

FPGArm Final Report

Russell Perez-Tomas

*Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, USA
perezr@mit.edu*

Anthony Donegan

*Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, USA
ajzd@mit.edu*

Abstract—We present the FPGArm, a scalable system that integrates the high- and low-level control systems of a five-axis robotic arm into a single FPGA, directly translating keyboard inputs into updates to the arm’s physical configuration. The FPGArm system receives inputs through a PS/2 controller, uses a motion planner with cylindrical inverse kinematics to generate target joint-space angles, and actuates each servo in the arm using a PWM controller. The FPGArm successfully implements parallel control of all servos in the arm with lower latency than the servo refresh rate, joint-space and task-space motion planning, and sequence replaying for autonomy. The arm was evaluated by moving along a single axis in its task space, manipulating objects, and executing a series of recorded or pre-programmed movements.

Index Terms—Digital systems, FPGA, Robotic arm control, Inverse kinematics, Motion planning, CORDIC

I. INTRODUCTION

Robotic arms play a key role in multiple fields, from manufacturing to cooking, being able to perform repetitive, precise, and even hazardous tasks with unmatched consistency. Robotic arms are multi-servo systems that traditionally require a complex network of a high-level motion planner to determine joint space angles and low-level microcontrollers to individually control each servo. However, this interconnection complexity can lead to poor synchronization between the various servo motors. Furthermore, the serial execution of microcontroller code limits the real-time responsiveness of the system. Using specialized hardware for control, such as an FPGA, allows for true natively parallel control and near real-time responsiveness in hardware, solving both these problems.

To demonstrate this, we have designed and constructed the FPGArm, a full system folding high- and low-level controllers of a traditional arm into a single FPGA solution. The FPGArm system is capable of parsing PS/2 keyboard inputs, joint space jogging, cylindrical task space jogging, going to specified coordinates in the task space, and control sequence replaying. The FPGArm is capable of translating inputs into arm configuration updates faster than the 50 Hz servo refresh rate while maintaining stable inverse kinematics convergence and enforcing servo angle limits to prevent unsafe movements and damage to the arm.

II. OVERALL DESIGN

A. Physical Construction

The FPGArm system, seen in Figure 1 consists of:



Fig. 1. The physical construction of the FPGArm system. From left to right are the Cherry PS/2 keyboard, generic power supply, Adafruit Servo Shield, AMD Spartan 7 FPGA, and Lynxmotion SES-V1 five-axis robotic arm.

- A Lynxmotion SES-V1 five-axis robotic arm, which contains six individually PWM-controlled servos. The FPGArm replaced the control and power electronics of the arm, which were removed.
- A generic 5V 10A power supply and an Adafruit 16-Channel Servo Shield, which was used exclusively as a power distribution board for the arm. The PWM control component of the servo shield was left unconnected.
- A RealDigital Urbana board with an AMD Spartan 7 FPGA, which directly provides a PWM control signal to each of the six servos from its PMODB output and shares ground with the servo shield.
- A Cherry PS/2 keyboard and Diligent PS/2 to PMOD interface, which are used as an input controller for the FPGArm. The keyboard runs on the 3.3V provided by the FPGA and connects the data stream and clock to the FPGA’s PMODA input.

B. System Architecture

The high-level block diagram of the FPGArm system is shown in Figure 2. The FPGArm takes inputs directly from a PS/2 keyboard and provides a PWM control signal to each of the six servos in the five-axis robotic arm. The PS/2 controller

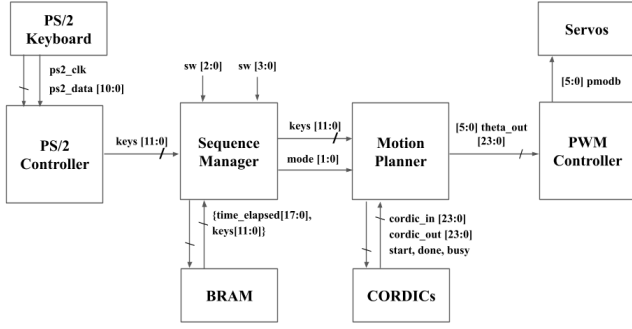


Fig. 2. The high level block diagram of the FPGArm. Greater focus will be given to the motion planner, the most complex module of the system.

module takes in the keyboard clock and data stream and produces a multi-hot data packet encoding the keys pressed, along with signals identifying the control mode of the arm and sequence controller. The sequence controller then takes the outputs of the PS/2 controller and, depending on its control mode, may store inputs in BRAM and replay them back. The motion planner module takes in the multi-hot keypress data packet and control mode, and depending on these inputs, will produce joint space angles using an inverse kinematics engine. These updated angles are then executed on each servo by the PWM controller, which drives their PWM line directly.

C. Data Representation

In the motion planner and PWM controller, 24-bit signed Q4.20 fixed-point numbers were used to represent both angles and positions whenever possible. The servos in the FPGArm system had angles in the range $-\pi/2 \leq \theta_i \leq \pi/2$ radians, and link lengths were well in the range $0 \leq L_i \leq 1$ meters. Using these units, these values and intermediate computations typically fit well within the range $-8 \leq Q4.20 \leq 8$ provided by the 4 integer bits, with the 20 fractional bits providing precision well beyond the physical resolution of the servos and accuracy of the CORDICs. When values were too large for the Q4.20 range, Q12.20 was used, with multiplication and other operations remaining relatively simple between Q12.20 and Q4.20.

III. PS/2 CONTROLLER

The PS/2 controller receives input from an external PS/2 keyboard through a PMOD interface and encodes the keys being pressed as a multi-hot data packet representing keystrokes that can be read easily by later modules.

A. PS/2 Receiver (Russell)

The PS/2 receiver module is a standard PS/2 keyboard interface, taking in the 11-bit serial output of the keyboard and internal clock. Following the PS/2 protocol, the module produces the 8-bit scan code and valid signal for the key currently being pressed. It watches the serial output of the keyboard for the start bit, reads the following 8-bit data packet,

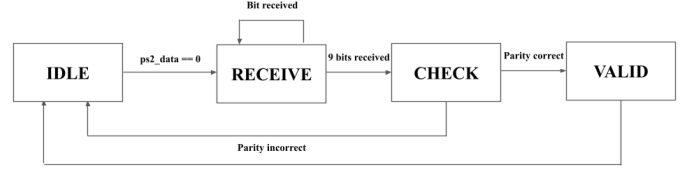


Fig. 3. The FSM for the PS/2 receiver module.

verifies its parity against the odd-parity bit, and then watches for a validating one-bit at the end of the packet. It uses the internal keyboard clock to progress through the FSM in Figure 3.

B. Encoder (Russell)

The encoder converts the scan code outputs of the PS/2 receiver into multi-hot data packet 12'b{wasd_tgfh_ijkl}, which encodes the keys being pressed, which can be easily used by later modules. In all control modes, the FPGArm requires two keys per servo/axis for six total servos and axes, hence requiring 12 total keys and a 12-bit encoding. The encoder watches the PS/2 receiver output for any of the keys [w, a, s, d, t, f, g, h, i, j, k, l] and ignores all other keys. It also watches for the BREAK signal, followed by the scan code for the key being released. Hence, based on when keys are pressed and released, we may encode multiple simultaneous keypresses.

IV. MOTION PLANNER

The motion planner module of the FPGArm takes the place of a high-level controller in a traditional system. It updates either the joint or task space position of the arm based on inputs provided from the PS/2 controller and produces the joint space angles necessary to execute these updates. The FPGArm is capable of both jogging in and going to specific coordinates in its cylindrical task space (ρ, z, ϕ, ψ) while also being able to directly jog its joint space angles $(\theta_0, \dots, \theta_5)$. A full motion planning solution for a Cartesian (x, y, z, ψ) task space was also developed and successful in simulation but found to be incompatible with the physical construction and servo limitations of the arm. In all modes, the rotation θ_4 and gripper actuation θ_5 of the end effector do not affect the arm's task space position and hence are always jogged directly in joint space.

The block diagram of the motion planner is given in Figure 4 with much of its complexity folded into the inverse kinematic module's FSM. Note that although it takes around 300 cycles to perform forward and inverse kinematics, the chosen FPGA has a clock rate of 100 MHz, while the servos refresh rate is around 50 Hz. This provides a window of around 2 million cycles to perform all the necessary computations to update the servo angles after a keyboard input, and hence the system has been significantly pipelined with no detriment to responsiveness.

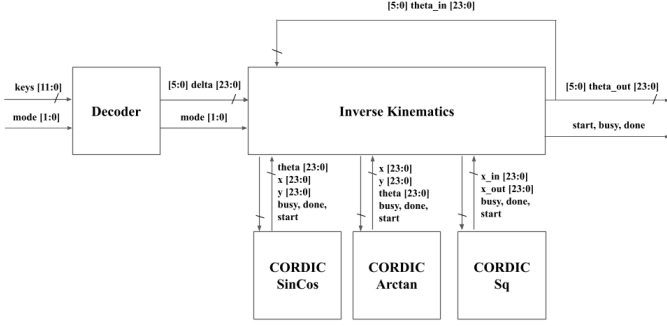


Fig. 4. The block diagram for the motion planner module. Note that the CORDIC modules shown here are themselves wrappers for CORDIC IPs.

A. Decoder (Both)

The decoder interprets the multi-hot encoded keypress data packet and, based on the control mode, sets the six signed Q4.20 delta values Δ_i , which are then fed into the inverse kinematics module. For joint and cylindrical task space jogging, each Δ_i represents an increment applied to an individual joint angle or task space position. The Δ_i values will be one of $\{+\delta, 0, -\delta\}$, where δ is some tuned step size. For going to coordinates in the task space, the Δ_i values directly represent the new coordinates the arm should go to instead of an increment, which is possible as updates and positions share the same Q4.20 representation. The rate of movement may be further tuned using the baud rate, whose period determines how long to wait between sending Δ_i values while a given key is held down. The inverse kinematics module takes a single set of deltas at a time, and once latched, will run for several hundred cycles uninterrupted, and so logic is in place to queue a set of Δ_i values to the inverse kinematics module while it is busy.

B. Inverse Kinematics (Anthony)

At a high level, the inverse kinematics module first performs forward kinematics, converting the joint space servo angles $(\theta_0, \dots, \theta_5)$ into a position in the cylindrical task space (ρ, z, ϕ, ψ) . It then updates the task space positions to obtain a set of task space target positions $(\rho_*, z_*, \phi_*, \psi_*)$, and lastly uses inverse kinematics to compute servo angles $(\theta_0^*, \dots, \theta_5^*)$ needed to obtain the target position within a safe range. One instantiation each of a $(\sin \theta, \cos \theta)$, $\arctan_2(y, x)$, and $\sqrt{x}$ CORDIC IP are time multiplexed and used to perform some of the computations for both forward and inverse kinematics. Although the method used for forward and inverse kinematics is unique to the degrees of freedom of the arm and its physical configuration, much of the computation used here is reusable for different arms and more or fewer degrees of freedom.

1) *Other Methods*: In our design, we chose to use the analytic solution for inverse kinematics in cylindrical coordinates with 4 effective degrees of freedom (cylindrical IK). We originally attempted to use the Jacobian Transpose method for inverse kinematics, a gradient descent-based method that

iteratively pushes the arm towards a given target position using the update rule

$$\theta_i \leftarrow \theta_i + \alpha \left(\frac{dp}{d\theta} \right)^T dp$$

where p is the (x, y, z, ψ) position vector [1]. However, in simulation this method failed to converge, likely due to the simulated limitations of the arm, and hence was not used.

As a result, we then implemented the analytic solution for inverse kinematics in Cartesian coordinates with 4 effective degrees of freedom (Cartesian IK), and this method performed well in simulation, providing joint angles that produced the desired position. This formulation required significant mathematical overhead, taking around 500 cycles to run, calling CORDIC IPs multiple times, and was implemented using a 27-state FSM. Upon deployment to the physical arm, it was discovered that the angles outputted by the Cartesian inverse kinematics were fundamentally incompatible with the physical limitations of the arm and servos we used. In particular, the Cartesian IK implementation we chose assumed a neutral position, which has the arm fully extended forward and flat to the ground, with the shoulder able to move down from this, which our arm could not do. Furthermore, our arm's neutral position was a "right angle" shape, where the upper arm link was vertical and the forearm link was forward horizontal, and hence fundamentally incompatible with Cartesian IK. As a result, although this solution worked in simulation, it was not used in the final hardware implementation.

2) *Cylindrical Inverse Kinematics*: Cylindrical IK was chosen as it could be easily modified to account for the physical limitations and configuration of the arm. The module is implemented as a 25-state FSM, which broadly can be split into the IDLE, FORWx, BUMPIN, CLIPIN, INVx, BUMPOUTx, CLIPOUT, and DONE states, as can be seen in the FSM diagram given in Figure 5. Note that for FORWx, INVx, and BUMPOUTx, to meet timing, several FSM stages were used to pipeline certain tasks in order to meet timing, although these all could be implemented as a single stage given the right timing logic. Due to the high clock rate of the FPGA relative to the servo, the module is able to latch onto a given $(\theta_{in}, \Delta_{in})_i$ input and performs its logic exclusively on that input until it completes, and hence could be implemented as a large FSM. In the IDLE state, the module waits for a $(\theta_{in}, \Delta_{in})_i$ input to latch onto and transitions into FORWx once latched while raising a busy flag.

3) *Forward Kinematics*: In the FORWx states, we would like to compute the height z , base rotation ϕ , radial distance from the z axis ρ , and pitch ψ of the arm from the joint space angles $(\theta_0, \theta_1, \theta_2, \theta_3)$. We are given link lengths L_1, L_2, L_3 , and L_4 , and here we combine the wrist and forearm length into $L_e = L_3 + L_4$. Due to the neutral position of the arm being a right angle as opposed to flat forward, shifted angles

$$t_1 = \theta_1 + \frac{\pi}{2} \quad t_2 = \theta_2 - \frac{\pi}{2} \quad t_{12} = \theta_1 + \theta_2$$

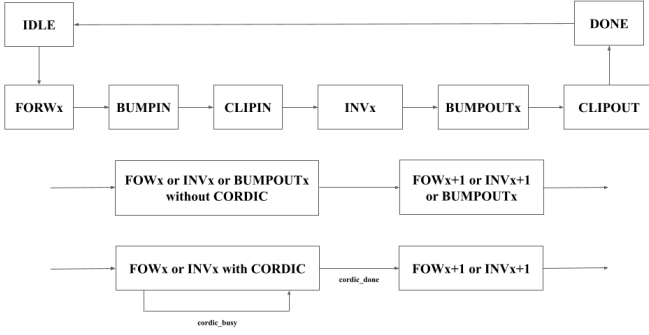


Fig. 5. The FSM for the inverse kinematics module. The FORWx, INVx, and BUMPOUTx states actually represent several states, with their state transitions shown below.

are used internally. The upper arm vector and forearm vectors are given as

$$\mathbf{v}_1 = (L_2 \cos t_1, L_2 \sin t_1) \quad \mathbf{v}_2 = (L_E \cos t_{12}, L_E \sin t_{12})$$

giving us total displacement from shoulder to end effector of

$$\mathbf{r} = \mathbf{v}_1 + \mathbf{v}_2 = (L_2 \cos t_1 + L_E \cos t_{12}, L_2 \sin t_1 + L_E \sin t_{12})$$

giving us height z' as the y component and radial distance ρ as the x component

$$\rho = L_2 \cos t_1 + L_E \cos t_{12} \quad z' = L_2 \sin t_1 + L_E \sin t_{12}$$

with z' being offset from the ground by L_1 and hence

$$z = L_1 + z' = L_2 \sin t_1 + L_E \sin t_{12}$$

The base rotation and wrist angles are then given as

$$\phi = \theta_0 \quad \psi = \theta_1 + \theta_2 + \theta_3$$

Note that to compute these values, we must use a $(\sin \theta, \cos \theta)$ CORDIC IP, and we constructed a wrapper that handles the angle formatting and range so angles can be directly inputted into the CORDICs. Angles are passed into the CORDICs using a handshake, which waits for the CORDIC to become available, passes in the input value(s), and upon seeing the done flag of the CORDIC wrapper, latches the values and moves to the next state.

4) Task Space Update: Once the task space position (ρ, z, ϕ, ψ) has been determined, the FSM progresses to the BUMPIN and CLIPIN stages, in which the target task space position $(\rho_*, z_*, \phi_*, \psi_*)$ is computed. If the module is jogging in joint space, the targets are simply set to the current position, as although inverse kinematics will be run, its output is ultimately ignored. If the module is jogging in task space, the targets are updated by adding each Δ_i to the current task space position, whereas if the module is going to a coordinate in task space, the target task space position is simply set to the Δ_i values. The values can also be clipped to ensure compliance with some safe range, although this was unnecessary in our design as the arm was very well behaved even at singularities.

5) Inverse Kinematics: Once a target task space position is obtained, the FSM transitions to the INVx stages. Here, we would like to obtain a set of servo angles θ_i^* needed to safely obtain the target task space position. We then shift the z coordinate back so the shoulder is at the origin and compute the squared distance to the target

$$z' = z_* - L_1 \quad r^2 = \rho_*^2 + z'^2$$

we may then note that because $\mathbf{r} = \mathbf{v}_1 + \mathbf{v}_2$, we can have

$$r^2 = \|\mathbf{v}_1 + \mathbf{v}_2\|^2 = L_2^2 + L_E^2 + 2L_2L_E \cos(\theta_2^*)$$

and hence we find $\cos(\theta_2^*)$ and then $\sin(\theta_2^*)$ using the Pythagorean identity

$$\cos(\theta_2^*) = \frac{r^2 - L_2^2 - L_E^2}{2L_2L_E} \quad \sin(\theta_2^*) = -\sqrt{1 - \cos^2(\theta_2^*)}$$

Note that here we have selected the elbow-down configuration with the negative square root; we found experimentally the elbow-up configuration produced nonphysical angles. We may then find the angle as

$$\theta_2^* = \arctan_2(\sin(\theta_2^*), \cos(\theta_2^*))$$

The derivation for the θ_1^* is more complex [2] and does not follow the computations done in hardware, and has hence been omitted, but by computing constants

$$b_{\text{new}} = L_2 + L_E \cos(\theta_2^*) \quad d_{\text{new}} = L_E \sin(\theta_2^*)$$

and then finding the intermediate angle

$$\beta = \arctan_2(z', \rho^*)$$

we may obtain the shoulder angle

$$\theta_1^* = \beta - \arctan_2(b_{\text{new}}, d_{\text{new}})$$

At this point, the FSM transitions to the BUMPOUTx and CLIPOUT states, where we must re-apply our original offset to obtain the joint space servo angles

$$\theta_1^* = \theta_1^* - \frac{\pi}{2} \quad \theta_2^* = \theta_2^* + \frac{\pi}{2}$$

Note that because the wrist pitch is given as $\psi^* = \theta_1^* + \theta_2^* + \theta_3^*$ and we have updated θ_2 and θ_3 , we obtain

$$\theta_3^* = \psi^* - (\theta_1^* + \theta_2^*)$$

these angles are clamped to the range $[-\pi/2, \pi/2]$ and outputted as θ_i^* . Note that if the module is jogging in joint space, the θ_i^* values will be overwritten as pure joint space increments $\theta_i^* = \theta_i + \Delta_i$, and the angles computed by IK will be discarded. The FSM moves to the DONE state, where it puts down the busy flag and raises the done flag, outputs θ_i^* , and resets back to IDLE. The FSM diagram for the inverse kinematics modules is given in Fig. 5, and as previously noted, the FORWx, INVx, and BUMPOUTx states represent several intermediate states that either transition to the next every cycle or wait for a CORDIC output.

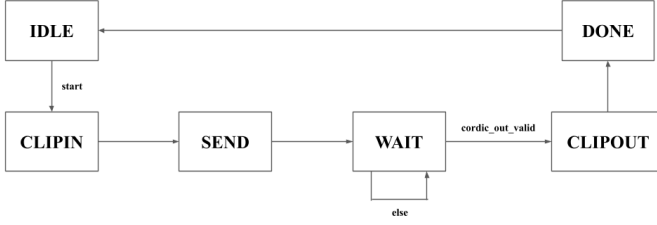


Fig. 6. The FSM for the CORDIC wrapper modules.

C. CORDICs (Anthony)

Within the motion planner module, one instantiation of each of a $(\sin \theta, \cos \theta)$, $\arctan_2(y, x)$, and $\sqrt{x}$ CORDIC IP were time multiplexed to perform computations for forward and inverse kinematics. Similar CORDIC wrappers were developed for each CORDIC IP to Q4.20 values used elsewhere in the module were compliant with the limits of each CORDIC. At a high level, values were clipped and sent into the CORDIC IP, and when the CORDIC had completed its computation, the values were clipped again and sent out.

This process was implemented as a 5-state FSM, as seen in Figure 6. with the IDLE, CLIPIN, SEND, WAIT, CLIPOUT, and DONE states. In the IDLE state, the module awaits input and advances to CLIPIN once a value is received, raising a busy flag. Next, in CLIPIN, generally each CORDIC has a certain fixed-point format it will take, and hence the Q4.20 values must be clipped and shifted into this format. In the $(\sin \theta, \cos \theta)$ case, the CORDIC may only take values in the range $[-\pi/2, \pi/2]$, however values internally may be as large as $[-3\pi/2, 3\pi/2]$, and hence the periodicity of sine and cosine is applied to extend the input range. In the $\arctan_2(y, x)$, the values are clipped by $-1 \leq y \leq 1$ and $-1 \leq x \leq 1$, and in the $\sqrt{x}$ case, the values are clipped by $0 \leq x \leq 2$, and in both cases all expected inputs fall in this range, so no extension is necessary.

V. PWM CONTROLLER (ANTHONY)

The PWM controller takes signed Q4.20 angle values with values in the range $-\pi/2 \leq \theta_i \leq \pi/2$ as inputs and directly actuates the PWM line to each servo in the physical arm. For future flexibility, the PWM controller has been parameterized to take the minimum pulse width TMIN, pulse width span TSPAN, and period of the servo motor in terms of clock cycles of the FPGA clock. To do this, $/2$ is first added to all angles to shift the range to $0 \leq \theta_s \leq \pi$, which was then multiplied by constant $K = \text{TSPAN}/\pi$ in Q4.20 and offset again by TMIN, giving us a pulse width in the range [TMIN, TMIN + TMAX]. The PWM controller then uses a counter to output a PWM signal for each line corresponding to its respective pulse width.

VI. SEQUENCE CONTROLLER (RUSSELL)

The Sequence Controller module implements the FSM shown in Figure 7 to record keyboard inputs into BRAM and play them back to allow for autonomous control over repetitive actions. The controller also takes in $\text{sw}[15:0]$

inputs, with $\text{sw}[15:14]$ corresponding to the control mode and $\text{sw}[3:0]$ corresponds to different states for the sequence controller depending on the values of the switches, and outputs the appropriate 14-bit keys_out value. The controller makes use of the Xilinx True Dual Port RAM, Read First, Dual Clock BRAM module, as well as two event counter modules.

$\text{sw}[0]$ corresponds to the *RECORDING* state. When this switch is on (and $\text{sw}[3]$ is off), the input 14 bits (2 for control mode and 12 for motor control) are continuously saved to the previous_keys register every clock cycle, and two registers increment. The first register increments every cycle until it reaches the 100 MHz/1000 cycle count, at which point it resets. This causes the second register, an 18-bit time_elapsed counter, to increment, producing a 1 ms timing granularity. At the same time, the BRAM module is continuously written to on port B. Regardless of the current state, the controller writes the 32-bit value $\{\text{time_elapsed}, \text{previous_keys}\}$ to BRAM every clock cycle, but only in the *RECORDING* state are the input value and write address updated. To update the write address, the controller checks whether previous_keys equals the current keys each cycle. If they differ, the controller triggers one of the event counter modules to increment the write address and resets the time_elapsed counter. When the controller leaves the *RECORDING* state, it preserves the values of time_elapsed , previous_keys , and the write address.

$\text{sw}[1]$ corresponds to the *PLAYING* state, which is entered only if $\text{sw}[0]$ and $\text{sw}[3]$ are low and $\text{sw}[1]$ is high. This state is similar to the *RECORDING* state in that it uses two registers: one that increments every cycle and another, timer , that counts the number of 1 ms cycles since the last line was read from BRAM. In this state, the output keys_out is always the first 14 bits of the output from port A of the BRAM module. The last 18 bits correspond to the saved time_elapsed counter associated with those keys. Each cycle, the timer counter is compared against these last 18 bits. If they match, both counters are reset, and the second event counter triggers an increment of the BRAM read address. If at any point the read address is greater than the write address, a constant 14-bit string of zeros is output as keys_out , and the read address no longer increments. This ensures that playback never reads beyond the recorded sequence. Additionally, playback may be paused by enabling $\text{sw}[2]$ while in the *PLAYING* state, which freezes both counters and outputs a constant string of zeros. Disabling $\text{sw}[2]$ resumes playback.

$\text{sw}[3]$ corresponds to the *CLEARING* state, which resets all registers, including the read and write addresses and the associated timers. This state does not erase the contents of the BRAM module but allows them to be overwritten during recording. The *CLEARING* state has the highest priority; if $\text{sw}[3]$ is ever enabled, the controller immediately transitions to this state.

Finally, the *IDLE* state occurs when all $\text{sw}[3:0]$ bits are 0, at which point all recorded values are preserved and no values are updated.

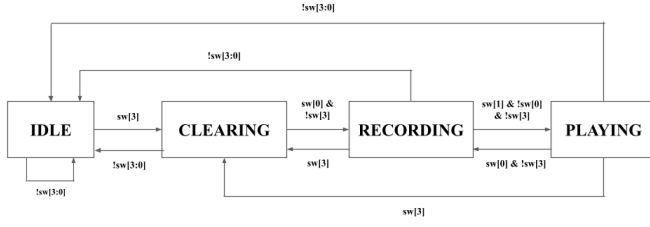


Fig. 7. The FSM for the Sequence Controller. Note that the states depend on the values of $sw[3:0]$.

Note that the `sequence_top.sv` file creates an instance of the `sequence_manager` module and directly connects it to the PS/2 controller for input and the motion planner for output. Within this top-level file, the values sent to the motion planner are wired either to the raw key inputs or to the sequence controller outputs. If $sw[1]$ is high and $sw[0]$ is low, that is, in the *PLAYING* state, the output is wired to the sequence controller output. Otherwise, the output is wired to the keys from the PS/2 controller. In this way, while recording a sequence of commands, the user can simultaneously observe the real-time execution of those same commands.

VII. DESIGN EVALUATION

A. Timing and Resources

The chosen FPGA has a clock rate of 100 MHz, while the servo's refresh rate is significantly slower, at around 50 Hz. This provides a generous window of around 2 million cycles for input keystrokes to propagate through the system while still maintaining the maximum responsiveness supported by the servos. As a result, less priority was needed for optimizing the time efficiency of the design, and more focus was instead placed on minimizing resource usage.

We may estimate the total latency of the FPGArm controller by noting that aside from inverse kinematics, no other module has a latency of $t \leq 20$ cycles. Note that although parts of the PS/2 controller operate on the slower 10 kHz PS/2 clock, this input delay does not affect other modules and is also still inconsequential in comparison to the significantly slower 50 Hz servo refresh rate or human response time, and hence is largely negligible. The inverse kinematics module takes less than 250 cycles total to run completely, and hence, assuming a delay of at most 20 cycles for all the other 5 modules, ignoring the PS/2 clock delay, the complete system takes no more than 350 total cycles to run and hence has a latency of around 3.5 microseconds. The inverse kinematics module latches into a given set of input angles and Δ_i 's and performs computation on the latched values, ignoring all others for around 250 cycles, making the throughput of the design around 6 servo angles/ $3.5\mu s = 1.71$ million joint space servo angles per second. Although the inverse kinematics module could be implemented more time efficiently, the generous timing requirements and time multiplexing requirements for the CORDICs made implementing the module as a many-state chained FSM.

In total, the design uses 6998 LUTs, which is around 1/4th of the total number on the chosen FPGA. With Cartesian inverse kinematics, although the design did not function properly, it used around 8000 LUTs. Both designs used 3 CORDICs, which were time multiplexed so they could individually take in and compute several different values across the overall FSM of the inverse kinematics module. The design uses 20 DSPs for hardware multiplication. Our design also uses 2 total BRAMs, one for sequence controller and another likely used for CORDICs.

The mathematical overhead to extend the current implementation to 6 degrees of freedom is nontrivial, although it can reuse many of the blocks that have already been developed here, simply using a longer pipeline than what is currently implemented. Assuming time multiplexing of the current CORDICs is possible, somewhere on the order of 5-10 additional CORDIC calls and 20-30 additional DSPs would be needed, with the LUT count likely no more than doubling to less than 15k LUTs. This generous estimate still fits well within the chosen FPGA, and hence the design is scalable to account for additional degrees of freedom. Furthermore, if larger links or higher precision is needed, the Q4.20 representation can be easily adjusted to something like Q12.20 or Q4.28 for these purposes.

B. Checklist

We were able to successfully accomplish all of our commitments, goals, and stretch goals as originally proposed. We demonstrated the parsing of inputs from a keyboard using the PS/2 protocol and a multi-hot encoding and joint space jogging control mode, where these inputs were used to increment servo angles using a PWM controller. Hence, the commitments were all accomplished. We then demonstrated using forward and inverse kinematics to translate from both cylindrical and Cartesian task space coordinates to target joint space angles using several CORDICs. Of these, cylindrical task space jogging control mode was implemented in hardware, where keyboard inputs were used to jog across a cylindrical task space. Forward kinematics were used as a subset of the inverse kinematics engine to determine the current position of the end effector in task space. The original goals did mention Cartesian control of the arm, and although this was not implemented in hardware, it was successful in simulation, and a roughly equivalent formulation of inverse kinematics was successfully implemented in hardware. The scalability of the design was assessed, with the additional resources required to add more degrees of freedom and the minimum resources required for the current design being evaluated, and with this, all goals were met. We demonstrated storing sequences of keyboard inputs in BRAM and replaying them on command and sequence replay control mode. We also implemented pre-programmed arm movements and configurations the arm could automatically execute, moving through a sequence of pre-programmed task space coordinates, such as a small hexagon, and hence all stretch goals were accomplished.

C. Implementation Insights

The most challenging aspect of this project was certainly inverse kinematics, as the complexity of this component and its interdependence with the limitations of the physical arm we were using made development very difficult. We initially attempted to simplify the problem by using the Jacobian transpose method of inverse kinematics, a gradient descent-based method that iteratively moves the arm towards the target task space position. Due to the complexity of the module, we developed a Python "simulation" of the module, in which a class was used to store state in "registers" and an enum was used to emulate an FSM with state transitions. This proved immensely helpful in understanding how to convert the Jacobian transpose formulas into something that could be done in hardware and also verified that, at least on paper, the formulas produced the desired result. Upon implementation in Verilog, the module was simulated but failed to produce the same results it had in Python. This could be due to a variety of reasons, from not being able to run enough times to being limited in convergence by the 16-bit Q4.12 angle representation we used at the time of development. After significant attempts at tuning and modification, the module was scrapped and a new method was attempted.

The second iteration of inverse kinematics used the analytic solution for inverse kinematics in Cartesian coordinates with 4 effective degrees of freedom. This module was significantly more complex than the previous, requiring a 26-state FSM, whereas the last attempt used a 12-state FSM, also requiring 2 additional types of CORDICs. Like the last iteration, a Python "simulation" of the module was first developed and tested to prove the mathematics was sound. Next, a Verilog module was developed based off of this, now using a Q4.20 angle representation, and following its completion, it was testbenched in simulation, where it produced the desired joint space angles to produce the target task space position. At this point, we believed we had a working solution for inverse kinematics. However, upon implementation in hardware, and very close to the project deadline, we realized the approach and mathematical assumptions for the Cartesian inverse kinematics solution we used were fundamentally incompatible with the physical hardware of our arm.

At this point, we chose to reformulate the problem in cylindrical coordinates, where the hardware limitations and properties of the physical arm could be easily accounted for. This implementation was done extremely quickly, assisted by first developing a Python "simulation" of the FSM, which was very close to the actual implementation in hardware, and verifying the results it produced against the actual hardware of the arm before proceeding with Verilog development. If we could do the project again, we would have done more prototyping using Python, as this allowed for much faster verification and overall development while also routinely validating our software results against the actual hardware.

Within our Sequence Controller, our method of recording and storing key values differs from an earlier design in which

`time_elapsed` represented the number of 20 ms cycles since the beginning of the recording, allowing us to record only a $\frac{2^{16}}{50 \times 60} \approx 21$ mins sequence. Now, since we use an 18-bit representation that resets after each line in BRAM, this value represents the number of 1 ms cycles since the previous line in BRAM, allowing us to determine how long each set of input keys persisted relative to the previous set. Thus, each line in BRAM can potentially represent a $\frac{2^{18}}{1000 \times 60} \approx 4$ mins long key command. And with our use of a 36KB BRAM module with each entry 32 bits long, we allow for a $\frac{36 \times 10^3}{32} \times 4 = 4500$ min long sequence, virtually eliminating any storage space issues.

VIII. CODE REPOSITORY

Link to our code repository. The `integrated_system` branch is the working final branch with the `full_system` subdirectory housing the completed project files.

IX. CONTRIBUTIONS AND ACKNOWLEDGMENTS

Russell worked on the PS/2 controller, sequence controller, decoder, incrementer, and clamp modules. Anthony worked on the inverse kinematics, decoder, and PWM controller modules, along with the physical assembly of the arm. The written sections of the report were lead by the corresponding member who built the code, with collaboration on most sections.

REFERENCES

- [1] S. R. Buss, "Introduction to inverse kinematics with Jacobian transpose, pseudoinverse and damped least squares methods," Univ. of California, San Diego, CA, USA, Oct. 7, 2009.
- [2] J. C. Hoe and D. Trumper, "Design and Testing of a Stewart Platform Augmented with a Two-Link Planar Robot Arm", Massachusetts Institute of Technology, Cambridge, MA, USA, 2002, sec. 3.1.2.2.