Tetris: A Frame Perfect Game Adventure

Authors: Eric Chen, Alton Olson, Deanyone Su:
Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—A FPGA-based system that is capable of generating a modern 2-player Tetris game that follows the Tetris Design Guideline, though it is not necessarily Guideline-compliant. This system provides frame-perfect responsiveness to the user(s) in addition to displaying information about both the user's game state and the opponent's game state over VGA. The user primarily interacts with a custom-designed, reversible controller and receives auditory information from an external DAC which drives a 3.5mm jack. 2-player battles are enabled over a custom, local network connection.**

***Index Terms*—arcade, emulation, FPGA, frame-perfect, game, local area network, low-latency, networking, PCB, synthesizer, SystemVerilog, Tetris, Verilog**

1 INTRODUCTION

Most modern Tetris implementations rely on a CPU to express the intricacies of the fundamental game mechanics. Mechanics like the Super Rotation System (SRS), Delayed Auto Start (DAS), and T-Spins are easier to implement and debug in a traditional software programming language. However, CPU-based implementations suffer from two major flaws: input latency and interference.

The input/output (I/O) stack in modern computers attempt to strike a balance between performance and load on the processor to avoid wasting valuable processor time. As a result, I/O latency in modern systems tends to be rather long as most applications have no need for near-instant I/O latency. However, Tetris is one such application. The user expects that their input is reflected by the game state instantaneously. This is difficult for CPU-based implementations to service as their responsiveness is bottle-necked by their I/O stack. In our implementation we enforce that user inputs must be reflected in the next frame that is loading onto the monitor, a latency that we term "frame-perfect".

Despite multi-core and simultaneous multi-threading technologies in modern CPUs, the vast majority of programs are still largely single-threaded. This means that the various services that are running the game, the network stack, the graphics subroutine, the game logic, etc. can interfere with each other and cause unintentional stalls for the queued processes. By nature, FPGAs are inherently parallelized and can avoid these issues entirely. In design, the separate components, graphics, logic, networking, etc. can be built to operate independently such that, unless logically required, no process will waits on the completion of another independent process. The cost of this parallelism is area on-chip, so our design must reasonably fit into an economical FPGA.

2 DESIGN REQUIREMENTS

The most important design requirement in our system is the frame-perfect nature. A standard display runs at 60 hz. Therefore we expect our user to see their inputs reflected by the display within $\frac{1}{60}$ of a second. This will be tested using hardware counters which record from user input to the resulting change being loaded onto the VGA pins on the FPGA. This measurement ignores the latency from the controller to the FPGA and from the FPGA to the monitor intentionally. These are parameters that are outside the scope of our design, and are consistent across different game implementations.

In terms of game mechanics, we are following the Tetris Design Guideline for the majority of our implementation, as described in [5]. These game mechanics are either implemented or not, and they are verified through playtesting. While it is possible to test these inputs in simulation, at the time-scale that these mechanics work at, it is more streamlined to playtest for verification. Further detail of the game mechanics are provided in Section 4.1 .

Of course, individual small components are verified for correctness using simulation testbenches, while more complex components are verified for correctness using hardware testbenches. As each component verifies for correctness, it is integrated into the final "testbench" that houses the finished game implementation.

Our sound synthesizer produces "Korobeiniki", the classic Tetris music. The music is sampled at 50 KHz, slightly above the industry standard of 44.1 KHz. This sampling rate is measured by the clock rate generated over GPIO for the external DAC. In addition to music, the synthesizer can produce sound effects based on in-game events.

Our network is a custom protocol over a subset of the GPIO pins available on our FPGAs. The requirements on this network is that it does not interfere with the single-player mode(s) and that it can communicate the necessary data within the latency of a single frame since the data it carries is to be displayed to the user. Therefore the required network latency is less than $\frac{1}{60}$ of a second. The true requirement is somewhat less than that though, since the data needs to be received, processed, and then prepared to be shown to the user in a timely manner. The network takes advantage of the available pins to transmit data in parallel and enable more robust encoding techniques.

As a side note, there are 7 types of tetrominoes in Tetris, I, O, T, J, L, S, and Z. They are cyan, yellow, magenta, blue, orange, green, and red, respectively as depicted in [5]. They will be referred to as such for the rest of this document.

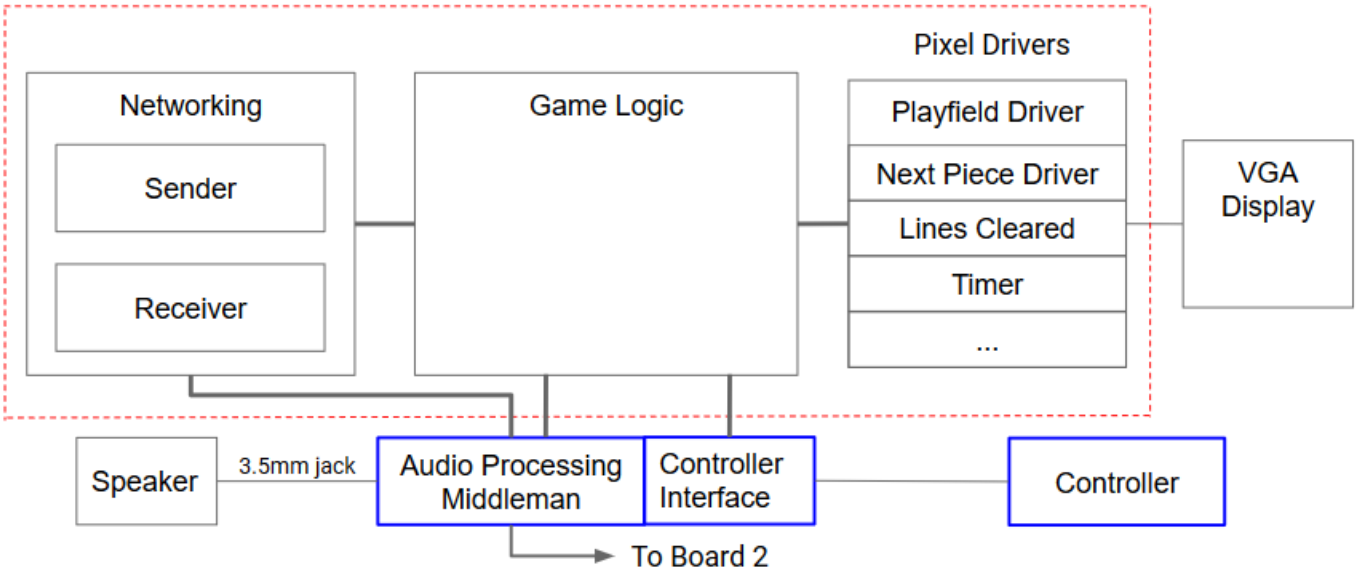


Figure 1: Overview of the full system per FPGA. The red box indicates the components on the FPGA itself while the blue boxes are housed on an external PCB, interfaced with via GPIO. The PCB with the audio processing middleman and the controller interface also houses a network interface to connect to the second FPGA for Battle mode.

3 ARCHITECTURE OVERVIEW

Our system is architected with division of labor in mind. Given our team of 3, we wanted each contributor to be able to work in parallel as long as possible. This maximizes the efficiency of our individual work, and also reduces the number of errors than can occur due to miscommunication. With this in mind, we split our design into 5 major sections.

1. Game Logic
   This subsystem is responsible for the majority of game mechanics. This holds both system state and game state. These are used to provide data as needed to other sub-systems in addition to allowing the other subsystems to communicate to each other as needed.

2. Graphics
   This subsystem is responsible for graphical output via (S)VGA. The data pulled from the Game Logic subsystem is re-organized into either tiles or blocks and rendered into an understandable form for the user. Each independent portion of the graphical output has a dedicated pixel driver to detail that portion of the display. This reduces complexity of each individual pixel driver and also makes graphical errors quicker to debug as each error can be instantly isolated to a particular driver. This subsystem is tightly integrated with the Game Logic subsystem as the majority of data that needs to be displayed is directly tied to the game state.

3. Network Protocol
   This subsystem is responsible for communication between FPGAs. This is only used in the 2-player Battle mode and communicates data over the GPIO pins. This system requires full send and receive stacks to encode and decode data.

4. Audio Synthesis
   This subsystem is responsible for producing audio for the user experience. The data is pulled from the Game Logic subsystem for sound effects as well as a smaller separate module in the FPGA (not pictured) to read data from a memory file to produce music. This includes a table-lookup to reference notes to waveforms of the correct frequency. The external DAC that drives the 3.5mm jack is housed on an external PCB, the middleman PCB. This middleman PCB is wired into directly using GPIO, which is then broken into components for the external DAC, the network protocol, and the controller.

5. Controllers
   This subsystem is responsible for the primary interaction with the user. The user(s) will use the controllers to provide inputs to the Game Logic subsystem. The controller has a dedicated PCB which is cabled to the middleman PCB. The buttons are arranged in a layout that mirrors a generic pair of human hands.

Further detail of each system is discussed in section 5. It is important to note that while the network protocol is designed and set, the interfaces between each of the subsystems is not fixed as the information that needs to be shared between each module is not static. As new mechanics are added, the Graphics subsystem grows larger and may require more information from the Game Logic subsystem to drive that graphical output. It is within expectations for the system architecture to expand the interfaces for each system as mechanics are added.

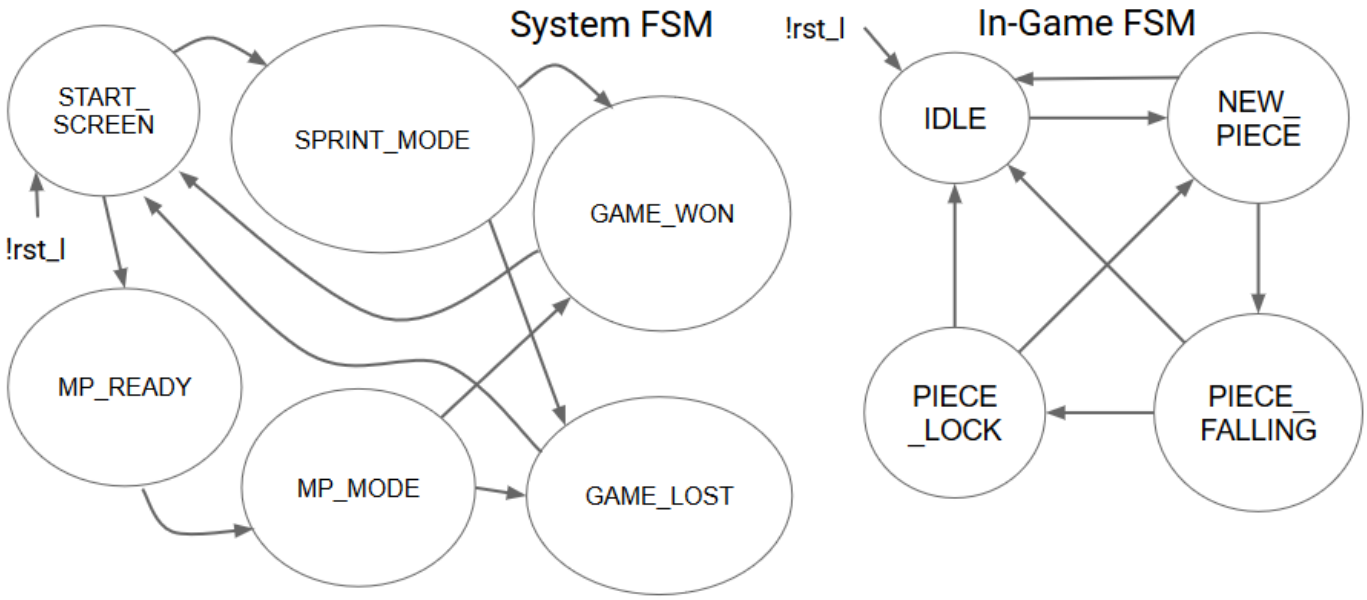


Figure 2: Overview of the FSMs involved with managing game logic

4 SYSTEM DESCRIPTION

4.1 Game Logic

In a user-oriented game, it is important to manage the user's interactions with the system. We manage this using a series of "screens" that are shown to the the user in sequence. On launch, the user is shown a "start" screen which displays the Tetris logo and the various options available to the user. Then the user can opt into a single-player Sprint mode, which can begin immediately, or they can opt into a multiplayer Battle mode. For the multiplayer option, they are moved into the "MP_READY" state which stalls until the other player is also in the "MP_READY" state.

In both the "SPRINT_MODE" and "MP_MODE" states, the user is presented with a classic Tetris screen, without and with their opponent's UI, respectively. In these states, the user is able to play Tetris as expected and the game concludes as defined by the game mode. Then, the user is presented with a winning or losing screen, depending on the outcome of the game, with some statistics about the game that concluded, and then allowed to begin a new one.

In-game, the state is handled as a loop of spawning a new piece, having it fall to the "floor" of the playfield, then "locking" the piece into place. This FSM can be interrupted, and can be forced into an "IDLE" state by the game ending. While this FSM drives the Seven Bag (described below), it is not the only trigger to spawn new pieces. It is also possible for the Hold logic (also described below) to spawn new pieces.

The mechanics of Tetris are largely implemented within the Game Logic subsystem. As such the remaining description will be structured as a breakdown of some of the more interesting game mechanics and the implementation of such. All mechanics are described at a high level in [5].

- Super Rotation System (SRS) [4]
  The SRS is the current Tetris Guideline for how tetrominoes rotate and wall-kick when in the playfield area. All tetrominoes have 4 orientations: 0, R, 2, L. All tetrominoes spawn horizontally, in the 0 orientation.

  Basic rotations are defined such that each tetromino appears to be rotating about a single point. This single point is a individual mino for the J, L, S, Z, and T tetrominos. The I and O tetrominos appear to rotate about an intersection of gridlines.

  Wall kicks are an important aspect of rotations because it enables rotations that are otherwise impossible. Importantly, when a piece is pressed against a wall of floor, wall-kicks define how a piece is shifted to enable the rotation to occur. SRS has a defined set of 5 rotations (basic rotations plus 4 different kicks) per rotation. The I tetromino has its own set of wall-kicks while the other tetrominos share a set of wall-kicks. The actual tables themselves and more information can be found in [4].

  Wall-kicks are arranged in order of priority. As such, the first valid wall-kick (in-bounds and non-overlapping) is the one that is used, and a rotation only fails if all 5 wall-kicks are invalid. In our implementation we check the wall-kicks sequentially, but in parallel with all other movement options. This is a trade-off between area and latency. Since validity must be determined by comparing the new position of the piece against the current playfield state, each validity check requires an area cost roughly proportional to the number of validity checks being done.

  In a purely parallel implementation, we have 5 right rotations, 5 left rotations, 1 move left, 1 move right,

1 soft drop, and 1 hard drop to be checked. This is 14 checks per cycle, which translates to roughly 30K logic elements (LEs). We deemed this unfeasible due to area cost. By continuously checking the wall-kicks in sequence, we reduce the number of checks to 6 per cycle. This lowers the LE usage to roughly 14K. This we deemed acceptable, though we could reasonably do more the checks in sequence which could reduce the number of checks to as few as 1. This latency is of minimal concern to the user. This game designed for human players. In practice, the fastest a human can spam a button is somewhere in the range of 200 presses per second. Therefore, using several to a dozen cycles to evaluate input validity is acceptable.

- Delayed Auto Shift (DAS) [2]
  Also known as autorepeat, this mechanic defines the behavior of a held button in game. A standard cool down is necessary to have the user be able to play the game, since a piece shifting or rotating at the board's clock rate is useless to a human player. With DAS, a held move causes the piece to shift initially at a high cool down period, than repeatedly shift at a lower cool down period. This enables the user to efficiently move and rotate pieces.

  We implement DAS into our input handler for the controllers. This module integrates a synchronizer chain with a cool down counter and a validity check. This integration allows the module to vary the cool down based on an FSM, and also refine the input to a single-cycle pulse, which is easier to manage in the remainder of the system.

- Spawning Position
  Pieces spawn at the 21st and 22nd rows of the playfield, which are hidden from the user and move down instantaneously on spawn. We deviate in an unnoticeable manner from the Guideline by spawning pieces in the 20th and 21st rows of the playfield and not instantly moving the piece down on spawn. Effectively, these are identical, so long as the top-out logic handles overlaps in addition to locking above the visible playfield.

- Move Reset Lock Down
  The Guideline defines 3 different lock down mechanics, the most common of which is move reset lock down. In classic Tetris, the pieces will lock onto the floor or another piece it is stacked on top of after 0.5 seconds. Move reset lock down resets the timer if the piece is moved or rotated. Naturally, this could allow users to infinitely spin a piece to delay the game, but most games implement a limit of 15 resets before the piece locks with no delay. We follow this limit.

- Hold
  Hold is a mechanism that allows that player to store an active piece to swap with another piece later in the game. At the beginning of the game, the hold is empty. As such, the first time a piece is held, the Seven Bag needs to spawn a new piece, but thereafter the piece held is swapped with the active piece. Upon swap, the active piece (that was just beign held) is spawned at the top of the playfield. This hold can only be done once per piece, so a swapped piece cannot be held.

- The Seven Bag
  The Seven Bag is the mechanic by which pieces spawn as defined by the Guideline. This is intentionally setup to avoid strings of the same piece being given to the player, which is possible using a naive random number generator (RNG). As the name suggests, tiles are provided to the user as though drawn from a bag containing the 7 different tetrominoes. When empty, the bag is refilled. While this mechanism does provide some unfavorable strings of tetrominoes, like S, Z, S, Z, it does avoid most of the issues with simpler mechanisms.

  Our pseudo-RNG is a set of 31-bit Galois Linear Feedback Shift Register (LFSR) as described in [7]. Each LFSR generates a bit that is concatenated to produce a tetromino. This generation logic runs continuously in the background, which means the Seven Bag is generated based upon how the user plays the game. While this is an awful randomness source for any cryptography application, it is sufficient and efficient for our use.

- Piece Preview
  The next 6 pieces that are provided to the user are shown ahead of the user actually dropping and placing the tetrominoes. This is implemented as a modified queue that is continuously filled by the Seven Bag. The modified queue has its contents output to be able to communicate with the NextPixelDriver (described below) to show the values to the user.

- T-Spins
  T-Spins are a special kind of line clear, where the last movement of a T tetromino is a rotation and it moves the piece into a "hard to fit" location. The exact detection method is unclear since the Guideline has changed the definition of a T-Spin multiple times over the course of the past 2 decades. As such we will be using the 3-corner method, which was used in past SRS-based games, in addition to other heuristics to restrict the definition. This will avoid some of the issues that plagued Tetris DS, which purely implemented the 3-corner T-spin.

- Notable Omissions
  Since the Tetris Guideline is not publicly available, and online resources can only provide most of the user-facing details of the game, it is impossible for our implementation to be fully Guideline-compliant. With that in mind, we have attempted to build a

version of the game that is sufficiently Guideline-complaint such that any user familiar with official versions of the game will be able to instantly play our version as well, with minimal changes. That being said, there are definitely some deviations from the Guideline in our implementation.

- Lack of Marathon or Ultra Modes
  Marathon is an endless mode where the player is able to continue playing Tetris until they top out and lose the game. Ultra is a timed game mode where the player attempts to clear or send as many lines as possible within a fixed time limit. Both of these modes are less popular today than either Sprint or Battle modes. We have chosen to exclude these modes because of this, but will be including them in the event we have time to do so after integration steps.

  Past versions of the game have omitted different modes, usually because of hardware limitations.
- Controller Mappings
  The Guideline defines standard mappings for consoles and handheld gamepads. Since we're building custom controllers for our implementation, without joysticks, our controllers are not going to be Guideline-compliant. Nonetheless, they will be intuitive to use.

4.2 Graphics

The Graphics subsystem is entirely based on the VGA controller that is provided in 18240 Lab 5 for implementing Mastermind. There are minor modifications to the protocol to make it work at a higher resolution and refresh rate (SVGA). These specifications are defined in [6].

The pixel drivers that compose the Graphics subsystem are independent drivers of VGA_R, VGA_G, and VGA_B pins which drive the 8-bit color values to the display. These independent drivers are multiplexed based on context. Here context can be the part of the screen that is being rendered (the row and column) or the current screen being displayed to the user, as defined by the System FSM in Figure 1. The data that each driver needs are generated in the Game Logic subsystem. This data is then wired across into the Graphics subsystem and then passed down to the individual drivers as needed. As a result, the Graphics subsystem is deeply interconnected with the Game Logic subsystem.

This organization lends itself to being modular and expandable which is important in our project as we implement features section by section. It is also important in enabling us to identify issues since an error on-screen can immediately be isolated to a particular driver and/or the logic associated with providing values to that driver.

The multiplexers between drivers is based on an active signal that each pixel driver produces. The active signal is one-hot, which is efficient for the logic that dictates which driver is providing valid color values for the controller.

At the moment we have several graphics drivers. The following list is a short description of each.

- Playfield Pixel Driver
  This driver is responsible for displaying the playfield in-game. This is effectively a translation from a 10x20 array array of enumerated tile types to a color value, based on the row and column from the VGA controller.

- Next Pixel Driver
  This is similarly structured to the Playfield Pixel Driver, albeit on a smaller scale. This region is only 6 x 19 as it only needs to display 6 tiles in a set of fixed positions. This region needs to be 6 tiles wide as the widest tetromino, the I tetromino is 4 blocks wide, which means that the region needs to be 6 tiles wide to enable buffer space on either side of the tetromino.

- Hold Pixel Driver
  Again, this is similarly structured to the Playfield Pixel Driver, albeit on a smaller scale. This region is only 6 x 4 as it only needs to display a single tile in a fixed position.

- Timer Pixel Driver
  Time is a set of values ranging from hours down to milliseconds generated in the Game Logic subsystem. The driver has the system time as an input and uses this to compute the individual digits to be displayed to the user based on the time inputs. These digits are translated to scaled 6x6 pixel arrays. From there the driver produces color when the row and column match an "on" area of the scaled pixel arrays and blank otherwise. See below for a more detailed description about text rendering.

- Lines Cleared Pixel Driver This is very similar to the Timer Pixel Driver, but showing a count of lines cleared with some text.

Text rendering is important for communicating information to the user. We implement text rendering by referencing a 6x6 pixel font, found in [1]. We imported this font by hand into an ASCII lookup table that returns a 6x6 binary array. Each character is an individual module instantiation that uses an input to determine the character for which to generate the binary array. This module uses parameterized coordinates and scaling to determine where the character is displayed on the screen, and how large the character should be. Scaling text in this way uses division to determine which "pixel" of the 6x6 array is currently being rendered. Therefore, it is ideal if the scaling is by a power of 2 since this reduces the logic complexity of this pixel driver. However, this is not crucial since modern FPGAs have hardened division blocks which can be inferred such that the scaling logic is not terribly expensive, even if the scaling is not a power of 2.

4.3 Network Protocol

Two-player Tetris Battle differs from Sprint in a few ways. When the player clears lines, a corresponding number of "garbage" lines are sent to the opponent. Garbage lines are extra lines with a single random gap, appearing at the bottom of the opponent's playfield. Sent lines are stored in a pending queue of up to 12 lines, which appear on the playfield after a set delay. The goal of the multiplayer game mode is to make the opponent top out by sending them garbage lines.

Garbage per line clear

Line clear	Lines sent
Single	0
Double	1
Triple	2
Tetris	4
T-Spin Single	2
Mini T-Spin Single	0
T-Spin Double	4
Mini T-Spin Double	1
T-Spin Triple	6
Back-to-Back bonus	+1
All Clear bonus	+4

Garbage per combo

Combo	Lines sent
0	+0
1	+1
2	+1
3	+2
4	+2
5	+3
6	+3
7	+4
8	+4
9	+4
10+	+5

Figure 3: Garbage Table based on Tetris 99 mechanics

To track garbage lines being sent and update the opponent's board state on the screen, the following information must be communicated by each player on every frame.

- Garbage
  Number of garbage lines being sent.
- Hold Register
  Content of the hold piece register, as described in Section 4.1.
- Piece Preview
  Contents of the next piece queue/piece preview, as described in Section 4.1.
- Playfield
  Current state of player's playfield, as described in Section 4.1.

To enable multiplayer communication between game instances on separate boards, we describe the Tetris Synchronous Parallel INterface, or TSPIN communication protocol.

Pinout

Pin	Abbreviation	Description
0	CLK	Clock
1	MOSIH	Master Out Slave In Handshaking
2	MOSI0	Master Out Slave In Data 0
3	MOSI1	Master Out Slave In Data 1
4	MOSI2	Master Out Slave In Data 2
5	MOSI3	Master Out Slave In Data 3
6	MISOH	Master In Slave Out Handshaking
7	MISO0	Master In Slave Out Data 0
8	MISO1	Master In Slave Out Data 1
9	MISO2	Master In Slave Out Data 2
10	MISO3	Master In Slave Out Data 3
11	GND	Ground

Figure 4: TSPIN Pinout

TSPIN is a 4-bit parallel, stop and wait protocol with dedicated handshaking lines. The pinout is shown above, utilizing 11 GPIO pins with one board being designated the master and the other the slave. These designations are determined when the games are synthesized onto the boards. The master sends the clock used for synchronization, and the designation is used for naming purposes. Master and slave are otherwise functionally identical.

In designing this protocol, a number of factors were taken into account. For an optimal game experience, the opponent's playfield must update on the player's screen every frame, or 1/60th of a second. Transmission must succeed within this timeframe. From past projects we know that the worst case clock rate we can send over GPIO is 50kHz, giving us at worst 833 cycles per frame to work with. Transmitted data in total is 832 bits, not including overhead such as syncwords or sequence numbers. Due to the high number of GPIO pins available, we are not bandwidth limited. As such, we use the available bandwidth to send data in parallel, allowing us to attempt to send packets multiple times per frame. Stop and wait is chosen for flow control due to simplicity, and the fact that data only needs to be successfully received once per frame. This dictates that, after sending a packet, the sender must wait for an acknowledgement from the receiver before sending the next packet, or time out before re-sending the same packet. Sequence numbers are used to distinguish fresh packets and avoid sending garbage twice. Handshaking is given its own dedicated lines for simplicity.

Error correction is handled via Hamming Codes on each line. Due to the short range of communication we expect error rates to be low, so hamming encoding is selected to perform single error correction and double error detection.

Handshake packets incorporate additional redundancy for safety.

Synchronization for multiplayer game start/end is handled using the handshaking lines. When a player enters the MP_READY state (Figure 2), the sender will continuously send ACK packets on the handshaking line, and the receiver will begin listening for ACK packets in return. When acknowledgement is received from the other board while in this state, the game will begin.

Data for sending is loaded into the sender from the game logic, via an update_data signal that is asserted for one cycle when fresh data can be loaded in. This is set to occur once per frame so that the sender does not attempt to resend updated data with the same sequence number, which could cause sent garbage lines to be lost. Upon loading in data, the sender constructs an overall data packet, before dividing it into four chunks for each data line and encoding them individually. These encoded data chunks are combined with the syncword to form the data packets, which are sent serially on each of the 4 data lines. Once sending is complete, a send_done signal is asserted and the timeout counter begins to increment. The sender then waits until an acknowledgement is received or timeout is asserted. Data packets with their bit mappings are shown in Figures 5 and 6.

Data Packets (pre-encode)

Bit	Field	Description
835-832	SN	Sequence Number
831-828	GBG	Number of garbage lines being sent
827-824	HLD	Hold Register
823-800	PQ	Piece Queue
799-0	PFD	Playfield Data

Figure 5: Data prior to division and encoding

Handshaking operates similarly, but does not require data from the game logic. Handshaking packets essentially consist of a sequence number and packet identifier, the latter of which can either be an acknowledgement (ACK), or game end signal. Figure 7 details these packets.

The receiver for each wire works by listening for the syncword, an 8-bit sequence of 1s. This pattern is selected because it cannot otherwise appear in the encoded data. Upon detecting the syncword, each receiver shifts in bits equal to the length of the packet, which is specified for each line by the protocol. Once the full packet is assembled, it is decoded and reconstructed, and sent back to the game logic via the update_opponent_data signal, which is asserted for one cycle when there is fresh data available. Handshaking works similarly, with separate signals for ack_received and game_won based on the decoded data.

These modules are implemented as a set of individual serial data senders/receivers for each data/handshaking line, with overall sender/receiver modules handling packet construction and interfacing with game logic. Several FSMs are used to track game state and control the individual sender/receiver modules for each line. These FSMs are depicted in figures 8 and 9.

MOSI0 Packet

Bit	Field	Description
224-217	SYNC	Syncword
216-0	DATA	DATA[835-628] (encoded)

MOSI1 Packet

Bit	Field	Description
224-217	SYNC	Syncword
216-0	DATA	DATA[627-420] (encoded)

MOSI2 Packet

Bit	Field	Description
224-217	SYNC	Syncword
216-0	DATA	DATA[419-212] (encoded)

MOSI3 Packet

Bit	Field	Description
224-217	SYNC	Syncword
216-0	DATA	DATA[211-0] (encoded)

Figure 6: Data packets post division and encoding

Handshake Packets

Bit	Field	Description
13-7	SYNC	Syncword
5-0	HEAD	Encoded header containing packet identifier (ACK, GE) and sequence number

Header (pre-encode)

3	2	1	0
PID	SN	~PID	~SN

PID

Name	PID	Description
ACK	1	Acknowledge/Game Start
GE	0	Game End

Figure 7: Handshaking packet specification

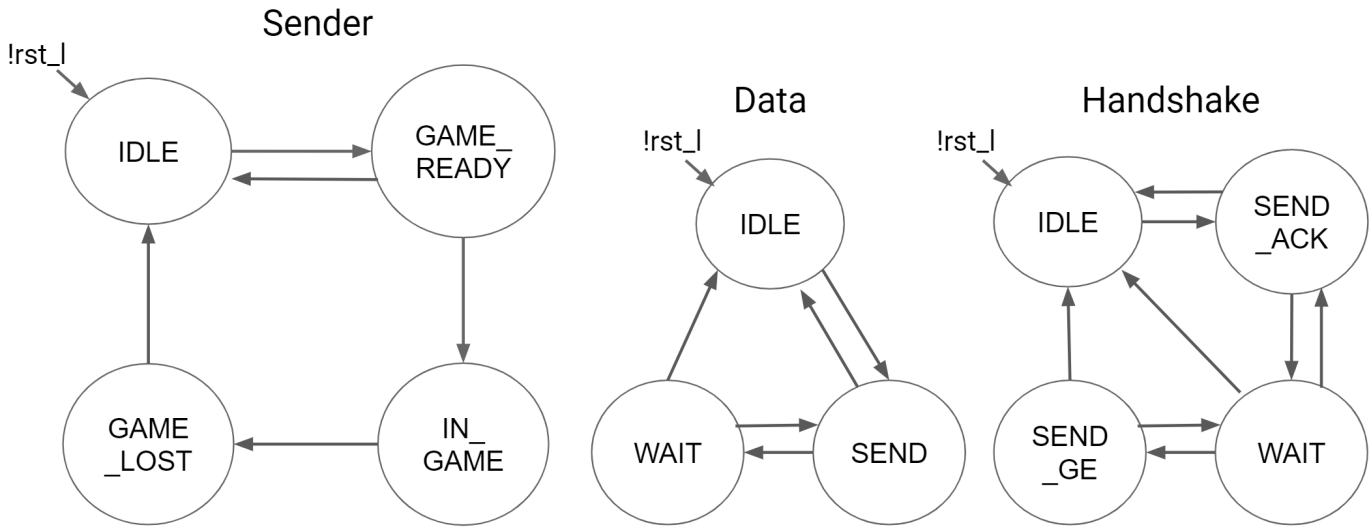


Figure 8: Send stack control FSMs

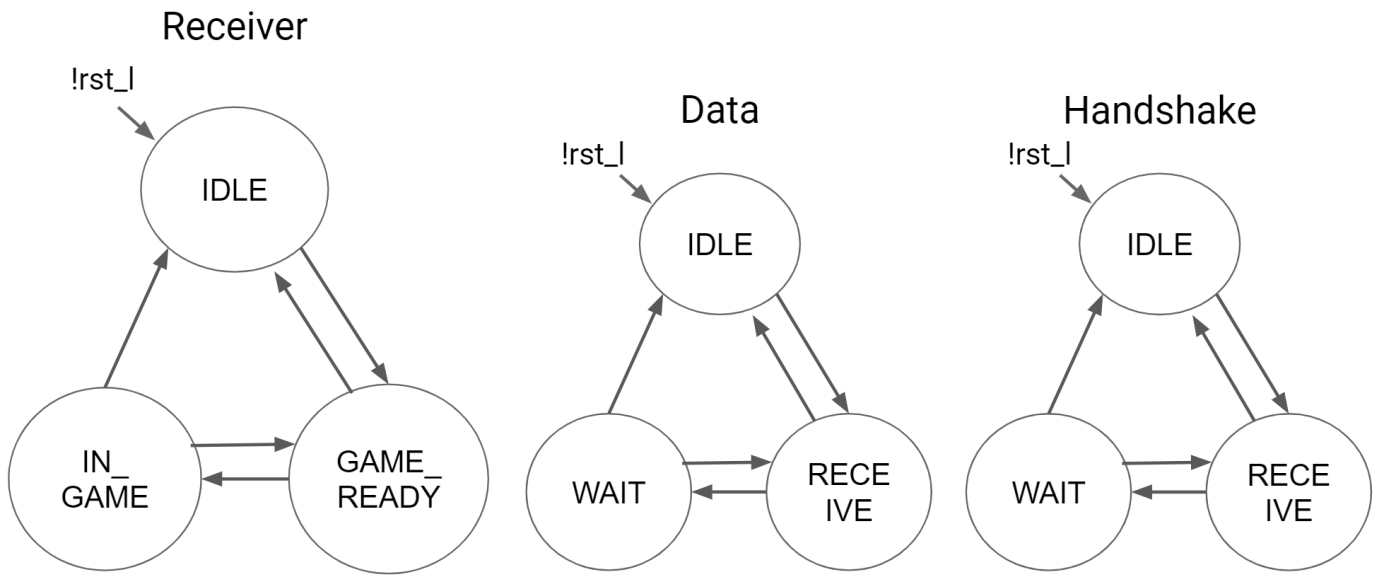


Figure 9: Receive stack control FSMs

4.4 Audio Synthesis

We chose to synthesize audio using the FPGA GPIO pins. The DE2-115 boards have an onboard audio codec and 3.5mm jack which we considered, but we chose the GPIO pins as a more generic and portable output option in case we decided to switch boards. In addition, we were not I/O limited by number of pins, so using 8 pins for audio was not a problem. To convert the digital signal to an analog audio signal, we chose the TLC7528C [3], a cheap R-2R DAC with a 100ns settling time. We operate the 7528 in voltage-mode, meaning the output ranges from 0-5V. 8 GPIO pins directly interface with the digital input pins of the 7528.

The responsibilities of the music module can be divided up into four parts, which happen in roughly this order:

- Reading the note number from memory
- Converting note number to note frequency
- Generating a waveform at note frequency
- Mixing multiple waveforms together to create the final output signal

The top Music module is responsible for loading the note number and mixing the waveforms. It sends the note numbers to two Wave Generator modules, which generate the actual waveforms. Each of these contains a Note Frequency Lookup module, which reads frequency information from a lookup table stored in memory. All of the logic in these modules is clocked at 50Mhz. We encoded each note frequency by storing the note wavelength divided by

50MHz, or in other words, how many clock cycles long each period of the wave is. This make it easy to output a square wave at this frequency: all we have to do is repeatedly count clock cycles up to half the period, then invert the output signal. Mixing is done by simply performing a weighted sum of two Wave Generator output signals (melody and bass). The Music module also includes counters for determining position in the song and 50kHz clock timing. On each 50kHz edge, the 8 GPIO pins sample the current value of the mixed signal, which is then held until the next 50kHz edge.

The primary motivation behind the Music module's design was for it to be lightweight in terms of board area, as we need to save space for game logic. This meant precomputing note frequencies and storing them in BRAM rather than using expensive logic to calculate frequencies using floating-point math and exponentiation, allowing the Music logic to be composed of simple counters and adders.

4.5 Controllers

Our controllers need to be responsive and precise to align with our goal of frame-perfect inputs. There is little point to having hardware that can process inputs within $\frac{1}{60}$ of a second if the user cannot consistently perform the inputs they want.

We considered several options for controllers before settling on an 8 arcade button layout that mimics Tetris controls on a computer keyboard. The user has translation in one hand, rotation in the other hand, hard drop on one thumb, and hold on other thumb. The layout is ergonomically similar to the universal computer game standard of WASD/spacebar. We considered just using keyboard switches in that exact layout, but ended up choosing arcade buttons instead for their superior durability and user satisfaction (you can't slam a key the same way you can slam an arcade button).

We chose to make the controls hand-agnostic, so users who prefer to translate with their left hand could do so. We did this by using a 2x8 connector and designing the pinout to be reversible. Power and ground are rotationally symmetric, and the other pins rotate to their left-handed counterparts.

Each button is wired with a 1KΩ pullup resistor and directly connected to its corresponding pin on the 2x8 connector. The FPGA will see a digital high or low value indicating the state of the button.

The controller is housed in a sturdy laser-cut plywood box, with cutouts for the buttons and the connector port.

5 PROJECT MANAGEMENT

5.1 Schedule

See Appendix A at the end of the report for our schedule in the form of a color-coded Gantt chart.

Our first major milestone was to have a working prototype by spring break. This included a working 40-line sprint mode, ability to generate audible music, and a hardware testbench with data being transferred across FPGAs.

Post spring break, the plan was to spend time on the multiplayer mode and integration. Any leftover time was allocated to work on details, like graphical assets or area optimizations. Significant amounts of slack were left at the end to handle integration issues as well as debugging any major incidents that occurred along the way, that could potentially side-track significant portions of the project.

5.2 Team Member Responsibilities

Here is a list of responsibilities per team member. Each of us were tasked with implementing a subset of the main subsystems in the full system.

- Deanyone Su
  Primary Responsibilities
  - Game Logic
  - Graphics
- Eric Chen
  Primary Responsibilities
  - Network Protocol
- Alton Olson
  Primary Responsibilities
  - Audio Synthesizer
  - Game Controllers

5.3 Budget

See Appendix B for our budget spreadsheet. This table includes all purchases made for this project, and therefore includes redundancies in the event that parts arrived nonfunctional. There will be a bill of materials included in the final report that includes only the necessary parts, without redundancy, for the final implementation.

5.4 Risk Management

This project was initially planned to be built in parallel to reduce risk of miscommunication. By building our components as nearly stand-alone, we could have inflexible interfaces, that were clearly defined, between the three of us, while leaving "internal" interfaces to be flexible and more amenable to modification. This is only possible with a limited number of interfaces between each of our components so our responsibilities were allocated to reduce these intentionally. As an example, having different people work on the Game Logic subsystem and the Graphics subsystem would be unwise as both subsystems are closely linked and need a very high amount of inter-connectivity.

For parts, we ordered 50% to 100% more than needed for our implementation. This reduced the risk and delays

associated with re-ordering parts due to the parts being faulty and/or us damaging the parts as we were building the projects. This was feasible due to the relative cost of our parts, a lot of the components we were using are relatively cheap.

Additionally, we sourced and ordered parts as early as possible. We originally planned to do so to be able to begin prototyping as early as possible and reduce the impact of discovering that we needed more parts. It turns out this was a wise choice as many of our parts have some portion of their supply chain in mainland China and the ongoing pandemic is negatively impacting production in that part of the world.

6 RELATED WORK

This project shares many aspects with emulation projects. FPGAs are well suited for emulating retro game systems or late 20th-century hardware since clock speeds and data rates of the era tend to be well below the capabilities of modern FPGAs. Therefore, our work shares facets with other works that attempt to emulate systems such as the NES or Gameboy. Full emulators do exist, emulating the NES, SNES, and Gameboy (Original, Color, and Advance). Our implementation is game-specific and is addressed at improving the experience in comparison to modern systems, by addressing specific short-comings of those modern implementations.

It would be remiss to not mention the other emulation project in our own capstone group, Team C0's GameBoi. They built a Gameboy Original cycle-accurate emulator onto a DE10-Standard FPGA. We also credit inspiration for this project to the many 18240 Lab 5 implementations of retro games, implementing Pong, Breakout, and Mastermind on FPGAs. Our original idea was largely based around taking a retro game, building it onto an FPGA, and then taking it to the next level.

The Analogue Pocket is a consumer emulation product that is FPGA-based. Many open-source implementations can be found of Github/Gitlab and other nooks on the internet.

7 SUMMARY

Our primary metric was response time relative to user input. This is achieved in our design since user inputs are reflected on either the same or next frame sent to the VGA display. The current implementation is limited by the frame rate of the screen it is connected to and indirectly by the clock speed of the FPGA. Being "frame-perfect" is a nice phrase to use but it needs a refresh rate to provide quantitative meaning. We define refresh rate using a standard monitor with 60hz refresh rate. However, today there exist many monitors that can go to 75hz, 120hz, 144hz, or even 240hz. Then, the term "frame-perfect" takes on even stricter meaning. Therefore, our system metrics could be further improved by driving a higher refresh rate, which provides information to the user at an even higher rate. Unfortunately, we are somewhat capped for the frame rate we could reasonably drive from our FPGA since higher resolutions and refresh rates require faster clock speeds to send across VGA. The fastest we could reasonably do at 50 MHz is 72hz, 800x600 VGA output. To push this further would require pushing a faster pixel clock than our native on-board crystal. This means locking a PLL to a desired frequency and passing data across clock domain crossings. This is a reasonable course of action to take for a future, long-term project.

For this implementation, and the Tetris game, 60hz frame rate is reasonable and our design stops here as a demonstration of the improvement possible over a traditional platform by using an FPGA implementation.

Some lessons learned in the course of building this project:

- Start work early. Many aspects of what was being built were not clear until we actually attempted to implement the module and realized there was a significant challenge in the implementation details.
- Think about what is being implemented before implementing it. Several rushed decisions ended up causing us significant efforts to re-write, for example:
    1. We originally decided to use 640x480 @ 60hz over VGA for our display but it turned our, upon closer inspection of the VGA standard, we could do 800x600 @ 72hz over VGA. This afforded us more space for our game and also enabled us to achieve a lower I/O latency to the user, which we identified as our primary metric.
    2. Using SRS, both I and O pieces do not have a center "tile". Thinking that the center coordinate was then arbitrary, we chose to use the bottom right tile of the O piece as the origin for that tile. It turns out, to spawn tiles correctly, it was a lot more elegant to have the bottom left tile be the center, which meant rewriting parts of our rotation logic and rendering logic.

References

[1] Alexander Atkishkin. *8-bit Monospace 6x6 Pixels Font*. Mar. 2020. URL: https://previews.123rf.com/images/iunewind/iunewind1607/iunewind160700049/60848823-8-bit-monospace-font-6x6-pixels-on-glyph-vector-set-of-alphabet-numbers-and-symbols.jpg.

[2] *Delayed Auto Shift*. Sept. 2019. URL: https://tetris.wiki/DAS.

[3] Texas Instruments. *Dual 8-bit Multiplying Digital-to-Analog Converters*. Mar. 2020. URL: http://www.ti.com/lit/ds/symlink/tlc7528.pdf.

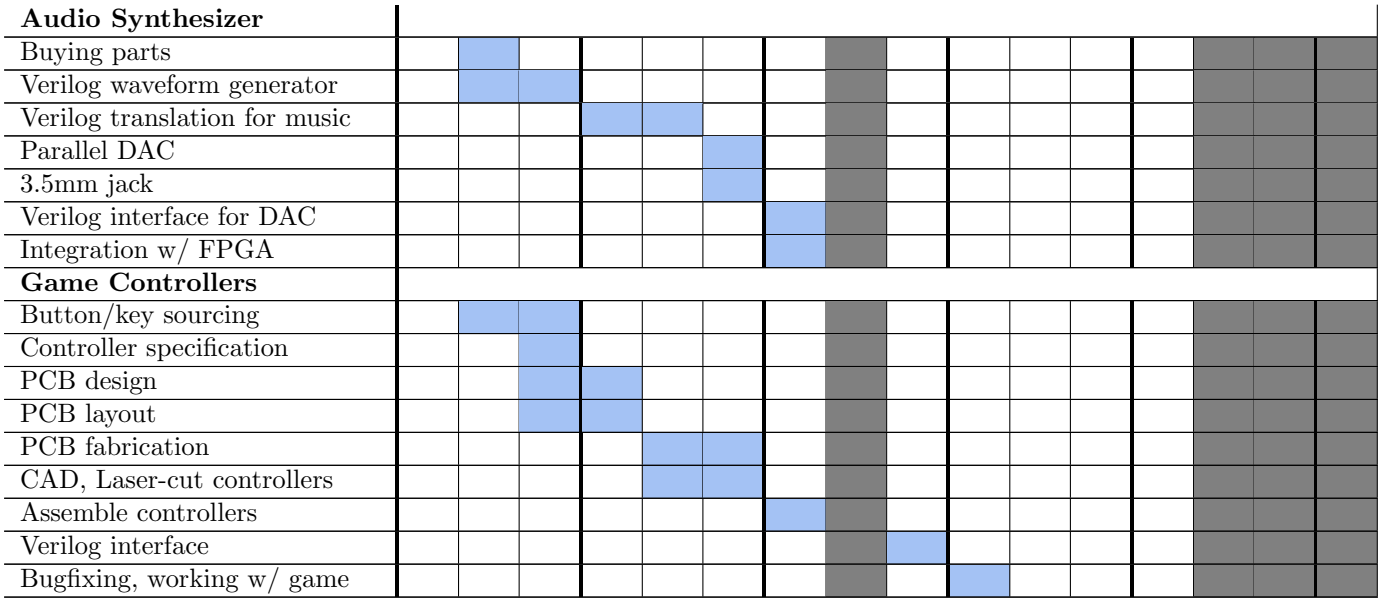
[4] *Super Rotation System*. Jan. 2020. URL: https://tetris.wiki/Super_Rotation_System.

[5] *Tetris Guideline*. Feb. 2020. URL: https://tetris.wiki/Tetris_Guideline.

[6] *VGA Controller (VHDL)*. Mar. 2020. URL: https://www.digikey.com/eewiki/pages/viewpage.action?pageId=15925278#VGAController(VHDL)-SignalTiming.

[7] R.W. Ward, T.C.A. Molteno, and University of Otago. Electronics Group. *Table of Linear Feedback Shift Registers*. Electronics technical report. Electronics Group, University of Otago, 2012. URL: http://courses.cse.tamu.edu/walker/csce680/lfsr_table.pdf.

Week of Jan 20 to May 4	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16
Game Logic																
Lateral movement																
Rotation																
Soft drop																
Hard drop																
Ghost Piece																
Wall collision																
Floor collision																
Rotation kick tables																
Piece sequence generator																
Piece Preview																
Local game start/stop																
Move reset piece lockdown																
Gravity/Auto drop																
Delayed auto shift																
Line clearing																
Hold piece																
T-Spins																
Combos																
Networked game start/stop																
Graphics																
Empty playfield																
Falling tile																
Locked board state																
Ghost piece																
Next tiles area																
Rendering text over VGA																
Lines sent area																
Upgrade from VGA to SVGA																
Start screen																
Waiting/Ready screen																
Opponent's board state																
Graphical assets																
Graphical assets																
Network Protocol																
Determine viable clock rate																
Protocol design/specification																
Receive stack																
Send stack																
Prototype data across boards																
Garbage queue and generation																
Synchronize game start/end																
Multiplayer game integration																



Appendix A: Gantt Chart

Red: Deanyone Su, Green: Eric Chen, Blue: Alton Olson, Grey: slack

Part Name	Qty	Cost/Item	Total Cost	Provided by Course
Sanwa Arcade Buttons w/ Microswitches (White)	12	$2.45	$29.40	No
Sanwa Arcade Buttons w/ Microswitches (Blue)	12	$2.45	$29.40	No
Sanwa Arcade Buttons w/ Microswitches (Black)	3	$2.45	$7.35	No
TLC7528CN Digital to Audio Converter	4	$4.76	$19.04	No
SJ1-3513 3.5mm Barrel Jack	4	$1.42	$5.68	No
PRT-12794 0.1mm 6” 20pc Ribbon Jumper Cables	8	$1.95	$15.60	No
M3BBA-1618J 16pin 2x8 Female Ribbon Cable	6	$3.67	$22.02	No
302-S161 16pin 2x8 Male Header	16	$0.40	$6.40	No
H3CCS-4036G 40pin 2x20 Female Ribbon Cable	4	$4.03	$16.12	No
SBH11-PBPC-D20-ST-BK 40pin 2x20 Male Header	8	$0.73	$5.84	No
Middleman PCB	5	N/A	$45.00	No
Controller PCB	x	N/A	$xx.xx	No
DE0-CV Altera Cyclone V FPGA	4	$99.00	$396.00	Yes
VGA Monitor	2	$69.99	$139.98	Yes
Quartus Prime Standard*	3	$2995.00	$8985.00	Yes
Total Budget Cost (w/o course equipment)			$201.85	Yes

Appendix B: Budget

* Quartus Prime Lite is free and also compatible with the DE0-CV. This tool can be used as an alternative to Quartus Prime Standard, for our purposes it only lacks support for multi-threaded compilation.