petris

A Tetris Clone

Pushkar Mohile 180260027

Sankalp Gambhir 180260032

June 2020

Abstract – `petris` is a bad attempt at a portmanteau of EP and Tetris.

1 Project Details

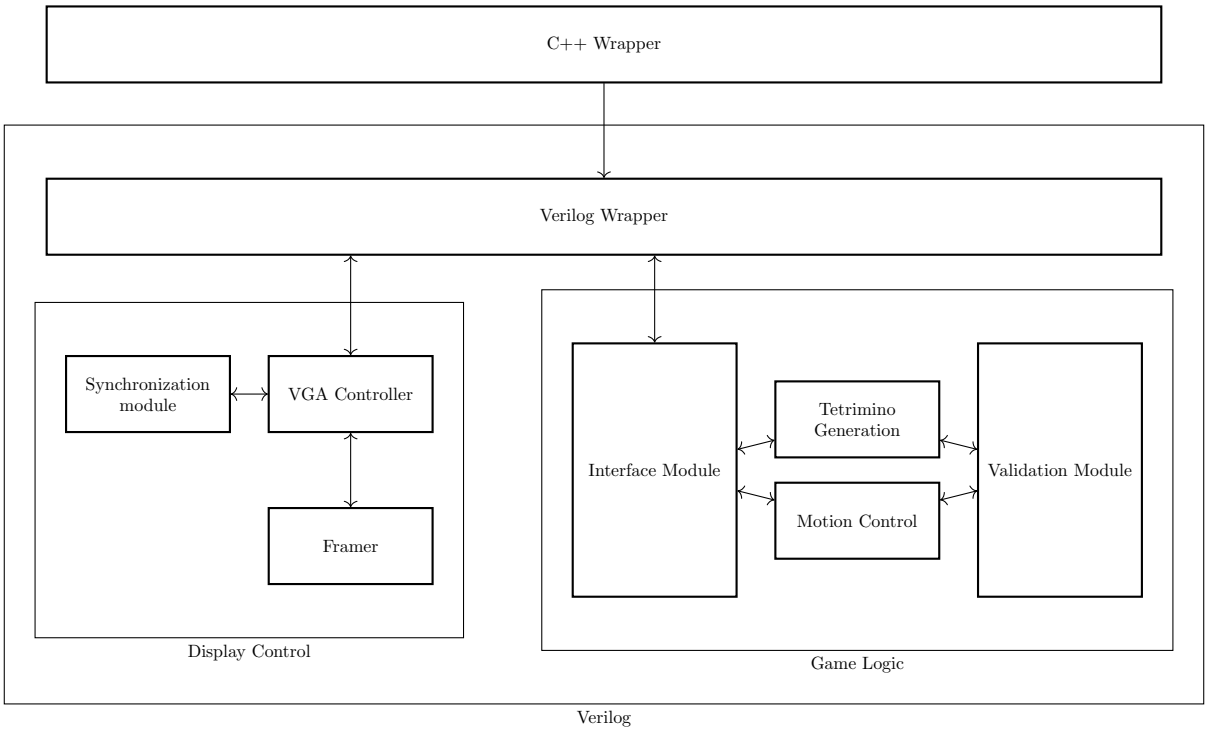

Figure 1: Block diagram *Sankalp : Better caption?*

1.1 Verilog Modules

1.1.1 Wrapper

1.1.2 Logical Modules

Interface Module

Tetrimino Generation Module

Motion Control Module

Validation Module

1.1.3 Display Modules

VGA Control Module

Framer Module The Framer Module maintains a frame buffer with controlled public access, maintained using a semaphore [**?**]. This allows other modules and tasks to write directly to the frame buffer instead of maintaining copies of the data potentially causing overwriting and synchronisation issues. The module writes this frame buffer onto an output buffer at `vsync`, in turn sent onto VGA rails serially (1.1.3) over the course of the next frame cycle. After flushing this data, the buffer is cleared, i.e. written over with black, preparing it for the next output cycle.

Synchronisation Module The Synchronisation Module receives a `clock` signal from the VGA controller (1.1.3) and calculates several signals critical to the VGA control flow:

- `vga_clock` — a signal generated by dividing the input `clock` signal by a known `multiplier` variable. The multiplier variable can be easily adjusted at runtime to allow for adjusting frame times in the event of a performance bottleneck or sudden throttle to prevent a drop in frametimes. This is akin to a primitive version of modern Adaptive Syncing [**?**], now popularized as often proprietary technologies built into modern GPUs, such as AMD Freesync [**?**] and Nvidia G-Sync [**?**]. Modern implementations instead use a specialised controller on the display side as well, communicating actively with its counterpart in the GPU, as opposed to our host-only solution.
- `vsync / hsync` — sync signals standardized as part of the VGA standard [**?**]. Intended to be passed via a DAC to the monitor. In our implementation, `hsync` sees little use other than this, while `vsync`, marking, the end of a frame, is at several points (**??**, **??**, 1.1.3, 1.2) used to flush a frame buffer or to advance a logical step.

1.2 C++ Wrapper

In the absence of an FPGA to run described modules, we had to look to other options. With a strong desire to maintain real-time playability, we tried several options — standard simulations had to be rejected due to not being anywhere close to real-time, with the inability to accept inputs. Having considered the idea of running a simulated clock, we then moved to forcefully driving a clock for the simulation. We attempted to use `verilog-vga-simulator` [**?**], a C-based program interfacing with the simulation via Verilog Procedural Interface (VPI) [**?**], drawing the resultant VGA output via Simple DirectMedia Layer [**?**] referred to as SDL herein. We were driven away by a cocktail of issues — primarily the performance,

combined with the relative inflexibility of VPI and the now obsolete [**?**] SDL API. However, the cocktail was more than resolvable.

Saving grace came in the form of Verilator [**?**], a compiler for Verilog which translates modules top-down into C++ code with I/O interfaces handled via classes [**?**]. This allowed us to bypass having to deal with VPI's idiosyncrasies and limited nature, paired with the finicky fixes required to obtain a realtime driven clock.

Armed with a way to handle inputs and outputs to our modules in realtime, we needed a VGA simulator to handle the outputs. This was executed using SDL2 [**?**] and a low-level frame-buffer, rendering to screen via a cross-platform hardware accelerated rendering interface (with OpenGL [**?**] primarily used for testing).

However, a software based simulation is unable to drive a clock fast enough to run a VGA simulation in any of the common modes [**?**]. As such, a myriad of performance-centric changes had to be made. Alongside several minor optimizations was a major change to the system — a drastic reduction in resolution or frame rate. Realistically, keeping the same resolution, the frame rate would have been cut to approximately `1/20 fps`, far from playable. The resolution would have to be cut similarly to maintain a playable frame rate. Since most of our clock cycles are used up driving the VGA controller, keeping true to the design, we chose to downscale it. Outputting a much smaller resolution from the simulated Verilog modules and writing an integer scaling [**?**] system within the relatively fast C++ wrapper.

We would like to add that much higher raw resolution and frame rates are possible and have been seen after removing the limitation to output serial VGA. Providing the frame buffer as output to the display simulator in parallel reduces the time complexity of drawing a frame from $\mathcal{O}(n^2)$ to $\mathcal{O}(1)$ in screen dimensions. However, having achieved desired levels of responsiveness and playability without resorting to bending the limits of the FPGA, we chose not to do this.

2 Main Components

2.1 I/O Modules

2.2 Hardware Description

2.3 Structural Description

2.4 Behaivoural Decription

3 Results