Spencer Carter

Tavana Alailima

Chapter 2: Game Console and Engine

Section 2.1: Game Console

Subsection 2.1.a: Digilent ATLYS Board

The Digilent Atlys board is a wonderful candidate for our embedded system processor project. It hosts a wide range of on board peripherals including USB and audio ports as well as a 16 bit DDR2 memory. Of the many peripherals offered, the Space Invaders project would make use of the memory, push buttons, and interval timer peripherals. The Atlys board is compatible with all Xilinx provided CAD tools, thus, all of our software development for the board was able to be completed within the Xilinx SDK.

Subsection 2.1.b Xilinx Spartan-6 and Microblaze

The Spartan-6 FPGA family from Xilinx offers 45nm technology optimized for low power consumption. The dynamic power utilization, including hibernate and suspend mode, allow for minimum power consumption. The Spartan-6 also includes integrated memory controller blocks in the form of DDR as well as high speed serial transceivers. The Microblaze is a configurable soft processor that can be placed on the FPGA. It supports more than 70 different configuration options. Because of the Microblaze’s high degree of configurability, the user can implement a processor tailored specifically to the needs of any project. The Microblaze can be configured to operate in either a 3 stage pipeline mode or a 5 stage pipeline mode which respectively optimize for size and speed.

Subsection 2.1.c System Organization

The first step was to download the new provided hardware given. The next step was to unzip the files. After that the provided EDK project was opened, and then the Atlys\_AXI\_BSB\_Support/lib directory was selected to solve the error. The next step was to “Export Design” to the SDK, and create a new C project in a different directory. The last thing to do was generate a linker script, and change the stack size to 4KB, and place the code sections in DDR memory.

Section 2.2 Game Engine

Subsection 2.2.a Game Engine (Main Game Loop)

The Space Invaders project became very large and complicated and thus required a high level of top level organization during the software development process. The Space Invaders game is a collection of multiple different parts, much of which are required to have communication with each other in some form or another. For example the bullets must know the positions of the bunkers, aliens, and tank. The score board must receive information from the aliens about what any updates to the score and what numbers to show, and the lives at the top must know when the tank is hit. These examples illustrate just a small portion of the degree of complexity of this game. In order to maintain organized flow of the development of this game, each constituent part was isolated and separated from all other parts of the game. For each of these different parts, a unique header and source file were created. Thus a header and source file were created solely for tank bullets, for the tank, bunkers and score, to name a few. By thus breaking the game into as small of parts as possible, development and debugging was far simpler and efficient.

Many of these source files representing different aspects of the game were then required to communicate with each other. In order to accomplish this task, setter and getter functions for any important variables or pieces of information required to be seen by other parts of the project were created and placed at the top of each respective source file. This made identification and retrieval of necessary information very easy. A globals header file was also created and stored constants that would be needed throughout the project. This eliminated the need to create repetitive constants in each source file for the same variables.

All of the source files were brought together in a main source file where the interrupt handlers for the FIT and push buttons peripherals were also placed. The main function contained all necessary initialization of the board and blanking and drawing of the screen. There was also a function for the interrupt handler dispatcher that would call either the push button handler function or the timer handler function depending on which interrupt it received. The majority of the interrupt handling code was placed in the timer interrupt handler function since most of the project was based on interval timing. Every time the timer interrupt handler was called due to an interrupt firing from the FIT every 10ms, there were counters placed inside the handler that would increment and upon reaching a final value would initiate some functionality such as updating alien position. Much of the game’s functionality was based on the use of these counters incrementing every 10ms.

Subsection 2.2.b Meeting the Game Specifications

One aspect of the game’s functionality that proved challenging was the bunker erosion. This required much thought because a single erosion required three pieces of information: bunker number, erosion section and erosion state. Checking all three of these pieces of information required the use of many clever algorithms. The erosion states for all of the sections of the different bunkers were stored in a 2 dimensional matrix. This allowed the erosion state to be saved at a location in the matrix corresponding to bunker number (1st dimension) and erosion section of that bunker (2nd dimension). The bunker number and erosion section information also had to be obtained. These were checked against the position of both the alien and tank bullets to identify where erosion would need to occur. Bunker number identification was made possible by use of a switch statement. Erosion section was also made possible by use of a switch statement, however rather than having 40 different switch cases, only 10 were used since the known bunker number was cleverly used as a multiplier for the x positioning of the erosion section. Thus the 10 different erosion sections for the far left bunker were identified, and then the x positioning of that erosion section was multiplied by a factor pertaining to the bunker number to arrive at the correct bunker.

One other challenging aspect of the game’s implementation was the printing of the score to the game console. This was a challenge because this meant repeatedly printing a dynamic number. Thus 10 sprites were made for digits 0-9 and interpretation had to be done on the variable containing the current score as to what digits would be necessary to print. A switch statement covered this implementation with 10 cases, one for each of the possible digits needed to be printed as a sprite. It became more challenging, upon realization that leading zeros would need to be ignored and the entire score number shifted left appropriately to account for any ignored leading zeros. The implementation of this functionality required nested for loops that would simultaneously check what digit was being interpreted and the x positioning of the overall score. With these algorithms, successful printing of a dynamic score with no leading zeros was accomplished.

Section 2.3 Application Programming Interfaces

The project utilizes multiple API’s that were created specifically for this project. As mentioned in the Main Game Loop section, function allowing interface to other source files were necessary so that access to information contained in other sections of the lab would be possible. In most of the source files can be found setter and getter functions as well functions such as update position and get position. These all functioned as programming interfaces to other parts of the project. These were the only programming interfaces that were used. None were used that weren’t made specifically for this project.

Timing and Memory Report

The CPU usage was calculated for the Space Invaders project. In order to calculate this number a counter was placed in a while loop in the main function and the program was run with interrupts enabled. After 10 seconds had passed the counter value was printed to the screen. The interrupts were then disabled and the process was repeated to get a new counter value while interrupts were disabled. These numbers represent the idle time of the program. The number with no interrupts was larger because there was less idle time. In order to calculate CPU usage, the difference between these two numbers was found and then divided by the number retrieved from running the program with no interrupts. Our CPU usage was calculated to be 8.2%.

The program consumes a total of 60776 bytes of memory. The elf file printout is included below which distinguishes the allocation of the different types of memory. As can be seen text consumes the most memory, then bss and data consumes the least amount of memory of the three.

text data bss dec hex filename

53770 1544 5462 60776 ed68 SpaceInvaders.elf

Bug Report

Throughout the course of the project, on multiple occasions, the use of fixed bit ints such as uint8\_t led to confusion simply because attempts were being made to store numbers larger than the allowable size in that case. In one instance in particular, the mystery score of the saucer needed to be printed at the location of the saucer. The y position was relatively small, only 45 pixels or so, thus uint8\_t was capable of storing this number. The variable used to hold the x position was also set as a uint8\_t and this led to problems since the x position needed to range from 0 to 640, much higher than the allowable 255 for 8 bits. In this instance it was especially difficult to identify this as the problem since the x and y coordinates for the position of the mystery score print location were passed through multiple functions and across multiple source files. In all but one of these instances the x variable was set as 16 bit, but in one single instance the x variable was passed in as 8 bits and this led to the above mentioned problems. In hindsight, the problem should have been identified quicker had it been noticed that the mystery score was being printed only on the left half of the screen and that the x position variable was not attaining values higher than about half the width of the screen, 320, which is very close to the max number capable of being held by 8 bits.