Taylor Simons

Joseph DeVictoria

ECEN 427 Embedded Systems

October 17, 2014

**Chapter 2: Game Console and Engine**

**Section 2.1: Game Console**

**Subsection 2.1.a: Digilent ATLYS Board**

The platform that was chosen to host Victory Game’s rendition of Space Invaders is an FPGA development board manufactured by Digilent Inc. called the “Atlys Spartan-6”. The central component on this build is the Xilinx Spartan-6 Field Programmable Gate Array which will be described in more detail in the following section. The Atlys development board integrates several key components that make it very attractive to the modern embedded system engineer.

The board hosts the following important components:

* 128 Mb DDR2 Memory.
* Gigabit Ethernet PHY connection.
* Five USB 2.0 serial ports (two micro-USB, and three full-size USB).
* Integrated USB-UART and USB-HID (mouse/keyboard).
* Four HDMI video ports (two input and two output).
* AC-97 audio codec with all four standard audio connections (line-in, line-out, mic, and headphone).
* Integrated power monitoring for all power rails.
* 16 Mb x4 SPI flash memory.
* 100 MHz CMOS oscillating system clock.
* 48 parallel I/O pins connected to expansion connectors.
* Eight integrated LEDs.
* Five user buttons (Up, Down, Left, Center, and Right).

For our application, the main components that are used are an HDMI out port, the five GPIO user buttons, the AC-97 audio device, and the DDR2 RAM. All of these devices are driven by controllers implemented on the FPGA hardware. The functionality of these devices can then be configured and programmed in C code. The HDMI port is used to display the game visuals to an external monitor, the buttons allow the user to control both the flow and characters of the game, and the AC-97 interface provides sound to accompany the game.

The Spartan-6 processor is capable of supplying limited BRAM for code storage, but the capacity is very limited. Because of this reason, one of the most convenient features that was heavily used in our game was having access to 128 Mbytes of DDR2 randomly accessible memory in which to easily store our game code which was quite large. Due to the singular purpose of our design, this feature was extremely useful.

**Subsection 2.1.b: Xilinx Spartan-6 and MicroBlaze**

The FPGA that is included on the Digilent Atlys development board is called the Spartan-6 LX45. This FPGA is manufactured by Xilinx Inc. and is part of their low-cost, low-power lineup for entry level devices. One of the most common applications for this particular FPGA is education and rapid prototyping on development boards like the Atlys. The other main use for this board is for video system testing as it does well to implement video controllers. The Spartan-6 is very useful for developers as it come pre-embedded with many hard-IP units such as Block RAM, Memory interface blocks, a 1080 MHz clock and a very flexible I/O controller.

One of the most attractive features of the Spartan-6 FPGA is its capability to host embedded applications through instantiating a Xilinx MicroBlaze soft-core embedded processor in the programmable portion of the device. The MicroBlaze is a 32-bit CPU with a variable three or five stage pipeline, capable of running a wide range of modern applications. The MicroBlaze comes with a host of libraries and IP blocks that allow the embedded programmer to develop a wide range of programs written in C. These programs can be compiled from source through standard gcc specifically for the MicroBlaze system. After a brief introduction and some basic knowledge of the Xilinx libraries, an embedded systems engineer can instantiate the device and run his or her first piece of code in a matter of minutes.

The MicroBlaze is a key component to the success of Victory Games’ rendition of Space Invaders as it performed the majority of processing needed to render and update the game. Although the clock speeds of our Spartan-6 MicroBlaze may be a bit slow compared to competing embedded processors, the flexibility and ease of learning the Xilinx SDK combined with the relatively simple architecture of the MicroBlaze and the simple nature of our program led this FPGA and microprocessor to be a very fair choice for our game.

**Subsection 2.1.c: System Organization**

As the entire scope of our embedded system spans both hardware and software implementations, it is important to understand how these two pieces interact. Essentially the processor on which we actually run our game is composed of a hardware intellectual property that is instantiated within the LUTs/slices of the FPGA. The main pieces of IP that were used in our design were included the Microblaze system (With its included Block Ram and memory controller), but had to be enabled and configured correctly.

One of the most important devices that we had to enable for our design was the HDMI out controller. This controller had to be initialized with the correct parameters to output a 640x480 display over HDMI. The image that is displayed on the screen is directly connected to a frame buffer that is instantiated within the memory system of the FPGA. This frame buffer consists of a 480x640 array of integers representing the color that is meant to be displayed on the screen. By writing and reading to this output frame buffer, we are able to use our application code to decide what to display on the screen. In addition to the HDMI controller, we also implemented a controller to drive the five GPIO buttons that are wired on our Atlys board. The entire GPIO device is read as a single integer in the memory system of the MicroBlaze, with different values indicating different buttons have been pressed. Although these buttons can be registered to a interrupt handler, we decided that the overhead for simply polling the buttons was small enough that its simplicity and ease of design warranted such an approach.

Another important piece of IP that was heavily used during the debugging portion of our design was the USB-UART device that let us communicate over USB to a host computer console via std in and out mapping. This piece of hardware was created like all the other hardware IPs during the initial building of the MicroBlaze system. After creating the hardware block, it could be further configured in software. Although this was an extremely useful tool to use during the initial steps of our design, we found that we used it less and less as the game progressed. In the end we wanted all user interface to take place through the on-board buttons and the output display.

After all of these useful tools were implemented, the bulk of the software portion of our embedded system was actually stored outside of the FPGA in DDR2 RAM. The large size of this memory made it ideal for storing the somewhat complex logic of our game as well as the bitmaps for every game object that can be seen on screen. Using the DDR2 memory as storage for the code allowed us great freedom in the software architecture that we chose to pursue.

**Section 2.2: Game Engine**

**Subsection 2.2.a: Game Engine (main game loop)**

This specific design of space is driven principally by a single timer interrupt. This interrupt is used in determining the timing of all components of the game, everything from the rate at which the buttons are polled to the rate that the rate that objects on the screen are updated and refreshed. It is set to go off every 1,000,000 clock cycles. In the main loop, after system initialization, the program enters an infinite loop which idles the system between timer interrupts. Once the timer goes off, a list of functions are called. These functions compose the bulk of the game engine.

Aside from the initialization the functions called by the timer interrupt include the functions used for polling the buttons, updating bullet positions and updating alien positions. All of these functions start by checking to see if enough time has passed to match its specified rate of execution. From here the function used to poll buttons will calculate the required adjustments to the game data and then re-update the screen if needed. For the alien and bullet/missile update functions, a control function is used to determine how game data needs to be changed, and then render function will be called that use bitmaps and the current frame buffer to update the display. Collisions are also calculated in these functions as well.

When calculating collisions, the game engine uses the current pixel color right in front of the bullets/missiles. For alien missiles, if green pixels are detected, then either a bunker or tank was hit. For tank missiles there is a unique off color white that is used for alien missiles/debris that is different from the white used to display the aliens. This way the collisions detected by the bullets can determine if it is an alien rather than debris or a missile. After any collision, functions are called to manipulate game data and the parts that were hit are then re-rendered.

Other special case features included in the game engine are the next level function, initialization and games status logic. Along with game data, the control functions used to determine the changes that need to be made to game data and the objects displayed, we implemented a set of status states that let the program know what is allowed to be changed. These states are STOPPED,RUNNING, and GAME\_OVER. With this structure we can ensure that gameplay stops when needed and then can be resumed flawlessly after delays have been removed.

By using a set of render and control function within the timer driven functions, implementing and expanding Space Invaders became a straightforward task. By examining the list of functions that were used in the timer handler it is an easy task to determine what and when the system will be required to run.

**Subsection 2.2.b: Meeting the Game Specifications**

An example of Space Invaders game implemented in with flash became the main template from which we based the gameplay and flow of the space invaders game. Details of the proportions, speed, control and game rules for our system were pieced together as we played though and experimented with this flash version of the game. In order to see the different animations that were used for the bullets/missiles, the game screen had to be paused. While determining the control of the alien blocks and rate at which they marched, the flash game was run alongside our implementations in order to compare. While proportions could be approximated or even ignored, we aimed to get as close to the flash version as possible. This meant that a large margin or border had to be implemented around the game area. One of the more undetectable aspects of the flash game was the animation of the bullets/missile. This included a gradual change of bit map guise to reduce flashing as they were updated.

Aside from the display aspects that were replicated, there were many details of game play that were deciphered and implemented. These included movement by the player, score calculation, game flow and initialization.

**Section 2.3: Application Programming Interfaces**

Initially our design focused on a very rigid case based system of determining what event had happened and what result should take place in response to that event. This system was functionally sound, yet severely limited our ability as programmers to troubleshoot and develop the game. As time and development progressed, we moved our design from a very un-organized two function system (control() and render()) to a much more fluid and dynamic programming interface.

We broke our code down into discrete steps (implemented as functions) that happened in the execution of our game engine. These steps included interrupt handlers, a game flow control function responsible for calling the correct update functions for each scenario, update functions, and a variety of helper functions to cut down on code repetition and make the program smaller. Our final product resulted in much cleaner and more readable code, following the mantra of “Don’t Repeat Yourself.”

**Timing and Memory Report**

The Victory Games version of Space Invaders has the following CPU usage characteristics:

* Most time for single game loop: 1,000,000 cycles (one fit timer)
* Best case CPU usage: 93.33%
* Worst case CPU usage: 95.24%
* Average case CPU usage: 94.14%

These performance characteristics were calculated by taking a ratio of active program cycles to idle cycles. The active cycles were measured by the total number of fit timers that go off minus one million idle cycles (equal to one fit timer). As the number of fit timers per million idle cycles increased, our CPU usage went up, when the processor was not heavily tasked, more idle tasks (a counter in the while(1) loop) would run for every active cycle.

The most time that we ever spent running a single loop of our main routine was one fit\_counter which is equal to 1,000,000 instructions. The maximum CPU usage that we observed was 20,000,000 active program instructions (20 fit timers) for every 1,000,000 idle instructions, giving us a worst case CPU usage of 95.2381%. The best case scenario for CPU usage was 14,000,000 active program instructions (14 fit timers) for every 1,000,000 idle instructions, giving us a best case CPU usage percentage of 93.3333%. The average CPU usage taken from a window of 1,160,000,000 instructions under normal-heavy use was 94.1379% (1,092,000,000 active / 1,160,000,000 total instructions).

Our program had a fair amount of code space due to the optimization levels under which we compiled, and the depth, robustness and and complexity of our code. The size of our resulting code:

* Text: 45402 = ~45.4kB
* Data: 4408 = ~4.4kB
* Bss: 68374 = ~68.4kB
* Total Size (Decimal): 118184 = ~118.2kB
* Total Size (Hex): 0x1CDA8 = ~118.2kB

**Bug Report**

During the course of our development we encountered a wide variety of bugs that we had no anticipated. Some of the bugs were cosmetic, such a bullets being erased when they shouldn’t, or drawing the wrong color when erased, other bugs were more subtle, such as collision detection and the random spawning to alien bullets. Because we as developers were set on implementing the best possible game that we could, we spent extra time to hunt down and correct every bug that we encountered as our game was designed. Below we will list the major bugs that we had to fix as we worked through our project:

* The algorithm designed to move the aliens from position to position did not initially erase the previous alien positions correctly and some tracing or a black box were apparent.
* Our original code was very slow and jittery upon first build, and it was not until we enabled compiler optimization that we got up to the performance level that we had expected.
* Bullets would collide with bunkers and erase a portion of the bunker below them in the shape of the bullet. This was corrected by changing the erase distance to erase the bullet the moment that it detected that it was about to hit the bunker.
* Missiles were cosmetically ugly and we had to add additional guises to smooth out their motion.
* Game control flow initially allowed the user to move the tank after the game over state had been reached. We had to add in additional logic that force the system not to update any game objects when this state had been reached.
* Occasionally a missile would hit a bunker and the bunker next to it would take damage. This was corrected by changing the way in which the bullets detected a collision.
* The Mothership did not initially move smoothly off of the edge of the screen. Additional logic was implemented to make this transition as smooth as butter.
* A variety of similar cosmetic bugs were encountered during the course of our development, but many of them we were able to correct after a short bout of troubleshooting.