Tetris Project Proposal

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

Spring 2025

1. Introduction

This project aims to develop a hardware/software system capable of playing Tetris. The project will be loosely based on how Tetris worked on 8-bit consoles like the NES, where instead of manually writing to a frame buffer, tile and sprite based graphics were used instead.

The FPGA portion of the DE1-SoC board will be used to generate the VGA video signal. As mentioned before, the FPGA will not just be a simple frame buffer, but more akin to a Picture Processing Unit (PPU) from something like the NES. The reason for using a tile and sprite-based PPU is due to VRAM limitations. If we build our VRAM out of M10K blocks, we have about 480KB of VRAM to work with, since there are roughly 390 M10K blocks, each of which stores 10Kb of data[1]. Since the VGA DAC expects 24-bit RGB values[2], 8-bits for each of red, green, and blue, we need 24-bits per pixel. With a minimum resolution of 640x480[2], direct RGB control of every pixel on screen would require 921.6KB of VRAM, about double the memory capacity we have available. Furthermore, if we want to smoothly scroll the background in any direction, we need to store 4 screens of data in VRAM, or over 3.6MB of data. This isn't doable with the limited memory we have available, so we instead turn to tile-based graphics like those used on the NES, which allow saving memory through reusing tiles. Another trick we can borrow from the NES's playbook is to use color palettes. Rather than assigning a full 24-bit RGB value to each pixel, we can instead define a set of color palettes in VRAM. Tiles, then, are stored as an array of numbers, with each number representing which entry in the color palette color table its corresponding pixel should be. Because we have quite a bit more VRAM to play with than something like the NES, we can define a color palette per tile rather than per sprite and per background. This allows further tile reuse since a single tile can be used with various color palettes to change its look significantly. Because the VRAM, along with the Object Attribute Memory (OAM) for the sprites, will be internal to the PPU, the PPU will need memory mapped IO to allow the CPU to write to VRAM and OAM. To allow for easier writing of VRAM, the PPU will have an auto increment option that allows the CPU to continuously write data to sequential locations in VRAM without having to manually update the address register every clock cycle. Our PPU will also need a VBLANK signal to tell the CPU when it is safe to access VRAM and OAM.

The CPU portion of the DE1-SoC board will be used to run the game code. However, once per frame the CPU also needs to read the controller data and write relevant data to VRAM and OAM. Because we have two cores[2], and thus two simultaneous threads, we don't have to deal with a messy and complex VBLANK interrupt system like what is found on the NES. Instead, we can have one thread solely responsible for processing game logic, and a second thread responsible

[1] https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/Cyclone5/index.html

[2] DE1-SoC User Manual

for executing the VBLANK routine, including updating VRAM and OAM, polling the controller, and anything else that should be run exactly once per frame. The advantage of this system is not performance but simplicity, since we don't have to worry about saving the internal registers of the CPU to somewhere in memory so we can jump execution to the VBLANK routine without corrupting the game logic. To be clear, this doesn't really offer a performance advantage, since we still need to pause execution of the program thread during VBLANK to avoid updating the state of the game while we are trying to construct the new frame; we don't want to for example, be processing logic that moves a sprite on the game logic thread as we are writing that sprite's position data to OAM on the VBLANK thread.

2. Specifications

Framerate: We will use a standard framerate of 60Hz, to allow compatibility with the widest range of VGA monitors. 60Hz also allows more time for VBLANK, which doesn't hurt.

VBLANK: We will take advantage of the VBLANK period to update graphics data, poll the controllers, and run any other routines that need to happen exactly once per frame.

Resolution: We will use the base VGA resolution of 640x480, since, as discussed earlier, we are severely VRAM limited and even this level of resolution requires some level of tile reuse.

Color Palettes: Our color palettes will be 6-bits, allowing 64 colors per palette and resulting in a palette size of 192B. Color palettes will have their own dedicated area of VRAM and we will allow for up to 64 of them, meaning the dedicated color palette area of VRAM will be about 12.2KB in size.

Tiles: Our tiles will be 16x16, for a tile resolution of 40x30 per screen and 80x60 per full 4-screen screen buffer. This means that one screen consists of 1,200 tiles, while the full size buffer consists of 4,800 tiles. The 16x16 tile size was chosen since it allows more detail and granular control than 32x32 tiles without having an overwhelming number of tiles as would be the case for 8x8 tiles. With a 6-bit color palette, each tile will be 192B. If we allow for 1024 tiles to be stored in VRAM, so a 10-bit tile ID, we need slightly less than 200KB of VRAM for the tile graphics data.

Tile buffer: With each tile slot in the screen buffer needing a 6-bit color palette specifier and a 10-bit tile ID, so 16-bits per tile, and the buffer being made up of 4,800 tiles, the buffer size becomes 9.6KB.

Sprite Data: We have about 200KB of VRAM left over which can be used to store the sprite graphics. To keep things simple, each sprite will be a single 16x16 tile. If larger sprites are desired, they can simply be constructed out of multiple sprites. With 200KB of VRAM to work with, we can support storing 1024 16x16 sprite tiles. 1024 is an absurd amount of sprites though, so we'll support 256 sprites being on screen at once, which is nice because we can use a single byte as the sprite ID.

OAM: Each sprite in OAM will consist of 7 bytes. Bytes 0 and 1 store the 6-bit palette ID and 10-bit tile ID for the sprite. Bytes 2 and 3 store the X position of the sprite. Bytes 4 and 5 store the Y position of the sprite. Byte 6 stores attributes about the sprite, such as whether or not it should be flipped vertically and/or horizontally, and whether it should appear in front of or behind the background. With 7 bytes per sprite and 256 sprites, OAM is about 1.8KB in size.

Controller: The controller will be a standard USB HID game controller, with a layout mimicking the SNES controller, mainly so we can use the shoulder buttons for piece rotation which is a nice quality of life thing over something like an NES controller. The controller will be polled during VBLANK, meaning it will have an effective polling rate of 60Hz which should be plenty for this kind of game.

Audio: Audio will be generated using the DE1-SoC board's built-in 24-bit audio CODEC.[2] Because we are already facing memory limitations with the FPGA, the HPS will handle driving the audio CODEC, meaning we can store the raw audio data on the SD card. Even though we only have two logical threads on the CPU, one of them does nothing during VBLANK and the other only does things during VBLANK. Thus, if we create a third virtual thread for handling the music, the operating system should have little trouble effectively scheduling the three virtual threads using the two logical ones.

Hardware-Software Interface: A device driver will be used to communicate with PPU. The device driver will include functions to manually address the PPU registers as well as general purpose functions that perform higher level tasks such as moving sprites, writing to VRAM, etc.

Software: The game we will run using this system will be an implementation of Tetris. Ideally, we would like to have levels that increase in speed as you complete them, just like the classic NES game.

3. Major Tasks

- Finalize all major design decisions such as resolution, tile size, VRAM capacity and layout, specifics of memory mapped IO, etc. The finalized design specifications will be recorded in the design document.
- Create a Verilator-based testbench for the Picture Processing Unit (PPU) Verilog code before synthesizing it on the FPGA. The testbench will write out each VGA frame as an image file.
- Design and write the PPU Verilog code based on the finalized specifications.
- Implement the verified PPU code on the FPGA, connecting it to all the required IO.
- Create a Linux device driver to call from C for interfacing with the PPU to make both testing the synthesized PPU and writing the game code easier.
- Design a C-based test program for the PPU so the synthesized hardware can be tested.
- Implement the game code for Tetris itself, splitting up the logic as applicable between the game-logic thread and the VBLANK thread.