit.edu

***Abstract*—We propose a robotic arm system, implemented on a NEXYS 4 DDR FPGA, to mimic the movement of a human arm in front of two orthogonal cameras. The system is run at a high system speed of 65 MHz, allowing the image processing and angle calculations to be done quickly for minimum response delay between the arm and the robot arm. The design consists of the camera data, image processing, angle calculation of each arm joint, and the mechanical driver of the robotic arm. We discuss the design of the system, the performance, testing and debugging methods, and any potential problems or improvements.**

***Index Terms*—Field Programmable Gate Arrays, Robotic Arm, Blob Detection, CORDIC**

I. Introduction

The Mirror Me system consists of two orthogonal cameras facing the user's arm, and an addressable mechanical arm, both displayed in Fig. 1. The person's arm has its finger, knuckle, wrist joints and forearm each covered in pink patches. As the person moves their arm, the robot arm moves to mimic the person's movement by copying the angle formed at each joint.

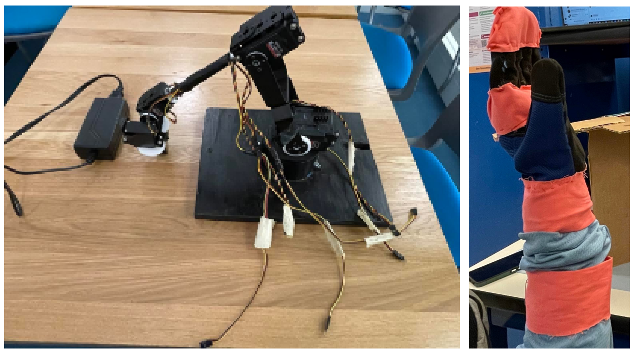


Fig. 1. The left image displays the robotic arm and its power source. The right image shows the human arm with its pink patches.

In order to achieve the goal, our system has two main components: an image processing component and a computational component. The image processing involves dual camera operation and joint detection. The computational component, directly following the image processing, calculates the angles at each joint and generates a signal to drive the mechanical arm. Fig. 2 shows a block diagram of the high level components, Fig. 3 shows a block diagram of the top level flow of information, and the remainder of the paper will elaborate on each component, our design choices, and the trade-offs.

Our design's code base can be viewed here: https://github.com/janpark49/6205_mirror_me.

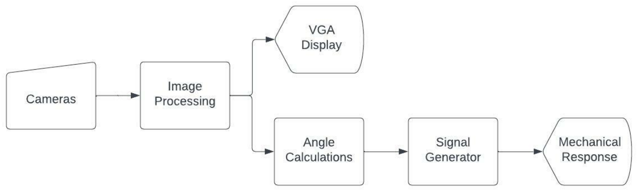


Fig. 2. High Level Components of the Mirror Me Design

II. General Overview

The robot has three joints to move; its shoulder, elbow, and wrist joints. The robot's wrist joint angle is defined by the angle formed by the human's finger, knuckle, and wrist. The robot's elbow joint angle is defined by the angle formed by the human's knuckle, wrist, and forearm. The robot's shoulder joint angle is defined by the angle formed by the human's wrist, forearm, and the horizontal axis. Thus, the system must find the locations of each of the four human joints in each camera frame, associate these coordinates to the correct joint, extrapolate the two 2D coordinates into a 3D coordinate for each joint, then calculate the angles between the joints listed above.

A. Interesting Challenges

Both the image processing and angle calculation components of the system are interesting problems to solve. Regarding image processing, since all the joints are covered in the same pink patch, the joints are indistinguishable to the camera. Thus, the image processing algorithm needs to not only find the locations of each spatial conglomeration of pink, but it also needs to associate each blob to the right joint. This would require a calibration at the beginning to initialize each blob location to the correct joint, then utilizing previous blob

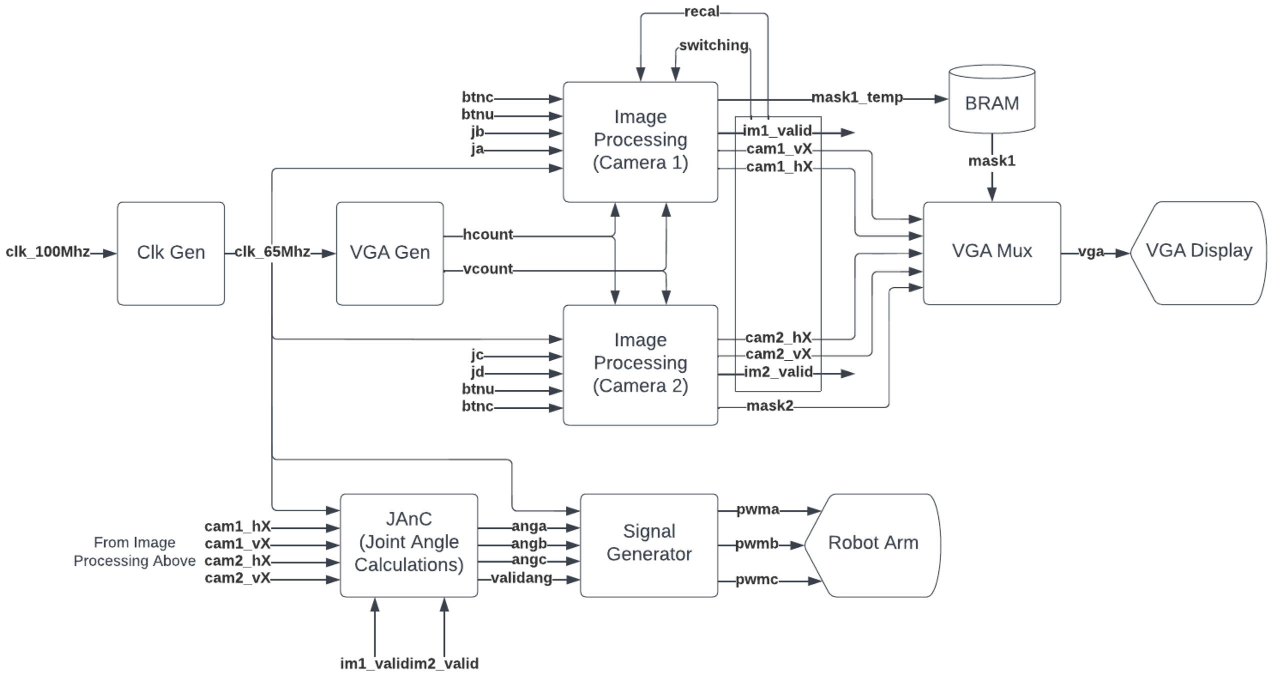


Fig. 3. Block Diagram for Top Level

locations to associate the next frame's blob locations to the right joints.

Another challenge with image processing is that in the cameras' view, two blobs can look like one blob when adjacent or one behind another, as shown in Fig. 4, or a blob can disappear from the camera reading due to lighting. The system would need to account for these situations as accurately as possible. Another consideration to be taken is the possibility of pink noise in the camera frame that might affect blob detection.

On the more mathematical side, angle calculation on an FPGA is equally non-trivial. Typically, the simplest approach to 2D angle calculations are trigonometric functions. Unfortunately, trig and inverse trig functions can be quite difficult and costly to implement on an FPGA. Arcsine and arccosine functions do not natively exist in Verilog and must be implemented with its own module. This would require extensive uses of lookup tables to relate angles to their trig equivalent, ultimately costing a large amount of area. In addition to large area, utilizing inverse trig function requires division, which may add to large overall delays as the division would be quite large. A different approach to calculating angles would be required, one that is neither too large in size or too slow.

Aside from trigonometric functions, another challenge with angle calculation is extending calculations into 3D space. It is simpler to find an angle in 2D space than an angle in 3D space, so it is necessary to map the point in 3D space back into the 2D Cartesian space to calculation the angle between two vectors. To achieve this, dot products and cross products are necessary, which are slow and heavy in calculations. This

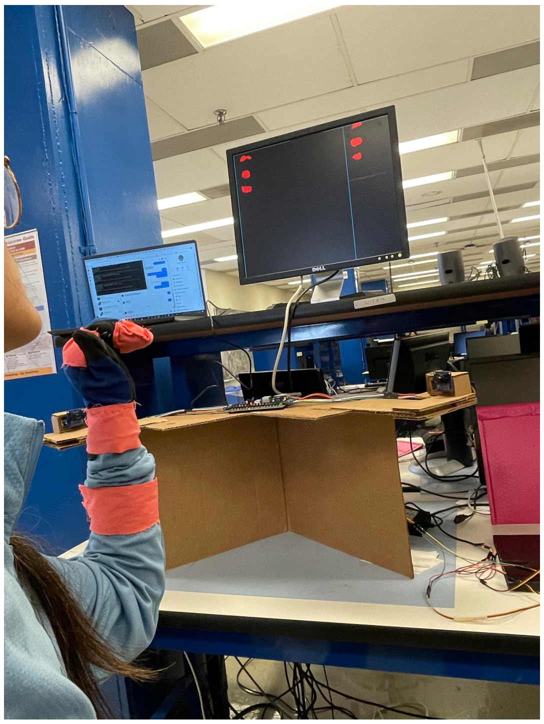


Fig. 4. When the hand is bent and oriented as shown, both cameras read the finger and knuckle joints as the same conglomeration of pink pixels, as we only see three blobs for each VGA display. The left camera should have distinct coordinates for the two joints because it sees them adjacently, while the right camera should have similar coordinates for the two joints because one is behind the other.

produces issues with delay and timing violations, and it is necessary to optimize the method to function.

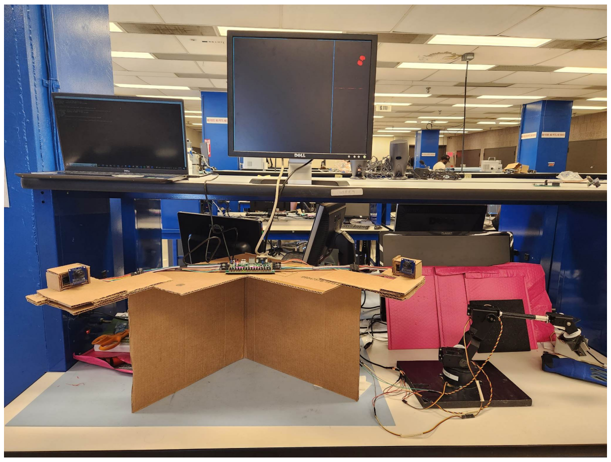


Fig. 5. Full Physical System Setup

B. Project Checklist

COMMITMENT - One robot arm joint responding to three different colored patches defining one angle. This is a much simpler problem to solve, as it eliminates the blob detection challenges listed above since the color of the patch defines which blob it's associated to.

GOAL - One robot arm joint responding to three pink colored patches defining one angle. Most of the blob detection challenges still remain, but are less likely to cause issues since there's only three blobs that could overlap with one another.

STRETCH GOAL - Three robot arm joints responding to five pink colored patches defining three angles. A working project would show the robot arm responding to a human arm with five patches.

UPDATE ON STRETCH GOAL - We realized that the camera does not zoom out enough to fit five human joints in a frame, and that the robot arm's shoulder joint makes an angle with respect to the horizontal surface. Thus, we modified our stretch goal to still drive three robot arm joints, but responding to four pink colored patches, with the fifth necessary coordinate corresponding to a point on the horizontal surface that the bottom-most pink patch is on.

III. Physical Hardware and Setup

The full system setup is shown in Fig. 5. The physical hardware includes:

- NEXYS 4 DDR FPGA
  The FPGA runs on 65 MHz for all modules.
- Human arm
  The human arm is in a well-lit room with dark-colored surroundings. The finger, knuckle, wrist joints, and mid forearm are covered in pink patches. The first three joints' patches pink fabric are sewed onto a dark glove, around 1-1.5 inches thick. The mid forearm patch is a 1.5 inch thick pink fabric that must be wrapped around a long sleeved shirt and fastened by a paper clip. The long sleeve shirt must cover any skin between the forearm patch and the glove to eliminate noise from skin color. The human arm setup is shown in Fig. 2
- Two cameras
  The two cameras connect to JA, JB (Camera 1), and JC, JD (Camera 2) on the NEXYS4 DDR FPGA and are positioned to both face the human arm at orthogonal angles.
  The camera and NEXYS 4 DDR FPGA sit on a cardboard setup with two slots for the cameras 22 inches away from the intersection of where the cameras point to, as shown in Fig. 6.

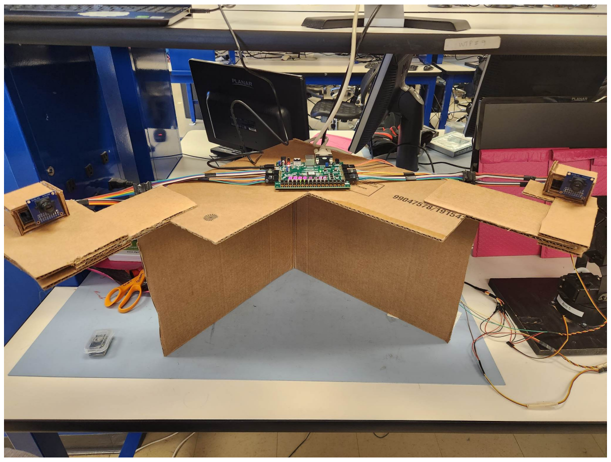


Fig. 6. Camera System Setup

- Robotic arm
  Each joint on the mechanical arm has a Servos motor which is driven with 5 Volts and 2 Amps from an external power supply and addressed with a 50 Hz PWM signal from the FPGA pins xa_p[2:0], as shown in Fig 7. The PWM signal indicates the joint's angle, with a pulse of 1 ms for 0 degrees and 2 ms for 180 degrees (pulse length increases linearly). The operating voltage range for the motor is between 4.8 V and 6 V. No external circuitry is required.

IV. Image Processing

The data from the two cameras are individually processed to find the locations of the four pink patches. Each camera has an instance of the Image Processing module, which outputs an (h, v) coordinate pair for each joint blob. The system first calibrates, on a button click, the initial locations of the joints when the arm is vertically extended, then finds the next locations for each joint given the previous locations. Thus, the image processing system follows the flow of information in the block diagram in Fig. 8.

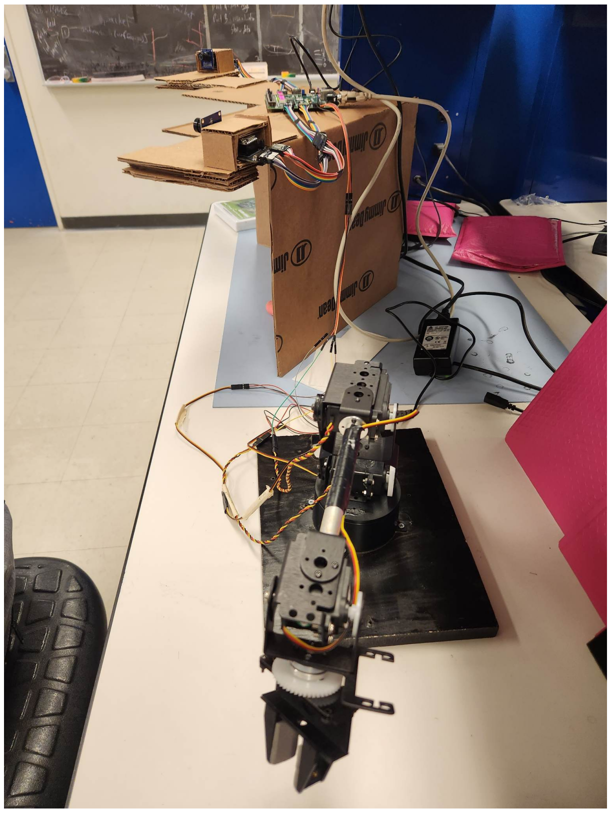


Fig. 7. Robot Setup

Throughout this section, we define hcount and vcount to be 11 bit and 10 bit inputs indicating the current h and v coordinates from VGA Gen in Top Level, respectively.

A. Initial Camera Data

The cameras send their pixel data into the NEXYS 4 DDR FPGA, and the pixel information are appropriately rotated, mirrored, scaled, and stored into a BRAM for further processing, using submodules from labs 3-4. Each camera frame reads left to right, then top to bottom, with dimensions 240x320. The pixels then undergo a masking filter after converting the pixel data into YCrCb, which represents each pixel as logic High if the original pixel reaches a threshold of chrominance red, and a logic Low otherwise, called *mask*.

B. Calibrate Blobs

The button *btnu* indicates a cue for calibration in the next full video frame. The human arm must be vertically positioned in front of the two cameras when calibrating, with no two blobs connecting, as shown in Fig. 9. The submodule iterates over the mask value at each coordinate, reading left to right then top to bottom with hcount and vcount inputs. The submodule then identifies a rectangular box that encloses each blob by roughly finding the beginning and end coordinates of a conglomeration of masked pixels. It then outputs the coordinates of the centers of the four boxes for a single clock cycle at the end of the video frame.

On a lower level, the algorithm works as follows, with everything in sequential logic:

Important variables:

- data_in - Single bit input; High if the pixel at hcount, vcount is pink, low otherwise.
- data_cache - 3 bit storage of the most recent 3 data_in, resets to 3'b0 when hcount = 0.
- is_colored - Single bit indicating whether that row of pixels has had masked pixels yet. Resets to Low 1'b0 at hcount = 0, or the beginning of each row.
- box_num - Four bit one-hot encoding indicating which blob the module is currently looking for.
- box_made - Single bit indicating whether a box has been made yet for the current box_num blob.
- (tlh_X, tlv_X) - Top left coordinates of the X'th blob.
- (brh_X, brv_X) - Bottom right coordinates of the X'th blob.

Pseudocode and Explanation:

```
If reset or calibrate:
    (tlh_X, tlv_X) = (11'd240, 10'd320)
    (brh_X, brv_X) = (0, 0)
    box_num = first blob
    is_colored, data_cache, box_made = 0
```

The module keeps track of the top left and the bottom right (h, v) coordinates for each of the four blobs, starting with the top blob. When the user resets or calibrates the system, all values are initialized, and we tart looking at the first blob.

```
If (calibrate called and
    hcount = 11'1024 and vcount = 10'd768):
    Start calibrating the next clock cycle
```

If there's a calibration cue, the module waits until the next frame to start the actual calibration.

```
If calibrating:
    Pipeline hcount and vcount
    If hcount = 1, reset data_cache.
    Otherwise, update data_cache
```

Once actually calibrating, the data_cache updates, or resets if in a new row. Hcount and vcount are also pipelined once to account for the sequential data_cache update.

```
If calibrating:
    If data_cache = 3'b111 and ˜is_colored:
        If box_num = blob X:
            If hcount < tlh_X:
                tlh_X = hcount
            If ˜box_made:
                box_made = 1
                tlv_X = vcount
            brv_X = vcount
```

If there have been 3 pink pixels in a row horizontally, it's likely that this pixel is a part of a blob rather than noise. If it's the first pink pixel of the row (is_colored) and its h coordinate

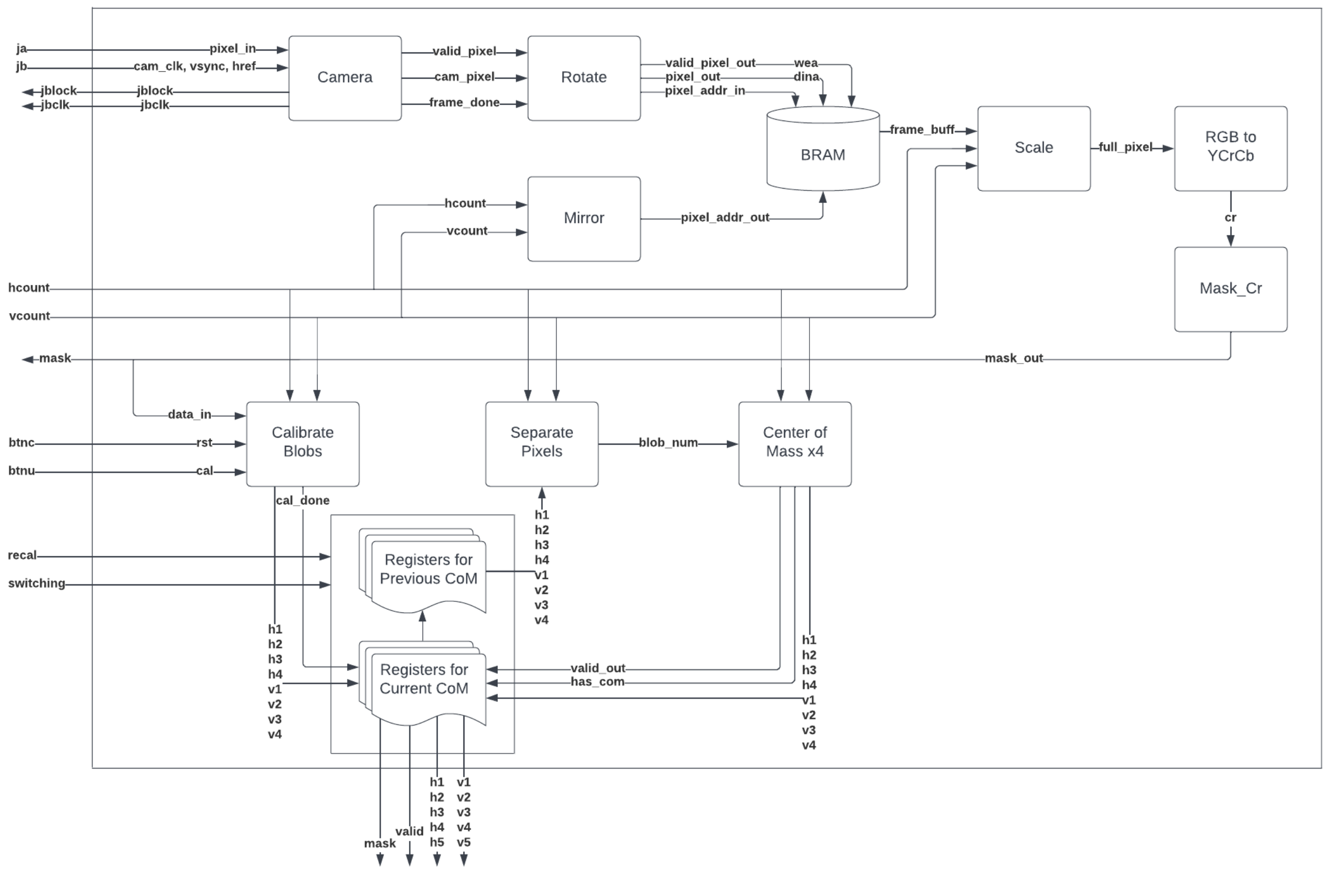


Fig. 8. Block Diagram for Image Processing

is not yet enclosed in the blob's current box, then we update the top left h coordinate. If this is the first pixel in this box, we also know the top left v coordinate is vcount. Since we go top to bottom, we know vcount updates the bottom right v coordinate too.

```
If calibrating:
    If data_cache = 3'b000:
        If is_colored for blob X:
            is_colored = 0
            If hcount > brh_X:
                brh_X = hcount
```

If there are 3 non-pink pixels in a row horizontally, it is likely we're not in a blob. If we are just getting out of a blob and know that hcount updates the bottom right h coordinate.

```
    If data_cache = 3'b000:
        If box_made for blob X:
            If vcount - brv_X > 5:
                box_made = 0;
                If brv_1 - tlv_1 <= 5
                    reset tl and br coords
                    for X to initial vals
                Otherwise:
                            Update to box_num
```

This part checks whether we should move on to the next blob or discard the current blob if it exists. We only consider moving on or discarding a blob if we've had 5 rows of no pink pixels. If so, we check whether the height of the current blob is too small to be considered a valid blob. If it's too small, it's likely noise, and we can reset the blob's enclosing coordinates to recalculate with the following pixels. Otherwise, we can move on to finding the next blob.

```
    If hcount = 1024 and vcount = 768:
        Output valid = 1
        Output h_X = avg tlh_X and brh_X
        Output v_X = avg tlv_X and brv_X
        No longer calibrating

    If output valid == 1:
        Output valid = 0
        Reset all values
```

Once we've seen an entire camera frame, we output the coordinates of the center of each blob box we created for one clock cycle.

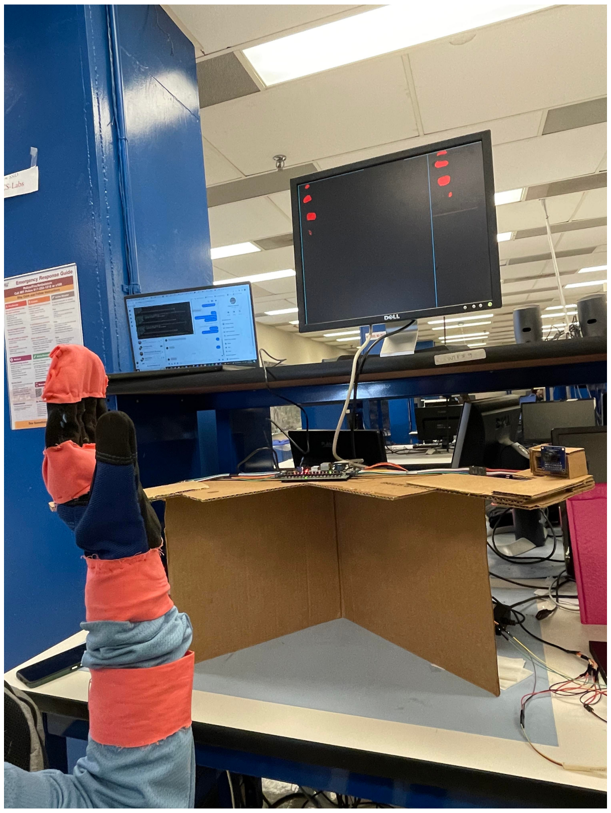


Fig. 9. Calibration Setup

C. Separate Pixels

At each subsequent camera frame, the masked pixels are separated between the four blobs based on the previous frame's coordinates. This submodule takes the inputs a single bit indicating whether that pixel is pink or not and the previous (h, v) coordinates of each of the four blobs. For each pink pixel, the submodule calculates the distance squared between the pixel's coordinates and each of the previous CoM coordinates. The pixel should be associated with the joint that has the shortest calculated distance squared. The submodule outputs a one-hot encoding indicating which blob the pixel is a part of, with 4'b0001 being blob 1 (fingers), 4'b1000 being blob 4 (forearm), and 4'b0000 if the pixel is not masked or not a part of any blob. This submodule has zero latency (fully combinational logic).

This algorithm to reassign pixels to their closest previous blob can allow outlier noise pixels to skew subsequent CoM calculations. A version was tested where the pixels were only assigned to its blob if both the h and v distances between the pixel and blob CoM are less than a threshold, but this skewed our CoM calculations by eliminating pixels that are part of large blobs, making some large blobs have a CoM at its edge. Raising the threshold to safely include all pixels in large blobs ended up not showing a significant enough effect on eliminating outliers, so the final system does not perform this check.

D. Center of Mass

The image processing module has four instances of the Center of Mass submodule, one for each blob. If the blob's corresponding bit from the Separate Pixels output is high, it contributes to the blob's next center of mass coordinate calculation. The submodule sums all the h and v coordinates of its blob's pixels, then divides it by the number of pixels at the end of a camera frame. The division occurs with a Divisor submodule, which has a variable latency depending on the size of division and outputs a valid CoM coordinate until the next CoM calculation starts.

E. Updating Coordinates in Image Processing Module

The Image Processing module has registers for the current joint coordinates and the previous joint coordinates. After a calibration, the resulting calibration coordinates are stored in both the current and previous joint coordinates for the next frame's coordinate calculations. After calibration is completed, the module sequentially updates its previous CoM coordinates with its current coordinates.

When the human arm bends in a way such that a blob no longer shows any masked pixels in the camera (due to lighting or being covered), the system should still preserve it's location for when the blob reappears in a future frame. In other words, if there are no coordinates being passed into a Center of Mass instance for a frame, then the blob's coordinates should not update. Examples of these situations are shown earlier in Fig. 4.

Blobs that fade out are also more susceptible to being skewed by noise, since nearby outliers hold greater weight in the CoM calculations. In these situations, the newly calculated coordinates would be a large distance from the current coordinates. Thus, the module combinationally calculates the h and v distances between a new CoM and current CoM, and only updates the current CoM if the h and v distances are small enough, or less than 40.

The Image Processing module outputs five coordinates; four being the CoM coordinates of the four blobs, and the fifth being (0, v of 4th blob). This allows the bottom joint of the robot arm to be calculated as the angle between the 4th blob and the horizontal plane.

F. Re-Calibration in Image Processing Module

A problem posed by this algorithm is that when the hand bends such that one blob is directly behind another blob in one camera's view, that camera cannot differentiate between the two blobs when the re-emerge as two separate blobs. This can cause the coordinates of two blobs to swap. After testing different scenarios, we saw that the forearm blob does not have this issue given the physiology of a human arm and the frame that the cameras can capture. Thus, the system needs to re-calibrate the CoM coordinates for the first three blobs when the coordinates get swapped between blobs.

Note that the two cameras share the same vertical v axis, so we know that two coordinates have been swapped if the vertical ordering of blobs in one camera is different than the vertical ordering of blobs in the other camera. Also note that given our algorithm, even if two CoM coordinates are swapped, the two coordinates are still coordinates of an existing blob, just not the correct blob assignment, so all we need to do is swap the coordinate values. This requires comparing the two camera's Image Processing outputs in the Top Level, then passing in a bit back into Image Processing to indicate whether to re-calibrate, and information on how to re-calibrate.

We orient and move our human arm in the physical system so that Camera 2 will never have one blob hiding behind another blob; Camera 2 sees more of the side of the hand and less of the palm, as we see in Fig. 4. Then, Camera 2 should never have the problem listed above, so its vertical CoM ordering should always be correct, and we can re-calibrate Camera 1's blobs based on Camera 2's blobs coordinates. The vertical ordering of camera 2 is the most accurate representation of the actual blobs' vertical ordering when the arm is in a vertical position, or if blob1 is above blob2, above blob3, above blob4. Thus, in Top Level, we do a combinational loop checking if both Image Processing values are valid, and if Camera 2's blobs' vertical coordinates are ordered blob1, blob2, blob3 top to bottom. If so, we check the order of Camera 1's blobs' vertical coordinates. If the order is incorrect, we send a High 1'b1 into Camera 1's Image Processing instance for re-calibration, as well as a number indicating which blobs' CoM's need to be switched (Camera 2's re-calibration is hardwired to Low). Then, in Camera 1's Image Processing module, if recal is High, all the necessary current and previous coordinates are appropriately swapped for the next clock cycle.

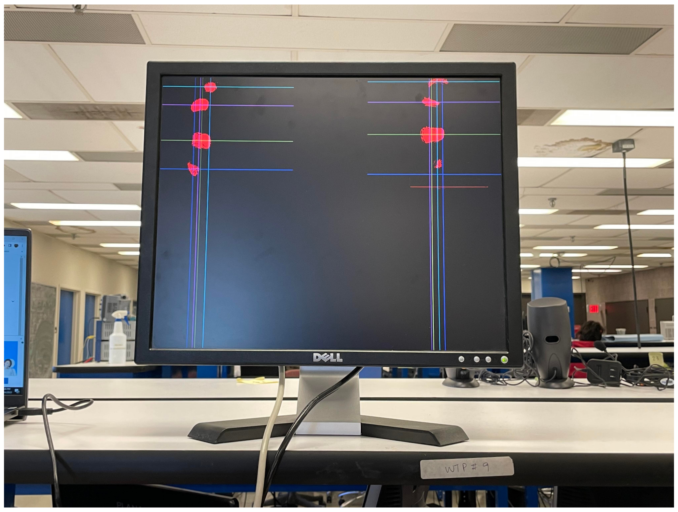


Fig. 10. VGA Display of Both Cameras and Their Blob CoMs

G. Testing and Debugging

The system displays both of the cameras' masked frame and crosshairs at each blob's coordinates on a monitor via VGA connections, as seen in Fig. X. Camera 2 is displayed on the left, and Camera 1 is displayed on the right (750 pixels to the right). To do this, we use a BRAM to store the mask values of Camera 1, then read them out 750 hcount pixels later.

H. Trade Offs with Other Algorithms

I originally considered a different approach for separating pixels into the four different blobs. Instead of assigning each pixel to its nearest blob, I considered detecting the locations of conglomerations of blobs in each frame, similar to the Calibrate Blobs module, then assigning each blob's center to the closest previous coordinates. However, this approach cannot distinguish between the two situations shown earlier in Fig. 4. Two blobs that are adjacent would result in the same output as two blobs where one is hiding behind the other, and it poses a challenging problem of obtaining accurate coordinates from a merged blob that represents two joints. The algorithm we decided on is more accurate for these situations because it associates each pink pixel to a proper joint, so two merged blobs can still be split accurately rather speculating how to split a conglomeration of pixels.

V. Angle Calculation

From our image processing system, we are able to identify the center of mass points on a given frame, however, we cannot directly interface the mechanical arm with the coordinate points. In order to translate the center of mass coordinate points from the cameras into angles that are programmed onto the robotic arm, we developed the joint angle calculation (JANC) module. The JANC module mainly does the bulk of the calculations but utilizes instances of two submodules to assist in calculations: the root module and the CORDIC module. The high level explanation of how the JANC module calculates the area is given in the following section with later section elaborating further.

$$|A| * |B| * \cos(\theta) = A \cdot B \tag{1}$$

$$|A| * |B| * \sin(\theta) = |A \times B| \tag{2}$$

$$\sin(\theta)/\cos(\theta) = \tan(\theta) = |A \times B|/(A \cdot B) \tag{3}$$

$$\theta = \arctan(|A \times B|/(A \cdot B)) \tag{4}$$

The general idea given in eq. (4), which relates theta to two vectors. Knowing that the dot product is equivalent to the product of both vectors and the cosine of the angle between them (1), and that cross product is the product with the sine of the angle between the vectors (2), a formula can be derived for the tangent of the angle between the vectors (3) and subsequently the angle itself. This is particular useful as representing the angle as an arctangent of a cross product

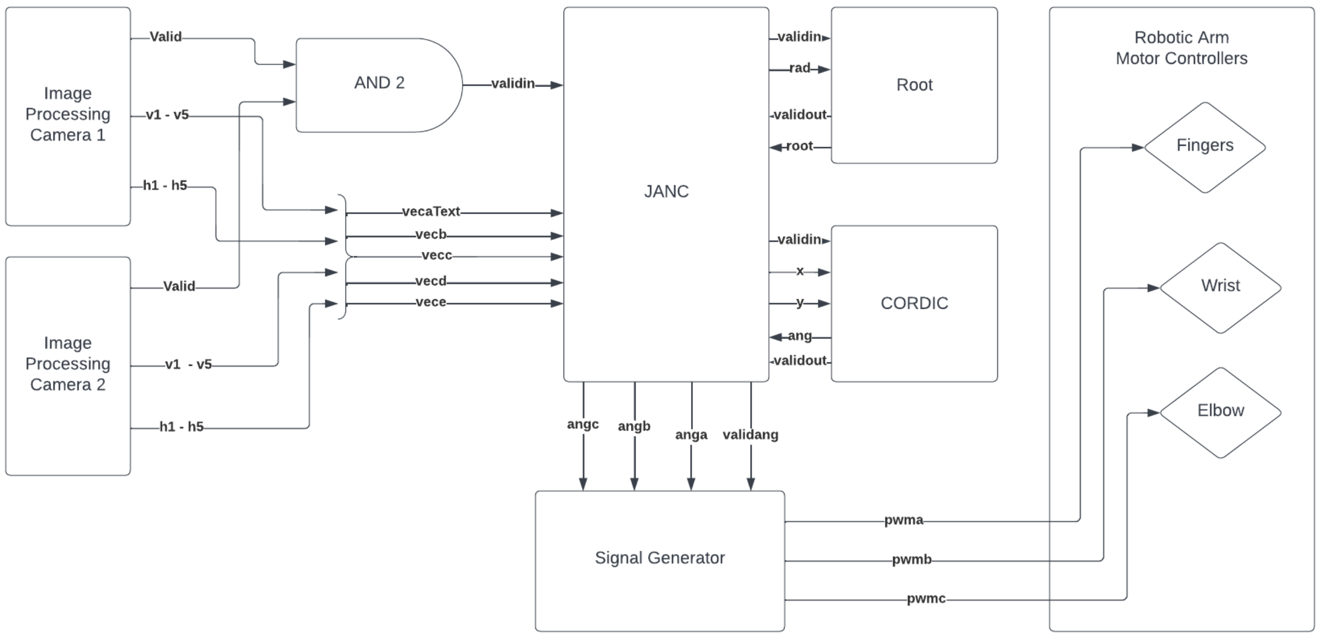


Fig. 11. Block Diagram for Angle Calculation and Mechanical Response

and a dot product simplifies the calculations the FPGA must preform. Dot products are a sum of products, which can efficiently calculated on the FPGA. The magnitude of the cross product is similar to the dot product but requires the use of the root module, which will be discussed later, to take the square root. Arctangent is a particular special case as it can be estimated using the CORDIC algorithm and the CORDIC module, reducing the size of the lookup table significantly.

A. JANC Setup

The JANC module has a one bit validin input and five 3D vector inputs each with a bitwidth of 11. Before the outputs from the image processing system can be used in the JANC module, it must be slightly altered since the system requires two cameras. Firstly, the valid bits from each camera must be passed through an AND gate, ensuring that the module only starts when both frames are processed and ready. Each instance of the image processing system will output vertical counts and horizontal counts, which represents the (x, y) coordinates of the center of mass points. Each camera produces its own set of five center of mass coordinates, which are concatenated with the other camera's outputs to extrapolate the point from 2D to 3D space. Since the cameras are orthogonal to each other, they share the same height value allowing for concatenation between the center of mass coordinates.

$$(x, y) + (z, y) \rightarrow (x, V - y, H - z) \qquad (5)$$

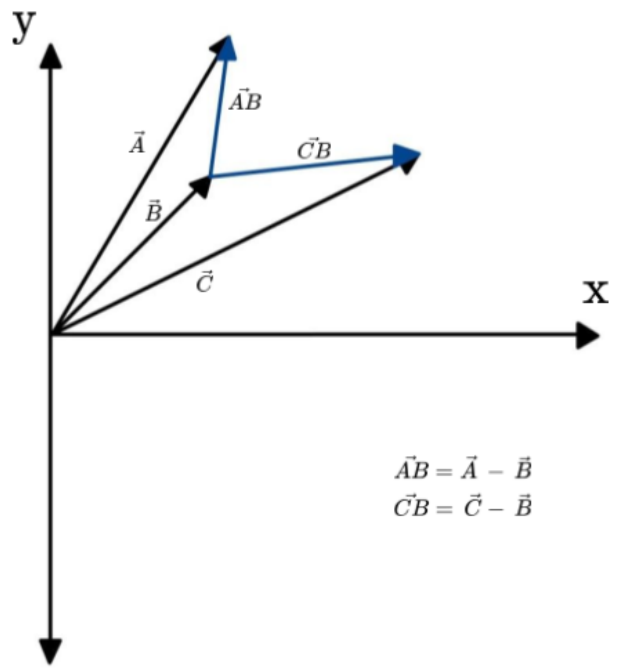
The full translation is given in eq. (5) where (x, y) represents a coordinate pair on camera A and (z, y) represents the same coordinate pair on camera B. The values V and H are the vertical and horizontal dimensions of the frame respectively. Both of the cameras have origins at the top left of the frame so to translate to 3D space where each axis originate from the same point, y must map to V - y while z must map to H - z. In our design, the dimensions of our frame is 240x320, translating to H = 240 pixels and V = 320 pixels.

B. JANC: IDLE

Once valid 3D center of mass coordinate points are received, the Joint Angle Calculation module is able to proceed with its calculation. JANC uses a finite state machine to control its operation, with four main states of operation being: IDLE, VECTORPRODUCTS, ROOTCALC, and CORDIC. The module begins on and resets to IDLE, where it waits for a high of the validin input wire, signaling a set of new coordinate points. For the remainder of this paper, we will refer to these set of coordinate points as vectors on our 3D representation of space, with each incoming vector originating from our set origin. Upon receiving this signal, the module will reset from any previous calculations, store the difference of each pair of vectors, which we will refer to as joint vectors, into registers, and progress the state to VECTORPRODUCTS.

```
If validin:
  Repeat for all coordinate pairs:
    vector21 = COM_point1 - COM_point2
  state = VECTORPRODUCTS
```

Recall that in our coordinate system, every vector is relative to a set origin, so we must make each vector relative to their respective joint by translating the vectors such that two vectors



y
x
$\vec{A}$
$\vec{AB}$
$\vec{CB}$
$\vec{B}$
$\vec{C}$
$\vec{AB} = \vec{A} - \vec{B}$
$\vec{CB} = \vec{C} - \vec{B}$

Fig. 12. 2D Translation Example

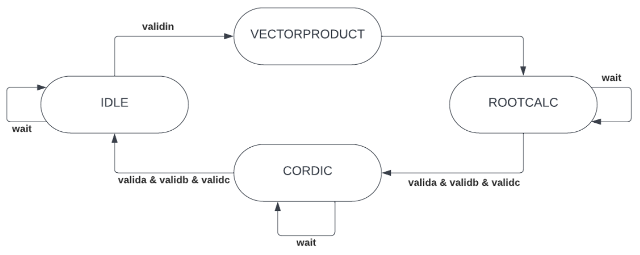


Fig. 13. JANC FSM Diagram

will originate from the third vector, forming the angle we are looking for. A 2D example is shown in Fig. 12, where the angle of interest, joint angle at B, is the angle between vector AB and vector CB. Similarly, our system will take a set of 3D vectors and translate it.

C. JANC: VECTORPRODUCTS

The VECTORPRODUCTS states serves as an intermediary state to calculate dot products and cross products of the two vectors. During this state, the dot product and cross product for all of the angles are calculated in parallel with a maximum of three multiplication operations in series. The dot product is calculated completely while the cross products are calculated for each direction. The state transitions to ROOTCALC.

```
dot = vec[0]² + vec[1] ^2 = vec[2] ^2
cross[k] = vec[i] * vec[j] - vec[j] * vec[i]
state = ROOTCALC
vectors abbreviated to vec
```

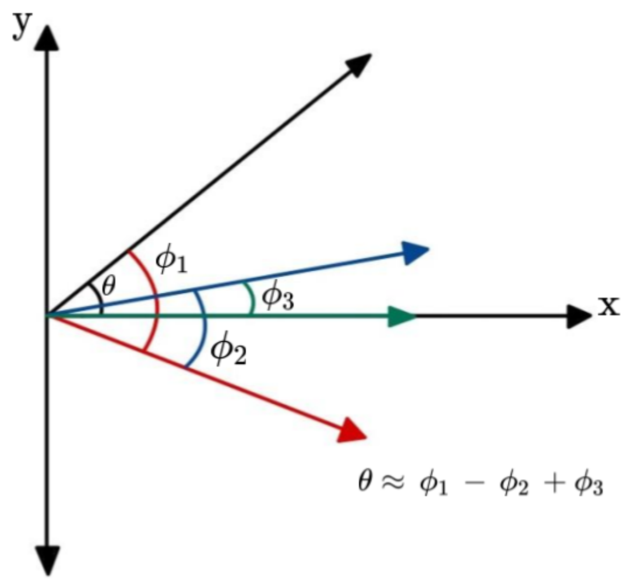


Fig. 14. Three Cycle CORDIC Example

D. JANC: ROOTCALC

In the ROOTCALC state, the cross product from the previous state is taken. The magnitude of the cross product is needed for the CORDIC module so this state calculates the y value that will be passed into the CORDIC module. This is done over a number of cycles first by taking the sum of products of each unit vector in the cross product.

```
crossmagnitudesquared =
cross[k]² + cross[i]² + cross[j]²
```

After getting the squared magnitude of the cross product, it is sent to the ROOT module along with a validin signal. The ROOT module is discussed in further detail in the following section. The ROOT module acts as a minor finite state machine, which the JANC module must wait for all three square roots to be calculated to continue. Once all cross products magnitudes have been calculated, the module transitions to the CORDIC state.

E. JANC: CORDIC

In this state, the dot product is passed in as the x value, the magnitude of the cross product is passed in as the y value and a validin signal is pulsed to indicate a new calculation request. Refering back to eq. (4), CORDIC replaces the need for an arctan and estimates the angle. Similar to the square root calculations, the JANC module must wait until all angle calculations are done, at which it will output the angles and pulse a valid out signal before resetting and return back to IDLE. A notable detail is that due to the CORDIC algorithm only being able to calculate angles between 0 degrees and 90 degrees, if there is an obtuse angle, which is identitfied by have a negative dot product and x value, the JANC module must pass in the absolute value of x. This is to calculate the supplementary angle so that it can be subtracted from 180 degrees to get the true joint angle. Further details about the CORDIC module and CORDIC algorithm is elaborated in later in this paper.

VI. ROOT MODULE

To calculate the square root of a number, we developed our own simplified integer square root calculator. The module uses binary search to estimate the square root of a given number. For an n-bit number, where n ¡= 32, the module will return the square root in n/2 cycles. The module utilizes a finite state machine with the states: IDLE, SIMPLIFY, and CALC.

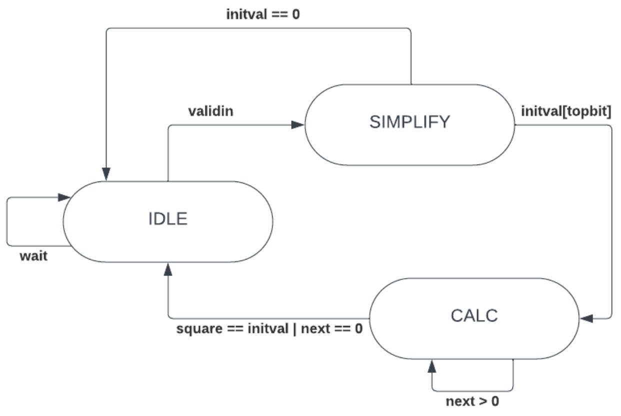


Fig. 15. root module FSM Diagram

A. root: IDLE

The IDLE section is the start of a new calculation. Upon a validin pulse, the initial value is set to the radical, the state is set to SIMPLIFY, and the topbit set to 31. The topbit is used in the SIMPLIFY state to reduce the number of cycles taken for larger n values.

```
If (validin):
  initval = radical
  state = SIMPLIFY
  top = 5'b31
```

B. root: SIMPLIFY

As the name implies, this state aims to simplify the problem for larger bit widths. The topbit variable is slow decremented to find the most significant bit in the rad value. Once the MSB is located, the guess value is set to half the bitwidth of the value. This is done due since multiplication commonly doubles the bitwidth of a value, so by starting the guess at n/2 bits, the number of guesses is reduced by approximately n/2, which simplifies the problem for larger n values. Along with guess, square is set to the square of guess, stat is set to CALC, and the variable ”next” is bit shift to the right by topbit bits. The main use for simplifying the guessing is so that the size of the registers used for guessing is kept small. For example, if rad is a 32-bit integer, without simplification guess would start at half the value of rad which is a 31 bit integer, thus the square register would need to be at least 62-bit wide. However with the simplification, the guess would start at 16 bits and so square can be kept at 32-bits.

```
If n = topbit:n = radical bandwidth
  guess = radical >>> topbit/2
  square = guess * guess
  next = radical >>> topbit/2 + 1
Else:
  topbit = topbit - 1
```

C. root: CALC

Every cycle, during calculation state, the root module checks if guess is correct by comparing square with the saved rad value. If the square equals the rad value, then the guess is correct, the output register root is set to the square value, validout is pulsed for one cycle, and the state returns to IDLE. If the guess is not correct, the next guess is calculated by adding next if guess was too lower or subtracting next if guess was too high. In the case that the next value is 0, the search is exhausted and the one less than the current guess value is returned. It is one less since we are aiming to underestimate.

```
If square = rad:
    return guess
    pulse validout
    state = IDLE
Else if next = 0:
    return guess - 1
    pulse validout
    state = IDLE
Else if (square > rad)
    guess = guess - next
    square = guess * guess
Else if (square < rad)
    guess = guess + next
    square = guess * guess
next >>> 1
```

VII. CORDIC ALGORITHM

The coordinate rotation digital computer, or CORDIC, algorithm is an efficient way to estimate the joint angle without having to calculate inverse trigonometric functions. The algorithm takes advantage of rotations to implement binary search to approximate the angle between a vector in 2D space and the horizontal axis of the plane it lies on. The vector is rotated around the origin, with each angle rotation being saved, the sum of all angle rotations when the vector aligns with the horizontal axis will give a reasonable approximation to the value of the angle.

We implemented a simplified CORDIC module to estimate the joint angles of the human arm at a given frame. For our module, we set the max number of cycles per estimate to be 10 cycles, thus results should always come within that time frame. The range of angles the module can calculate is from 0 degrees to 90 degrees with a worst observed accuracy of ±2 degrees. This limitation is due to how CORDIC operates as the y value is used to determine when the calculation is finished. Our module is a two state FSM that accepts an x and y value with a valid in signal and outputs the angle between them with a valid out signal. Additionally for simplification, we represent angles as a 13-bit value that is 10 times the actual angle. i.e. 45 degress is represented as 450, 69.9 degrees is represented as 699. We only care up to one place after the decimal value since we are just estimating. This representation avoids the use of decimal bits and allows for much better accuracy than if we had excluded the decimal bits.

$$\begin{aligned} x &= x * \cos(\theta) - y * \sin(\theta) \\ y &= y * \cos(\theta) + x * \sin(\theta) \end{aligned} \quad (6)$$

$$\begin{aligned} x &= x - y * \tan(\theta) \\ y &= y + x * \tan(\theta) \end{aligned} \quad (7)$$

$$\begin{aligned} x &= x - (y >> i) \\ y &= y + (x >> i) \end{aligned} \quad (8)$$

For our implementation we begin with the initial rotation matrix for a point around the origin, given by eq. (6). This is the general rotation matrix for a vector in 2D space around the origin. We can simplify eq. (6) further into eq. (7) since we only care about the angle, giving us an equation with tangent. However, by using certain theta values (i.e. taking theta where tan(theta) = 1, 1/2, 1/4, 1/8 etc), the tangent can be simplified bit shifts as shown in eq. (8) where i is current loop iteration. Although this introduces error, bit shift takes up less space than utilizing a lookup table which was an initial concern when implementing the module.

A. CORDIC: IDLE

During its IDLE state, the module is ready to accept new values. On a valid input signal indicating new x and y coordinates, the CORDIC module resets all previous values and begins a new estimation. The values of x and y are saved into a register, the angle is set to zero, the counter is reset and the state is set to CALC.

```
If validin:
  angle = 0
  xreg = x
  yreg = y
  counter = 0
  state = CALC
```

B. CORDIC: CALC

On each cycle, the module will first check whether the yreg value of rotated vector is 0 or if the cycle count is at 10. If it is the former, the vector is aligned with the horizontal axis and the output angle is set the angle sum. If it is the latter, even though yreg is not 0, the angle adjustments are too small after 10 cycles to make a difference so this will limit the number of cycles that the CORDIC module is spend to calculate an angle. The module also checks whether the x value is 0, which normally only occurs at the very beginning of the algorithm, or at cycle 0. If the xreg value is 0, the vector is perpendicular to the horitzontal axis and the angle is set to 90 degrees. In both cases, the angles are set to their respective values, a valid out signal is pulsed and the state is set back to IDLE.

```
If xreg = 0:
  angle = 90 degree
  state = IDLE
  validout = 1
Else if yreg = 0 or counter = 10:
  angle = sumangle
  state = IDLE
  validout = 1
```

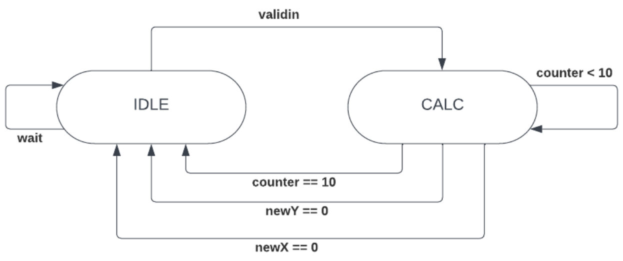


Fig. 16. CORDIC FSM Diagram

If the y value is not 0, the vector is rotated: clockwise if the yreg value is positive and counterclockwise if the yreg value is negative. Each time this occurs, the angle adjustment is approximately halved, starting from an initial value of 45 degrees. As mentioned previous, the actual angle adjustments are chosen such that when rotating the vector, the xreg and yreg adjustments values can be calculated with only bit-shifts. These angle adjustments are: [45, 26.5, 14, 7.1, 3.5, 1.8, 0.9, 0.4, 0.2, 0.1]. The angle adjustment are added to the sum angle if yreg ¿ 0 and subtracted from the sum angle if yreg ¡ 0, and the value depends on the current loop iteration, with earlier loops having more weight on the angle sum. With each iteration, the values of the xreg and yreg are also updated to reflect the rotation.

```
If yreg < 0:
    sumangle = sumangle - adjustments[i]
    xreg = xreg - (yreg >> i)
    yreg = yreg + (xreg >> i)
Else:
    sumangle = sumangle + adjustments[i]
    xreg = xreg + (yreg >> i)
    yreg = yreg - (xreg >> i)
```

VIII. Mechanical Response

The mechanical response requires some outside circuitry and setup for the mechanical arm to operate but is fairly straightforward in execution. The angles from the previous Joint Angle Calculation module are translated into PWM signals in the Signal Generator module, and output to an outside amplifier circuit which will act as the signal input for the mechanical arm.

A. Signal Generator

The Signal Generator module receives the three angles, angleA, angleB, angleC, as an input and generates a PWM signal as an output out of the ports xa_p[2:0], with a duty cycle that corresponds to the angles. Note that the angles are passed in to the tenths, as in 180 degrees is passed as 1800 rather than 180. This module should also run on the system 65 MHz, but after researching the servos motors, the PWM output must be at 50 Hz with duty cycle linearly increasing with angle such that 0 degrees is a 5% duty cycle and 180 degrees is a 10% duty cycle. Note that the PWM's rising edge signifies the beginning of a 50 Hz clock cycle.

Important variables:

- counter - counts how many 65 MHz clock cycles have passed since the current 50 Hz clock cycle has started.
- MAX - the number of 65 MHz clock cycles in a 50 Hz clock cycle. Calculates to $65 \cdot 10^6 \div 50 = 1300000$.
- targetA - number of 65 MHz clock cycles in the high pulse of the PWM to drive the motor's angle to angleA. We know two coordinates for (angle, target) are $(0, \text{MAX} \cdot 5\%) = (0, 65000)$ and $(1800, \text{MAX} \cdot 10\%) = (1800, 130000)$. Thus,

$$\begin{aligned} \text{targetA} &= 65000 + \text{angleA} \cdot \frac{130000 - 65000}{1800 - 0} \\ &= 65000 + \text{angleA} \cdot 36. \end{aligned}$$

Pseudocode (all sequential logic):

```
If (counter == MAX - 1):
    counter <= 0
    targetA <= 65000 +
        (avg of most recent 4 angleAs)*36;
    Same for target B, C
```

Once we've output an entire 50 Hz clock cycle, we start a new clock cycle with the next angle and update targetA, B, C accordingly. We take the running average of 4 most recent angle inputs.

```
Otherwise:
    Output pwmA <= 1 if counter < targetA
                  0 otherwise
    Increment counter
    Same for pwm B, C
```

Our PWM signals should output 1 if we're still in the high pulse and 0 otherwise.

```
If valid angle input:
    angA_pipe <= angA
    angB_pipe <= 1800 - angB
    angC_pipe <= 1800 - angC
    Pipeline each angle 3 more times
```

Based on the geometry of the robot arm and the direction each motor rotates, we want angA (wrist angle) to be as is, angB (elbow angle) to be the supplementary angle (since 0 degrees is a straight robot elbow joint), and angC to be the supplementary angle.

In the actual code, we offset targetA and targetC by 60000 instead of 65000, and targetB by 50000. This is because after some trial and error, we saw that these offsets produce more accurate responses, likely because of the external forces acting on the robot arm that prevent it from its ideal response, to be discussed in Challenges and Potential Improvements later.

IX. Design Evaluation

A. Memory

The Image Processing components utilize three BRAMS:

- Camera 1's pixel data memory - RAM width of 16 bits and RAM depth of 320*240.
- Camera 2's pixel data memory - RAM width of 16 bits and RAM depth of 320*240.
- Camera 1's VGA output memory - RAM width of 1 bit and RAM depth of 320*240.

The Angle Calculation and Signal Generator modules don't utilize any BRAMs.

We can eliminate the third BRAM if we didn't need to display both cameras on the VGA monitor at the same time. Initially, we didn't use the third BRAM because we used a switch to toggle which camera's information to display. However, only showing one camera's data at a time made it difficult to test and debug the full system, so we found it necessary to display both and utilize this BRAM.

B. Latency

Regarding the Image Processing components, the following outlines the latency of the sequential submodules:

- mirror - 2
- BRAMs - 2
- rgb_to_ycrcb - 3
- center_of_ mass - variable latency

The latency within the Image Processing module requires pipelining hcount and vcount 7 clock cycles for Calibrate Blobs, Separate Pixels, and Center of Mass, and the Center of Mass outputs at variable latencies. However, since the outputs of Image Processing are only updated once every camera frame, which is at fastest $1024 * 768 = 786,432$ clock cycles because of the VGA system, the latency within Image Processing are not a problem in providing pipelined data to the rest of the system.

With regards to the Angle Calculation, the max, worst case latency of the submodules are:

- JANC - 6
- root - 32

- CORDIC - 11

The latency of the angle calculation is very large, totaling in at 49 cycles. However, it is still significantly less than the rate at which frames are updated. The angle calculation is fast enough that the main module will be finished and ready for the next frame long before the next frame arrives.

C. Resource Utilization

The following are some key resource utilization of the entire system:

- Slice Logic: 9.88%
- LUT: 6.42%
- Block RAM: 71.11%

Initially we were concerned about the resource utilization of our project, and had designed with this constraint in mind. We had expected the image processing to use a significant amount of SRAM, and ultimately it utilized around 71.11% of the total BRAM. Thus, we avoided trying to use BRAM in the angle calculation. Similarly, we expect the angle calculation to require significant amount of slice logic and lookup tables, which ultimately took up 9.88% and 6.42% respectively. This ended up being much lower than we had expected, though still quite significant. Overall, we had used a minimal amount of resources on the FPGA, as we never needed to optimize for timing due to being bottlenecked by the camera framerate.

D. Timing Constraints

Timing constraints become a main concern mainly within the modules that include heavy calculations, specifically our main concern was calculating dot products and cross products in the JANC module. Initially our approach was to calculate all products in a single clock cycle which had violated setup slack timing. So we had to break our product calculations into multiple cycles to meet the timing constraints. In our final build, we managed to achieve a WNS of 2.339 ns and a WHS of 0.001 ns. We managed to meet timing constraint.

E. Goal Evaluation

We met our stretch goal of driving 3 robot joints with our human arm. Because of the challenges listed below, though, the mechanical response is not as smooth or accurate as we expected.

Our system can be used in cases where high speed blob detection or angle calculation is needed. In particular, it would be useful in 3D image mapping and projection, as our system can be applied to edge detection to map edges, possibly useful in AR technology.

X. Challenges, Insights, and Potential Improvements

1) Camera: The cameras we interfaced with were not the best quality; they were very zoomed in and susceptible to noise. To safely fit our hand and forearm in the camera frames, we needed the cameras to be at least 20 inches away from the hand, which poses the problem of noise susceptibility when using such long wires between the camera and the FPGA board. Fortunately, having the cameras 22 inches away from the hand still gave accurate enough readings through the long wires, but it would be nice to improve this project with less noise-prone cameras.

2) Robotic Arm: Note that with the VGA system, each camera frame can be displayed in $1024 * 768 = 786,432$ clock cycles of 65 MHz. However, the robot arm runs on PWM of 50 Hz, which is $1,300,000$ clock cycles of 65 MHz, which is nearly twice as slow as the camera frames. This makes the robot arm the bottle neck of the entire system, making the entire system slow despite the impressively fast computations in the rest of the system.

Another problem with the robot arm is that gravitational forces from the motors and metals can act against the motor's movement, thus giving the robot arm a hard time opening all the way upwards when given a large robot elbow angle. This makes the response inaccurately reflect the angles calculated in the rest of the system. To solve this problem and the speed issue above, perhaps the only solution is to build a better and faster robot arm.

Another problem is that there is a delay between when the motors receive the next angles' pulse and the motor actually reaching that angle physically, which causes some backlog in the response. In addition to our running average of angles in our signal generator to smooth the change in angles per pulse, we could also potentially consider a feedback loop to make it more accurate.

3) Image Processing: The Image Processing module's calculations of coordinates are entirely estimations. The Calibration results are the centers of each blob's roughly estimated enclosing rectangles, and each subsequent coordinates can be skewed towards pink noise or not even accurately represent at all because a blob is hidden behind another blob. In addition, the human hand must be oriented a certain way (palm not facing camera 2) for the current algorithm, as mentioned earlier. Perhaps there are better algorithms to approach these problems and minimize estimation. If I were to continue this project, I would love to find a way to improve my algorithm so that the human hand can be oriented in any way and provide more accurate locations of each blob, perhaps through considering other characteristics at each pixel (such as lighting or other colors).

4) Angle Calculation: The angle calculation modules currently scales very terribly with increasing input bit width. When I initially designed the module, I had a focus on reducing the error as much as I can. This led to me making the registers used in the modules a set bit width as to save on area. While this does work for our system, it does not do well when the inputs have a different bit width than the chosen value. I would have to change every register bit width to accommodate. If I were to do this again, knowing that the resource utilization is not that much overall, I would use more parameterized modules to vary the bit width for my registers. Not only would that have made my modules more scalable if I choose to use them again in the future, it would have saved me a lot of time and headaches.

XI. Contributions

The overall project contribution was completed as follows:

A. Janette (Jan) Park

- Image Processing module, including VGA displays
- Building the physical setup, including sewing pink patches to the glove, building the cardboard camera setup, and building the wire extensions for the cameras
- Within the final report: everything related to Image Processing module; Project Checklist, Physical Hardware and Setup, Mechanical Response, Memory, Challenges, Insights, and Potential Improvements 1-3
- Top Level and Image Processing block diagrams

B. Thienan Nguyen

- Angle Calculation module
- Setting up and testing the functionality of the robot arm
- Within the final report: Everything related to Angle Calculation; Timing Constraints, Resource Utilization, Challenges, Insights, and Potential Improvements 4; General formatting
- Angle Calculation block diagram, Finite State Machine Diagrams

C. Together

- Fixing the robot when a motor broke and screws fell out.
- Signal Generator module; Thienan wrote the general algorithm, and Jan researched the Servos motors and calculated the correct numbers for the module.
- Within the final report: Abstract, Introduction, Interesting Challenges, Latency, Goal Evaluation

XII. References

We utilized submodules from labs 3-4 in class for the camera and VGA display.
https://www.servo-faqs/how-do-i-control-a-servo
https://www.eit.lth.se/fileadmin/eit/courses/eitf35/2017/CORDIC_For_Dummies.pdf