

EMBEDDED SPACE INVADERS GUIDE

Produced by
Luke Hsiao & Jeff Ravert
Revision 1.3
8 December 2014

SPACE INVADERS GUIDE

TABLE OF CONTENTS

- Chapter 1: Space Invaders Overview.....2**
 - Section 1.1: Space Invaders History2
 - Section 1.2: Game Play2
 - Section 1.3: Game Details and Specifications4
- Chapter 2: Game Console and Engine6**
 - Section 2.1: Game Console6
 - Subsection 2.1.a: Digilent ATLYS Board6
 - Subsection 2.1.b: Xilinx Spartan-6 and MicroBlaze6
 - Subsection 2.1.c: System Organization7
 - Section 2.2: Game Engine7
 - Subsection 2.2.a: Game Engine (Main Game Loop)7
 - Subsection 2.2.b: Meeting the Game Specifications10
 - Section 2.3: Application Programming Interfaces13
- Chapter 3: Game Audio15**
 - Section 3.1: WAV Files and the AC’9715
 - Subsection 3.1.a: WAV File Conversion15
 - Subsection 3.1.b: AC’97 Operation16
 - Subsection 3.1.c: Sound Triggering16
- Chapter 4: DMA Controller17**
 - Section 4.1: Driver API18
 - Section 4.2: Performance Analysis18
- Appendix A: Timing & Memory Report20**
- Appendix B: Bug Report21**
- References.....22**

CHAPTER 1: SPACE INVADERS OVERVIEW

Section 1.1: Space Invaders History

Space Invaders, synonymous with classic arcade game culture, was first released in 1978. Its pixelated aliens and chromatic sound loops have become pop-culture icons referencing a simpler age of gaming. Its gameplay have been cloned, mimicked, and remade hundreds of times. It was one of the earliest shooting games and a forerunner of modern video gaming. Its impact helped push video gaming into a global industry.

Space Invaders was originally launched in Japan by Taito, a Japanese publisher of video game software and arcade hardware now owned by Square Enixⁱ. Tomohiro Nishikado, the game's designer, drew ideas from popular science fiction media from the time like *The War of the Worlds* and *Star Wars* and from popular level-based games like *Breakout*. Unlike the creation of modern games, Nishikado created Space Invaders singlehandedly over the course of a year. He not only designed and programmed the game, but also created artwork, sounds, and arcade hardware to support it.

At its inception, Space Invaders was run on a custom-made arcade board that featured an Intel 8080 CPU, raster graphics on a CRT monitor, and monaural sound created by analog circuitry and a Texas Instruments SN76477 sound chip. Nishikado found that despite his best efforts, the CPU lacked the power to display more colors or move enemies faster. He also discovered that the CPU was able to render graphics faster the fewer aliens were on the screen. Today, the challenge of progressively faster enemies is a gameplay mechanic used prolifically in the industry.

By the end of 1978, Taito had installed over 100,000 machines and grossed over \$600 million in Japanⁱⁱ. In two more years, Taito had sold over 300,000 arcade machines in Japan with an additional 60,000 in the United Statesⁱⁱⁱ. By 1982, Space Invaders grossed \$2 billion in quarters^{iv}. Overall, Space Invaders has been a significantly influential piece of hardware and software, and this manual will explain yet another rendition on a modern FPGA.

Section 1.2: Game Play

Space Invaders is a two-dimensional shooting game. The player can maneuver a tank, positioned at the bottom of the screen, to the left or to the right, and fire bullets at the descending block of aliens. The objective is to eliminate the five rows of eleven aliens before they reach the bottom of the screen. The aliens simply move horizontally left and right on the screen, and drop vertically each time they hit the edge of the screen. As more aliens are killed, the entire block moves more quickly. If the entire block is destroyed, a new block will appear. Players earn points each time an alien or spaceship is killed.

Aliens attack the player by firing bullets downwards while they approach the bottom of the screen. The player can either dodge the bullets, or seek cover behind one of the four stationary, destructible bunkers. If the aliens reach the bottom, the game ends. Bonus points can be acquired by destroying the spaceship which occasionally appears at the top of the screen. Score is kept track at the top left of the screen. The number of lives the player has is tracked at the top right of the screen.

For a summary of each of the game's units and components:







Unit Icon	Unit Description
	[40 points] There are only 11 of these aliens in the back row of the alien block. These are the alien elite which have earned the privilege of being away from the front lines.
	[20 points] There are 22 of these aliens. These are the middle class and form the middle two rows of the alien block.
	[10 points] These are the alien grunts. 22 of them form the front two lines. This alien, along with the other two are animated in the game by toggling between two appearances.
	[50-350 points] The spaceship. Occasionally, the space invaders' spaceship itself will move through the top of the screen. It is worth significant bonus points if the tank can get a precisely aimed shot past the aliens and take it out!
	There are four stationary, destructible bunkers that the tank can use as a shield from alien bullets. However, any time the tank bullet or the alien bullets hit the bunker it will begin to be destroyed, so watch out!
	The tank. The player is a tank is equipped with a top-of-the-line cannon that has unlimited ammo. This unit is earth's last line of defense from the space invaders and has to make sure to wipe all those aliens out! The game starts with three additional lives, but it only takes one hit by an alien bullet to destroy the tank.

Table 1: Space Invaders Unit Summary

Section 1.3: Game Details and Specifications

Below are the details of the game that must be implemented for the game to play correctly. This guide discusses the tank, bunkers, aliens, and bullets individually.

Gameplay

First, the game must run smoothly. This not only means the game must not stall or freeze, but there must be no flickering or visual artifacts left on the screen.

The game ends and “GAME OVER” is displayed if (1) the player is killed and has no more extra lives or (2) the alien block drops below the bottom of the bunkers. That is, if the lowest live alien drops vertically lower than the bottom border of the bunkers.

The level resets if the player kills all of the aliens, and the score carries over to allow the player to pursue a high score and the player’s extra lives are replenished. The player can push any of the five pushbuttons to restart the game.

Tank

The tank is controlled by the player using the buttons on the ATLYS board. It is able to move and fire at the same time, and only has one bullet on the screen at a time. The tank’s movement is limited to horizontal movement only, and it can only move from the left edge of the screen to the right. If the bullet hits an alien or spaceship it is killed. If tank bullets hit the bunkers, the bunker will also be eroded.

When a tank is hit, a death animation is shown by toggling between two bitmaps of an exploding tank. Then, a life is subtracted and the tank’s position is reset. If the tank is hit and has no extra lives remaining, the game ends.

Bunkers

The four stationary, destructible bunkers are at the bottom of the screen. They are built up of 12 sub-blocks and erode when hit by either the tank bullet or alien bullets, until finally they allow bullets to pass through. The aliens drop down on top of the bunkers, and if they drop low enough to pass the bottom of the bunkers, the game ends.

Aliens

There are 55 aliens (5 rows of 11 aliens) in the alien block. Each alien has two *guises* or forms so that the game can simulate movement by toggling between them. All of the alien guises are shown in Figure 1.

Alien 1		Alien 2		Alien 3		Death
						

Figure 1: Alien Guises

Aliens explode with the death guise when hit by the tank bullet. If an entire outer column of aliens is killed, the alien block still appears to move edge-to-edge. For example, if only one column of aliens remains, that single column will still move horizontally across the screen until it hits the edge of the screen.

Aliens can have a maximum of four bullets in flight at a time. Bullets are fired from a randomly selected alien that is in the lowest position of its column. An alien that is on top of another will never fire through the lower alien in a given column.

Aliens slowly speed up as more aliens are killed, and they also speed up as they get closer to the bottom of the screen. When killed, the appropriate amount of points are added and displayed to the player.

When aliens are rendered on the bunkers, they are printed in the foreground over the bunkers, and no flickering occurs. There is no black box rendered around the aliens.

Spaceship

Occasionally throughout the game a red spaceship will appear at the top of the screen. The purpose of this spaceship is for the player to get bonus points. The points that this spaceship is worth is a random multiple of fifty between 50 and 350. Unlike the aliens that bounce on the edges of the screen, the spaceship will fly off of the edge, giving the player only a small window of opportunity to score the extra points.

When killed, the alien spaceship score will be flashed where the spaceship was when it was killed.

Score



Figure 2: Score Display

Score is displayed on the top left of the screen as shown in Figure 2. The number is updated each time an alien or spaceship is killed. Aliens and the spaceship are worth the amount of points shown in Table 1.

Lives



Figure 3: Lives Display

Lives are shown on the top right hand side of the screen by using the same tank bitmap as shown in Figure 3 above. When a tank is hit, its death animation will play, and one of the life icons will disappear. When there are no remaining extra lives, the game is over.

Note that this implementation of Space Invaders mimics the gameplay and appearance of the online flash version found at www.freeinvaders.org.

CHAPTER 2: GAME CONSOLE AND ENGINE

Section 2.1: Game Console

Subsection 2.1.a: Digilent ATLYS Board

This version of Space Invaders was designed and developed for use on the ATLYS™ Spartan-6 FPGA Development Board. This board was developed by Digilent and houses a Xilinx Spartan-6 FPGA. Its features are shown in Table 2 below:

IC:	Xilinx Spartan-6 LX45 FPGA in a 324-pin BGA package
Connectors:	One Vmod™ (high-speed VHDC) connector One 12-pin Pmod™ connector One RJ-45 connector for 10/100/1000 Ethernet PHY and RS-232 serial Two HDMI video input ports & two HDMI output ports Two on-board USB2 ports for programming & data transfer AC-97 audio with line-in, line-out, mic, & headphone
Programming:	JTAG programming interfaces compatible with Xilinx's iMPACT™ and Digilent Adept™
Memory:	128Mbyte DDR2 16-bit wide data 16Mbyte x4 SPI Flash for configuration & data storage
CMOS Oscillator:	100 MHz
Ethernet:	10/100/1000 Ethernet PHY
Power:	Real time power monitors on all power rails Ships with a 20W power supply and USB cable
I/O:	48 I/O's routed to expansion connectors GPIO includes 8 LEDs, 6 buttons, & 8 slide switches

Table 2: ATLYS Spartan-6 Development Board Features

After being programmed with the hardware described in section 2.1.c and loaded with the Space Invaders software, this board contains all the I/O and resources to run the game standalone.

Subsection 2.1.b: Xilinx Spartan-6 and MicroBlaze

The Xilinx Spartan-6 LX45 FPGA included in our development board contains integrated hard memory. It is built using a 45nm, low-power copper process utilizing 6-input LUTs. The LX series are optimized for applications that require the lowest costs. The Spartan-6 LX supports up to 147K logic cell density, 4.8Mb memory, integrated memory controllers, DSP slices, and high-performance Hard IP. Xilinx markets the Spartan-6 LX FPGA as ideal for meeting low-cost, high-volume applications where reduced power consumption is important. It contains 1080MHz clock management tiles, and 320MHz Block RAMs.

To utilize the features of the Spartan-6, a soft-core processor runs this version of Space Invaders. A soft-core processor is a processor that is implemented in FPGA logic, rather than cast directly in silicon. Specifically, Space Invaders uses Xilinx's proprietary MicroBlaze IP. The MicroBlaze contains over 70 user-configurable options. It can operate in a 3-stage pipeline to optimize size, or a 5-stage pipeline to optimize speed. It also has advanced architecture options like AXI or PLB interface, Memory Management Unit, instruction and data cache, Floating-Point Unit, and more. It is a 32-bit RISC Harvard architecture processor.

Subsection 2.1.c: System Organization

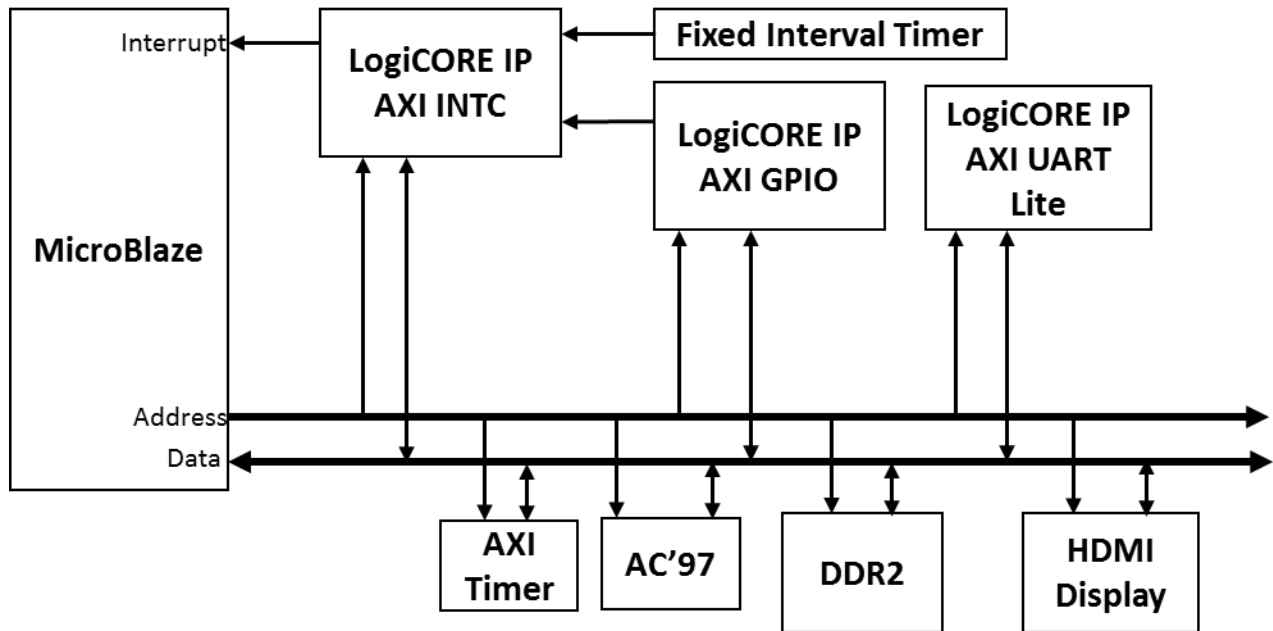


Figure 4: Block Diagram of System Organization

Figure 4 shows a block diagram of the hardware system. The Xilinx MicroBlaze connected to an interrupt controller and utilizes the AXI bus to communicate with other system components. A Fixed Interval Timer (FIT) is used to provide an interrupt every 10ms, which drives all the game's animations. The General Purpose Input Output (GPIO) IP provides interrupts corresponding to the push buttons on the ATLYS development board. However, rather than using these interrupts, Space Invaders simply polls the buttons at regular intervals for simplicity. The AXI Timer makes it possible to measure the time it takes for portions of code to execute, and can be used to estimate the CPU utilization. The UART is used to communicate (e.g. debugging printouts) with the PC.

Game graphics are displayed by writing data to frame buffers located in DDR2 memory. This memory is then being constantly read from and displayed on the HDMI monitor connected to the ATLYS board at the monitor's refresh rate.

Section 2.2: Game Engine

Subsection 2.2.a: Game Engine (Main Game Loop)

The main Space Invaders file simply sets up the hardware, initializes the state machines, and waits indefinitely in an empty `while(1)` loop. After this initialization, the game is completely interrupt driven.

All of the game's visual updates are based upon FIT timer interrupts. This simplifies the design and the state machines by allowing the software to simply poll the buttons at fixed intervals rather than deal with the complexities of responding to button interrupts and debouncing.

When a FIT interrupt is detected the following state machines may be called. These state machines are illustrated on a high-level on the following page in Table 3.

Tank Movement and Bullet

TankMovementAndBullet_SM cycles every time an interrupt occurs. When the tank has been hit this state machine transitions to a dead state and calls the respective bitmaps to animate the tanks death. When the tank is alive this state machine polls the buttons and fires a tank bullet if the center button is pressed. It also moves the tank left or right when left or right button is pressed. The user can fire a tank bullet while moving left or right because the center button doesn't cause a state transition. The tank moves smoothly because this state machine is called every cycle and only moves the tank a few pixels.

Tank Bullet Update

TankBulletUpdate_SM cycles every 5 interrupts because the tank bullet does not need to move seamlessly smooth because it moves so fast. This state machine simply updates the tank bullet location and redraws the bullet.

Alien Movement and Bullets

AlienMovementAndBullets_SM cycles no slower than every 60 interrupts. This state machine gets called more often as the number of aliens decrease and move down the screen to meet the game specification of the increasing speed. The fastest this state machine is ever called is every cycle. The state machine also randomly fires bullets from a random alien at a programmable rate.

Alien Bullet Update

AlienbulletsUpdate_SM cycles every 5 interrupts because the alien bullets don't need to move seamlessly smooth because they move so fast. This state machine simply updates the alien bullets locations and redraws the bullet. Note that the alien bullets move fewer pixels per update than the tank bullet, making them appear to travel slower than the tank bullet.

Spaceship Movement

SpaceShipUpdate_SM cycles every time an interrupt occurs. When the spaceship has been hit this state machine transitions to a dead state and calls the respective bitmaps to flash a point value for hitting the spaceship. This state machine also causes the spaceship to appear at random intervals. The spaceship moves smoothly because this state machine is called every cycle and only moves the spaceship a few pixels.

Alien Death

AlienDeath_SM cycles at the same variable rate as AlienMovementAndBullets_SM. This state machine waits until there is an alien death. When a death occurs the state machine transitions and animates an alien death.

Each state machine also has a game over state for when the user has lost all extra lives or the aliens have dropped too low on the screen. The only way out of this state is to reinitialize the state machines. When the user presses any game button the state machines are reinitialized.

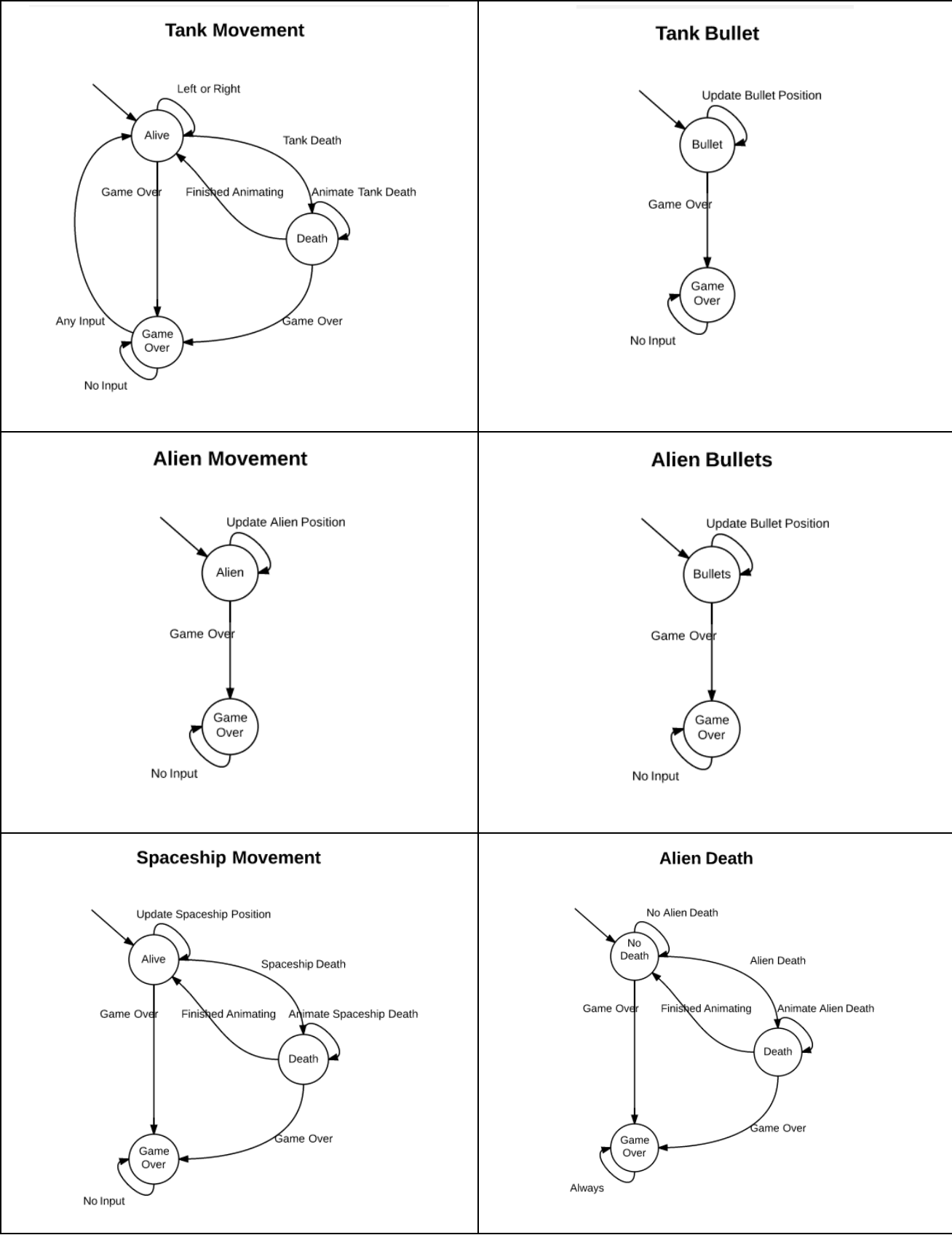


Table 3: State Machine Diagrams

Subsection 2.2.b: Meeting the Game Specifications

In order to meet the game specifications, lots of consideration went into global variables and data structures. In order to properly animate the game, software state machines were used. Note that this game was created to be run on a 640x480 resolution monitor, and all of the pixel calculations for all rendering and movement are based on this dimension.

Tank

The tank is rendered from a 32x16 bit bitmap, stored as an array of unsigned integers. For convenience, the software simply tracks and draws the tank based on an (x,y) coordinate storing the location of the tank's top-left corner. The y-value was made constant so that the tank cannot move vertically. Rather, when the user presses one of the tank's movement keys, the x-coordinate is updated and the tank is re-rendered. This means that the range of the tank's x-coordinate values ranges from 0-608.

In regards to unrendering, the software simply blacks out the entire 32x16 box that contains the tank before rendering the tank in its new location.

In order to fire the tank's bullet, the tank bullet's position is calculated based on the tank's current location and the corresponding global variables are set. Bullet mechanics are discussed more below.

Bunkers

The four stationary, destructible bunkers are made up of 12 sub-blocks (as shown in Figure 5), arranged in 3 rows of 4 blocks. The bunkers are numbered 0-11 going from left-to-right, top-to-bottom. Bunkers are made up of bitmaps for the 4 corner cases (at the top left and right as well as the middle left and right) along with a generic solid block for the rest. Each block can take 4 bullet hits before disappearing.



Figure 5: Bunker Blocks

Similarly, there are bitmaps that contain the 4 erosion patterns for the blocks. All the blocks will erode in the same pattern. These erosion bitmaps are ANDed with the bitmap that is currently being drawn. This effectively turns off the pixels that need to be turned off, without altering the pixels that were already off. Each bunker's location is tracked using global variables of (x,y) coordinates of their top-left corners.

Each block's erosion state is managed using a single unsigned integer. Because each block has five states—(1) full health, (2) hit once, (3) hit twice, (4) hit thrice, (5) destroyed—it only requires 3 bits to track each block's status. The organization of each block's status is shown in Figure 6 below. Bits 31 and 30 are unused.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
		Block 11				Block 8				Block 7				Block 6				Block 5				Block 4				Block 3				Block 2				Block 1				Block 0			

Figure 6: Bunker Block State Layout

Note that because blocks 9 and 10 are always “off” and will only display black, space didn't need to be allocated to track their status. This design choice allows the data to be compact as possible.

Aliens

In order to track, render, and unrender the aliens, the software keeps several global variables. First, it maintains the (x,y) coordinate of the top-left position of the entire alien block. Next, it saves a 16x5 bit array to track whether each alien is dead or alive. If an alien is killed, the death bitmap is shown and then the space is blanked.

When rendering, all of the alien's white pixels are drawn, and then if there are white pixels leftover from a previous render, they are erased with either black or green, depending on whether the pixels are above a bunker or not. A second frame buffer is used to store the bunker data to allow this smooth drawing.

The same numbering convention as the bunker blocks is followed, the top-left alien is 0, and incrementing going left-to-right, top-to-bottom, ending on alien 54 at the bottom-right. These two global variables are all that is needed to draw each alien at the correct location.

In order to allow the aliens to continue to bounce off the edges of the screen when entire outer columns are destroyed, there are two variables that serve as imaginary "walls". The alien block will automatically move horizontally until one of these abstract walls is hit, and then it will move down, and reverse horizontal direction. The wall variables are updated whenever an entire outer column of aliens is killed. This allows for the (x,y) coordinate of the top-left corner to actually wrap around past zero when moving to the left, but the wall variables compensate for the wrap around.

Similarly, the position of the lowest level of live aliens is tracked so that when the aliens get far enough down on the screen, the game will end.

Calculations about whether an alien was hit or not are made using the tank bullet's (x,y) coordinate, the array tracking alien's life/death status, and the (x,y) coordinate of the alien block itself.

Spaceship

The spaceship is also rendered from a 32x16 bit bitmap. It is also placed by storing a global variable containing the (x,y) coordinates of its upper left-hand corner. In order to calculate whether the spaceship is hit or not, the software simply utilizes the (x,y) coordinates to calculate whether the point in question falls within the width and height of the spaceship.

In order to accomplish the flashing points, a random number is generated between 1 and 7, and then multiplied by 50 to give random point values between 50 and 350 in multiples of 50. Then, the position that the spaceship was in when hit is saved and state machines are used to repeatedly render and unrender the point value of the spaceship before finally leaving the area blank.

The spaceship's bonus points are then added to the global score variable and the screen is updated.

Bullets

The aliens can fire two types of bullets. In order to show them as animated, each bullet also has guises as shown in Figure 7.

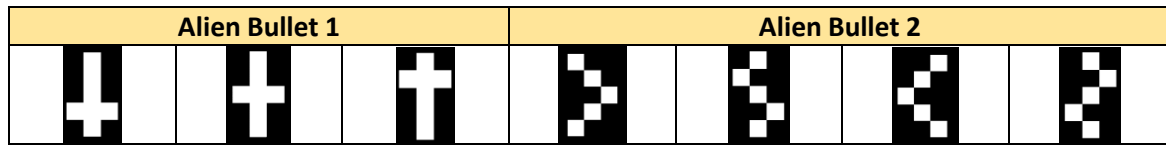


Figure 7: Alien Bullet Guises

Like the aliens, each bullet rotates through each of its guises as it moves so that it appears animated. All of the alien bullets update and move at the same rate, while the tank bullet moves at a much quicker rate.

Because the only the lowest alien of a column can shoot, a random column number is generated, then the column is iterated over from bottom to top until an alien is found. If no alien is found, then an adjacent column is selected and iterated over.

In order to coordinate all of these variables, the software stores an *alienBullet* struct. This struct contains the bullet's (x,y) coordinate, its current guise, and its type. In order to make sure that only four bullets can in flight at any given time, once a bullet hits something or flies off of the screen, its y-position is set to a number larger than 480, indicating that it is now available to be shot. If a bullet's y-location is less than 480, it indicates that it is still being drawn on the screen, and thus, still in flight.

The tank bullet is rendered in a similar way. However, since the bullet is just a simple line, no bitmap is used for rendering. The only variables that need to be associated with the tank bullet is its (x,y) location.

Score

A global variable is stored containing the player's current score. Using bitmaps for all the digits zero through nine, the score is rendered at a set location at the top left of the screen. In this version, the score will not render correctly if it exceeds four digits. However, since an entire alien block only yields 1060 points, this should not be an issue. The amount of digits printed can be expanded in a later version of the software if needed.

When the player eliminates an entire alien block, all of the game's variables are reset except for the score.

Lives

Lives are also stored as a global variable. The maximum number of lives allowed is 3, and they are drawn using the tank's bitmap. Currently, when all aliens in a level are killed, the lives simply reset to 3. When lives reaches zero, a flag is raised to bring the state machine into the game over state.

Section 2.3: Application Programming Interfaces

An Application Programming Interface (API) is a set of routines, protocols, and tools for building software and working with hardware. It specifies inputs and outputs and defines functionality. Space Invaders utilizes existing APIs to access hardware, and contains custom APIs to ease the work of programming visual components.

Space Invaders utilizes many of Xilinx's provided APIs to interface with the hardware of the ATLYS Board. For example, Space Invaders relies heavily upon the APIs shown in Figure 8, along with comments on their purpose. These APIs give access hardware components on the AXI bus, interface with interrupts, and interface with memories.

```
#include <stdio.h>
#include "platform.h"      // Needed to both initialize and cleanup our hardware platform
#include "xparameters.h"   // Defines all of the base addresses we use
#include "xaxivdma.h"      // this DMA engine transfers frames to or from the AXI Bus
#include "xio.h"           // Basic Input Output functions to hardware
#include "xtmrctr.h"       // Allows us to interface with the AXI Timer
#include "time.h"
#include "unistd.h"        // Defines standard constants and types
#include "render.h"        // Our rendering file.
#include "xuartlite_1.h"   // Allows us to read from the UART for user input
#include "mb_interface.h"  // Provides the microblaze interrupt enables, etc.
#include "xgpio.h"         // Provides access to PB GPIO driver.
#include "xintc_1.h"       // Provides handy macros for the interrupt controller.
#include "stateMachines.h" // Provides access to our state machines.
```

Figure 8: APIs used by Space Invaders

In addition to these Xilinx APIs, Space Invaders includes several software APIs to help organize the rendering of the game. These were separated into global game variables, game bitmaps, and rendering respective components of the game.

Bitmaps

Bitmaps.h contains all of the functions necessary to access Space Invaders bitmaps (aliens, guises, tank, text, etc). This allows outside code to simply call functions and determine when and where to draw pixels for a given object.

Globals

Globals.h contains all of the necessary initializations, getters, and setters for all of the global positions, alien and bunker statuses, score, lives, and flags. That is, it contains the interface to get all of the data necessary to render the game.

Aliens

Aliens.h contains the functions for rendering aliens, the alien bullets, and calculating alien intersections.

Spaceship

Spaceship.h contains the functions for rendering the spaceship and calculating when the spaceship is hit.

Tank

Tank.h contains the functions for rendering the tank and calculating when the tank is hit. It also contains the functions used to move the tank left and right, as well as the functions for firing and rendering the tank bullet.

Bunkers

Bunkers.h contains the functions for rendering the bunkers and calculating when the bunkers are hit. These functions will correctly erode the hit sub-blocks of the bunkers and update the global variables associated with them.

Render

Render.h contains the functions for general game rendering. That is, the functions to draw the text, lives, and score. It also contains the functions to blank the screen, render everything, and render the bottom green border.

These APIs created very convenient interfaces for rendering and updating game components from the state machines that drive the game. The details of these functions are outlined in the source code. For illustration, the function of rendering the spaceship on the screen is shown below:

```
/**
 * If the spaceship is activated, update its location and draw the Spaceship.
 */
void renderSpaceShip() {
    u32 col;
    u32 row;
    point_t position;
    const u32* arrayToRender;
    if(getSpaceshipActivated()) {
        updateSpaceShipLocation();
        position = getSpaceshipPosition();
        arrayToRender = getSpaceShipArray();
        for(row =0; row <16; row++) {
            for(col =0; col <32; col++) {
                if((arrayToRender[row]>>(31-col)&0x1)==1) {
                    //Slide off the screen
                    if((position.x+col)<640) {
                        framePointer0[(position.y+row)*640+(position.x+col)] = RED;
                    }
                }
                else{
                    framePointer0[(position.y+row)*640+(position.x+col)] = BLACK;
                }
            }
        }
    }
}
```

By using the custom API, the state machines could simply call renderSpaceShip() and the spaceship will update and draw accordingly. Similar functionality exists for all of the other objects on the screen.

To view all of the API functions, please refer to the header files mentioned.

CHAPTER 3: GAME AUDIO

Section 3.1: WAV Files and the AC'97

Space Invaders utilizes WAV files to play sounds. However, in order to use the data, an external WAV converter program was created to convert a WAV file into a simple array of integers, along with a sample rate and number of samples. This generated source file was then utilized in the game.

Subsection 3.1.a: WAV File Conversion

WAV files are binary files that are organized into “chunks” or sections as shown in Figure 9. We created a WAV converter program in using C, that parsed through the binary file until it reached the Format (“fmt”) chunk and extract the sample rate in hertz. Next, the program parses to the Data(“data”) chunk and stores the number of samples in the file. Finally, it uses a loop to iterate over all of the digital audio samples and outputs them to a file.

It is important to note that there are 8-bit WAV files and 16-bit WAV files which store the digital audio samples as an *unsigned* 8-bit number, or a signed 16-bit number, respectively. The wave converter program had to accommodate these changes in its internal data structures. Also note that the data alternates values for both the left and right channel.

The wave converter would then simply output this data into a C-language source file with the format below:

```
int alienSound1_soundData[] = {1, 2, 3, 4}; // Sound data go here.
int alienSound1_numberOfSamples = 50; // This tells you how many samples you have.
int alienSound1_sampleRate = 1000; // This is the sample rate.
```

As other WAV files are converted, they append their respective data onto the end of the C source file. This provides the necessary raw sound data.

Within the Space Invaders software, internal data structures track which sounds are active (need to be played), what sample number it is on, and the total number of sounds that are active. Then, if a sound is active, its sample data is simply added together with all of the other active sounds. This sum is then attenuated and placed in the AC'97 FIFO to be played. Specifically, we used a struct with the following fields:

```
typedef struct
{
    unsigned int active; // Whether to play this sound
    unsigned int sampleRate; // Sample Rate
    unsigned int currentSample; // The index of the array to use next
    unsigned int size; // Total length of the array
    int *arrayAddress; // Pointer to the array's location
} soundData;
```

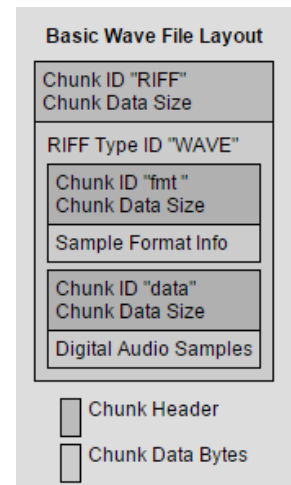


Figure 9: Basic WAV Format

Subsection 3.1.b: AC'97 Operation

The ATLYS board that is running Space Invaders contains a National Semiconductor LM4549A AC'97 audio codec. The Xilinx OPB_AC97 core is used to support this codec. Sound is played by writing 32-bit values to a FIFO. 16 bits are dedicated to the right and left channels. Then, this FIFO is continuously emptied and played by audio controller.

The AC'97 controller also contains several sets of registers that can be used to manipulate the audio. For example, writing to the AC97_AuxOutVol registers can increase or decrease the volume of the output.

The key to playing smooth sounds is keeping the FIFO in a known state. That is, not under or over filling it and making sure that no random data is inserted. For this purpose, the AC'97 will send an interrupt when the FIFO is half empty, allowing the processor to respond by filling it up with more data. The FIFO is implemented using Block RAM that is 512 samples deep. For this to work, the FIFO must be filled with known values on initialization.

It is also important to note that the AC'97 interrupt is level sensitive and positive asserted. This means that the interrupt must be acknowledged after servicing so that another interrupt is not generated.

Subsection 3.1.c: Sound Triggering

Sound is triggered by the same events that are used for rendering. For example, the same event that causes a tank bullet to begin to be drawn (a player pushing the fire button) also causes the tank bullet sound to become active. Similarly, the same event that causes an alien death guise to be drawn (an intersection between a tank bullet and an alien) will also trigger the alien death sound to become active.

Then, all of the active sounds are mixed by adding their digital samples using simple addition. Because we are using 8-bit audio samples in Space Invaders, the 16-bits allocated for the left and right channels allow this simple mixing without having to deal with clipping. Each sound plays until it is complete, and then becomes inactive until its next trigger event. Because sounds are mixed, no extra code was necessary in selecting which sound to play. Table 4 summarizes these triggers.

Sound	Trigger
Marching Sound of the Aliens	Becomes active when alien locations are updated
Flying Spaceship Sound	Always active while the spaceship is on the screen
Tank Explosion Sound	Becomes active when the tank is hit and death guise is rendered
Tank Bullet Sound	Becomes active and resets when the tank fires a bullet
Spaceship Explosion Sound	Becomes active when a tank bullet intersects with the spaceship

Table 4: Space Invaders Sounds and Triggers

This was convenient because each sound structure would simply return a baseline value of 127 if it was inactive, or return its respective sample if it was active. This organization made it so that adding sound to the FIFO of the AC'97 was as simple as summing all sounds.

In order to implement the four different alien marching sounds, a static variable was used and looped through the four possible sounds.

CHAPTER 4: DMA CONTROLLER

Space Invaders also includes both a software-based and hardware-based screenshot feature. This feature is controlled by the switches on the ATLYS board. The instructions are summarized in Table 5 below. Note that the instructions below are activated on the rising edge of the switch (e.g. flipping switch 7 from down to up will use hardware to capture a single screenshot, and will not do anything until the switch is lowered and raised again). Also note that the only one screenshot can be stored at a time. That is, as soon as another screenshot is taken, the old screenshot is overwritten.

Instruction	Switch Configuration								Description (X = switch is ignored)
	7	6	5	4	3	2	1	0	
Hardware Screenshot	↑	X	X	X	X	X	X	X	On the transition from DOWN to UP, the <i>hardware</i> will use DMA to capture a single screenshot of the current screen.
Software Screenshot	X	↑	X	X	X	X	X	X	On the transition from DOWN to UP, the <i>software</i> will capture a single screenshot of the current screen.
Display Screenshot	X	X	↑	X	X	X	X	X	While switch 5 is UP, the gameplay is paused and the screenshot is displayed.
Display Game Screen	X	X	↓	X	X	X	X	X	While switch 5 is DOWN, normal gameplay is displayed.

Table 5: Screenshot Instructions

Register Descriptions

In order to achieve this, the DMA controller contains three software-accessible, 32-bit registers.

Register name	Offset from Base Address	Function
Source Address	0x0	Stores the starting address to read from. This address will be incremented each time a word is read.
Destination Address	0x4	Stores the starting address to write to. This address will be incremented each time a word is written.
Transfer Length	0x8	The 32-bit unsigned integer representing the number of words to copy from Source to Destination.

Table 6: DMA Controller Register Summary

This registers are written to using the parameters passed into the API function `DMA_CONTROLLER_CopyData` described in Section 4.1.

Details on the registers functionality is described below:

Source Address Register

When DMA is performed, the 32-bit value stored in this register is interpreted as an address to memory. The DMA starts from this address, then increments it by 4 (the number of bytes in a word on the MicroBlaze). This means that if DMA is started from Source Address 0x4 and *Length* = 3, the words at locations 0x4, 0x5, and 0x6 would be copied to the Destination Addresses.

Destination Address Register

Like the Source Address Register, the 32-bit value stored in this register is interpreted as an address to memory. Values read from the source and written to this address. This address also increments by 4 each write.

Transfer Length Register

The 32-bit value in this register is interpreted as unsigned and specifies the number of words to transfer using DMA. For example, if the value in this register is 0x5, 5 words will be copied from source to destination.

Section 4.1: Driver API

The Application Programming Interface defined to interact with the hardware DMA is outlined below.

```
void DMA_CONTROLLER_CopyData(Xuint32 BaseAddress,
                             Xuint32 SrcBaseAddr,
                             Xuint32 DestBaseAddr,
                             Xuint32 Length);
```

This function copies the specified number of words from the source address to the destination address. *Length* specifies the number of words to transfer. *SrcBaseAddr* is the base address of the location to copy from, and *DestBaseAddr* is the base address of the location to copy to. The first parameter, *BaseAddr* is the base address of the DMA controller as assigned in `xparameters.h`.

Example usage of this function is shown below:

```
void hardwareCapture() {
    int screen_size = 640*480;
    // Perform the DMA
    DMA_CONTROLLER_CopyData(XPAR_DMA_CONTROLLER_0_BASEADDR, // IP BaseAddr
                           (Xuint32) framePointer0, // Source Address
                           (Xuint32) captureFramePointer, // Destination Addr
                           (Xuint32) screen_size); // Number of Words to transfer
}
```

Note that a single-pulse interrupt will be sent from the hardware DMA controller after all the data is finished copying.

Section 4.2: Performance Analysis

The hardware DMA functionality was added to illustrate the significant performance difference between hardware and software screen captures. Because DMA can occur while the processor is doing other things, rather than consuming processing power, it can capture the screen with essentially no effect on the gameplay. In contrast, using software to capture a screen requires iterating over 307,200 pixels, which is much slower.

Measurements were taken using the system AXI Timer, which provides a way to measure functions with clock by clock accuracy. Table 7 below shows the average time it takes to perform both software and hardware screen captures. Note that time was derived knowing that the system is running on a 100MHz clock.

Software Screen Capture	144.7 ms
Hardware Screen Capture	130.4 ms

Table 7: Screenshot Performance Times

Although the hardware screen capture appears to only be about 10% faster than the software capture, the fact that it is done in using Direct Memory Access allows the game to play without any degradation in game performance. In contrast, using the software capture produces a slight, but noticeable, pause in gameplay as the CPU time is spent copying the data.

Note also that the software screen capture was performed using the code below:

```
void softwareCapture() {  
    int captureLoop;  
    for(captureLoop = 0; captureLoop < 640*480; captureLoop++) {  
        captureFramePointer[captureLoop] = gameFramePointer[captureLoop];  
    }  
}
```

Although the software optimization may improve the software screen capture time, the ability for hardware DMA to read/write memory while the CPU does separate work still provides the best performance gains.

APPENDIX A: TIMING & MEMORY REPORT

Space Invader's memory usage (in bytes) is shown below:

text	data	bss	dec	hex	filename
75230	504896	5866	585992	8f108	space_invaders_0.elf

Space Invaders Memory Footprint

Using the AXI Timer, we measured the shortest, longest, and average execution time. We also estimated the CPU utilization. Our results and methodologies are summarized below.

Shortest Time	6.31μs
Longest Time	48,291μs
Average Time	1,897μs
CPU Utilization	19%

Timing Statistics

Shortest Time

The shortest time was found by starting the AXI timer when each state machine is called then stopping the timer when the state machine returns. Each state machine has a best case execution time. These times were added together to get the shortest execution time for one loop. This occurs when none of the bullets need to be updated, the tank isn't moving the spaceship isn't on the screen and there is no alien death. Thus, this time is spent just checking conditions and found that there is nothing for the state machines to do.

Longest Time

The longest time was found by starting the AXI timer when each state machine is called then stopping the timer when the state machine returns. Each state machine has a worst case execution time. These times were added together to get the longest execution time for one loop. This occurs when the bullets need to be updated, the tank is both firing and moving, the spaceship is moving, the aliens have hit a wall and are shifting down a row, and there was an alien explosion shown. This time is about 48ms, four times our FIT timer interval. The major factor in this time is when the alien block shifts down a row because it hit a wall. However, this does not affect the smoothness of the visuals.

Average Time

The average time was computed by running the game for 43 seconds and timing how long the state machines ran during that time. The AXI timer is a 32 bit timer, the max value is $2^{32} = 4,294,967,296$ and runs at 100MHz. 4,294,967,296 clock cycles at 100 MHz is 42.9 seconds. The FIT timer interrupts at 100 Hz. For the game to run for about 43 seconds the FIT needs to interrupt 4300 times. After 43 seconds the AXI timer value divided by 43 to get the average number of clock cycles per second. This was then converted to time.

CPU Utilization

The CPU utilization was computed by running the game for 43 seconds and timing how long the state machines ran during that time. The FIT timer interrupts at 100 Hz. For the game to run for about 43 seconds the FIT needs to interrupt 4300 times. After 43 seconds the AXI timer value divided by the max value of the AXI time. This resulted in the percent time the CPU spent doing things that were game related.

APPENDIX B: BUG REPORT

This appendix contains a list of bugs and solutions encountered throughout the lab.

Bug Identifier	Symptoms	Solution
Move to DDR	After moving to DDR, the screen display is incorrect.	The solution to moving to the DDR correctly was making sure that the FRAME_BUFFER_0_ADDR was changed correctly everywhere in the program. It had not worked the first time because a second macro definition was not properly updated.
Slow Alien Render	The Alien block appears “flashy” as it draws.	Initially, the software had functions that calculated the color for each pixel of each alien and returned it to be drawn. To significantly reduce the number of function calls and calculations made, the software was altered simply returned a pointer to the bitmap, and used array indexing to draw the entire alien.
Missing Collisions	Both the spaceship and the aliens would not register collisions if their top-left position was off of the left-edge of the screen.	This was caused because we use unsigned ints to track (x,y) coordinates. This meant that when an x value was less than zero, it would overflow and prematurely jump out of our logic. To fix this, we added constants that would compensate for the overflow on the bounds.
Flashing Tank	Every second or so, our tank would flicker when rendering.	Rather than unrender the entire tank each time, we utilized the fact that it only moves a set amount each update. We then render extra black columns of the width of the movement on both the left and right side of the tank. This effectively “erases” its previous location, and allowed us to speed up the tank draw so that no flicker was visible.
Popping Sounds	At the end of our sounds, we heard a brief pop.	This was an error in our wave converter program. In our converter, we had inadvertently truncated read data samples as 16-bits although the wav file was of 8-bit format. This meant that although our total sample size was correct (say, 8000 samples) our array only contained 4000 samples. This meant that at end of each sound, the FIFO was filled with garbage data resulting in an audible pop.
Hardware DMA Crash	When activating the hardware DMA, Space Invaders would crash.	Luckily, this was a simple fix. The pointers to each frame buffer were <i>pointers</i> , and we had accidentally passed in the values as pointers to pointers into DMA_CONTROLLER_CopyData (e.g. &framePointer0 rather than just framePointer0). This resulted in the hardware DMA corrupting whatever data happened to be at that location, and crashing the game.

REFERENCES

ⁱ Square Enix History (2003-). (n.d.). Retrieved October 1, 2014.

ⁱⁱCan Asteroids Conquer Space Invaders? (1981, Winter). Electronic Games, 30-33.

ⁱⁱⁱJiji Gaho Sha, inc. (2003), Asia Pacific perspectives, Japan 1, University of Virginia, p. 57, retrieved October 1, 2014.

^{iv}"Space Invaders vs. Star Wars", Executive (Southam Business Publications) 24, 1982: 9, retrieved October 1, 2014.