Tetris Project Design Document

Michael Lippe (ml5201), Bhargav Sriram (bs3586), Garvit Vyas (gv2361)

Spring 2025

1. Introduction

Tetris is a classic puzzle video game revolving around the strategic placement of falling geometric shapes known as Tetrominos. The goal is to rotate and arrange these pieces in such a manner that forms complete horizontal lines, which are then cleared from the screen, and points are given based on the number of lines cleared. As the game goes on, the falling speed of the blocks increases, and thus so does the difficulty.

Our project aims to implement a hardware-software system capable of playing Tetris. To accomplish our goal of playing Tetris, we will use an SNES-style USB controller, the DE1-SoC FPGA board, a VGA monitor, and some speakers.

1.1 Tetrominos Overview

In Tetris, gameplay revolves around manipulating geometric shapes known as Tetrominos, each consisting of exactly four squares arranged in seven distinct configurations. These configurations are labeled based on their shapes: I, O, T, S, Z, J, and L.

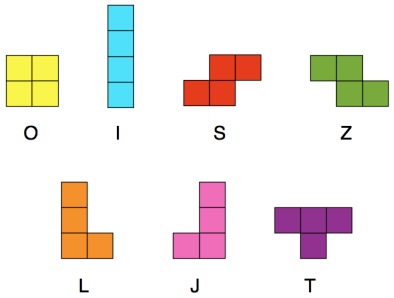

Figure 1: The Seven Standard Tetrominos

Each Tetromino can be rotated by 90-degree increments, allowing four possible orientations for each piece (except the O-Tetromino, which remains unchanged upon rotation). Players strategically rotate and position these pieces as they descend, aiming to fill horizontal rows completely to clear them from the playing field and score points.

The specifics for each Tetromino are:

- **I-Tetromino**: Straight-line shape, can rotate between vertical and horizontal orientations.
- **O-Tetromino**: Square shape, rotation does not affect its configuration.
- **T-Tetromino**: T-shaped configuration, has four distinct rotational states.
- **S- and Z-Tetrominos**: Mirror-image zigzag shapes, each has two rotational states due to symmetry.
- **J- and L-Tetrominos**: Mirrored L-shaped configurations, each with four rotational states.

2. System

2.1 System Block Diagram

The project can be broken down into five major sections: Input devices, software run on the HPS, hardware instantiated on the FPGA, built-in peripherals on the DE1-SoC board, and output devices. The block diagram from the project is shown in Figure 2, where the various components have been grouped together into the five major sections.

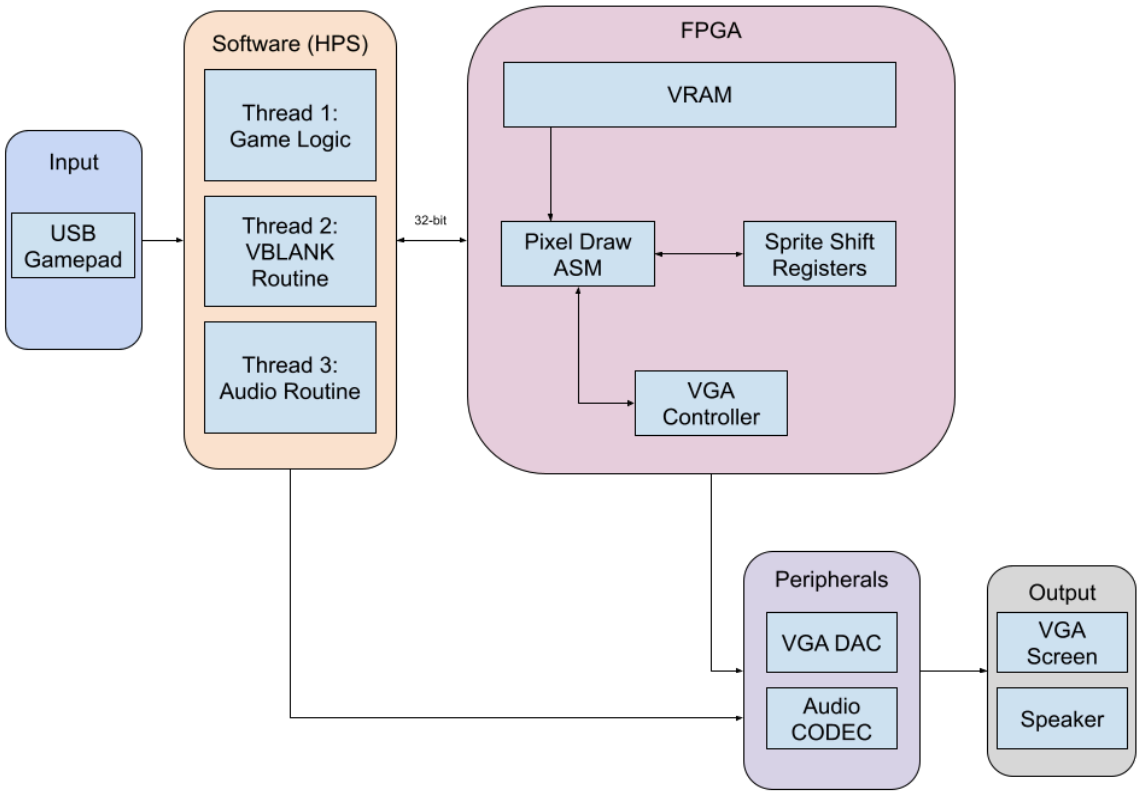

Figure 2: Tetris Project Block Diagram

2.2. PPU Hardware Summary

The Picture Processing Unit (PPU) is designed to drive a 640×480 VGA display at 60Hz. Graphics are built from 16×16 tiles where each pixel is encoded in 4 bits, and each tile is referenced via 16 bits of combined attribute data, specifically a 6-bit palette ID and a 10-bit tile ID. We support storing 1024 unique background titles in memory and 1024 unique sprite tiles. The tile buffer holds 1,200 tiles since the tile resolution of the screen is 40 x 30. Each color palette has 16 colors, and there are a total of 64 palettes. Our system allows for up to 256 sprites on screen at once, with a maximum of 16 sprites per scan line. Object Attribute Memory (OAM) is used for storing data about each of the 256 sprites, where each sprite is described with an 8-byte entry and is addressed via 16-bit words. Word 0 contains both the 6-bit palette ID and the 10-bit tile ID, word 1 stores the X coordinate, word 2 stores the Y coordinate, and word 3 holds the bits corresponding to horizontal and vertical flipping of the sprite's graphics. Table 1 shows the per-sprite mapping of OAM.

Table 1: OAM Map Per Sprite

Word Range	Field
Word 0	Sprite Graphics Palette ID and Tile ID
Word 1	X Coordinate
Word 2	Y Coordinate
Word 3	Rotation Flags

3. Algorithms

3.1. Hardware Algorithms

The PPU hardware implements an algorithm, referred to as Pixel Draw, which provides a mechanism for translating the tile, background, and sprite data from VRAM into an output frame. Pseudo code for the general pixel draw algorithm is shown below:

```
/* This procedure is executed every clock cycle with inputs H (current horizontal
pixel) and V (current vertical pixel) provided by the VGA controller.
Preprocessing is assumed to load sprite graphics into OAM.
A separate process (or pipeline stage) handles the sprite line determination when V
changes. */

if (V has just changed) then:
   // --- SPRITE LINE DETERMINATION (per scanline update) ---
   // Clear the list of active sprites for the current scanline.
   activeSprites = []

   // Check each sprite in OAM (from slot 0 to 255, order defines priority)
   for spriteSlot from 0 to 255 do:
       sprite = OAM[spriteSlot]
       if (V >= sprite.Y) and (V < sprite.Y + 16) then:
           // Calculate the row index within the sprite graphics.
           if sprite.verticalFlip is set then:
               rowIndex = (15 - (V - sprite.Y)) << 4   // Multiply by 16 to get byte
offset.
           else:
               rowIndex = (V - sprite.Y) << 4

           // Retrieve the corresponding 16-bit chunk from sprite graphics.
           if sprite.horizontalFlip is set then:
               // Load the 16-pixel data in reverse order.
               spriteLineData = loadReversedSpriteData(sprite, rowIndex)
```

```
            else:
                // Load the data in normal order.
                spriteLineData = loadSpriteData(sprite, rowIndex)

            // Append sprite info (maintaining OAM order for priority) with its data
and X coordinate.
            activeSprites.append({
                slot: spriteSlot,          // lower slot number means higher priority
                X: sprite.X,
                shiftRegister: spriteLineData, // initial content for the shift
register.
                enabled: False             // shift register initially disabled.
            })

            // Only consider up to the first 16 active sprites (by OAM order).
            if activeSprites.length == 16 then break

// --- SHIFT REGISTER CONTROL AND PIXEL OUTPUT (every clock cycle) ---

// For every clock cycle, using current H (and preloaded activeSprites for current V):
for each spriteInfo in activeSprites do:
   // Enable the sprite's shift register when H equals the sprite's starting X
coordinate.
   if H == spriteInfo.X then:
       spriteInfo.enabled = True

   // Disable the sprite's shift register when H reaches the end (X + 15) of the
sprite.
   if H == spriteInfo.X + 15 then:
       spriteInfo.enabled = False

// Now determine the final pixel color using the current outputs of the enabled sprite
shift registers.
outputPixelColor = None

// Check activeSprites in ascending OAM order (sprite slot 0 highest priority, etc.).
for each spriteInfo in activeSprites do:
   if spriteInfo.enabled then:
       // The current pixel color is taken from the shift register output.
       // The shift register is assumed to shift its data with each clock,
       // so its present output corresponds to the pixel at (H - spriteInfo.X).
       spritePixelColor = spriteInfo.shiftRegister.currentOutput()
```

```
      // If the sprite pixel is not transparent (nonzero), it is selected.
      if spritePixelColor != 0 then:
          outputPixelColor = spritePixelColor
          break  // Use the highest-priority sprite pixel available.

// If no sprite produced a nontransparent pixel, compute the background pixel.
if outputPixelColor is None then:
   // --- BACKGROUND PIXEL COMPUTATION ---
   // Calculate the tile buffer index using the high-order bits of the coordinates.
   tileBufferIndex = ((V >> 4) * (SCREEN_WIDTH >> 4)) + (H >> 4)
   // Calculate the pixel offset within the tile from the lower-order bits.
   tilePixelOffset = ((V & 0xF) << 4) + (H & 0xF)meaid
   // Retrieve background pixel from the tile graphics.
   outputPixelColor = getTilePixel(tileBufferIndex, tilePixelOffset)

// Finally, output the computed pixel color at (H, V).
drawPixel(H, V, outputPixelColor)
```

To minimize the critical path for the circuit that determines which of the 16 sprites or the background a given pixel should come from, we will use divide and conquer so that rather than a chain of 17 muxes, it will be a chain of length 5.

3.2. Software Algorithms

Our software will be split among three virtual threads. One thread will handle the game logic, one thread will handle the Vblank routine, and the last thread will feed raw audio data from the SD card to the audio CODEC for the background music. Because the game logic and Vblank routines are mutually exclusive, only two threads will be running at a given time, making it easy for the OS to schedule our three virtual threads onto the two physical threads of the HPS.

The following pseudo-code details the functions needed to implement the Tetris logic, based on analysis of existing Tetris code and a general understanding of how the game works.

```
ROWS 20 // play-field height
COLS 15 // play-field width

// Block shapes
SHAPES = { I, O, T, S, Z, J, L }

// Matrices
BlockMatrix[4][4] // current falling tetromino
```

```
Playfield[COLS][ROWS] // fixed blocks on the board
main() {
   initMatrix() // clear the playfield
   prepareFirstBlock() // choose initial shape & colour
   loop until Quit OR GameOver
       userInput() // non-blocking; may set flags
       stepTimer() // advances a tick
   end loop
   if GameOver
       showGameOver()
   end if
}

initMatrix()
// Set every cell of Playfield to BLANK (empty)

prepareFirstBlock()
// 1. Pick a random shape & colour for CurrentBlock
// 2. Initialise BlockMatrix accordingly
// 3. Place CurrentBlock at top-centre start position

userInput()
// Get most recent button presses polled during Vblank

stepTimer()
// Called once per frame during vblank
// Move or rotate the current block based on userInput() unless user input is Down
// If accumulatedDelay < CurrentSpeed and userInput() is not Down
// 1. accumulatedDelay++
// Else
// accumulatedDelay = 0
// 1. moveBlock(DOWN)
// 2. If the move failed due to collision with bottom of playfield or fixed blocks:
//      • fixCurrentBlockIntoPlayfield()
//      • checkForLineClears()
//      • spawnNextBlock() (may set GameOver)
// 3. update score display

moveBlock(direction)
// Attempt to translate CurrentBlock by one cell in the specified direction
// Call detectCollision() first; if collision then return FAILURE
// Otherwise update newBlockX / newBlockY and return SUCCESS
```

```
rotateBlock()
// Compute the 90° rotated version of BlockMatrix
// If the rotated piece collides with walls or fixed blocks then cancel
// Else copy rotated data back into BlockMatrix

detectCollision(candidateX, candidateY)
// Given a hypothetical position of CurrentBlock:
//   • Check left/right boundaries
//   • Check bottom boundary
//   • Check overlap with filled cells in Playfield
// Return TRUE if any collision is detected, FALSE otherwise

fixCurrentBlockIntoPlayfield()
// Copy all non-blank cells from BlockMatrix into Playfield
// at the current (BlockX, BlockY) offset

checkForLineClears()
// Iterate every row of Playfield
// If a row is completely filled:
//     • removeLine(rowIndex)
//     • increment LinesCleared and call calculateScore()
//     • after every 20 cleared lines call increaseSpeed()

removeLine(rowIndex)
// Delete the specified row by shifting all rows above it downward by one
// Insert a new blank row at the top of Playfield

increaseSpeed()
// Decrease the logical delay between automatic DOWN steps
// Increment Level counter (both affect difficulty)

spawnNextBlock()
// 1. Select NextShape & colour (random)
// 2. Reset BlockX, BlockY to the spawn coordinates
// 3. Initialise BlockMatrix for the new shape
// 4. If the new block immediately collides then set GameOver

calculateScore()
//Calculates score based on the number lines cleared at once and the current falling
speed
```

4. Resource Budget

4.1. Memory Usage

Using the M10K blocks, we have about 480KB of memory to work with. Table 2 shows how we have utilized that memory.

Table 2: Total Memory Used

	Size
Tile Buffer	2.4KB
Tile Graphics	131.072KB
Sprite Graphics	131.072KB
Color Pallets	3.072KB
OAM	2.048KB
Total	269.664KB

5. Hardware/Software Interface

5.1. PPU Memories

The main memory blocks in the PPU are the Tile Buffer, Tile Graphics Memory, Sprite Graphics Memory, Color Palette Memory, and Object Attribute Memory; each is implemented individually, with different widths based on the data it stores. The various PPU memories are exposed to the HPS as a single virtual VRAM. Table 3 shows the virtual VRAM mapping, width, and size of each memory.

Table 3: VRAM Map:

	Start	**End**	**Width**	**Size**	**Number of Addresses (Dec, Hex)**
Tile Buffer	0x00000000	0x000004B0	16-bits	2.4KB	1200, 4B0
Tile Graphics	0x10000000	0x10004000	64-bits	131.072KB	16384, 4000
Sprite Graphics	0x20000000	0x20004000	64-bits	131.072KB	16384, 4000
Color Palettes	0x30000000	0x30000400	24-bits	3.072KB	1024, 400
OAM	0x40000000	0x40000400	16-bits	2.048KB	1024, 400

5.1.2. Avalon Bus and PPU Registers

Ideally, we would use a 64-bit wide bus to communicate between the HPS and PPU, so that we could write the data for the widest memories in a single transfer. However, we are limited to a 32-bit bus width since the ARM SoC in the HPS is 32-bit. Thus, we will have two 32-bit registers for data, PPUDATA_LOW and PPUDATA_HIGH. We will also have PPUADDR for specifying the VRAM address and PPUCTRL to control the automatic address incrementation, Vblank interrupt, and Vblank CPU write blocking. Table 4 shows the PPU registers, their read/write direction, and their purpose.

Table 4: PPU Registers

Register	**Read/Write**	**Function**	**Misc**
PPUCTRL	W	PPU Control	The lowest 16 bits are used to control the PPUADDR autoincrement settings. Setting them to 0 disables autoincrement, while setting them to any other value sets the number of addresses to autoincrement by after a write. The MSB of this register is used to enable or disable the Vblank interrupt. The second MSB of this register is used to enable or disable allowing CPU writes to VRAM during non-Vblank periods.
PPUADDR	W	Specify VRAM Address	The address given is the virtual VRAM address.
PPUDATA_LOW	W	Lower 32 bits of VRAM data	Writing to this register initiates a write to VRAM, so it should only be written to after PPUADDR and PPUDATA_HIGH are set correctly. If autoincrement is enabled, the PPUADDR register will then automatically be updated by the specified increment value.
PPUDATA_HIGH	W	Upper 32 bits of VRAM data	Only relevant for the tile and sprite graphics memories.

Writing to VRAM thus goes as follows:

1. Set the lower 16-bits of PPUCTRL as desired to control autoincrement.
2. Write the virtual VRAM address you want to start the write at to PPUADDR.
3. If writing to the Tile Graphics or Sprite Graphics memory, set PPUDATA_HIGH to the upper 32-bits of the 64-bit data value you wish to write.
4. Set PPU_LOW to the lower 32-bits of the value you wish to write. If the memory you are writing to has a width of less than 32-bits, the extra topmost bits of PPUDATA_LOW will be truncated and ignored.
5. If you have set the auto increment to a value other than zero, repeat steps 3 and 4 for as long as you wish to continue this write burst.

For the Vblank interrupt from the PPU to the HPS, we will use the irq signal from the Avalon Bus.