FPGA Royale Final Report

Maxwell Jiang
Department of Mathematics
Massachusetts Institute of Technology
Cambridge, MA, USA
mdjiang@mit.edu

Frederick Tang
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
ftang@mit.edu

***Abstract*—We implement *Clash Royale*, the real-time, player-vs-player mobile game, on an FPGA. Our system functions on a custom processor that reads game-logic instructions written in a domain-specific language (DSL) of our own design. The system implements 2D graphics to render and animate sprites following the game's logic, and it allows for user interactivity by way of PS/2 mouse control.**

***Index Terms*—Field-programmable gate array, gaming**

I. Introduction

Clash Royale is a popular real-time, player-vs-player mobile game in which each player aims to destroy the other's *towers* by deploying *troops* to a battlefield. To deploy troops, the player selects one of four *troop cards* (henceforth simply *cards*) displayed in the user interface on their side of the battlefield, then indicates a deployment location on the battlefield. Deployment costs *elixir*, a self-regenerating quantity associated to each player. Once deployed, troops advance and attack automatically, engaging with enemy troops and towers following pre-programmed rules that vary by troop type.

We implement a system enabling two players to engage in Clash Royale gameplay on a shared interface run by a single FPGA. The system consists of five components:

- The processor: responsible for executing the core game logic and directing the graphics module accordingly.
- The graphics module: responsible for rendering to the HDMI display.
- The mouse interface: responsible for interpreting inputs from two PS/2 mouse hardware units (one for each player).
- The game logic program: responsible for determining gameplay rules and behaviors. It is written in a domain-specific language (DSL) that resembles RISC-V assembly but is customized for our Clash Royale implementation.
- The assembler: responsible for converting the game logic program into raw binary for the processor to read at runtime.

The diagram in Figure 1 shows the high-level flow of information in our system. The code can be found in the repository here.

More generally, our system provides a general framework for a simple 2D game engine on an FPGA, capable of rendering and animating a variable number of sprites simultaneously (in our case, up to 64), at 60 FPS, supporting custom, programmable game logic and meaningful user input.

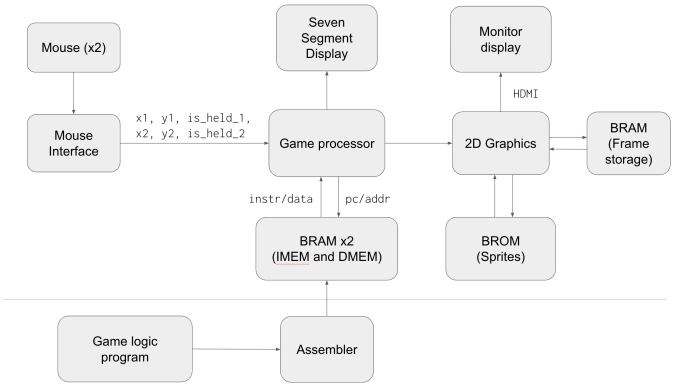

Fig. 1. Overview of FPGA Royale

II. Graphics

The graphics module is responsible for rendering all *sprites* to an HDMI display, which is the players' visual interface with the game. Troops, towers, cards, mouse cursors, elixir, and user interface text are all considered sprites. The rendering follows a standard, 720p, left-to-right and top-to-bottom HDMI rasterization protocol. We use *frame* to refer to a unit of 1650×750 clock cycles comprising one out of the 60 frames rendered each second. Correspondingly, the graphics module operates at a clock speed of 74.25 MHz.

A. Spritesheet

The visual data of the game sprites is precomputed and then accessed via read-only memory at runtime. This data is organized into two files, a *spritesheet* and a *palette*. The palette holds a small number of 24-bit RGB values, while the spritesheet holds indices referencing RGB values in the palette. The spritesheet holds a value for each pixel of every animation frame of every sprite in the game, so it is comparatively large. The two-file design saves memory by significantly reducing the width of each entry of the spritesheet. The creation of the spritesheet and palette is carried out by a Python script. The script starts with a PNG file containing all the sprites and applies the image discretization tools available in Python's PIL library. At runtime, the spritesheet and palette are loaded into BROM memory for the graphics module to read.

Our spritesheet contains 24 total sprites (animation frames), each one measuring 48×48 in pixels. Our palette consists of 16

colors, the final three of which are hardcoded values separate from the values extracted from the spritesheet PNG: a green for the background terrain color and a blue for the background water color, and a grey for the user interface banner containing the cards.

B. Rendering Logic

At a high level, the graphics module acts as a "register for frames". Namely, at any given time, it outputs the data of the current frame to the HDMI display while simultaneously receiving the data of the next frame from the processor. The module accomplishes this with two BRAM memory modules (which we call *frame memory*), writing to one of them to store the data for the next frame, and reading from the other to output the data of the current frame. The two frame memory modules must swap roles on each subsequent frame.

We now describe the mechanism by which the processor interfaces with the graphics module. The processor controls the following inputs to the graphics module:

- `sprite_valid`: when the signal is high, the processor is indicating to the graphics module that it is communicating the location and appearance of a sprite to be rendered in the next frame.
- `sprite_x`: indicates the horizontal position of the sprite to be rendered.
- `sprite_y`: indicates the vertical position of the sprite to be rendered.
- `sprite_frame_number`: indicates the index in the spritesheet that the graphics module should use to render this sprite.

When `sprite_valid` is high, the graphics module reads the above signals and then begins reading from the spritesheet, obtaining palette indices that it stores to frame memory. This process takes a number of clock cycles equal to the number of pixels in a sprite animation frame, namely 48^2. The graphics module controls a signal `sprite_ready`, indicating to the processor when it is ready to receive new sprite data. The processor and graphics module interact this way many times over the course of a frame, allowing the processor to dictate the appearances of a variable number of sprites, so long as the total time it takes for the graphics module to read and store all the sprites fits within the duration of a frame.

Reasons why this system of processor-graphics communication is convenient include:

- Modularity: the communication protocol helps to decouple the implementations of the processor and the graphic module. In particular, the graphics module does not need to concern itself with *what* is being rendered, only how to render it; control of what is being rendered and where is left up to the game logic.
- Scalability: the processor has a frame's worth of clock cycles to specify the next frame's sprites and their locations to the graphics module. Hence, our system's graphics module is capable of rendering $1650 \times 750/48^2 \approx 500$ sprites which is far more than enough for our purposes. (Our actual system only actually uses up to 64 sprites).

Figure 2 gives a high-level but slightly more detailed view of the graphics module – it is a simplified version of our true implementation, but it is detailed enough to convey the important ideas. The reader wishing to understand the full details should consult Section II-C.

The diagram shows the following signals at the left, which the graphics module computes internally:

- `reading`: a bit that is high when the module is using the processor's inputs to read from the spritesheet.
- `spritesheet_addr`: indicates an address in the spritesheet BROM. When the module starts reading a new sprite, `spritesheet_addr` is set to `sprite_frame_number * PIXELS_PER_FRAME` where `PIXELS_PER_FRAME` is the number of pixels in a single sprite animation frame.
- `frame_loc_ptr`: indicates an address in a frame memory BRAM to write to. When the module starts reading a new sprite, `frame_loc_ptr` is set to `sprite_y * FRAME_WIDTH + sprite_x` and is then incremented on each subsequent clock cycle.
- `output_index`: the (flattened) index of the current pixel begin rendered to HDMI, computed combinationally as `hcount + FRAME_WIDTH * vcount` where `hcount` and `vcount` are the pixel's horizontal and vertical locations. `output_index` is used to index into the frame memory used to render the current frame.
- `write_mem_1`: a bit that is high when the first BRAM frame memory is being written to for next-frame storage, and low when it is being read for current-frame output. `write_mem_1` is toggled on each new frame.

C. Additional considerations

For completeness' sake, we outline some additional implementation details regarding the graphics module:

- The full 1280×720 screen size proves too large to be contained in the frame memory BRAMs on a single FPGA. Moreover, Clash Royale is played in a portrait orientation. Hence we define paramaters `CANVAS_WIDTH` and `CANVAS_HEIGHT` defining the region that holds all gameplay, and maintain a signal indicating whether `hcount` and `vcount` lie in that region. The final output color is set to black if not.
- In each 48×48 sprite animation frame, some of the pixels should be transparent because sprites never occupy he entirety of their animation frame. To implement this, the spritesheet PNG has uses white for transparent pixels so that the palette-making script includes white as the 0th palette color. In hardware, a `read_color_index` value of 0 is treated as a command to not overwrite the existing color value in frame memory, thus implementing transparency.
- While the graphics module is reading from the frame memory BRAM that holds the data for the current frame, it is *simultaneously* writing to that frame memory in order to "reset" the BRAM's contents to the background color (either green, blue, or grey depending on the values of

hcount and vcount). This prevents incorrect persistence of sprites in locations they no longer occupy.
- To correctly render sprites that extend partially beyond the edge of the playable area (i.e. "offscreen"), the graphics module maintains a signal indicating when frame_x and frame_y point offscreen. When the signal is low, writing to frame memory is disabled. Without this implementation detail, sprites "wrap around" when they go offscreen.

III. Processor

The logic of the game, including troop/building AI and user interaction, is coded in a domain-specific language (DSL) that we have created and run on a custom processor.

The processor is similar to a RISC-V processor. As shown in Figure 4, we have 5 stages: Instruction memory (in BRAM), a register file with 32 registers, a decoder, an ALU, and memory (BRAM). The processor is single instruction, meaning each instruction will go through all five stages before the next one starts.

A. Sprite File

Almost all game logic in Clash Royale involves troops, which can be represented as sprites. Thus, we decided to add a second register file called the sprite file. The sprite file consists of 64 sprites, where a sprite is an length-8 array of 13-bit integers. Each element in the array represents an attribute of the sprite. In our program, the attributes are listed in Figure III-A. We chose the sprite file to have 13 bit instead of the standard 32 bit integers because none of the attributes require numbers larger than 8191.

The attribute indices in Figure III-A are not hard-coded in. The only requirements the hardware has for the indices in the sprite file is that 1 is the x location of the sprite, 2 is the y location, and 3 is the frame number; the processor uses these indices to render the sprites at the correct location and with the correct frames. We do not specify how the rest of the attributes map to indices to give programmers freedom to choose. Without the sprite file, all this information must be stored in memory, which takes multiple cycles to load from and write to. The majority of operations in the game used sprites, so the sprite file decreased the number of cycles by 50 percent or more. In addition, the sprite file allowed us to add sprite specific instructions to our design specific language, which we will discuss in the next section. As mentioned before, the processor feeds sprite data to the graphics module for rendering each frame, and reading this data directly from the sprite file versus from memory facilitates this interaction as well. The sprite file will not only be used for troops, but for every entity on the battlefield, including buildings and each player's cards.

The sprite file has dedicated 2 sprites for mice, 62 and 63. The processor accepts input from the Mouse Interface, as will be discussed in the User Input section, and wires the information to these two sprite indices, allowing programmers to use it in software.

We summarize each stage of the processor below:

- Instruction Handler: If an instruction has just completed processing, this stage loads in the next instruction and passes it to the decoder stage, according to the instruction pointer. This stage takes 2 cycles since it has to read from instruction memory, stored in a BRAM module.
- Decoder: This stage takes in an instruction and computes *rs1*, *rs2*, and *rd*. *rd* is the destination register or sprite of the instruction (some instructions like memory writes). *rs1* and *rs2* are the value of any registers, sprites, or immediate numbers used for computation in the instruction. The stage takes 1 cycle.
- ALU: This stage does computation for the instruction using *rs1* and *rs2*. The stage takes 1 cycle.
- MEMORY: This stage loads and stores values from/to memory as necessary. Memory for our processor is a BRAM module. The stage also does any write backs to the register file, sprite file, and instruction pointer (for jumps). The stage can take either 1 or 2 cycles, depending on if the instruction requires memory accesses.

B. Rendering

The processor communicates with the graphics module to render sprites while running instructions. One of the inputs to the processor is a new_frame signal, and it has 4 outputs: sprite_valid, x, y, frame; these inputs and outputs are wired to the graphics module. If new_frame is high, then the processor will loop through the entire sprite file. At each index, if the sprite is alive, it loads the sprite's x, y, and frame data into the respective output registers. After looping through the sprite file, the processor also reads the elixir each player has from two dedicated registers (30 and 31), and renders the amount accordingly. This is all done in parallel with processing instructions, so rendering does not increase the processor's latency.

C. Tower Health

The goal for a player of FPGA Royale is to destroy the opponent's 2 towers while keeping their own 2 towers alive. In order to gauge the number of hit points remaining on player's towers, the processor sends the hitpoints of all 4 buildings to the FPGA's 7 segment display. Each tower starts with 255 hitpoints, which can be represented with 2 hexadecimal digits. Thus, the 8 possible digits on the FPGA can fit all the four buildings: the top players' tower hitpoints are the left four digits, and the bottom players' tower hitpoints are the right four. This functionality adds a requirement that the first two and last two sprites are the player's towers.

IV. Domain-Specific Language (DSL)

The DSL is a low-level, assembly-like language. The language has most of the RISC-V instruction set, including save word, load word, arithmetic, jump, and branch instructions. The logic of the game is coded in the DSL, and it is assembled into machine code and then run on our processor.

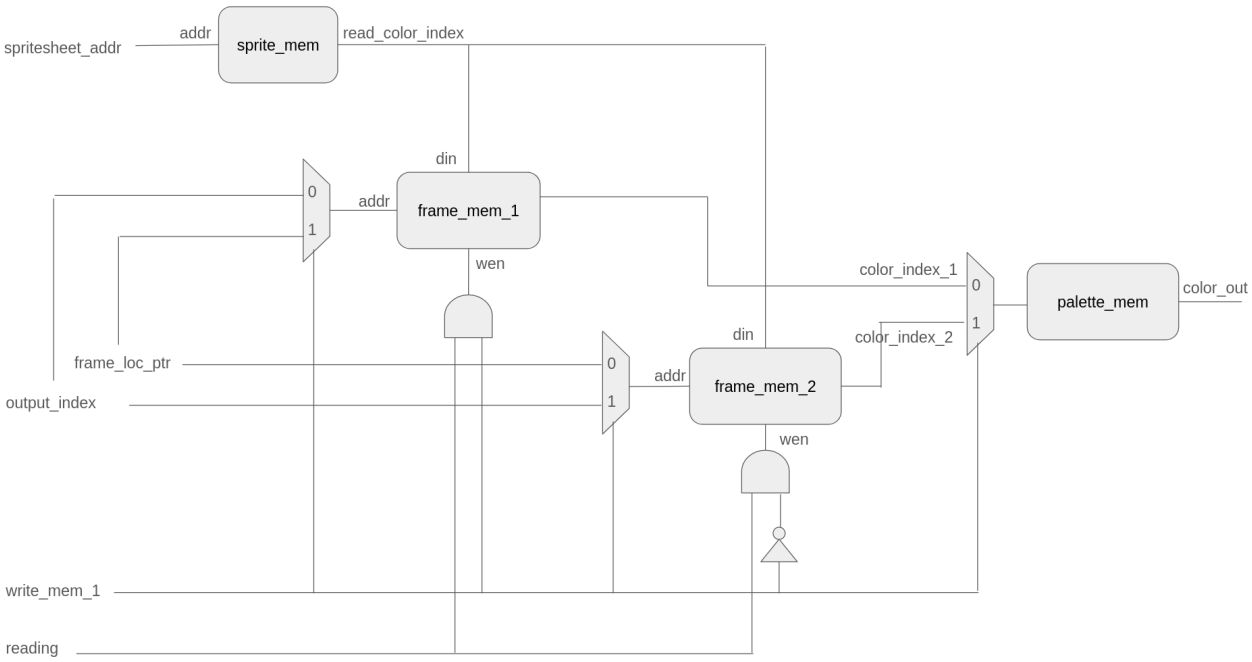

Fig. 2. Graphics module, a simplified view

Index	Description
0	sprite type index, where each number from 1 to 10 corresponds with a type (i.e barbarian, building, mouse)
1	X location of sprite
2	Y location of sprite
3	frame number in sprite sheet of sprite
4	health of sprite
5	damage of sprite
6	state of sprite. 0 means idle. 1,2 mean walking. 3,4 mean attacking
7	team of sprite (0 or 1).

Fig. 3. Sprite File Array

As mentioned previously, we have sprite file specific instructions such as loading between the sprite file and the register file. Because accessing sprites also involves indexing into the array of a sprite, our instruction size is 36 bits instead of the standard 32 bits of RISC-V. The 3 leading bits are for the sprite array index, and the 4^{th} is a flag to identify the instruction as a sprite instruction.

Since we do many distance calculations for sprite behavior, we have a distance instruction, which will store the manhattan distance between two sprites (given by their starting addresses) in a destination register.

When a troop attacks another troop in the game, the second troop's health is decremented by the first troop's damage. Since both these attributes exist in the sprite file, we have a special instruction called "attack," which handles this.
We also have a wait instruction, that tells the processor to wait a given number of cycles before continuing. Figure 5 shows a list of the instructions.

A. User input

We connect two pmod PS/2 mice connectors to the FPGA, and the `mouse_interface` module is responsible for updating the game with mouse movements and clicks. In particular, each of the two mouse interfaces sends signals `mouse_x`, `mouse_y`, and `clicked` to the processor, indicating the mouse's position and click status at some point in time.

The code for interfacing with the mouse using the PS/2 protocol is taken, in part, from the sample code provided by Digilent, the manufacturer of the `pmod` PS/2 connectors. Briefly put, the interface sends "initialization" signals to the mouse hardware, which responds by streaming data in 33-bit packets organized into chunks of 11 bits each; see Figure 6. The interface decodes the input using a FSM.

We implement a wrapper module for Digilent's code and also write logic that sets the mouse's maximum x- and y-coordinates to be `CANVAS_WIDTH` and `CANVAS_HEIGHT`, respectively. We also modified several "delay counter" vari-

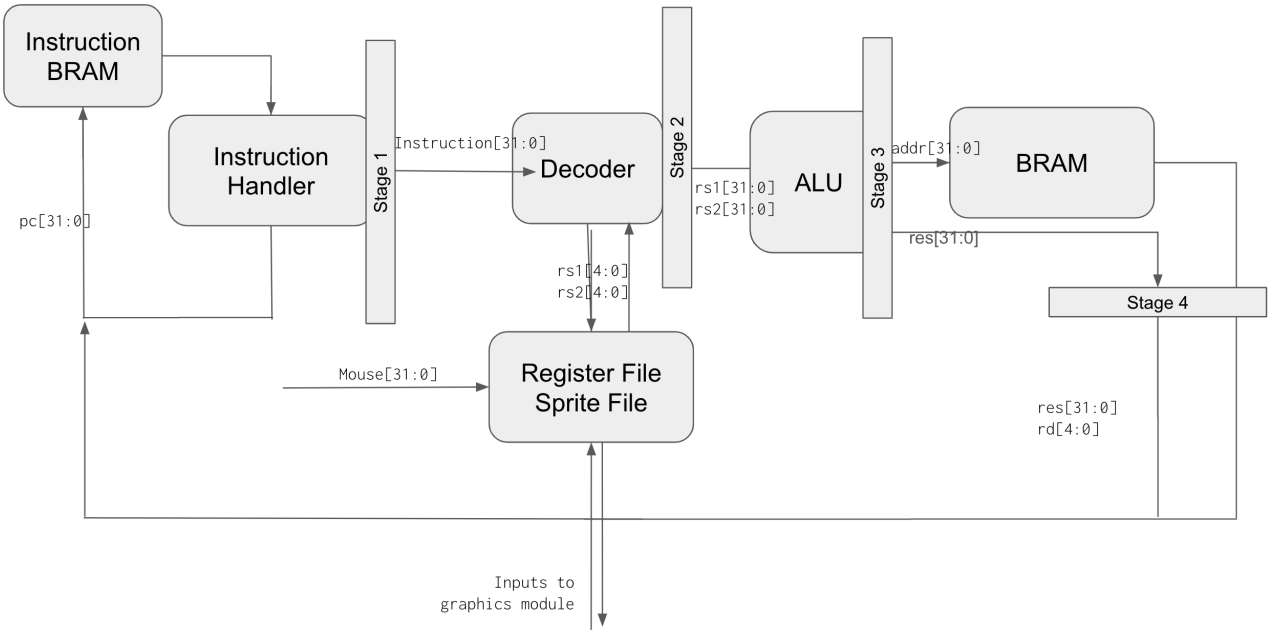

Fig. 4. Processor

ables in Digilent's code in order to adapt the system from 100 MHz to 74.25 MHz.

V. Results and Evaluation

We argue that the latency of our system, i.e. delay between user input and its effect on the rendered display, is at most 1/60 of a second (the duration of one frame). This is low enough to be undetectable by human players. The reasoning is as follows: because the graphics module stores the current frame's data during the previous frame, our claim holds as long as the latency between the user input and the graphics module writing changes to frame memory is less than the duration of one frame. This is indeed the case – the main bottleneck is the polling rate of the PS/2 mouse, which is 100 Hz (a standard estimate). The rest of the system contributes small latency, since the total pipelining in the mouse interface module and processor is on the order of tens of clock cycles. This puts total time well within a single frame's time.

Of the 648 instructions in our game logic, 330 use sprite instructions. These instructions on a standard RISC-V processor would require multiple memory accesses each. Conservatively, we assume that each sprite instruction would take 2 more cycles if replaced by RISC-V instructions, so our processor is two times more efficient for FPGA Royale.

Our system exhibits a slack (WNS) of 0.502 ns, so our system meets timing constraints.

All of our memory usage fits onto one FPGA. We used 2646 kilobits of BRAM, which is 98% of our FPGA's capacity. The bulk of the BRAM usage came from the two frame memory units in the graphics module, each using 32 RAMB36 units. We used 44.5% of our Slice LUTs.

Figure 7 is a picture of the gameplay, displaying multiple mechanics of the game. The top and bottom rectangles represent how much elixir each player currently has (two for the top four for the bottom). The mice are the orange and blue cursors. The four troops at the top and bottom of the screen are each players' cards, which they click with the mice to deploy.

There are a few troops on the battlefield, including mages shooting fireballs (the orange and yellow balls) and a bat flying over the moat.

Our implementation meets our qualitative goals:

- It runs smooth animations at 60 FPS (most obvious in the fluid mouse movement).
- It has the major Clash Royale gameplay mechanics, which involve nontrivial game logic: troop deployment (requiring the player to click and deplete elixir), automatic troop attack and movement behavior post-deployment, etc.
- It can simultaneously render 64 sprites, more than enough for the purposes of our Clash Royale implementation.

We have also met most of our stretch goals.

- We have implemented background details like the moat (and green bridges for troops to cross the moat) and a designated area where the player selects cards to play (on the top and bottom of the screen in the photo).
- We implemented more intricate details of Clash Royale, like having troops with different abilities, including air troops.

We did not implement more than 4 cards because the frame data took up too much memory on the FPGA. This would be possible if we attached more external memory.

Lastly, one of our goals in our abstract was flexibility for modifying the game. Since the processor only has a few requirements in software (frame data placement in the sprite file) implementing more functionality in Clash Royale, or even

Instruction	Description
LI rd const	load const to reg rd
JMP rd label	jump to label
JAL rd label	jump to label and put pc in reg rd
JALR rd const(rs1)	jump to addr rs1+const and rd=pc+4
BEQ rs1 rs2 label	jump to label if rs1=rs2
BNE rs1 rs2 label	jump to label if rs1!=rs2
BLT rs1 rs2 label	jump to label if rs1¡rs2
BGE rs1 rs2 label	jump to label if rs1¿=rs2
LW rd offset(addr)	load mem[offset+addr] into rd
SW rs1 offset(addr)	save rs1 to mem[offset+addr]
ADDI rd rs1 const	rd=rs1+const
SUBI rd rs1 const	rd=rs1−const
SLLI rd rs1 const	rd= rs1≪const
SRLI rd rs1 const	rd= rs1≫const
ADD rd rs1 rs2	rd=rs1+rs2
SUB rd rs1 rs2	rd=rs1-rs2 (unsigned)
SLL rd rs1 rs2	rd=rs1≪rs2 (unsigned)
SRL rd rs1 rs2	rd=rs1≫rs2 (unsigned)
ABS rd rs1 rs2	rd = |rs1 − rs2|
SPLI spd imm ind	load imm to sprite spd at index ind
LISP sp1 rsd ind	load the contents of sprite sp1 index ind into register rsd
SPLREG spd rs1 ind	load the contents of rs1 to sprite spd index ind
SPADDI spd rs1 imm	load rs1+imm into sprite spd index ind
SPSUBI spd rs1 imm	load rs1-imm into sprite spd index ind
SPADD spd rs1 imm	increment sprite spd index ind by rs1
SPSUB spd rs1 imm	decrement sprite spd index ind by rs1
ADDSP sp1 rsd imm	increment rsd by sprite sp1 index ind
SUBSP sp1 rsd imm	decrement rsd by sprite sp1 index ind
LW spd offset(rs1)	ind load memory address offset+rs1 to sprite spd index ind
SW spi offset(rs1)	store sprite index ind's contents to address offset+rs1
ATTACK sp1 sp2 ind1 ind2	decrement sp1 index ind1 by sp2 index ind2
DST rd, rs1, rs2	rd=Manhattan distance between rs1 and rs2.
WAIT imm	waits imm number of clock cycles.

Fig. 5. DSL Instructions

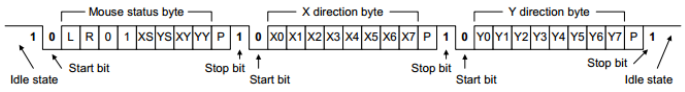

Fig. 6. Packets sent by the PS/2 mouse. Note L indicates left-click button status, and XS and YS are sign bits for the X and Y direction bytes.

constructing an entirely different sprite-based 2D game, is straightforward. This would only require changes in software (in the DSL code), not hardware or the design of the system.

If we were to redo the project, we would do two things: draw the sprites smaller (say, 32×32) and implement a pipelined processor. Our first processor was actually pipelined and only have 4 stages, but due to the sprite file, the processor's build used up all the LUTs on the FPGA. We simplified by removing the pipelines and also adding another stage to the processor to reduce the congestion of wires. We believe that re-adding the pipelines could work, because the addition of the new stage actually reduced the LUT usage by a lot.

The code can be found in the repository here.

VI. Acknowledgments

The work for this project can be roughly broken down as follows: Maxwell worked on the design and implementation of the graphics module and PS/2 mouse input handling. He also drew the spritesheet. Frederick worked on the design and implementation of the DSL (and the assembler), the processor, and the game logic written in the DSL. We coordinated to implement the interface between the processor and the graphics modules.

In terms of writing, we each wrote up what we worked on. Maxwell wrote the abstract and introduction. We collaborated on results and evaluations.

We would also like to thank Kosi Nwabueze, another student and friend in 6.205, for helping us use UART to flash our game logic program to our FPGA without having to rebuild our entire system each time we changed our game logic. This

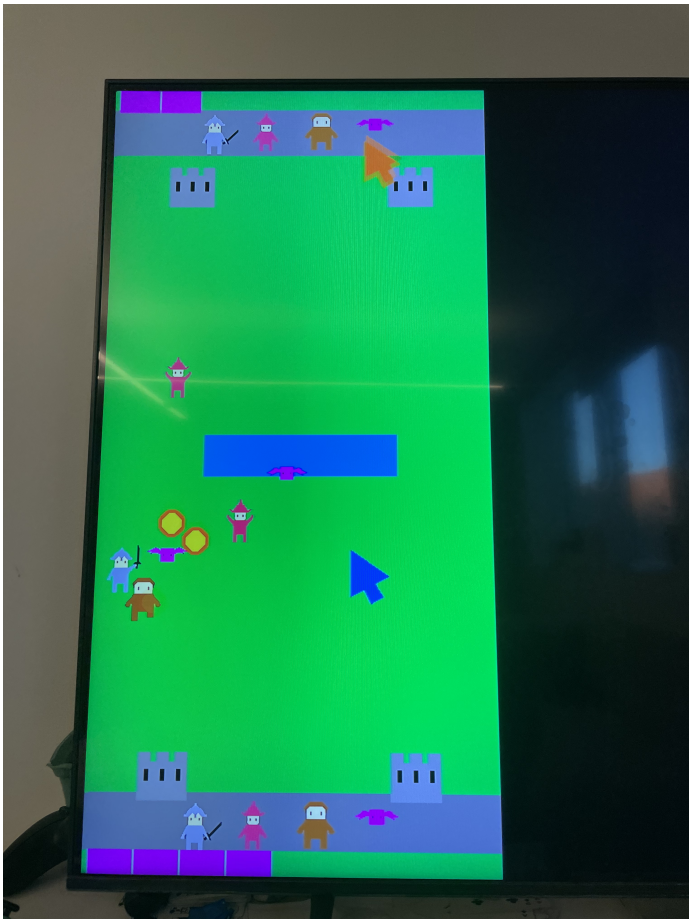

Fig. 7. FGPA Royale Gameplay

sped up development and testing, although we did not need UART in the final product.

Of course, a big thanks goes to Joe Steinmeyer and Adrianna Wojtyna for providing guidance for this project, and also to the 6.205 course staff at large for enabling this experience.