Project Report: Video Games on FPGAs

Adyanth Hosavalike, Chirag Dasannacharya

December 13, 2023

Please note that we presented our working games on the Pynq board in class on December 7th, 2023.

Abstract

This project explores the feasibility of configuring an FPGA as a simple console, programmed with a small collection of games and HDMI rendering capability. Three particular games are implemented - Snake, Tetris and Conway's Game of Life. Input is handled by keeping track of the Pynq board's built in buttons, while output is rendered through HDMI at a resolution of 1280x720.

We use a variety of methods to manage different challenges - an internal framebuffer is used to disengage frame updates from game logic. This framebuffer and additional buffers are used to scale the screen to a smaller virtual display for computational purposes. A number of FPGA-specific optimizations are used to improve end-to-end cycle time for our game logic.

We use an architecture that streams sequential pixels from a 'generator' through a 'frame processor', which sets its color based on state provided by the game update logic. The update logic itself is run after some configured number of frames are rendered. These pixel updates are collected at the framebuffer, which independently streams pixels to the output.

This report also covers our testing framework, which uses OpenCV to handle simulation. Finally, we observe the hardware intensity of this task in terms of resource usage on the FPGA board and go over some challenges faced in the course of this work.

1 Introduction

1.1 Motivation

This project was intended to observe the capacity of an FPGA to serve as a gaming platform. The motivation for this can be derived from the fact that the authors of this report like having fun while learning something useful. It was believed to lead to an increased likelihood of a rewarding project if the topic involved games.

Some initial research into emulating a Game Boy CPU suggested that this was not entirely beyond reach, as the instruction set itself is not too extensive. The Game Boy does, however, make extensive use of interrupts to generate and handle events, with programmer defined handlers specified for each kind of interrupt. These interrupts necessitate saving of state and introduce complexities that were considered potentially too challenging to address in the timeframe set for this project.

Accordingly, the project instead focuses on hard-coding[1] some games on the FPGA, with special focus on generating video output in a manner compatible with modern displays, through the use of HDMI. In particular, this project aims to achieve this without the use of a CPU core or off-board/off-chip memory for any tasks, a goal that was largely successful.

1.2 Games

The games chosen for this project include two common, popular games (Snake and Tetris) along with a cell-based automaton known as Conway's game of life. The first two have fairly minimal update logic that can run quickly on a regular CPU. The last, however is highly parallel and was chosen specifically for the capabilities of an FPGA.

Snake originated as an arcade game in 1976, and is the simplest of our chosen games. Updates change the direction of the head, check for an overlap with food and move (possibly elongate) the body. Chosen as a testbed, our Snake is limited to a length of 10 for reasons discussed in following sections.

Tetris is a puzzle game created in 1985 by engineer Alexey Pajitnov at the (then) Soviet Academy of Sciences. It spread quickly, with its US release in 1988. Tetris also has a fairly simple ruleset and corresponding update logic. Needs four buttons to play the game well and we did not have those many to spare.

Conway's game of life is an automaton developed by mathematician John Conway in 1970. It consists of a theoretically infinite grid of cells, each of which may be alive or dead based on interactions with their neighbors. These interactions are based on a simple count of live neighbors. Unlike the two games above, update logic here is proportional to the size of the grid. Noteworthy for an FPGA, each cell can also be handled in parallel.

1.3 Contributions

This project's main contributions lie in its development of a workable pipeline to write arbitrary data to an HDMI output from the FPGA. This includes processing either pixel by pixel or by scaled down frames, streaming from a framebuffer with the ability to update that buffer through the use of non-blocking reads. It also includes logic to follow the HDMI protocol, using the TLAST and TUSER bits (covered below).

Additional contributions lie in the exploration of two different strategies for updates, one looking up a logical framebuffer and the other passing each pixel through a static, bounded number of draw calls. This is explained in detail in the implementation, and is not dissimilar to the comparison between a raster and an older vector monitor.

2 Background

2.1 HDMI

This project depends on the HDMI requirements to decide most of the other blocks. HDMI (and its video only counterpart DVI) uses a signalling format known as TMDS, or Transition-minimized differential signalling, which sends data as two complementary positive and negative driven signal lines to prevent electromagnetic interference and ground noise [2]. HDMI also uses a pixel clock to separate the TMDS data packets with error correction. For a 720p30 video stream, this frequency needs to be above 30MHz to be able to transmit all the pixels for the frames at 30fps. The video timing is complex, including an active and a blanking interval for rows and columns, a remnant of the CRT days. The blanking interval is also used to send audio data, which makes the audio circuitry complex both at the transmitting and the receiving end.

To interface with HDMI on the Pynq board, Digilent has provided multiple helpful IP block on their Github repo [3], which includes the RGB to DVI block. Using that along with the built in Video timing generator and AXI-4 Stream to Video out block which outputs the needed timing and RGB data from an AXI stream correspondingly, one can use the HDMI output port on the board. The AXI stream uses a custom protocol described in the block's documentation, which uses TUSER to indicate the start of frame, TLAST to indicate end of line (pixel row) and uses the TDATA for pixel values. We used 24-bit full color RGB over this stream. Since the stream is sending data pixel by pixel, the video generation needs to feed this block with one pixel value every clock cycle. The converter does the clock domain crossing for us from the user IP logic frequency to the video clock frequency. Pynq also needs the HDMI module to be initialized in a certain way on boot, which we could not find documentation on, but just a warning in the Python module stating if we do not close the HDMI interface object, even after a bitstream flash, the HDMI module might not work. The impact of this was that, we have to use the Jupyter to enable the HDMI port after boot using the test pattern generator implementation discussed below, and then switch over to flashing our bitstream since that won't be controlled by the host code.

2.2 Related Work

MiSTer[4] is an open source project dedicated to providing emulation of old consoles without the performance degradation associated with pure software emulation. It does this by using FPGA hardware to handle performance sensitive parts of the emulation.

An important difference between MiSTer and this project is that MiSTer uses FPGAs as a hardware accelerator for a software emulator, while this project is based on using the FPGA as a standalone unit. In addition, this project specifically implements particular games rather than perform emulation in general.

3 Implementation

3.1 Architecture

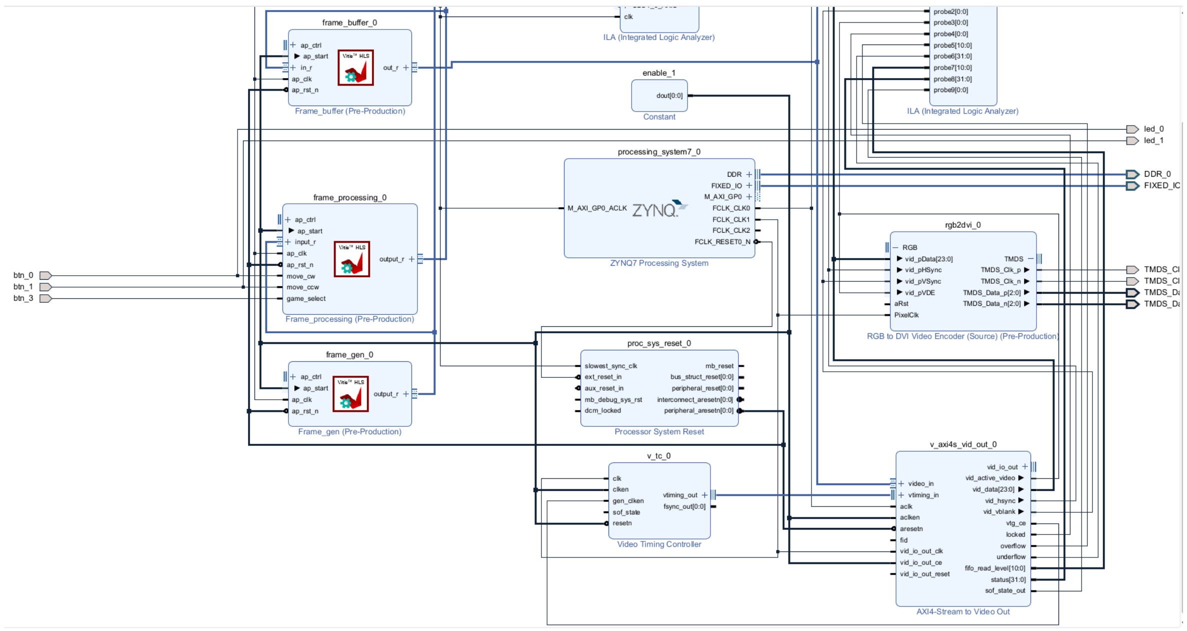

Figure 1: Block Diagram

The architectural approach used in this project involves separating the video and game logic into three blocks, shown on the left in the figure above. First, the frame generator streams 'pixels', consisting of X and Y coordinates, color (set to Black) and HDMI flags. These pixels are sent to the frame processing unit, which sets the pixel's color based on its location and the state of the game.

In case a certain number of frames has passed or a button press is detected, triggering frame processing also runs the game update logic. Button inputs can be seen coming from the extreme left of the image. When triggered, the game update runs a logic update, then returns to updating pixels.

Pixels are then collected at the frame buffer, a 10-to-1 scaled down version of the display, with 128*72 pixels. The frame buffer runs on a continuous loop with a non-blocking read facing the frame processing block, updates if any are applied. Independently, the frame buffer streams out pixels to the video out block, which sends it on to the HDMI port.

3.2 Display Logic

We started with a simple test pattern generator (TPG) IP provided by Digilent as the video source to make sure we have the video output working. We referred to the base overlay [5] for the required IPs, hardware and connection, but we could not use the base overlay itself since for one, it relies on the PS (processing system) to configure it, and two, it takes up too much board space with stuff that we would not use. After trying many different configurations, including different clock frequencies, different constraint files and IPs like the Video timing generator, AXI-4 Stream to Video out and RGB to TMDS blocks, we managed to get the HDMI output working with the missing HDMI HPD (hot-plug detection) pin pulled high.

Once we had something show up on the display, we naturally proceeded to add our processing logic in between the AXI link that connected the TPG to the AXI-4 Stream to Video Out converter using HLS.

In the HLS land, we started by writing a very simple graphics engine (GfxEngine) that we could reuse while writing games. It keeps track of the co-ordinates of the current pixel being streamed using the TUSER/TLAST fields, using that information to decide if the current pixel's color value should be changed or not. It also tracks how many frames were completed. We used that to "move" a colored dot diagonally across the screen to measure the approximate frames per second we could achieve, which was around 30. This was a vector based processing style which meant that we did not need to save the whole frame buffer in memory at the cost of computations for every pixel value, which we could afford to do. We did not see any output at first, and after debugging, we found that the AXI-4 Stream converter block needs a pixel value every clock cycle, meaning an Initiation Interval of 1. We optimized the graphics engine a bit to make this example work.

The architecture we selected, where there is a video source (TPG in this case) that can be swapped out with a HDMI input or something else, on top of which the frame processing block operates. This also means that the processing logic does not need to worry about setting the TUSER/TLAST at the right time. We created a frame generator block that spits out black frames as a background for all our games with an II=1, to be consumed by the processing block.

In hindsight, we should have read the video systems chapter of the PP4FPGAs [6] first to quickly get up to speed on many of the concepts.

3.3 Test Bench

Next step was to get a simulation working, since the testing flow was too slow to be quickly creating games. We needed an exact version of OpenCV and configure VitisHLS in a particular way to be able to use it while running the test bench. After that tedious task was out of the way, we could see what our processing logic spits out. We started coding Snake using the GfxEngine to draw rectangles and lines where the snake body should be and control the movement with direction and button input arguments. After it worked well on simulation, we found out that due to state that introduces dependencies, we could not pipeline the whole thing with an II of 1. We needed to come up with some other way of enforcing that.

3.4 Framebuffer

To isolate the logic from the video output side, we wrote yet another HLS IP block that would contain a frame buffer, consume the AXI-4 Stream asynchronously to fill the buffer while also streaming the available state of the buffer on the output without waiting for the input. This fixed the problem of II, but we could not implement a 1280x720 frame buffer due to lack of space on the PL (programmable logic), and thus decided to scale it down by a factor of 10 to 128x72, also making our simulation run much faster. We slowed down our game that is dependent on the frame rate to update every second frame to get half the speed. We also added a score system and quickly used the draw line and rectangle functions of the GfxEngine to display 7 segment style numbers.

3.5 Game Logic

3.5.1 Snake

For our implementation of Snake, with this being our initial test game, we use a very simple, 50 cycle update logic in the frame processor. The logic checks for button inputs, and if so sets the head's direction. It moves the snake in the direction of the snake head, and elongates it if the head overlaps with food. Food locations are statically coded into the program, with no randomization (this is introduced in other games).

For each element on the screen (snake head, body, food, score, etc), we use a corresponding 'draw' function call. Each pixel is streamed through each call. The call checks if the pixel position lies within the entity, and if so sets its color. Numbers are represented as a sequence of draw calls, one for each line segment.

As each draw call is a separate hardware unit with its own input and output streams, we are limited to a static number of draws, set by the program. As a result, we limit our snake to 10 food particles.

3.5.2 Randomization

We go to significant lengths to introduce randomization for our remaining games. We do this by using a linear feedback shift register (LFSR) configured to act as a pseudo-random number generator, and adding chaos to this register whenever we have an undetermined value we can use, such as a button press.

We use the algorithm of a pseudo-random LFSR as laid out in an open-source project [7]. Note that this is not a true random generator - it outputs every possible value before repeating any value, thereby covering the whole field.

This algorithm works by starting off the register in some non-zero seeded state. Every time a random number is requested, it shifts this register and sets the incoming bit as an XOR of a certain set of other bits, designed to give the characteristics described above.

We use this implementation to randomize the type, position and orientation of the next Tetris block as well as to set the initial state of the Conway automaton. Since we have no non-deterministic seed at startup, the automaton always starts and behaves the same. Accordingly, we introduce two mechanics - chaos and reset.

For chaos, when the chaos button is pressed, a random cell's liveness is flipped. For reset, we keep the value of the random register, without resetting it. This initializes the new automaton to a different state than the initial run. While Tetris also starts the same, button inputs and their timestamp as the game proceeds offer sufficient non-determinism to randomize the register.

3.5.3 Tetris

Our Tetris implementation uses a different method than the draw function described previously. Instead, we maintain an internal 'logical' framebuffer, distinct from the graphics framebuffer. This buffer is scaled down even further, in this case to 16*16, with a 4-to-1 scale to the graphics buffer and an 40-to-1 scale to the actual display. For simplicity, the game wraps around on the horizontal.

We track the 'active' or falling tetris piece, updating its position to move down. In case of button inputs to rotate and move, we simulate the action, check for overlaps on the logical buffer with existing entities and apply the action if there are none. The logical buffer is kept up-to-date with the type of the block occupying that location, if any.

When a pixel is streamed through, its coordinates are used to look up the relevant logical buffer entry, and its color updated. Compared to a draw call, this is significantly faster (a lookup takes 1-2 cycles, each draw takes at least 2 cycles, a sequence of draws can result in a 50+ cycle latency).

This comes with the tradeoff that it is difficult to draw non-grid based figures such as slopes, curves, numbers (except by rendering them very big), etc. It also leads to a longer update logic, in the order of hundreds of cycles.

3.5.4 Conway's Game of Life

For the Conway automaton, we use a similar graphics approach as Tetris, with an internal logical framebuffer. Like previously, the grid wraps around, but now both horizontally and vertically. Random initialization takes a large cycle count, as the random register is shifted after each use (i.e. after each cell is assigned a starting liveness).

On update, a first pass counts the neighbors of each cell. A second pass then sets a cell's liveness based on its neighbor count. These loops are unrolled as the operations are independent from one cell to the next. Passes must still occur in order, to prevent modifying the state before neighbor computation is complete.

FPGAs are seen to be able to perform Conway automaton updates at a very high rate. This is not unexpected, given their ability to perform massively parallel computations. Finally, we also accept button inputs to introduce chaos or perform a reset, as described in the section on randomization.

4 Results

We are successfully able to write to an HDMI screen purely (almost, disregarding initialization) from the FPGA. We also provide a successful implementation of a framebuffer and rendering unit and use these to develop games, with a successful demonstration. This covers our goals as outlined in the proposal.

A video of the demonstration can be accessed from the class recording on Dec 7th 2023. In addition, this link [8] is a recording of the games running on simulation and this link [9] is a recording of the games running on the FPGA. We were able to fix the bug in the Conway game logic as well as the line clear issue in the Tetris as seen in the above videos.

We showcased 3 games working off of the Pynq board without using the PS. We hit an II of 1 for both the frame_gen and frame_buffer IP blocks. Frame_processing has an II of 5̃0 cycles for Snake, Conway and Tetris taking 6̃00 cycles due to the difference in implementation styles. With a 10x upscaling in the frame buffer block and a further 8x upscaling for the internal framebuffer of Conway (4x for Tetris), we were able to easily fit within the resource constraints of the board after consciously applying loop pipelining and unrolling to loops that did not have dependencies.

Frame generator takes up a tiny area with only 29 FFs and 197 LUTs in total. Frame buffer takes up 30 BRAMs, 72 FFs and 336 LUTs due to the need of saving a 128x72 24-bit color pixel grid. Frame processing (with all three games) takes the highest with 10 DSPs, 10137 FFs and 32803 LUTs. Vivado shows the whole design utilization after place and route to be 34.5 BRAMs, 15375 FFs and 11513 LUTs.

Table 1: Resource Utilization as reported by Vitis and Vivado

Block	DSP	BRAM	FF	LUT
frame_gen	-	-	29	197
frame_process	10	-	10137	32803
frame_buffer	-	30	72	336
Total (Vivado)		34.5	15375	11513

5 Challenges and Future Work

The main challenge seen is to keep the timing down by breaking dependency chains while the game logic grows complex for the rendering to work well. Another significant challenge is to debug the HDMI connection logic, this was done using ILAs.

Further work could involve swapping the frame_gen block with an HDMI input stream from a laptop or other display, to render on top of the incoming video stream for an end-to-end video manipulation pipeline. This would need the Vitis Vision libraries to be set up. Going further, the next step would be to add synthesised audio to the HDMI output using existing open-source IPs like the ones in [10] and [11].

Another direction for further work would be to try to scale the games without increasing resource impact, and adding additional games on the side. One good candidate would be a 2D platformer such as Mario with simplified graphics.

We do still run into some apparent limits on total game update logic cycles, beyond which we see no output. Identifying and rectifying the reason for this would enable much more complex game logic. Similarly, repeated draw calls result in flickering and glitchy images at times, this is also an area for exploration.

6 Conclusion

We succeed in implementing video games on FPGAs. Our code can be found here [12]. FPGAs are seen to be highly versatile and capable of handling niche applications. In this case, we effectively build a custom Snake/Tetris/Conway automaton chip with potentially high performance at low power use.

We explore and utilize a number of image and video processing techniques, including framebuffers, pixel streaming and differential updates (for Tetris and Conway). FPGAs are found to be particularly capable of performing highly parallel computations at high frequencies.

We are able to present a working, end-to-end video game console capable of writing directly to a screen with button inputs, with a selection of games to choose from.

References

[1] Hehe, hard-coding, get it? Because FPGAs. Sorry.

[2] Cadence, "Differential signalling," last accessed 12 December 2023. [Online]. Available: https://resources.system-analysis.cadence.com/blog/msa2021-the-advantages-of-differential-signaling

[3] Digilent, "Vivado free-to-use IP library," last accessed 12 December 2023. [Online]. Available: https://github.com/Digilent/vivado-library/tree/master

[4] MiSTer, "MiSTer project to recreate classic computers, video game consoles, and arcade machines using modern hardware." last accessed 12 December 2023. [Online]. Available: https://mister-devel.github.io/MkDocs_MiSTer/

[5] Xilinx, "PYNQ-Z2 base overlay," last accessed 12 December 2023. [Online]. Available: https://github.com/Xilinx/PYNQ/tree/master/boards/Pynq-Z2/base

[6] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel Programming for FPGAs," *ArXiv e-prints*, May 2018.

[7] C. Taylor, "Linear feedback shift register based psuedo random number generator in Golang," last accessed 12 December 2023. [Online]. Available: https://github.com/taylorza/go-lfsr

[8] "Video of the project being simulated on OpenCV." [Online]. Available: https://drive.google.com/file/d/1rQauaKFQ16nY7BTeCzoh4SnHmnncnhfn/view?usp=sharing

[9] "Video of the project running on the FPGA." [Online]. Available: https://drive.google.com/file/d/1tItzSIGdtYFcsfqhD-l9iw2FjVvF6eXF/view?usp=share_link

[10] B. Sune, "Zynq HDMI IP with Audio," last accessed 12 December 2023. [Online]. Available: https://github.com/briansune/ZYNQ-HDMI-IP-with-Audio

[11] S. Puri, "HDMI video/audio output on FPGA," last accessed 12 December 2023. [Online]. Available: https://github.com/hdl-util/hdmi

[12] A. Hosavalike and C. Dasannacharya, "Video games on FPGAs," last accessed 12 December 2023. [Online]. Available: https://github.com/ahosavalike-ucsd-courses/cse237c-fp