CMPE 490

FINAL REPORT

VIDEO GAME WITH WIRELESS ACCELEROMETER-BASED CONTROLS

Project Summary: Space-shooter arcade emulator with wireless accelerometer-based controller

Group Members:	**Preferred email:**
Billy Kozak	kozak@ualberta.ca
Nathan Sinnamon	sinnamon@ualberta.ca
Jeff Theriault	jtheriau@ualberta.ca

(Available for both lab days, registered for Monday labs)

Submission Date: April 13th, 2011

Abstract

Our group has created an original video game similar to classic space-shooter arcade games such as Taito Corporation's Space Invaders. This project centers on the use of the Altera DE2 development board and the NIOSII microprocessor, and the necessary software was written using μC/OSII. The aim of the game is to shoot down and evade enemy spacecraft and their projectiles. We have also created our own controllers which will take advantage of accelerometers to provide the basic two-axis movement of the protagonist spacecraft along with a 'fire' and 'start' button which provide other functions which are standard to the genre. The controllers use ZigBee links for wireless communication and PIC-based microcontrollers. Due to the nature of the project's output, video (VGA) interfacing was required. The cost of all parts purchased for this project is $184.36.

Table of Contents

Functional Requirements....3
Design and Description of Operation....3
  Hardware Design....3
  Controller Design....4
  Gameplay Design....5
Parts List....6
Datasheet....6
  Controller I/O....6
  Main Board I/O....7
Software Design....8
  Game Engine Task....8
  Controller I/O Tasks....9
  Controller Firmware....9
Test Plan....10
Results of Experimentation....10
  Controller Testing....10
  VGA / Frame Reader Testing....11
Appendices....12
Declaration of Original Content: CMPE 490 Final Report....16

Functional Requirements

Functionally, our project must provide a smooth and consistent interface between user input and the video output to maximize the entertainment value and minimize the frustration of the user. Primarily, this means quickly and efficiently processing the user's physical movement of the controller and button presses, and converting that data into meaningful changes in the video output feed at a consistent rate.

Design and Description of Operation

The data flow for this project can be seen in Figure 1. Input signals from the user, in the form of button presses and analog accelerometer data will be passed to a local microcontroller on the controller. After being processed and encoded by the microcontroller, the data is sent wirelessly from the each controller to a Xbee wireless receiver which has a line to the NIOS II [1]. Responding to this input data, the video output frames are calculated according to the game engine and are passed to a video output monitor via VGA.

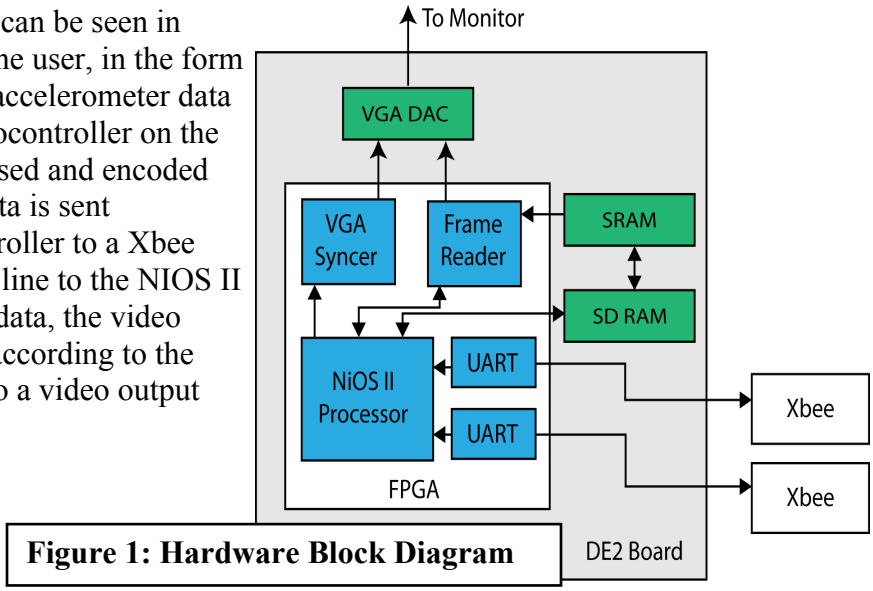


Figure 1: Hardware Block Diagram

Hardware Design

The hardware environment for our game's software includes the NIOS II/f processor and an SDRAM. For our project we found that we were able to clock the system up to 100 MHz using a PLL. This was crucial in allowing us to achieve a much higher performance than expected. Our program runs from the 8MB SDRAM.

The VGA buffer system (shown in Figure 2) makes use of two RAM memories, a DMA and two custom FPGA components (implemented in VHDL). The system works by having the NIOS generate graphical data and placing it into a temporary frame stored in SDRAM. From there

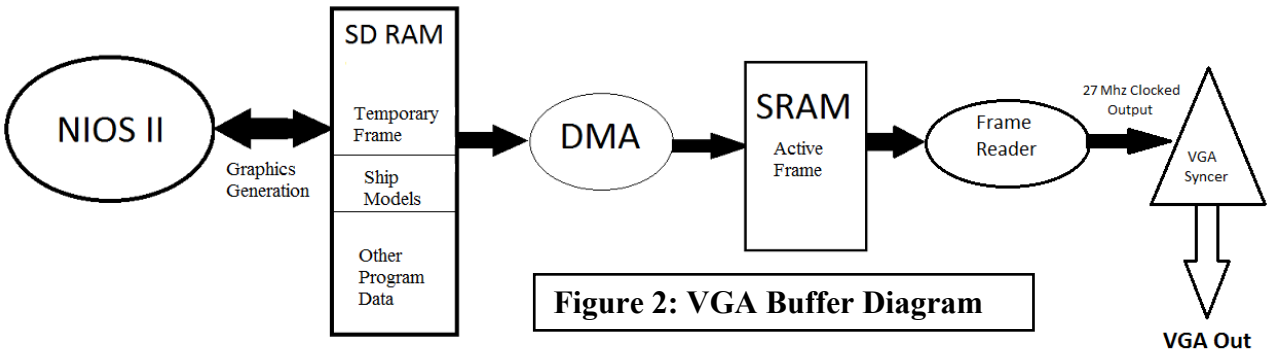


Figure 2: VGA Buffer Diagram

the NIOS queues up a DMA memory copy from SDRAM to SRAM. The DMA then moves the frame into the final frame buffer stored in SRAM. Using the temporary frame with DMA allows us to prevent the flicker and inconstancy in images that results from erasing and redrawing inside of the active frame. The use of the DMA means that this has only a small performance impact as the DMA transfers memory from the SDRAM in parallel with the CPU tasks (the instruction and data caches prevent the competition for the SDRAM between DMA and NIOS to be minimal). From there our two custom parts take the image stored in SRAM and output the digital VGA signals so that the image is displayed onscreen.

The Frame Reader was the most difficult piece of the puzzle to implement. The Frame Reader uses the Avalon memory mapped master interface to read from the SRAM as fast as possible, buffering pixels for output. The frame reader is needed because the SRAM is too slow for the syncer to read pixels on demand without errors. Arbitration is handled by the system interconnect fabric so long as the interface is used correctly. The Frame reader has two buffers: a read buffer and a write buffer. When a special signal from the VGA syncer arrives the frame reader swaps the read and write buffers. Each buffer is exactly large enough to contain an entire line of pixels from memory so that the syncer will request a swap in buffers only when it finishes reading a line. Originally we attempted to use a 640*480 resolution but it proved to be too hard to fill the entire buffer in time so we divided the resolution to 320*240 using the frame reader. When the syncer requests pixels it reads every horizontally adjacent pixel twice and then reads the entire line from the read buffer again to create pixels that are 4times larger. However it wasn't until we implemented burst reads using the Avalon memory mapped interface that we were finally able to achieve a performance that seemed sufficient.

The VGA snycer generates RGB, horizontal sync, and vertical sync signals at precise timings to produce an image on the screen. Furthermore the syncer drives x and y lines that show which pixel it wants to output and reads the pixel value from RGB input lines. Finally the syncer generates a signal when it is done reading a line and about to read the next so that the frame reader can switch read and write buffers in time. The signal is generated during a (rather lengthy) period where the syncer is not outputting RGB values so that it is safe to swap read and write buffers without getting graphical glitches.

Controller Design

Communication with the controllers will be over a wireless ZigBee link. Each controller has an accelerometer (MMA8452Q) to detect motion of the controller and two buttons to signal actions to the system. A PIC-based microcontroller (PIC16F877A) will be on the controller board to process the accelerometer data, and to communicate the current state of the controller to the main system. The PIC microcontroller was chosen because of previous team experience with the platform. (See Appendix C for controller hardware diagram.) The microcontroller communicates with the chosen accelerometer via the I2C bus. Fortunately, the microcontroller has I2C built into the hardware, which makes communicating with the accelerometer quite painless. To use the

ZigBee link, each controller has a Xbee wireless radio on board. Communication with the XBee is done over a simple serial link running at 3.3V.

The accelerometer unit chosen outputs acceleration data in both 8-bit and 12-bit formats. The 8-bit format yields 256 possible output values representing +/- 2g. During testing, it was decided that the 8-bit format provides sufficient resolution to determine the controller's movements. Using the 8-bit format also simplifies the controller software as the microcontroller being used is an 8-bit part. The state of the controller will be transmitted as a packet over the ZigBee radio link. Initially, the controller data format included a controller ID, and a battery level indicator. Further into the development of the game, it was determined that using a single base-station XBee would prove complicated as the two controller's output would interleave in a non-specific manner. The design was therefore changed to include two XBees on the main unit, one for each controller for a total of four radios in the system. This eliminates the need to transmit the ID of the controller. Due to time constraints having the battery level communicated to the main unit was deemed a non-essential feature and was eliminated.

Each packet transmitted from the controller contains the following things: the current acceleration in the X, Y, and Z axis, and the state of the start and fire buttons. Each of these values are encoded into ASCII format and separated by the colon (":") character. ASCII encoding was chosen to simplify the processing of the data at the main unit and to help in debugging.

Field	X	:	Y	:	Z	:	Fire	:	Start	\n
Width (bits)	8	8	8	8	8	8	8	8	8	8

Figure 3: Controller Packet Format

Each packet has a total length of 80 bits. Assuming a baud rate of 9600 and 8-N-1 format for the serial Zigbee link, roughly 120 controller updates can be transmitted (neglecting latency). Using documentation from Digi, it can be determined that after factoring in worst-case latency, we can expect only 20 updates per second. [2] After testing, we felt that 20 updates per second proved to be sufficient for smooth user input despite the fact that we were able to get more than 20 updates per second.

Additional RAM will be needed to facilitate the VGA output. If we use 8-bits to represent each pixel, we only need 300Kbytes. This will fit nicely into our 512Kbyte SRAM with room to spare. If 256 colors proves too limiting, we can increase color depth as needed until we run out of memory. The resolution of 640x480 was purely chosen to minimize RAM space, if there is time to implement, the resolution may be increased for better aesthetics. According to datasheet timing diagrams, the read cycles on both the on-board SRAM and SDRAM take roughly between 15-20ns. Given that for 640x480 our pixel clock is 25Mhz, we have 20ns left over after reading to send the pixel signal.

Gameplay Design

Gameplay emulates the control of a spaceship from a top-down view by the player via the controller(s). The accelerometer controls dictate the position of the ship in the two-dimensional world; tilting of the controller to the right/left will determine horizontal movement, where tilting of the controller to the front/back will determine vertical movement. Movement speed has been designed to be consistent for movement in any direction. As the player-controlled ship moves, enemy ships appear from the top of the screen moving downwards and exiting through the bottom of the screen via paths determined by one of several AI routines described in the game engine. The goal of the player is to fire as many projectiles – via a button press – at enemy planes as is needed to destroy them before they exit the screen, while avoiding both enemy ships and their respective projectiles. If the players' ships comes into contact with either of these obstacles too many times, their health bar depletes and the game ends. A scoring system has been implemented for the sole purpose of bragging rights, though no high-score screen has been implemented. The game consists of three unique levels for the player to progress through, with each level only accessible by completing the previous; if the players survive all three levels, they can start over at level one and continue improving upon their current score.

Parts List

Qty	Part Name	Unit Cost	Total	Order Status
2	Accelerometer (MMA8452Q) (I2C +/-2g) http://www.sparkfun.com/products/10955	9.95	19.90	Attained
3	PIC Microcontroller (PIC16F877A) http://ww1.microchip.com/downloads/en/devicedoc/39582b.pdf (2x controller, 1x spare)	7.38	22.14	Attained
4	XBee ZigBee Radio (2x Main System, 2x Controller)	35.58	142.32	Attained
			184.36	

Datasheet

Controller I/O

All parts use 3.3V power supply.

Signal Name	Description	Tx	Rx
Wireless Antenna	Antenna of Xbee	Bidirectional Air/Xbee	Bidirectional Air/Xbee
Xbee Tx	Serial Transmit line of Xbee	Xbee	PIC Microcontroller
Xbee Rx	Serial Receive line of Xbee	PIC Microcontroller	Xbee Radio

SDA	I2C Serial Data line	Bidirectional – PIC/ Accelerometer	Bidirectional PIC/ Accelerometer
SCL	I2C Serial Clock line	PIC Microcontroller	Accelerometer
Start Button	Digital I/O indicating wether the start button has been pushed.	Button	PIC Microcontroller
Fire button	Digital I/O for Fire button	Button	PIC Microcontroller

Main Board I/O

Signal Name	Description	Generated From	Received By
Wireless Antenna	Antenna of Xbee	Bidirectional Air/ Xbee	Bidirectional Air/ Xbee
Xbee Tx (x2)	Serial Transmit line of Xbee	Xbee	FPGA (NiOS II)
Xbee Rx (x2)	Serial Receive line of Xbee	FPGA (NiOS II)	Xbee Radio
SCLK	SPI Clock for SD Card (If Implemented)	FPGA (NiOS II)	SD Card
MOSI	SPI data line (If Implemented)	FPGA (NiOS II)	SD Card
MISO	SPI Data line (If Implemented)	SD Card	FPGA (NiOS II)
CS_N	Chip select for SD Card (If Implemented)	FPGA (NiOS II)	SD Card
VGA_BLANK_N	VGA Blanking signal	FPGA (VHDL)	VGA DAC
VGA_SYNC_N	VGA Sync Signal (tied high)	FPGA (VHDL)	VGA DAC
VGA_CLOCK	VGA Pixel Clock	FPGA (VHDL)	VGA DAC
VGA_HSync	VGA Horizontal Sync Strobe	FPGA (VHDL)	VGA DAC
VGA_VSync	VGA Vertical Sync Strobe	FPGA (VHDL)	VGA DAC
VGA_R	VGA 10-bit Red Signal	FPGA (NiOS II)	VGA DAC
VGA_G	VGA 10-bit Green Signal	FPGA (NiOS II)	VGA DAC
VGA_B	VGA 10-bit Blue Signal	FPGA (NiOS II)	VGA DAC
SRAM_ADDR	18-bit address bus for SRAM	FPGA	SRAM
SRAM_DATA	16-bit data bus for SRAM	Bidirectional FPGA, SRAM	Bidirectional FPGA, SRAM
SRAM_CE_N	SRAM Chip enable	FPGA	SRAM
SRAM_WE_N	SRAM Write Enable	FPGA	SRAM
SRAM_UB_N	SRAM Upper Byte Strobe	FPGA	SRAM
SRAM_LB_N	SRAM Lower Byte Strobe	FPGA	SRAM
SRAM_OE_N	SRAM Output Enable	FPGA	SRAM

Software Design

The software for our project will be a game that will showcase the motion controller and output graphics to a VGA monitor. We plan to use the MicroC/OS-II RTOS to implement our game.

The game is implemented as three separate tasks (described in more detail below) that handle user input, control the game's state (game engine), and create graphical output respectively. The user input task processes and interprets user input and sends the processed data to the game controller task. The game engine task updates the game's internal model as time progresses with new user data as it is received and sends select information concerning the game's state to the graphics thread. The graphics thread will interpret data from the game engine as graphical output and write to the frame buffer for VGA output.

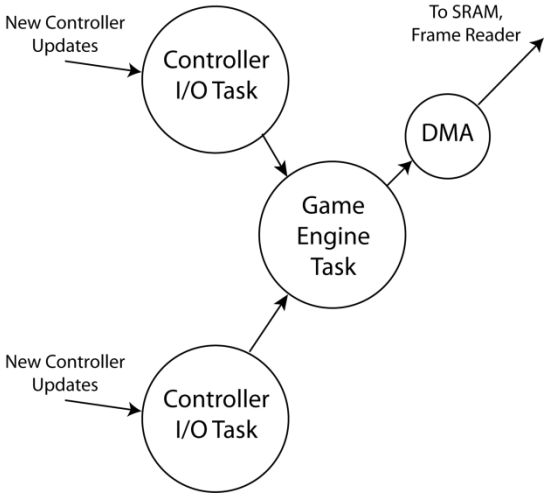


Figure 4: Software Data Flow Diagram

Game Engine Task

The game engine works by repeating 8 basic steps in a continuous loop.

- **Step 1:** Fetch new ships from memory and mark them for drawing to the screen. Ships are added as predetermined amounts of time pass.
- **Step 2:** Read data from the controller and update the position of the payer's ship (two short integers representing x and y coordinates). Projectiles can be created at the player's location.
- **Step 3:** Update the location of enemy ships (similar to updating player ship position) according to simple AI routines. AI routines can fire projectiles from the ship's position.
- **Step 4:** The locations of all onscreen projectiles are updated.
- **Step 5:** Detect and resolve collisions. A grid is kept in memory that mirrors the game's video output and tracks the position of every ship's pixels for the purpose of collision detection. If a ship is found to be trying to move into space occupied by another ship a collision has occurred and must be resolved by deleting enemy ships and projectiles or removing health from the player.
- **Step 6:** Redraw every ship and projectile in the temporary frame buffer according to their coordinates and the shape of the object then queue a DMA copy to place the frame into the frame buffer.
- **Step 7:** Check for victory or loss conditions. Advance Level or restart the game as required.
- **Step 8:** Update the game's time count and ensure that at least 15ms have elapsed since step 1 before going back to the first step. Done so that the game does not slow noticeably under moderate to heavy loads (some slowdown may occur under exceptional circumstances).

Some additional details about the engine are as follows:

1. Our software design is such that we are able to update the screen over 60 times per second (note that this works very well with the VGA refresh rate of 60Hz). We had originally anticipated only 17 frames per second but with some unexpected improvements to our hardware environment we were able to achieve over 3 times our expected performance.
2. We made extensive use of function pointers so we could run AI, collision detection, and draw/Erase routines on ships with completely different dimensions.
3. Levels are done by loading arrays of ships (structures) each with their own behaviour appearance starting location and starting time.
4. We explicitly defined the location and color of each pixel of each type of ship by creating arrays containing the appropriately placed pixel values. We then copy the array to memory at the target ship's location when we need to check for collisions or redraw the ship.

Controller I/O Tasks

On the main unit, controller input is handled by two MicroC/OS-II tasks, one for each controller. The tasks open a UART connected to an XBee (there is one for each controller) and sits in a loop. On each execution of the loop, it attempts to read a line from the serial line, blocking if one isn't available. Once a line has been read, it is broken up into its respective components according to the controller packet format. The acceleration data is used to compute the roll and pitch angles of the controller. These two values give us an indication of how the controller is tilted. The roll and pitch angles [3] are computed as follows:

$$Pitch = \tan^{-1}\left(\frac{A_y}{\sqrt{A_x^2 + A_z^2}}\right)$$

$$Roll = \tan^{-1}\left(\frac{A_x}{\sqrt{A_y^2 + A_z^2}}\right)$$

Once these two values have been calculated, a global structure for the controller's state is updated. This structure is used inside the game engine for computing changes in the game's state.

Controller Firmware

The controller's firmware is simple and straightforward. When the controller is powered up, the firmware initializes the hardware USART for communication with the XBee that is on the controller. Once this has been accomplished, it transmits a "Hello" message. This is purely a diagnostic message and is ignored by the main unit. After serial communications are up, the hardware I2C controller built into the microcontroller is configured and activated. Communication with the accelerometer is then verified by accessing a "WhoAmI" register

on the accelerometer that always returns a known value. If this value matches, the firmware configures parameters on the accelerometer and enters a tight loop. The loop reads acceleration data from the accelerometer, formats it into the controller packet format described in "Hardware Requirements" and transmits it down the serial link to the XBee where it is communicated wirelessly to the main unit. This loop also contains a delay at the end to implement data throttling.

Test Plan

Test-driven development is an important strategy in any engineering project, and it is especially important in the case of a project which requires interfacing of computer hardware and software. Therefore, our test plan encompassed breaking the project into multiple smaller units, to test these units individually, and to perform tests to ensure that each unit works as expected with the others when these units are combined into a larger system.

First, upon assembling the controller hardware and programming related microcontrollers, we tested the connection and transmission of data between the controller and the Altera DE2 board. The VGA connections were first tested with simple video output (e.g. a color pattern) from the DE2. Past a simple static test pattern, the frame reader component was tested to ensure that multiple frames could be read in sequence from a memory location in SDRAM and output via VGA during runtime and as directed by software. To test these units' interaction, a test was implemented by which the video output could be altered via controller input (e.g. a horizontal shift in the pattern displayed on the screen). Finally, as the game engine was designed and new features were added to game, testing was done at each step to ensure that the controller input affected the video output as directed by the game engine software.

Results of Experimentation

Controller Testing

Once the controller prototype was assembled and the firmware programmed into the microcontroller, a computer was connected to the microcontroller's serial port to examine output. Several iterations of the controller firmware were required to get reliable output to be displayed on the computer.

After serial data output was verified, the XBees were introduced into the mix. Two of the major concerns with the XBees were the latency of the connection, and whether or not one XBee at the main unit could be used to receive data from both controllers. By replacing the computer connection with an XBee and connecting the computer to another XBee, the data transmission could be evaluated. It was determined that the latency in the connection was minimal at the distances the controllers would likely be used at. The testing did show that an additional XBee would need to be installed in the main unit. When one is used to receive data from both controllers, it interleaves the two streams, making interpretation of the data difficult.

With wireless communication verified, a test harness was built using the DE2 board. The DE2 board would receive data from one controller to evaluate the NiOS's ability to keep up with the controller data stream. During testing it was found that 20 updates per second is taxing on the board when MicroC/OS-II multitasking is enabled. Error rates were seen as high as 20% as the software struggled to keep up. To manage this, throttling was implemented in the microcontroller code to reduce the updates to only ten per second. Following this, the error rate dropped dramatically however further performance testing will have to be done once the game engine is fully implemented on the board. Once the DE2 was receiving data from the controllers, the software was updated to light LEDs in rough similarity to how the controller is being rotated around.

VGA / Frame Reader Testing

A system was set up with SOPC builder into which the test pattern generator was inserted; this successfully showed the expected VGA output results. However, when the frame reader component was substituted for the working test pattern generator component, the output was all but incomprehensible. A single uniform color could be output to the screen by writing the same value to every pixel location in memory, but this was later found to be because the frame reader was only reading (and outputting) a single pixel per frame. With the aide of Signaltap II software, we were able to take a look into the signals corresponding to the (video) control packets and regular video packets, with the intent of establishing some source of error. However, the control packets' contents seemed to exhibit signs that writing to some of the control registers caused an overlap of information between said registers, but not in a predictable manner.

Due to these and other apparent bugs in the Altera-provided frame reader component, it was decided that a new frame reader should be written from scratch; in addition, a syncer component was also written to replace the clocked video output component to better suit our unique needs.

Appendices

Appendix A – Quick Start Manual

To re-assemble, connect, configure and demonstrate this project:

1. Launch Quartus II, create new project targeted for the DE2 board
2. Open niosII.sopc in SOPC builder, generate
3. Add all .vhd files in PROJECT_SOURCE directory except for niosII_inst file
4. Import all pin assignments
5. Set niosII_finalProject.vhd as top level file, compile
6. Program the device
7. Launch NiosII IDE, create new Nios II C/C++ Application project
8. Select the niosII.ptf file
9. Open the Project properties and under the System Library set the RTOS to MicroC/OS-II, set sdram_0 for all memory components, under C/C++ Build -> Nios II Compiler -> General set the compiler flags to "-std=c99" set the optimization level to O3, also set the optimization level to O3 for the Syslib project and apply all changes
10. Add all .c and .h source files to the project
11. Clean and Build the project
12. Connect receiver board to GPIO_0 on the DE2 board, connect monitor to VGA output on the DE2 board
13. Run project as NiosII hardware

Note: Quartus/SOPC builder are very unreliable for building our project.

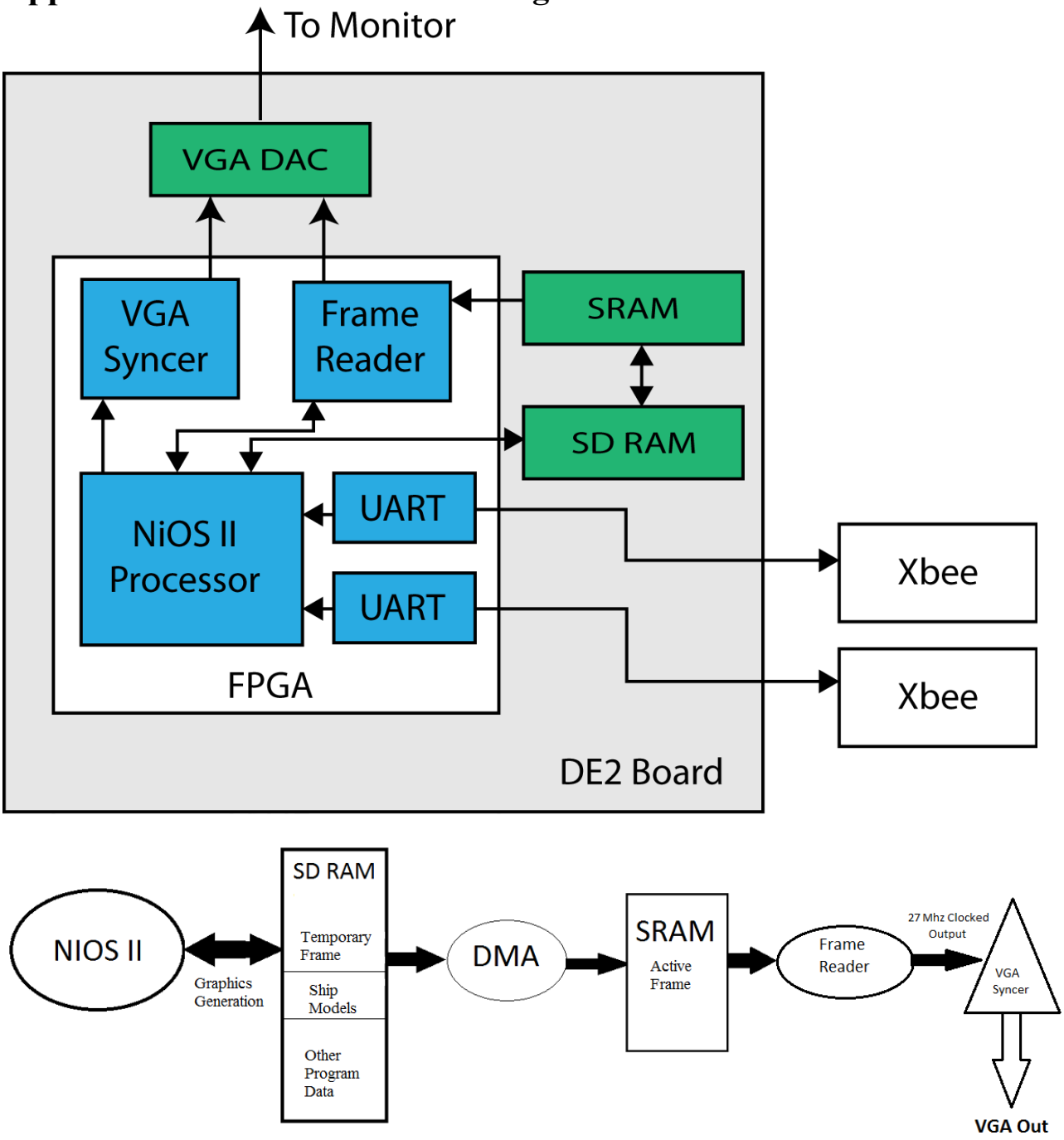
Alternatively, the niosII_finalProject.sof (and niosII_finalProject.pof) can be used to program the hardware and the niosII.ptf file can be used to program software.

Note: Switch 0 can be used to toggle multiplayer capabilities.

Appendix B – Future Work

To extend on this project we would create an audio output that would go along with the game which would entail sound effects as well as a background music track that plays continuously. The actual gameplay software could be altered as well; vertically-scrolling background graphics (eg. in the style of Xevious) could be implemented, features such as ship upgrades could be added and the number and length of game levels could be extended. It would also be possible to create other games, using the controllers and other hardware already in place from creating the first.

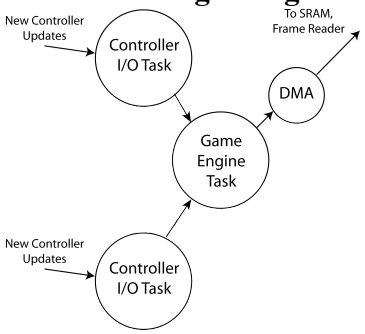
Appendix C – Hardware Block Diagram



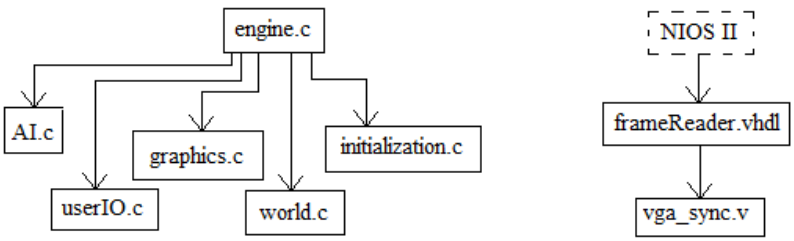
Appendix D – Source Code Documentation

File(s)	Description	Status
frameReader.vhdl	A substitute SOPC builder component written to replace the Altera-provided component by the same name	T
vga_sync.v	A substitute SOPC builder component modified from to replace the clocked video output component	T
AI.c, AI.h	Determines the movement and behavior of enemy ships within the game, and associated header file	T
engine.c	Mathematical modelling responsible for gameplay mechanics	T
graphics.c, graphics.h	Responsible for ship and other generalized graphics, and associated header file	T
fontGraphics.c, fontGraphics.h	Responsible for alphanumeric font and score area graphics, and associated header file	T
initialization.c, initialization.h	Responsible for game startup, and associated header file	T
levels.c, levels.h	Responsible for creating the game's list of ships via hard-coded patterns that can be repeated any number of times, and associated header file	T
moveCollide.c, moveCollide.h	Responsible for the checking and resolution of game projectiles, and associated header file	T
startScreen.c, startScreen.h	Responsible for the display of the title screen, and associated header file	T
userIO.c, userIO.h	Responsible for controller input tasks, and associated header file	T
vgaDrive.c, vgaDrive.h	Responsible for driving the DMA and VGA as needed by the game, and associated header file	T
world.c, world.h	File in which the current state of the game is held, and associated header file	T
main.asm	Reads triple-axis accelerometer data and outputs in a friendly format for main console to interpret.	T
ControllerDE2TestBench.c	Testbench for controller/DE2 board interaction	T

Software Design Diagram



Graphical Heirarchy of Source Code



Declaration of Original Content: CMPE 490 Final Report

The design elements of this project and report are entirely the original work of the authors and have not been submitted for credit in any other course except as follows:

[1] The Nios II embedded processor, DE2 Development and Education Board, and the Cyclone II FPGA are all Altera products. Altera's website: http://www.altera.com/

[2] Digi, Inc. "Sending data through an 802.15.4 network latency timing." Internet: http://www.digi.com/support/kbase/kbaseresultdetl?id=3065, [February 3, 2012]

[3] Freescale. "Tilt Sensing Using Linear Accelerometers." Internet: http://www.freescale.com/files/sensors/doc/app_note/AN3461.pdf, [Feb. 12, 2012]

Billy Kozak _____ Date

Nathan Sinnamon _____ Date

Jeff Theriault _____ Date

a signed hard copy of this page will be dropped into the CMPE 490 drop box