**Project 2: Asteroids**

**Introduction:**

The purpose of this project was to create a game similar to the classic arcade game Asteroids using VHDL, Altera Megafunctions and basic logic gates. In this game the player pilots a spaceship which must dodge asteroids which crowd the screen. If the player contacts an asteroid they lose the game.

The player controls the spaceship using the arrow keys on a PS/2 keyboard. The player is able to move throughout the entire screen. Two sizes of asteroids fly across the screen then disappear after exiting the screen. After an asteroid exits the screen it re-spawns at a random location at the top of the screen and with a random x and y velocity. The game begins with one asteroid, but the longer that the player survives, the more asteroids appear. After four minutes of game play, there are a total of 12 asteroids. Score for this game is kept based on time. For each second that the player survives they receive one point.

**Methods:**

A basic block diagram for this project can be seen in Fig. 1. The inputs to the design are a reset switch which resets the keyboard block and a PS/2 keyboard for game control. The Keyboard Processing block has responsibility for reading in values from the keyboard using a design block provided in a previous lab, and converting these into signals which can be easily interpreted by the ship to change its direction. The Keyboard Processing block looks for the key codes for the up, down, left, and right arrows and have an output for each which will be used by the ship to update its position. The Ship block is responsible for updating its position and outputting a visual of itself to the VGA output. The collision detection block is responsible for detecting collisions between the ship and with any asteroids. If a collision with an asteroid has occurred, the collision detection block outputs a reset to restart the game. The Asteroids block has the responsibility of producing and displaying asteroids onto the game field. The number of asteroids which are active is based on the count of the game clock. When the game starts there is only one asteroid, this number increases until there are 12 asteroids at the 240 second (four minute) count. A Random Position and Velocity Generation block provided initial information to each asteroid so that it can re-spawn after exiting the screen. This will provide for diverse gameplay. A Game Clock block houses the game clock which is used to keep track of how long the player has played since the last reset. This block also displays this time in seconds to the screen. The VGA output block was provided from a previous lab.



Fig. 1 Block Diagram of Asteroid Game

This block diagram was then implemented as a logic circuit in Quartus II. The top-level schematic for this design can be seen in Fig. 2. Each of the components in this schematic are discussed below.

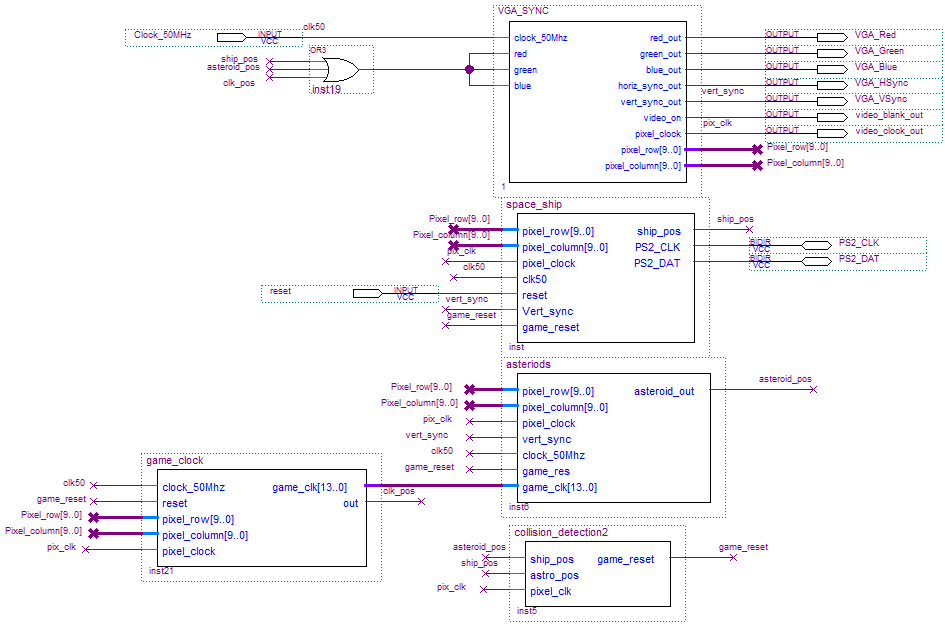


Fig. 2 Top-Level Schematic for Asteroid Game

The first block in this diagram is the “VGA\_SYNC” block. This block was provided in a previous lab and no changes to this block were necessary. As can be seen from the diagram, the inputs red, green, and blue were tied together. This allowed for black and white control. When the signal driving these inputs was HIGH, the output to the screen would be white, and when the input was LOW, the output would be black. These inputs were driven by an OR gate, which OR’ed together the output of the ship\_pos, asteroid\_pos, and clk\_pos signals. This allowed all three of these elements to output to the VGA screen. If any of the elements were driving that pixel white, then the pixel would be white, otherwise the pixel would be black.

The second block in this diagram is the space\_ship block. This block contained three sub-blocks, which can be seen in Fig. 3. These blocks were a keyboard block, an arrow\_control block, and the ship block.

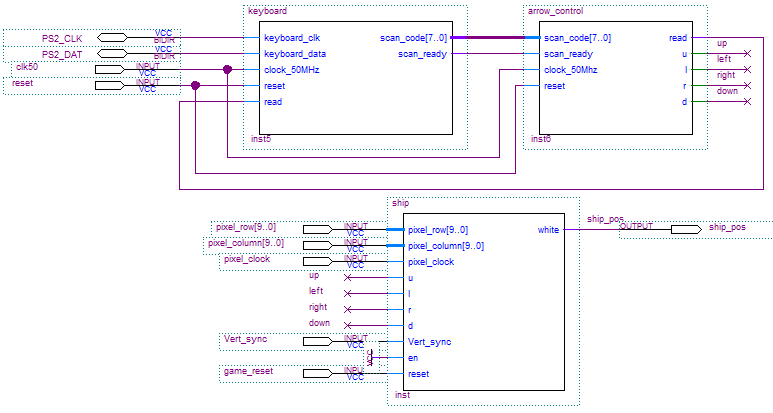


Fig. 3 Schematic for space\_ship Block

The first sub-block in the shace\_ship block is the keyboard block. This block was provided from a previous lab and no changes were made to it. The second sub-block is the arrow\_control block. This block was adapted from the color\_control block which was develop in Lab 10 from the code\_FIFO block which was provided. This block was written in VHDL. The inputs for the block were the 8-bit scan code from the keyboard, a scan ready input from the keyboard block, a 50 MHz clock, and a reset which was used to reset the internal storage in the block. The outputs of the block are a read output which is fed back to the keyboard block, and one output for each direction which is HIGH when that arrow key is pressed. The VHDL code for this block was broken into two processes. The first of these processes was a state machine which read from the keyboard block. This process was taken from the color\_control block. The second process read from the stored value to determine if one of the arrow keys had been pressed. In order to determine which key was pressed, the stored value was compared the make and break codes for each key. These code can be seen in Table 1.

Table 1. Make and Break Codes

|  |  |  |
| --- | --- | --- |
| Key | Make | Break |
| ↑ | E0 75 | E0 F0 75 |
| ↓ | E0 72 | E0 F0 72 |
| ← | E0 6B | E0 F0 6B |
| → | E0 74 | E0 F0 74 |

If least significant byte of storage was equal to the least significant byte of the make code for one of the keys and the most significant byte was not “F0” then it is known that that key was pressed. If least significant byte of storage was equal to the least significant byte of the make code for one of the keys and the most significant byte was “F0” then it is known that that key was released. Signals were used to store when a make code have been received for a key, but the corresponding break code had not been received. This indicated that the key was being pressed down. These signals were used to make sure that the output for that key was only effected when the key was first pressed and when it was released and not as it was held down. Any make code received for a key while its corresponding hold signal was HIGH was ignored. When a key was pressed the output corresponding to it would go HIGH. When that key was released, then the output corresponding to it would go LOW. The VDHL code for the arrow\_control block can be found in Appendix I.

The final sub-block in the spaceship block was the ship block. This block was responsible for controlling the motion and display of the ship graphic on the screen. The block was developed from the bouncing ball block which was provided in a previous lab. The block has inputs for the pixel\_row, pixel\_column, pixel\_clock, and vert\_sync from the VGA\_sync block, as well as inputs for the up, left, right, and down controls from the arrow\_control. The block also has a reset input which resets the ship back to its home position. The only output from this block is the white output, which is fed into the VGA\_sync block in order to display the ship. Inside the ship block, a one-port ROM stores the image of the space ship. The method for displaying the graphic was taken from the bouncing image example which was provided. This code was divided into two processes. The first process displayed the image at the correct point on the screen based on its position. This process was taken directly from the bouncing ball example with no modifications. This process works by comparing the ship’s position based on its center position and its size, with the row and column of the pixel currently being drawn. If the current pixel is within the boundary of a square with side length of *2\*size + 1*, centered around the x and y position of the ship, then the ball\_on signal was set HIGH, indicating that that pixel should be white, otherwise ball\_on was set LOW, indicating that pixel should be black. The second process was designed to move the position of the ball based on the inputs from the arrow\_control block. The sensitivity list for this block was based only on the vert\_sync signal from the VGA\_sync block. This meant that the ship would move once per screen refresh. Inside this process the reset signal was first read, to check if the ship should return to its home position. The home position of the ship was defined to be (320, 410) which was centered in the x-direction, and on the lower part of the screen. If reset signal was HIGH, and there was a positive edge from the vert\_sync signal, the process would then check each of the inputs for the arrow\_control block and update the position of the ship accordingly. To update the position of the ball, first the input from the arrow\_control block are read using a series of if statements. If the up input signal is high, then a value of -2 is added to the y-position of the ship. This is because the pixel rows are numbered top to bottom, so lower the y-position value of the ship moves it upward. Likewise if the down input is HIGH, then a value of 2 is added to the y-position of the ship. Similarly if the left or right inputs are high a -2 or 2 is added to the ships x-position respectively. Since it the ship cannot move both up and down at the same time, down was given priority over up when both are pressed. Likewise, left was given priority over right. In order to display a graphic instead of a square at the position of the ship, a ROM element was used. To determine where in the ROM to read from the signals Image\_Y\_pos and Image\_X\_pos were used to determine the relative position of the ship and the current pixel being read. The address of the ROM to be read from was taken from the five least significant bits of the Image\_Y\_pos. The 5 least significant bits of the Image\_X\_pos were then used to take a specific bit from the ROM data which was AND’ed with the ball\_on to control the output white, this was drew the ship on the screen. The VHDL code for the ship block can be seen in Appendix II. The .mif file which specified the contents of the ROM for the ship graphic can be seen in Appendix III.

The next block in the top-level schematic which will be discussed is the asteroids block. This block displays the asteroid images on the screen. It also increases the difficulty of the game the longer the player plays by increasing the number of asteroids as time goes on. The schematic for this block can be seen in Fig. 4. This schematic contains six small asteroid blocks and six medium asteroid blocks. Fig. 5 shows a schematic zoomed in on the top of the main schematic. This shows the detail at the top of the schematic. All asteroids are wired in the same way, with the exception of the compare values on each compare block.

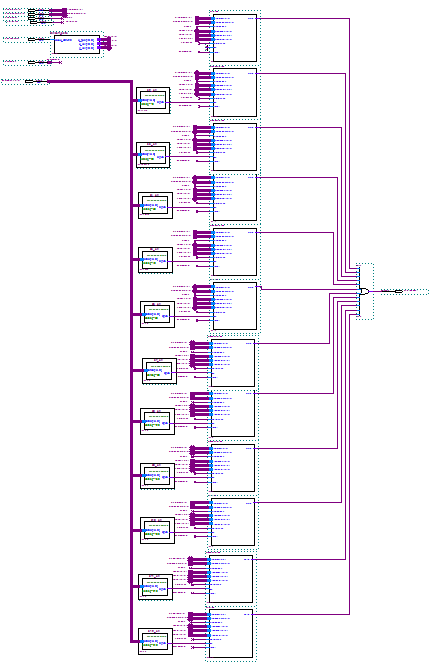


Fig. 5 Whole Schematic of asteroids Block

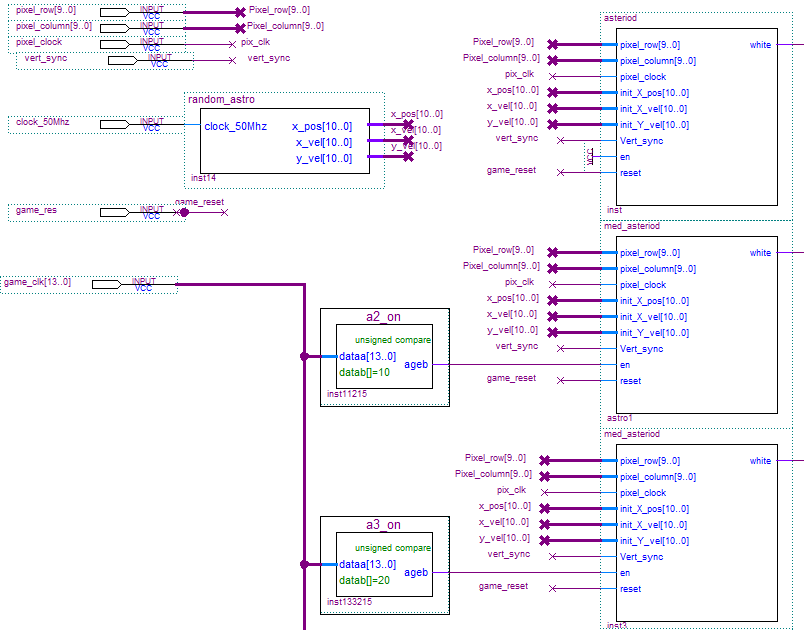


Fig. 6 Top-Left Schematic of asteroids Block

There are three type of sub-blocks within the Asteroids block. These are the random\_astros block, the asteroid block, and the med\_asteriod block. The first of these blocks which will be discussed is the random\_astro block. The input to this block is the 50 MHz clock. The block has outputs for the initial x-position, x-velocity, and y-velocity for the asteroids. The purpose of this block is to provide initial values for each asteroid when the re-spawn after leaving the screen. The schematic for the random\_astro block can be seen in Fig. 7.

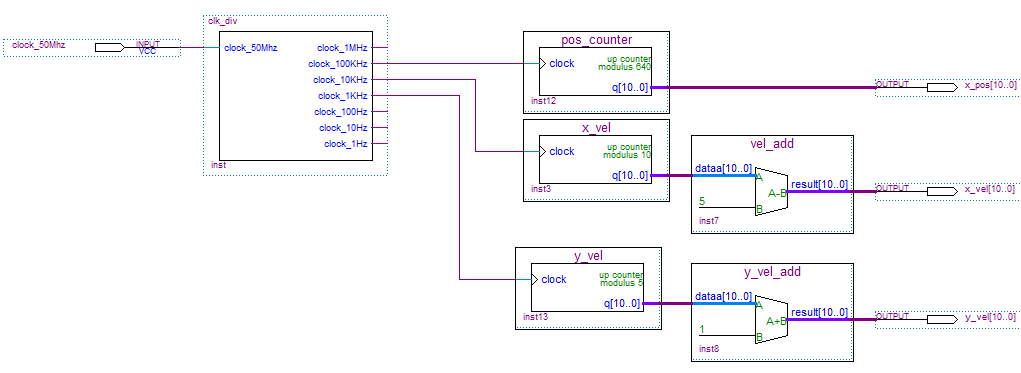


Fig. 7 Schematic of random\_astro Block

The random\_astro block is made up of Altera LPM Megafunctions and the clk\_div block which was provided. The principle of the block is that three counters of different mods are run off of three different clock speeds. This creates a pseudorandom pattern of combinations. The x-position counter was chosen to have a mod of 640 since the screen is 640 pixels wide, so this gives the asteroid equal chance of starting on any pixel at the top of the screen. The x-position counter was connected to the 100 kHz output of the clk\_div block. The x-velocity counter was chosen to have a mod of 10. The output of this counter was then attached to an adder which added -5 to the output of the counter. This allowed for a final output between -5 and 4. This allows the asteroids to move in either the left or right direction. The x-velocity counter was connected to the 10 kHz output of the clk\_div block. The y-velocity counter was chosen to have a mod of 5. The output of this counter was then connected to an adder which added 1 to the output of the counter. This allowed for a final output of 1 to 5. The addition of 1 was necessary to prevent asteroids with no y-velocity, which means that they would never interact with the game area.

The other two sub-blocks in the asteroids block, the asteroid block and the med\_asteroid block, are very similar to one another. The only difference between these blocks is the size of the asteroid which they produce. The asteroid block produces an asteroid with a diameter of 69 pixels, while the med\_asteroid block produces an asteroid with a diameter of 141 pixels. The VHDL code for the asteroids block can be seen in Appendix IV and its .mif file can be seen in Appendix V. The VHDL code for the med\_asteroid block can be seen Appendix VI and its .mif file in Appendix VII. Because of the similarities between these blocks, only the asteroid block will be discussed.

Like the ship block, the asteroid block was develop from the bouncing ball and bouncing image blocks. The block has inputs for the pixel\_row, pixel\_column, pixel\_clock, and vert\_sync from the VGA\_sync block, as well as inputs for the up, left, right, and down controls from the arrow\_control. The block also has a reset and enable inputs. The only output from this block is the white output, which is fed into the VGA\_sync block in order to display the ship. The asteroid block was very similar to the ship block, so only the differences between these blocks is discussed. The code for this block was divided into two processes. The first process displayed the image at the correct point on the screen based on its position. The only difference in this process and the similar process in the ship block is the addition of a buffer. This buffer is used to allow the asteroid to smoothly go off screen before resetting. In order to account for this buffer within the display process, the buffer size, which was equal to twice the size of the asteroid, was added to the right hand side of each of the inequalities. This accounted for the fact that the top left corner of the screen was at position (buffer\_size, buffer\_size) instead of (0,0). Aside from this change, the display process is unchanged from the bouncing ball example. The second process changes the position of the ball. This process is sensitive to the vert\_sync signal from the VGA\_sync block. This block first checks the reset and off\_edge signals to determine if the asteroid should re-spawn. If either of these signals is LOW, then the x-position as well as the x and y-velocity are set to their respective inputs from the random\_astro block. The y-position of the asteroid is determined by the init\_Y\_pos signal which is equal to the size of the asteroid. The off\_edge signal is also reset back to HIGH. If both reset and off\_edge are HIGH, then it is checked if the asteroid is completely off the edge of the screen. This is done by comparing if the bottom edge of the asteroid is past the bottom edge of the buffer or is the left or right edge are past the left or right edge of the buffer respectively. If any of these are the case then the off\_edge signal is set LOW, indicating that the asteroid should re-spawn. If none of these are the case, then the position of the asteroid is updated based on its x and y-velocity. This is done by adding the y-velocity to the y-position of the asteroid, and the x-velocity of the asteroid to the x-position of the asteroid. The drawing of the graphic is done in the same way as described in the description of the ship block. An additional constant of 5 was added to the Image\_Y\_pos and Image\_X\_pos in order to shift the image to be centered on the asteroids center position.

As can be seen in Figs. 5 and 6, the enables for all but the first asteroid were tied to the output of a compare block. These compare blocks were used to control when each of the asteroids activated. This is done by comparing the game clock, which increments every second, to a constant which is different for each asteroid. The output of the compare blocks is greater than or equal to, so it is high whenever the counter exceeds its compare delay. By making it so that more asteroids are enabled as the game goes on, the difficulty of the game increases with time. The delay for each asteroid can be seen in Table 2.

Table 2. Asteroid Delays

|  |  |
| --- | --- |
| Asteroid Number | Delay (s) |
| 1 | - |
| 2 | 10 |
| 3 | 20 |
| 4 | 30 |
| 5 | 40 |
| 6 | 50 |
| 7 | 90 |
| 8 | 120 |
| 9 | 150 |
| 10 | 180 |
| 11 | 210 |
| 12 | 240 |

From Figs. 5 and 6 it can also be seen that the outputs of each of the asteroids is OR’ed together to from the white output for the asteroids block. This allows all of the asteroids to be displayed by the VGA\_sync block.

The next block from the top-level schematic to be discussed is the game\_clk block. This block has inputs for the 50 Mhz clock and a reset input. As well as inputs for pixel\_row, pixel\_column, and pixel\_clock from the VGA\_sync block. The schematic for the game\_clk block can be seen in Fig. 8.

The sub-blocks which make up the game\_clk block are a clock divider, a LPM counter, a num\_to\_digits block, and four digit\_display blocks. The LPM counter was set up to be a 14 bit counter with an asynchronous reset. The clock to the counter was the 1 Hz output of the clk\_div block. The output of the counter was an output of the game\_clk block.

The num\_to\_digits block was developed in VHDL. The purpose was to convert the input number to four binary-coded decimal (BCD) digits. The input to the block was the 14-bit number output form the game\_counter. The outputs were four BCD digits. To accomplish this the number input was first converted to an integer. The individual digits were extracted by first dividing the input number by the number representation of the place of the digit desired, then using a *mod 10* to remove the all but the ones digit of the result. For example, to find the 10’s digit of the input number (1) was used. The individual digits were then converted back to standard logic vectors before being outputted. The rest of these equation can be seen in the VHDL code for this block which can be found in Appendix VIII.

(1)

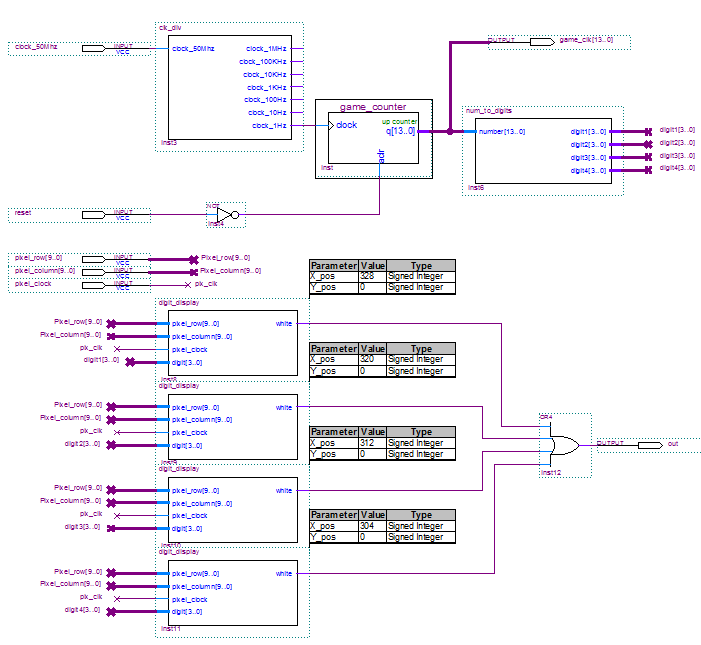


Fig. 8 Schematic for game\_clk Block

The four digit\_display blocks are exactly the same aside of the different parameters for their x and y-positions which can be seen in Fig. 8. Unlike the ship and asteroid blocks, where the position refers to the center of the shape, the position of the digit\_display block refers the top left corner of the digit. This was done because it was determined that it was simpler to visualize where the digit would be based on its top left corner than it would be to based on its center. It was decided to use parameters to determine the x and y-position of the digit because these digits are stationary, and thus the position is fixed so it can defined by the user as a parameter. Also, the use of parameters allowed for a single VHDL file to be used for all the desired digits even though each digit needed to have a different position.

The VHDL for the digit\_display block was adapted from the ship block. However the process to move the ship was removed since the digits are stationary. ROM was once again used to define the digits. The .mif file was configured in such a way that the six most significant bits of the address were the same as the number to be displayed, and the 3 least significant bits referred to each line of the graphic for that digit. The address in ROM was thus assigned to the digit input concatenated with the three least significant bits of the Image\_Y\_pos. The logic inside the display processes was changed to account for the position being the top left corner instead of the center of the graphic. This significantly simplified the logic to determine when the pixel being written was within the graphic’s area. The three least significant bits of the Image\_X\_pos were subtracted from 7, then used to take a specific bit from the ROM data which was AND’ed with the out\_on to control the output white in order to draw the digit to the screen. The subtraction was done to reverse the memory contents as it was found that the digits which were taken from the char\_rom example were written right to left instead of left to right in the .mif file. The VHDL code for the digit\_display block can be found in Appendix IX and its .mif file can be seen in Appendix X.

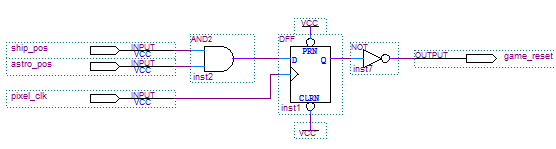


Fig. 9 Schematic for collision\_detect Block

The final block from the top-level schematic to be discussed is the collision\_detect block. The purpose of this block is to detect when the ship has collided with an asteroid and the reset the game when this occurs. The inputs to this block were ship\_pos and astro\_pos, which represented the pixel-by-pixel output of the spaceship and asteroids block. As well as the pixel\_clk from the VGA\_sync block. The schematic for this block can be seen in Fig. 9. In order to determine if the ship has collided with an asteroid, the pixel-by-pixel output for the spaceship and asteroids blocks were AND’ed together. The output of this AND gate would be HIGH whenever both the spaceship and asteroids were setting the same pixel to white. This meant that the two had collided. A D flip-flop using the pixel\_clk as its clock input was used to synchronize this AND gate and prevent glitches. The output of the flip-flop was NOT’ed in order to make the reset active LOW.

**Results:**

All of the requirements which I set out for this project were accomplished. The user is able to control the spaceship using the arrow keys of the keyboard in order to avoid asteroids. When the user’s spaceship collides with an asteroid the game ends and resets. The additional features of enhanced graphics and score keeping were also added.

Unfortunately however, not all the additional features were able to be implemented at this time. The most notable of these is the ability to control the spaceship using an ultrasonic sensor. While this would have been a very interesting feature, time restraints did not allow me fully explore how this would be implemented and instead I opted to implement other additional features such as graphics and scorekeeping. In addition, the additional feature of lasers to destroy asteroids was also not able to be implemented in the allotted time. I hope to implement these features at a later time in order to enhance game play.

**Conclusions/Discussion:**

From this project I have learned a variety of important insights about Digital System design.

* I learned how to break a large design project into smaller pieces to make the design more manageable. This is important because trying the design an entire system using a single block can be unwieldy and difficult to debug, by splitting a project into smaller pieces each piece can be tested and the development and debugging time required can be reduced.
* I learned to modify existing examples instead of trying to develop a block from scratch. For example, had I tried to create the ship and asteroids blocks from scratch, especially the ROM graphics portion, this project would have taken significantly longer.
* I learned how to use existing modules, such as the clk\_div module and LPM Megafunctions, to add functions to our circuits with minimal extra work. This is important because using modules allows us to speed development of our circuits.
* I learned that nesting blocks in a hierarchy can be used to create a simpler top-level design which is easier to follow. This is important because when others read our schematics we want them to be easy to understand.
* I learned how to use parameters to change values in a VHDL block without having to have separate VHDL files. This is important because we often desire to have a block where some value can be changed when setting up the block. The use a parameters allows us to do this without having to create and save a new VHDL file just to change a single value.
* I learned that using D flip-flops to synchronize signals in a circuit can be very useful in preventing glitches. For example, I have having a lot of problems with glitches in my collision\_detect block. By using a D flip-flop to synchronize the output of the AND gate with the pixel clock I was able to remove these glitches by ensuring that a reset signal would only be sent when a new pixel had been loaded.

**Appendix I: ARROW\_CONTROL BLOCK**

**LIBRARY IEEE;**

**USE IEEE.STD\_LOGIC\_1164.all;**

**USE IEEE.STD\_LOGIC\_ARITH.all;**

**USE IEEE.STD\_LOGIC\_UNSIGNED.all;**

**-- buffer for incoming keyboard keycode data**

**ENTITY arrow\_control IS**

**PORT(scan\_code : IN STD\_LOGIC\_VECTOR(7 DOWNTO 0); --key code from keyboard block**

**scan\_ready, clock\_50Mhz, reset : IN std\_logic;**

**read : OUT std\_logic;**

**u, l, r, d : OUT std\_logic);**

**END arrow\_control;**

**ARCHITECTURE a OF arrow\_control IS**

**TYPE STATE\_TYPE IS (wait\_ready, read\_data, read\_low);**

**SIGNAL state: STATE\_TYPE;**

**SIGNAL storage :STD\_LOGIC\_VECTOR(15 DOWNTO 0);**

**SIGNAL u\_make :STD\_LOGIC;**

**SIGNAL l\_make :STD\_LOGIC;**

**SIGNAL r\_make :STD\_LOGIC;**

**SIGNAL d\_make :STD\_LOGIC;**

**BEGIN**

**PROCESS (scan\_ready, reset, clock\_50Mhz)**

**BEGIN**

**IF reset <= '0' THEN state <= read\_low;**

**storage <= X"0000"; -- reset storage to 3 bytes of zeros**

**ELSIF (clock\_50Mhz'EVENT) AND clock\_50Mhz='1' THEN**

**CASE state IS**

**WHEN read\_low => -- temporary state**

**read <= '0'; -- reset the "read" pulse output back to low**

**state <= wait\_ready;**

**WHEN wait\_ready => -- waiting for new incoming data**

**IF scan\_ready = '1' THEN -- if a new keycode has come in**

**read <= '1'; -- set the "read" output to pulse high**

**state <= read\_data; -- for 2 clock pulses**

**ELSE**

**state <= wait\_ready; -- otherwise keep waiting**

**END IF;**

**WHEN read\_data => -- temporary state: add new byte to output buffer**

**-- shift left the first bytes, concatenate the new scan code as LSB**

**storage <= storage(7 DOWNTO 0) & Scan\_Code;**

**state <= read\_low; -- "read" output is still high during this state**

**END CASE;**

**END IF;**

**END PROCESS;**

**--key make break**

**--'U Arrow' E0 75 E0 F0 75**

**--'L Arrow' E0 6B E0 F0 6B**

**--'R Arrow' E0 74 E0 F0 74**

**--'D Arrow' E0 72 E0 F0 72**

**PROCESS (clock\_50Mhz, reset)**

**BEGIN**

**IF (reset = '0') THEN --reset all make and output variables**

**u\_make <= '0';**

**l\_make <= '0';**

**r\_make <= '0';**

**d\_make <= '0';**

**u <= '0';**

**l <= '0';**

**r <= '0';**

**d <= '0';**

**ELSIF (clock\_50Mhz'EVENT) AND clock\_50Mhz='1' THEN --syncronous**

**IF (storage(15 DOWNTO 8) /= X"F0") THEN --make sure it is not a break code**

**CASE storage(7 DOWNTO 0) IS --check which make code it was**

**WHEN X"75" => --"UP ARROW"**

**IF (u\_make = '0') THEN**

**u\_make <= '1'; --set u\_make**

**u <= '1';**

**END IF;**

**WHEN X"6B" => --"LEFT ARROW"**

**IF (l\_make = '0') THEN**

**l\_make <= '1'; --set l\_make**

**l <= '1';**

**END IF;**

**WHEN X"74" => --"RIGHT ARROW"**

**IF (r\_make = '0') THEN**

**r\_make <= '1'; --set r\_make**

**r <= '1';**

**END IF;**

**WHEN X"72" => --"DOWN ARROW"**

**IF (d\_make = '0') THEN**

**d\_make <= '1'; --set blue\_make**

**d <= '1';**

**END IF;**

**WHEN OTHERS =>**

**null;**

**END CASE;**

**ELSE**

**CASE storage(7 DOWNTO 0) IS**

**WHEN X"75" =>**

**u\_make <= '0'; --clear u\_make**

**u <= '0';**

**WHEN X"6B" =>**

**l\_make <= '0'; --clear l\_make**

**l <= '0';**

**WHEN X"74" =>**

**r\_make <= '0'; --clear r\_make**

**r <= '0';**

**WHEN X"72" =>**

**d\_make <= '0'; --clear d\_make**

**d <= '0';**

**WHEN OTHERS =>**

**null;**

**END CASE;**

**END IF;**

**END IF;**

**END PROCESS;**

**END a;**

**Appendix II: SHIP BLOCK**

**LIBRARY IEEE;**

**USE IEEE.STD\_LOGIC\_1164.all;**

**USE IEEE.STD\_LOGIC\_ARITH.all;**

**USE IEEE.STD\_LOGIC\_UNSIGNED.all;**

**ENTITY ship IS**

**PORT(pixel\_row, pixel\_column : IN std\_logic\_vector(9 DOWNTO 0);**

**pixel\_clock : IN std\_logic;**

**u,l,r,d : IN std\_logic;**

**Vert\_sync : IN std\_logic;**

**reset : IN std\_logic;**

**white : OUT std\_logic);**

**END ship;**

**architecture behavior of ship is**

**SIGNAL Ball\_on, Direction : std\_logic;**

**SIGNAL Size : std\_logic\_vector(10 DOWNTO 0);**

**SIGNAL Ball\_Y\_motion, Ball\_X\_motion : std\_logic\_vector(10 DOWNTO 0);**

**SIGNAL Ball\_Y\_pos : std\_logic\_vector(10 DOWNTO 0) := CONV\_STD\_LOGIC\_VECTOR(410,11);**

**SIGNAL Ball\_X\_pos : std\_logic\_vector(10 DOWNTO 0) := CONV\_STD\_LOGIC\_VECTOR(320,11);**

**SIGNAL Image\_Y\_pos, Image\_X\_pos : std\_logic\_vector(10 DOWNTO 0); -- keep track of which pixel of the image is being displayed**

**SIGNAL image\_data : std\_logic\_vector(30 DOWNTO 0); -- 30-bit data from the image rom**

**SIGNAL rom\_addr : std\_logic\_vector(4 DOWNTO 0); -- address bits for rom**

**--rom to store ship graphic**

**COMPONENT ship\_graphic**

**PORT**

**(**

**address : IN STD\_LOGIC\_VECTOR (4 DOWNTO 0);**

**clock : IN STD\_LOGIC;**

**q : OUT STD\_LOGIC\_VECTOR (30 DOWNTO 0)**

**);**

**END COMPONENT;**

**BEGIN**

**Size <= CONV\_STD\_LOGIC\_VECTOR(15,11); --size is measured from center position out**

**-- White Ship**

**white <= ball\_on and image\_data((CONV\_INTEGER(Image\_X\_pos(4 downto 0))));**

**-- the image pixels are determined relative to the "ball" position and the CRT pixel position**

**Image\_Y\_pos <= pixel\_row - Ball\_Y\_pos + Size;**

**Image\_X\_pos <= pixel\_column - Ball\_X\_pos + Size;**

**RGB\_Display: Process (pixel\_column, pixel\_row) --(reset, Ball\_X\_pos, Ball\_Y\_pos, pixel\_column, pixel\_row, Size)**

**BEGIN**

**-- Set Ball\_on ='1' to display ball**

**IF ('0' & Ball\_X\_pos <= pixel\_column + Size) AND**

**(Ball\_X\_pos + Size >= '0' & pixel\_column) AND**

**('0' & Ball\_Y\_pos <= pixel\_row + Size) AND**

**(Ball\_Y\_pos + Size >= '0' & pixel\_row ) THEN**

**Ball\_on <= '1';**

**ELSE**

**Ball\_on <= '0';**

**END IF;**

**END process RGB\_Display;**

**Move\_Ball: Process (vert\_sync)**

**BEGIN**

**IF (reset = '0') THEN**

**Ball\_Y\_pos <= CONV\_STD\_LOGIC\_VECTOR(410,11); --home y position**

**Ball\_X\_pos <= CONV\_STD\_LOGIC\_VECTOR(320,11); --home x positive**

**-- Move ball once every vertical sync**

**ELSIF (vert\_sync'event) and (vert\_sync = '1') THEN**

**-- Compute next ball Y position based on u and d**

**IF (u = '1') and (Ball\_Y\_pos > Size) THEN**

**Ball\_Y\_pos <= Ball\_Y\_pos + CONV\_STD\_LOGIC\_VECTOR(-2,11);**

**END IF;**

**--d takes presidence over u**

**IF (d = '1') and (('0' & Ball\_Y\_pos) < 480 - Size) THEN**

**Ball\_Y\_pos <= Ball\_Y\_pos + CONV\_STD\_LOGIC\_VECTOR(2,11);**

**END IF;**

**--compute next ball X position based on r and l**

**IF (r='1') and (('0' & Ball\_X\_pos) < 640 - Size) THEN**

**Ball\_X\_pos <= Ball\_X\_pos + CONV\_STD\_LOGIC\_VECTOR(2,11);**

**END IF;**

**--l takes presidence over r**

**IF (l='1') and (Ball\_X\_pos > Size) THEN**

**Ball\_X\_pos <= Ball\_X\_pos + CONV\_STD\_LOGIC\_VECTOR(-2,11);**

**END IF;**

**END IF;**

**END process Move\_Ball;**

**-- instantiate the rom and hook up the signals**

**ship\_graphic\_inst : ship\_graphic PORT MAP (**

**address => rom\_addr,**

**clock => pixel\_clock,**

**q => image\_data**

**);**

**-- rom address**

**-- MSB is from switch and selects which image to display**

**-- other 3 bits select rom of image to display**

**rom\_addr <= Image\_Y\_pos(4 DOWNTO 0);**

**END behavior;**

**Appendix III: SHIP\_GRAPHIC.MIF FILE**

**Depth = 31;**

**Width = 31;**

**Address\_radix = dec;**

**Data\_radix = bin;**

**Content**

**Begin**

**00 : 0000000000000001000000000000000;**

**01 : 0000000000000001000000000000000;**

**02 : 0000000000000011100000000000000;**

**03 : 0000000000000011100000000000000;**

**04 : 0000000000000111110000000000000;**

**05 : 0000000000001111111000000000000;**

**06 : 0000000000001111111000000000000;**

**07 : 0000000000011111111100000000000;**

**08 : 0000000000011111111100000000000;**

**09 : 0000000000111111111110000000000;**

**10 : 0000000000111111111110000000000;**

**11 : 0000000000111111111110000000000;**

**12 : 0000000001111111111111000000000;**

**13 : 0000000001111111111111000000000;**

**14 : 0000000011111111111111100000000;**

**15 : 0000000011111111111111100000000;**

**16 : 0000000111111111111111110000000;**

**17 : 0000000111111111111111110000000;**

**18 : 0000001111111111111111111000000;**

**19 : 0000001111111111111111111000000;**

**20 : 0000011111111111111111111100000;**

**21 : 0000011111111111111111111100000;**

**22 : 0000111111111111111111111110000;**

**23 : 0000111111111111111111111110000;**

**24 : 0001111111111111111111111111000;**

**25 : 0001111111111111111111111111000;**

**26 : 0011111111111111111111111111100;**

**27 : 0011111111111111111111111111100;**

**28 : 0111111111111111111111111111110;**

**29 : 1111111111111111111111111111111;**

**30 : 1111111111111111111111111111111;**

**End;**

**Appendix IV: ASTEROID BLOCK**

**LIBRARY IEEE;**

**USE IEEE.STD\_LOGIC\_1164.all;**

**USE IEEE.STD\_LOGIC\_ARITH.all;**

**USE IEEE.STD\_LOGIC\_UNSIGNED.all;**

**ENTITY asteriod IS**

**PORT(pixel\_row, pixel\_column : IN std\_logic\_vector(9 DOWNTO 0);**

**pixel\_clock : IN std\_logic;**

**init\_X\_pos : IN std\_logic\_vector(10 DOWNTO 0);**

**init\_X\_vel, init\_Y\_vel : IN std\_logic\_vector(10 DOWNTO 0);**

**Vert\_sync : IN std\_logic;**

**en : IN std\_logic;**

**reset : IN std\_logic;**

**white : OUT std\_logic);**

**END asteriod;**

**architecture behavior of asteriod is -- Video Display Signals**

**SIGNAL Ball\_on, Direction : std\_logic;**

**SIGNAL Ball\_Y\_motion, Ball\_X\_motion : std\_logic\_vector(10 DOWNTO 0);**

**SIGNAL Ball\_Y\_pos, Ball\_X\_pos : std\_logic\_vector(10 DOWNTO 0);**

**SIGNAL init\_Y\_pos : std\_logic\_vector(10 DOWNTO 0);**

**SIGNAL buffer\_size : std\_logic\_vector(10 DOWNTO 0);**

**SIGNAL Size : std\_logic\_vector(10 DOWNTO 0);**

**SIGNAL Image\_Y\_pos, Image\_X\_pos : std\_logic\_vector(10 DOWNTO 0); -- keep track of which pixel of the image is being displayed**

**SIGNAL image\_data : std\_logic\_vector(68 DOWNTO 0); -- 69-bit data from the image rom**

**SIGNAL rom\_addr : std\_logic\_vector(6 DOWNTO 0); -- address bits for rom**

**SIGNAL off\_edge : std\_logic; --stores if ship should re-spawn**

**--ROM for asteroid graphic**

**COMPONENT small\_astro**

**PORT**

**(**

**address : IN STD\_LOGIC\_VECTOR (6 DOWNTO 0);**

**clock : IN STD\_LOGIC;**

**q : OUT STD\_LOGIC\_VECTOR (68 DOWNTO 0)**

**);**

**END COMPONENT;**

**BEGIN**

**Size <= CONV\_STD\_LOGIC\_VECTOR(34,11); --radius of asteroid**

**buffer\_size <= Size(9 downto 0) & '0'; --multiply size by 2 to get buffer size**

**init\_Y\_pos <= Size; --start asteriod in middle of buffer**

**-- White Asteriod**

**white <= en and ball\_on and image\_data((CONV\_INTEGER(Image\_X\_pos(6 downto 0))));**

**-- the image pixels are determined relative to the "ball" position and the CRT pixel position**

**Image\_Y\_pos <= pixel\_row - Ball\_Y\_pos - Size + 5;**

**Image\_X\_pos <= pixel\_column - Ball\_X\_pos - Size + 5;**

**RGB\_Display: Process (pixel\_column, pixel\_row)**

**BEGIN**

**-- Set Ball\_on ='1' to display ball**

**IF (Ball\_X\_pos <= pixel\_column + Size + buffer\_size) AND**

**(Ball\_X\_pos + Size >= pixel\_column + buffer\_size) AND**

**(Ball\_Y\_pos <= pixel\_row + Size + buffer\_size) AND**

**(Ball\_Y\_pos + Size >= pixel\_row + buffer\_size) THEN**

**Ball\_on <= '1';**

**ELSE**

**Ball\_on <= '0';**

**END IF;**

**END process RGB\_Display;**

**Move\_Ball: process (vert\_sync)**

**BEGIN**

**IF (reset = '0') or (off\_edge ='0') THEN**

**Ball\_Y\_pos <= Init\_Y\_pos;**

**Ball\_X\_pos <= Init\_X\_pos;**

**Ball\_Y\_motion <= init\_Y\_vel;**

**Ball\_X\_motion <= init\_X\_vel;**

**off\_edge <= '1';**

**-- Move ball once every vertical sync**

**ELSIF (vert\_sync'event) and (vert\_sync = '1') THEN**

**IF (en = '1') THEN**

**--if bottom edge is past boundary**

**IF ('0' & Ball\_Y\_pos) >= 480 + Size + buffer\_size THEN**

**off\_edge <= '0';**

**--if right edgeis past boundary**

**ELSIF ('0' & Ball\_X\_pos) >= 640 + Size + buffer\_size THEN**

**off\_edge <= '0';**

**--if left edge is past boundary**

**ELSIF Ball\_X\_pos <= Size THEN**

**off\_edge <= '0';**

**ELSE**

**-- Compute next ball Y position**

**Ball\_Y\_pos <= Ball\_Y\_pos + Ball\_Y\_motion;**

**-- Compute next ball X position**

**Ball\_X\_pos <= Ball\_X\_pos + Ball\_X\_motion;**

**END IF;**

**END IF;**

**END IF;**

**END process Move\_Ball;**

**-- instantiate the rom and hook up the signals**

**small\_astro\_inst : small\_astro PORT MAP (**

**address => rom\_addr,**

**clock => pixel\_clock,**

**q => image\_data**

**);**

**-- rom address**

**-- 7 bits select rom of image to display**

**rom\_addr <= Image\_Y\_pos(6 DOWNTO 0);**

**END behavior;**

**Appendix V: SMALL\_ASTRO.MIF FILE**

**Depth=69;**

**Width=69;**

**Address\_radix=dec;**

**Data\_radix=bin;**

**Content**

**BEGIN**

**00:000000000000000000000000000000000000000000000000000000000000000000000;**

**01:000000000000000000000000000000000000000000000000000000000000000000000;**

**02:000000000000000000000000000000000000000000000000000000000000000000000;**

**03:000000000000000000000000000000000000000000000000000000000000000000000;**

**04:000000000000000000000000000000001111111100000000000000000000000000000;**

**05:000000000000000000000000000001111111111111110000000000000000000000000;**

**06:000000000000000000000001111111110000000011111111000000000000000000000;**

**07:000000000000000000011111111111000000000000001111110000000000000000000;**

**08:000000000000000001111111000000000000000000000001111000000000000000000;**

**09:000000000000000001110000000000000000000000110000011110000000000000000;**

**10:000000000000000111000000000000000000000000111100000111000000000000000;**

**11:000000000000001110000000000000000000000000110011110011100000000000000;**

**12:000000000000001100000000000000000000000000000001001001110000000000000;**

**13:000000000000011100000110000001110100000000000000001110110000000000000;**

**14:000000000000011000001001011111000000011000000000000010111000000000000;**

**15:000000000001111000011001110000000000001111100000000001111000000000000;**

**16:000000000111100011000000010000000000000010100000000000011100000000000;**

**17:000000001111001111000000110000000000000001110000000000001110000000000;**

**18:000000011100001010000001100000000000000000001000000000000111000000000;**

**19:000000011000011110000001000000000000000000001111100000000011100000000;**

**20:000000110000111100000010000000000000000000001000100000000000110000000;**

**21:000001110000100100000011000000000000000001011000010000000000111000000;**

**22:000001100000100111111101000000000000000011110000001000000000011000000;**

**23:000011100011000000100001000000000000000100000000001000000000011100000;**

**24:000011100010000000000001100000000000000010000000011000000000001100000;**

**25:000011000000000000000000010000000000000011011111111110000000001100000;**

**26:000011000010000000000000001000000000000000100000000000000000001110000;**

**27:000111000111000000000000001100000000000000000000000001000000000110000;**

**28:000011000111000000000000001100000000000000000000000001110000000110000;**

**29:000111000000000000000000000110000000000000000000000000011000000111000;**

**30:000011110000000000000000000100000000000000000000000000001100000011000;**

**31:000011001000000000000000000000000000000000000000000000000100000011000;**

**32:000011101000000000000000000000000000000000000000000000000000000011000;**

**33:000111110000000000000000000000000000000000000000000000000000000011100;**

**34:001110000000000000000000000001000000000000000000000000000000000011000;**

**35:001100000000000000000000000000001000000000000000000000000111000011000;**

**36:001100000000000000000000000000000000000000000000000000000011000011000;**

**37:001100000000000000000000000000000001000000000000000000000010000011000;**

**38:001100000000000000000000000000000000111000000000100000000000000110000;**

**39:001100000000010000000000000000000000000100001111011000000000000011000;**

**40:001110000000011000000000000000000000000010111100000110000000000110000;**

**41:000110000000011000000000000000000000000001100000000011000000000111000;**

**42:000111000000010000000000000000000000000000100000000001000000000111000;**

**43:000011000000000100000000000000000000000000010000000000100000000011000;**

**44:000011100000000011000000000000000000000000010000000000100000000011000;**

**45:000001100000000011011000000000000000000000010000000000100000000011100;**

**46:000001100000000001000100000000000000000000000100000001000000000011000;**

**47:000001110000000000100110000000000000001100000011111111100000000011000;**

**48:000000110000000000110001100000000000001110000000000000111000000011000;**

**49:000000111000000000001100010000000000000110000000000000001000000011000;**

**50:000000011100000000000011001000000000000000100000000000000100000011000;**

**51:000000001100000000000001101100000000000000010001111101000010000111000;**

**52:000000001110000000000000010100000000000000011010000100110000001110000;**

**53:000000000111000000000000011010000000000000000100000000010000001110000;**

**54:000000000011100000000000000011000000000000000100000000011001111100000;**

**55:000000000001111000000000000000100000000000000010000000001111111000000;**

**56:000000000000011100000000000000110000000000000011100001111111100000000;**

**57:000000000000001111000000000000010000000000000000010011111100000000000;**

**58:000000000000000011110000000000010000000000000000001111000000000000000;**

**59:000000000000000001111110000011110000000000000000001110000000000000000;**

**60:000000000000000000001111111111111110000000000000011100000000000000000;**

**61:000000000000000000000001111110001111100000000000111100000000000000000;**

**62:000000000000000000000000000000000011111110100011110000000000000000000;**

**63:000000000000000000000000000000000000111111111111100000000000000000000;**

**64:000000000000000000000000000000000000000011111100000000000000000000000;**

**65:000000000000000000000000000000000000000000000000000000000000000000000;**

**66:000000000000000000000000000000000000000000000000000000000000000000000;**

**67:000000000000000000000000000000000000000000000000000000000000000000000;**

**68:000000000000000000000000000000000000000000000000000000000000000000000;**

**End;**

**Appendix VI: MED\_ASTEROID BLOCK**

**LIBRARY IEEE;**

**USE IEEE.STD\_LOGIC\_1164.all;**

**USE IEEE.STD\_LOGIC\_ARITH.all;**

**USE IEEE.STD\_LOGIC\_UNSIGNED.all;**

**ENTITY med\_asteriod IS**

**PORT(pixel\_row, pixel\_column : IN std\_logic\_vector(9 DOWNTO 0);**

**pixel\_clock : IN std\_logic;**

**init\_X\_pos : IN std\_logic\_vector(10 DOWNTO 0);**

**init\_X\_vel, init\_Y\_vel : IN std\_logic\_vector(10 DOWNTO 0);**

**Vert\_sync : IN std\_logic;**

**en : IN std\_logic;**

**reset : IN std\_logic;**

**white : OUT std\_logic);**

**END med\_asteriod;**

**architecture behavior of med\_asteriod is -- Video Display Signals**

**SIGNAL Ball\_on, Direction : std\_logic;**

**SIGNAL Ball\_Y\_motion, Ball\_X\_motion : std\_logic\_vector(10 DOWNTO 0);**

**SIGNAL Ball\_Y\_pos, Ball\_X\_pos : std\_logic\_vector(10 DOWNTO 0);**

**SIGNAL init\_Y\_pos : std\_logic\_vector(10 DOWNTO 0);**

**SIGNAL buffer\_size : std\_logic\_vector(10 DOWNTO 0);**

**SIGNAL Size : std\_logic\_vector(10 DOWNTO 0);**

**SIGNAL Image\_Y\_pos, Image\_X\_pos : std\_logic\_vector(10 DOWNTO 0); -- keep track of which pixel of the image is being displayed**

**SIGNAL image\_data : std\_logic\_vector(140 DOWNTO 0);-- 141-bit data from the image rom**

**SIGNAL rom\_addr : std\_logic\_vector(7 DOWNTO 0); -- address bits for rom**

**SIGNAL off\_edge : std\_logic; --stores if ship should re-spawn**

**--ROM for asteroid graphic**

**COMPONENT med\_astro**

**PORT**

**(**

**address : IN STD\_LOGIC\_VECTOR (7 DOWNTO 0);**

**clock : IN STD\_LOGIC;**

**q : OUT STD\_LOGIC\_VECTOR (140 DOWNTO 0)**

**);**

**END COMPONENT;**

**BEGIN**

**Size <= CONV\_STD\_LOGIC\_VECTOR(70,11); --radius of asteroid**

**buffer\_size <= Size(9 downto 0) & '0'; --multiply size by 2 to get buffer size**

**init\_Y\_pos <= Size; --start asteriod in middle of buffer**

**-- White Asteriod**

**white <= en and ball\_on and image\_data((CONV\_INTEGER(Image\_X\_pos(7 downto 0))));**

**-- the image pixels are determined relative to the "ball" position and the CRT pixel position**

**Image\_Y\_pos <= pixel\_row - Ball\_Y\_pos - Size + 20;**

**Image\_X\_pos <= pixel\_column - Ball\_X\_pos - Size + 20;**

**RGB\_Display: Process (pixel\_column, pixel\_row)**

**BEGIN**

**-- Set Ball\_on ='1' to display ball**

**IF (Ball\_X\_pos <= pixel\_column + Size + buffer\_size) AND**

**(Ball\_X\_pos + Size >= pixel\_column + buffer\_size) AND**

**(Ball\_Y\_pos <= pixel\_row + Size + buffer\_size) AND**

**(Ball\_Y\_pos + Size >= pixel\_row + buffer\_size) THEN**

**Ball\_on <= '1';**

**ELSE**

**Ball\_on <= '0';**

**END IF;**

**END process RGB\_Display;**

**Move\_Ball: process (vert\_sync)**

**BEGIN**

**IF (reset = '0') or (off\_edge = '0') THEN**

**Ball\_Y\_pos <= Init\_Y\_pos;**

**Ball\_X\_pos <= Init\_X\_pos;**

**Ball\_Y\_motion <= init\_Y\_vel;**

**Ball\_X\_motion <= init\_X\_vel;**

**--Size <= size\_in;**

**off\_edge <= '1';**

**-- Move ball once every vertical sync**

**ELSIF(vert\_sync'event) and (vert\_sync = '1') THEN**

**IF (en = '1') THEN**

**IF ('0' & Ball\_Y\_pos) >= 480 + Size + buffer\_size THEN**

**off\_edge <= '0';**

**ELSIF ('0' & Ball\_X\_pos) >= 640 + Size + buffer\_size THEN**

**off\_edge <= '0';**

**ELSIF Ball\_X\_pos <= Size THEN**

**off\_edge <= '0';**

**ELSE**

**-- Compute next ball Y position**

**Ball\_Y\_pos <= Ball\_Y\_pos + Ball\_Y\_motion;**

**-- Compute next ball X position**

**Ball\_X\_pos <= Ball\_X\_pos + Ball\_X\_motion;**

**END IF;**

**END IF;**

**END IF;**

**END process Move\_Ball;**

**-- instantiate the rom and hook up the signals**

**med\_astro\_inst : med\_astro PORT MAP (**

**address => rom\_addr,**

**clock => pixel\_clock,**

**q => image\_data**

**);**

**-- rom address**

**-- 7 bits select rom of image to display**

**rom\_addr <= Image\_Y\_pos(7 DOWNTO 0);**

**END behavior;**

**Appendix VII: MED\_ASTRO.MIF FILE**

**Depth=141;**

**Width=141;**

**Address\_radix=dec;**

**Data\_radix=bin;**

**Content**

**BEGIN**

**000:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**001:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**003:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**004:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**005:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**006:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**007:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**008:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**009:000000000000000000000000000000000000000000000000000000000000000000000010101010000000000000000000000000000000000000000000000000000000000000000;**

**010:000000000000000000000000000000000000000000000000000000000000000001111111111111111111100000000000000000000000000000000000000000000000000000000;**

**011:000000000000000000000000000000000000000000000000000000000000001111111111111111111111111110000000000000000000000000000000000000000000000000000;**

**012:000000000000000000000000000000000000000000000000000000000000011111111111111111111111111111111000000000000000000000000000000000000000000000000;**

**013:000000000000000000000000000000000000000000000000000000001011111111110000000000110111111111111111000000000000000000000000000000000000000000000;**

**014:000000000000000000000000000000000000000000000010111111111111111110000000000000000000001111111111110000000000000000000000000000000000000000000;**

**015:000000000000000000000000000000000000000000111111111111111111111000000000000000000000000000111111111110000000000000000000000000000000000000000;**

**016:000000000000000000000000000000000000000011111111111111111111100000000000000000000000000000000111111111100000000000000000000000000000000000000;**

**017:000000000000000000000000000000000000001111111111111111011000000000000000000000000000000000000000111111110000000000000000000000000000000000000;**

**018:000000000000000000000000000000000000011111111100000000000000000000000000000000000000000000000000001111111100000000000000000000000000000000000;**

**019:000000000000000000000000000000000000111111000000000000000000000000000000000000000000000100000000000011111111000000000000000000000000000000000;**

**020:000000000000000000000000000000000001111110000000000000000000000000000000000000000000001111100000000000111111100000000000000000000000000000000;**

**021:000000000000000000000000000000000011111100000000000000000000000000000000000000000000001101111000000000001111110000000000000000000000000000000;**

**022:000000000000000000000000000000000111110000000000000000000000000000000000000000000000001001111110000000000111111000000000000000000000000000000;**

**023:000000000000000000000000000000001111100000000000000000000000000000000000000000000000001111100011111000000011111100000000000000000000000000000;**

**024:000000000000000000000000000000011111000000000000000000000000000000000000000000000000000010000001111111000000111110000000000000000000000000000;**

**025:000000000000000000000000000000111110000000000000000000000000000000000000000000000000000000000000010111100000111111000000000000000000000000000;**

**026:000000000000000000000000000000111110000000000000000000000000000010000100000000000000000000000000000000110000011111000000000000000000000000000;**

**027:000000000000000000000000000001111100000000000010000000000000001111000100000000000000000000000000000000111000001111100000000000000000000000000;**

**028:000000000000000000000000000001111000000000001111110000000000011110000000000000000000000000000000000000011110000111100000000000000000000000000;**

**029:000000000000000000000000000011110000000000011100111000001101111000000000000011100000000000000000000000000111110111110000000000000000000000000;**

**030:000000000000000000000000000111110000000000111000111001111111110000000000000001111000000000000000000000000000110111110000000000000000000000000;**

**031:000000000000000000000000001111100000000000110000011111100001000000000000000000011111111000000000000000000000011011111000000000000000000000000;**

**032:000000000000000000000000111111100000000011110000001111000000000000000000000000001111011100000000000000000000001111111000000000000000000000000;**

**033:000000000000000000000011111111000000011110000000000111000000000000000000000000000011000100000000000000000000000111111100000000000000000000000;**

**034:000000000000000000000111111100000001111000000000000111000000000000000000000000000001100110000000000000000000000011111110000000000000000000000;**

**035:000000000000000000011111111000001111110000000000000011000000000000000000000000000001111110000000000000000000000000111111000000000000000000000;**

**036:000000000000000000111111000000011111100000000000000110000000000000000000000000000000011111000000000000000000000000011111100000000000000000000;**

**037:000000000000000000111110000000110001100000000000001110000000000000000000000000000000000011100000000000000000000000001111111000000000000000000;**

**038:000000000000000001111100000001100001100000000000111100000000000000000000000000000000000000110000000000000000000000000011111000000000000000000;**

**039:000000000000000011111000000011100111000000000000110000000000000000000000000000000000000000011011100000000000000000000001111110000000000000000;**

**040:000000000000000111110000000011001110000000000001100000000000000000000000000000000000000000011111111100000000000000000000111110000000000000000;**

**041:000000000000000111100000000111111100000000000000100000000000000000000000000000000000000000001100001100000000000000000000011111000000000000000;**

**042:000000000000001111100000000111111100000000000001100000000000000000000000000000000000000000011000000110000000000000000000001111100000000000000;**

**043:000000000000011111000000000011001100000000000001100000000000000000000000000000000000000000110000000011000000000000000000000111110000000000000;**

**044:000000000000011110000000000111000100000000000011100000000000000000000000000000000000101001110000000011100000000000000000000011110000000000000;**

**045:000000000000111110000000001100001110000000001111110000000000000000000000000000000011111111100000000001110000000000000000000011111000000000000;**

**046:000000000000111100000000110100000011111111111100011000000000000000000000000000000001100101100000000000011100000000000000000000111100000000000;**

**047:000000000001111100000001111000000001111111100000010000000000000000000000000000000011000000000000000000010000000000000000000001111100000000000;**

**048:000000000001111100000011100000000000000000000000010000000000000000000000000000000110000000000000000000010000000000000000000000111100000000000;**

**049:000000000001111000000011000000000000000000000000111000000000000000000000000000000011000000000000000001110000000000000000000000111100000000000;**

**050:000000000001111000000011000000000000000000000000011100000000000000000000000000000011000000000000000011111000000000000000000000111110000000000;**

**051:000000000011110000000010000000000000000000000000001111000000000000000000000000000001100001111000111111111111000000000000000000011110000000000;**

**052:000000000011111000000000000000000000000000000000000011100000000000000000000000000001111011111111111110100111000000000000000000011111000000000;**

**053:000000000011110000000000000000000000000000000000000001110000000000000000000000000000011110000100000000000001100000000000000000011111000000000;**

**054:000000000011110000000011000000000000000000000000000000011000000000000000000000000000000000000000000000000000110000000000000000011111000000000;**

**055:000000000111110000000111110000000000000000000000000000011000000000000000000000000000000000000000000000000000010000000000000000001111000000000;**

**056:000000000111110000001100100000000000000000000000000000011100000000000000000000000000000000000000000000000000011000000000000000001111100000000;**

**057:000000000111100000001100110000000000000000000000000000001110000000000000000000000000000000000000000000000000011100000000000000000111100000000;**

**058:000000000111100000001111100000000000000000000000000000011011000000000000000000000000000000000000000000000000000111000000000000000111110000000;**

**059:000000000111100000001111000000000000000000000000000000010001100000000000000000000000000000000000000000000000000011110000000000000011110000000;**

**060:000000000111110000000000000000000000000000000000000000011011100000000000000000000000000000000000000000000000000000111000000000000011110000000;**

**061:000000000111111100000000000000000000000000000000000000001111000000000000000000000000000000000000000000000000000000011000000000000011110000000;**

**062:000000000111100111000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001100000000000001111000000;**

**063:000000000011110001000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001100000000000001111000000;**

**064:000000000011110001100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000110000000000001111000000;**

**065:000000000011110000110000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010000000000000111000000;**

**066:000000000011110000110000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010000000000001111100000;**

**067:000000000111111111100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010000000000000111100000;**

**068:000000001111111110000000000000000000000000000000000000000000001000000000000000000000000000000000000000000000000000000000000000000000111000000;**

**069:000000001111000000000000000000000000000000000000000000000000001100000000000000000000000000000000000000000000000000000000000000000000111100000;**

**070:000000011111000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010000000000111000000;**

**071:000000011110000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011111000000001111100000;**

**072:000000011110000000000000000000000000000000000000000000000000000000111000000000000000000000000000000000000000000000000011011000000001111000000;**

**073:000000111100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011011000000001111000000;**

**074:000000111100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011011000000001111000000;**

**075:000000111100000000000000000000000000000000000000000000000000000000000000100000000000000000000000000000000000000000000001111000000001111000000;**

**076:000000111100000000000000000000000000000000000000000000000000000000000000111000000000000000000000000000000000000000000000100000000001111000000;**

**077:000000111100000000000000000000000000000000000000000000000000000000000000011100000000000000000000000000000000000000000000000000000011110000000;**

**078:000000111100000000000000000000000000000000000000000000000000000000000000001111000000000000000000000100000000000000000000000000000011110000000;**

**079:000000111100000000000000000000000000000000000000000000000000000000000000000001111100000000000000111111100000000000000000000000000011110000000;**

**080:000000111100000000000000000000100000000000000000000000000000000000000000000000001110000000001011111000111000000000000000000000000011110000000;**

**081:000000011110000000000000000011110000000000000000000000000000000000000000000000000011000001111111000000011110000000000000000000000011110000000;**

**082:000000011110000000000000000011010000000000000000000000000000000000000000000000000011100011111100000000000111000000000000000000000011110000000;**

**083:000000001111000000000000000110010000000000000000000000000000000000000000000000000000110110000000000000000011100000000000000000000011110000000;**

**084:000000001111000000000000000011010000000000000000000000000000000000000000000000000000011100000000000000000001110000000000000000000011110000000;**

**085:000000001111100000000000000011110000000000000000000000000000000000000000000000000000001100000000000000000000011000000000000000000011110000000;**

**086:000000000111100000000000000001100000000000000000000000000000000000000000000000000000000110000000000000000000001100000000000000000011111000000;**

**087:000000000111110000000000000000000000000000000000000000000000000000000000000000000000000110000000000000000000000100000000000000000011111000000;**

**088:000000000011110000000000000000001100000000000000000000000000000000000000000000000000000010000000000000000000001110000000000000000011111000000;**

**089:000000000011111000000000000000001110000000000000000000000000000000000000000000000000000011000000000000000000000110000000000000000001111000000;**

**090:000000000001111000000000000000000111100000000000000000000000000000000000000000000000000011000000000000000000000110000000000000000001111000000;**

**091:000000000001111100000000000000000001111111100000000000000000000000000000000000000000000001100000000000000000000100000000000000000001111100000;**

**092:000000000000111100000000000000000001101110110000000000000000000000000000000000000000000000100000000000000000001100000000000000000001111100000;**

**093:000000000000111100000000000000000000100000011000000000000000000000000000000000000000000000110000000000000000001100000000000000000001111100000;**

**094:000000000000111100000000000000000000110000011000000000000000000000000000000000000000000000001100000000000000011000000000000000000001111100000;**

**095:000000000000011110000000000000000000010000001100000000000000000000000000000000011000000000000111000000000001110000000000000000000001111000000;**

**096:000000000000011110000000000000000000011000000111000000000000000000000000000000111100000000000011111110011111111000000000000000000001111100000;**

**097:000000000000011111000000000000000000001100000011100000000000000000000000000001100011000000000000111111111110011110010000000000000001111000000;**

**098:000000000000001111000000000000000000000111000000111110000000000000000000000000010011000000000000000001000000000111110000000000000001111000000;**

**099:000000000000001111100000000000000000000111110000000111000000000000000000000000011111100000000000000000000000000001111000000000000001111000000;**

**100:000000000000001111100000000000000000000000011100000001000000000000000000000000000111100000000000000000000000000000011000000000000001111000000;**

**101:000000000000000111110000000000000000000000000111000001100000000000000000000000000001110000000000000000000000000000001100000000000001111000000;**

**102:000000000000000011110000000000000000000000000001110000110000000000000000000000000000011100000000000000000000000000001100000000000011110000000;**

**103:000000000000000011111100000000000000000000000000111000111000000000000000000000000000000110000000000111000000000000000100000000000011110000000;**

**104:000000000000000001111100000000000000000000000000011100011000000000000000000000000000000011000000011111111101110000000001000000000111110000000;**

**105:000000000000000000111110000000000000000000000000001110001100000000000000000000000000000001100000111000011111111100000001100000000111100000000;**

**106:000000000000000000011111000000000000000000000000000110000110000000000000000000000000000000110011100000000000001110000000000000001111100000000;**

**107:000000000000000000001111100000000000000000000000000011000011000000000000000000000000000000011010000000000000000011000000000000001111000000000;**

**108:000000000000000000000111110000000000000000000000000001110001000000000000000000000000000000001110000000000000000001100000000000011111000000000;**

**109:000000000000000000000011110000000000000000000000000000111111100000000000000000000000000000000110000000000000000001100000000000111110000000000;**

**110:000000000000000000000011111100000000000000000000000000000111110000000000000000000000000000000110000000000000000000110000000111111100000000000;**

**111:000000000000000000000001111110000000000000000000000000000000011000000000000000000000000000000110000000000000000000110000011111111100000000000;**

**112:000000000000000000000000111111100000000000000000000000000000011000000000000000000000000000000011000000000000000000111010111111111000000000000;**

**113:000000000000000000000000011111110000000000000000000000000000001110000000000000000000000000000001100000000000001000111111111111110000000000000;**

**114:000000000000000000000000000111111110000000000000000000000000000111000000000000000000000000000001111110000000011111111111111110000000000000000;**

**115:000000000000000000000000000001111111000000000000000000000000000001000000000000000000000000000000001010000000111111111111111000000000000000000;**

**116:000000000000000000000000000000111111110000000000000000000000000001100000000000000000000000000000000011000001111111111111000000000000000000000;**

**117:000000000000000000000000000000001111111100000000000000000000000001000000000000000000000000000000000001110011111111010000000000000000000000000;**

**118:000000000000000000000000000000000011111111000000000000000000000011000000000000000000000000000000000000111111111000000000000000000000000000000;**

**119:000000000000000000000000000000000001111111111000000000000000000011000000000000000000000000000000000000001111110000000000000000000000000000000;**

**120:000000000000000000000000000000000000001111111110000000000000001011000000000000000000000000000000000000001111100000000000000000000000000000000;**

**121:000000000000000000000000000000000000000111111111111101000101111111111000000000000000000000000000000000011111000000000000000000000000000000000;**

**122:000000000000000000000000000000000000000000111111111111111111111111111110000000000000000000000000000000111111000000000000000000000000000000000;**

**123:000000000000000000000000000000000000000000001111111111111111111111111111100000000000000000000000000001111100000000000000000000000000000000000;**

**124:000000000000000000000000000000000000000000000000111111111111110000111111111000000000000000000000000111111100000000000000000000000000000000000;**

**125:000000000000000000000000000000000000000000000000000010010100000000001111111110000000000000000000001111110000000000000000000000000000000000000;**

**126:000000000000000000000000000000000000000000000000000000000000000000000011111111111000000000000000111111100000000000000000000000000000000000000;**

**127:000000000000000000000000000000000000000000000000000000000000000000000000011111111111100010011111111111000000000000000000000000000000000000000;**

**128:000000000000000000000000000000000000000000000000000000000000000000000000001111111111111111111111111100000000000000000000000000000000000000000;**

**129:000000000000000000000000000000000000000000000000000000000000000000000000000011111111111111111111110000000000000000000000000000000000000000000;**

**130:000000000000000000000000000000000000000000000000000000000000000000000000000000000111111111111111000000000000000000000000000000000000000000000;**

**131:000000000000000000000000000000000000000000000000000000000000000000000000000000000001111111100000000000000000000000000000000000000000000000000;**

**132:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**133:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**134:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**135:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**136:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**137:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**138:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**139:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**140:000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000;**

**End;**

**Appendix VIII: NUM\_TO\_DIGITS BLOCK**

**LIBRARY IEEE;**

**USE IEEE.STD\_LOGIC\_1164.all;**

**USE IEEE.STD\_LOGIC\_ARITH.all;**

**USE IEEE.STD\_LOGIC\_UNSIGNED.all;**

**ENTITY num\_to\_digits IS**

**PORT(number : IN std\_logic\_vector(13 DOWNTO 0);**

**digit1 : OUT std\_logic\_vector(3 DOWNTO 0);**

**digit2 : OUT std\_logic\_vector(3 DOWNTO 0);**

**digit3 : OUT std\_logic\_vector(3 DOWNTO 0);**

**digit4 : OUT std\_logic\_vector(3 DOWNTO 0));**

**END num\_to\_digits;**

**architecture behavior of num\_to\_digits is**

**SIGNAL int\_number : INTEGER range 0 to 9999;**

**SIGNAL int\_1 : INTEGER range 0 to 9;**

**SIGNAL int\_2 : INTEGER range 0 to 9;**

**SIGNAL int\_3 : INTEGER range 0 to 9;**

**SIGNAL int\_4 : INTEGER range 0 to 9;**

**BEGIN**

**int\_number <= CONV\_INTEGER(number);**

**int\_1 <= (int\_number mod 10); --find one digit**

**int\_2 <= (int\_number / 10) mod 10; --find tens digit**

**int\_3 <= (int\_number / 100) mod 10; -- find hundreds digit**

**int\_4 <= (int\_number / 1000) mod 10 ; --find thousands digit**

**digit1 <= CONV\_STD\_LOGIC\_VECTOR(int\_1, 4);**

**digit2 <= CONV\_STD\_LOGIC\_VECTOR(int\_2, 4);**

**digit3 <= CONV\_STD\_LOGIC\_VECTOR(int\_3, 4);**

**digit4 <= CONV\_STD\_LOGIC\_VECTOR(int\_4, 4);**

**END behavior;**

**Appendix XI DIGIT\_DISPLAY BLOCK**

**LIBRARY IEEE;**

**USE IEEE.STD\_LOGIC\_1164.all;**

**USE IEEE.STD\_LOGIC\_ARITH.all;**

**USE IEEE.STD\_LOGIC\_UNSIGNED.all;**

**ENTITY digit\_display IS**

**generic(X\_pos : INTEGER range 0 to 640 := 0; --position of top left corner**

**Y\_pos : INTEGER range 0 to 480 := 0);**

**PORT(pixel\_row, pixel\_column : IN std\_logic\_vector(9 DOWNTO 0);**

**pixel\_clock : IN std\_logic;**

**digit : IN std\_logic\_vector(3 DOWNTO 0);**

**white : OUT std\_logic);**

**END digit\_display;**

**architecture behavior of digit\_display is**

**SIGNAL Size : std\_logic\_vector(9 DOWNTO 0);**

**SIGNAL Image\_Y\_pos, Image\_X\_pos : std\_logic\_vector(9 DOWNTO 0); -- keep track of which pixel of the image is being displayed**

**SIGNAL image\_data : std\_logic\_vector(7 DOWNTO 0); -- 8-bit data from the image rom**

**SIGNAL rom\_addr : std\_logic\_vector(6 DOWNTO 0); -- address bits for rom**

**SIGNAL out\_on : std\_logic;**

**COMPONENT digits\_rom**

**PORT**

**(**

**address : IN STD\_LOGIC\_VECTOR (6 DOWNTO 0);**

**clock : IN STD\_LOGIC;**

**q : OUT STD\_LOGIC\_VECTOR (7 DOWNTO 0)**

**);**

**END COMPONENT;**

**BEGIN**

**Size <= CONV\_STD\_LOGIC\_VECTOR(7,10);**

**-- White Digit**

**white <= out\_on and image\_data((7-CONV\_INTEGER(Image\_X\_pos(3 downto 0)))); -- 7 - pos in order to reverse order since .mif file is backwards**

**-- the image pixels are determined relative to the digit position and the CRT pixel position**

**Image\_Y\_pos <= pixel\_row - Y\_pos;**

**Image\_X\_pos <= pixel\_column - X\_pos;**

**RGB\_Display: Process (pixel\_column, pixel\_row)**

**BEGIN**

**-- Set Ball\_on ='1' to display**

**IF (pixel\_column >= X\_pos) AND**

**(pixel\_column <= X\_pos + Size) AND**

**(pixel\_row >= Y\_pos) AND**

**(pixel\_row <= Y\_pos + Size) THEN**

**out\_on <= '1';**

**ELSE**

**out\_on <= '0';**

**END IF;**

**END process RGB\_Display;**

**-- instantiate the rom and hook up the signals**

**digits\_rom\_inst : digits\_rom PORT MAP (**

**address => rom\_addr,**

**clock => pixel\_clock,**

**q => image\_data**

**);**

**-- rom address**

**-- MSB is from switch and selects which image to display**

**-- other 3 bits select rom of image to display**

**rom\_addr <= digit & Image\_Y\_pos(2 DOWNTO 0);**

**END behavior;**

**Appendix X: DIGITS.MIF FILE**

**Depth = 80;**

**Width = 8;**

**Address\_radix = oct;**

**Data\_radix = bin;**

**% Character Generator ROM Data %**

**Content**

**Begin**

**000 : 00111100 ; % \*\*\*\* %**

**001 : 01100110 ; % \*\* \*\* %**

**002 : 01101110 ; % \*\* \*\*\* %**

**003 : 01110110 ; % \*\*\* \*\* %**

**004 : 01100110 ; % \*\* \*\* %**

**005 : 01100110 ; % \*\* \*\* %**

**006 : 00111100 ; % \*\*\*\* %**

**007 : 00000000 ; % %**

**010 : 00011000 ; % \*\* %**

**011 : 00011000 ; % \*\* . %**

**012 : 00111000 ; % \*\*\* %**

**013 : 00011000 ; % \*\* %**

**014 : 00011000 ; % \*\* %**

**015 : 00011000 ; % \*\* %**

**016 : 01111110 ; % \*\*\*\*\*\* %**

**017 : 00000000 ; % %**

**020 : 00111100 ; % \*\*\*\* %**

**021 : 01100110 ; % \*\* \*\* %**

**022 : 00000110 ; % \*\* %**

**023 : 00001100 ; % \*\* %**

**024 : 00110000 ; % \*\* %**

**025 : 01100000 ; % \*\* %**

**026 : 01111110 ; % \*\*\*\*\*\* %**

**027 : 00000000 ; % %**

**030 : 00111100 ; % \*\*\*\* %**

**031 : 01100110 ; % \*\* \*\* %**

**032 : 00000110 ; % \*\* %**

**033 : 00011100 ; % \*\*\* %**

**034 : 00000110 ; % \*\* %**

**035 : 01100110 ; % \*\* \*\* %**

**036 : 00111100 ; % \*\*\*\* %**

**037 : 00000000 ; % %**

**040 : 00000110 ; % \*\* %**

**041 : 00001110 ; % \*\*\* %**

**042 : 00011110 ; % \*\*\*\* %**

**043 : 01100110 ; % \*\* \*\* %**

**044 : 01111111 ; % \*\*\*\*\*\*\* %**

**045 : 00000110 ; % \*\* %**

**046 : 00000110 ; % \*\* %**

**047 : 00000000 ; % %**

**050 : 01111110 ; % \*\*\*\*\*\* %**

**051 : 01100000 ; % \*\* %**

**052 : 01111100 ; % \*\*\*\*\* %**

**053 : 00000110 ; % \*\* %**

**054 : 00000110 ; % \*\* %**

**055 : 01100110 ; % \*\* \*\* %**

**056 : 00111100 ; % \*\*\*\* %**

**057 : 00000000 ; % %**

**060 : 00111100 ; % \*\*\*\* %**

**061 : 01100110 ; % \*\* \*\* %**

**062 : 01100000 ; % \*\* %**

**063 : 01111100 ; % \*\*\*\*\* %**

**064 : 01100110 ; % \*\* \*\* %**

**065 : 01100110 ; % \*\* \*\* %**

**066 : 00111100 ; % \*\*\*\* %**

**067 : 00000000 ; % %**

**070 : 01111110 ; % \*\*\*\*\*\* %**

**071 : 01100110 ; % \*\* \*\* %**

**072 : 00001100 ; % \*\* %**

**073 : 00011000 ; % \*\* %**

**074 : 00011000 ; % \*\* %**

**075 : 00011000 ; % \*\* %**

**076 : 00011000 ; % \*\* %**

**077 : 00000000 ; % %**

**100 : 00111100 ; % \*\*\*\* %**

**101 : 01100110 ; % \*\* \*\* %**

**102 : 01100110 ; % \*\* \*\* %**

**103 : 00111100 ; % \*\*\*\* %**

**104 : 01100110 ; % \*\* \*\* %**

**105 : 01100110 ; % \*\* \*\* %**

**106 : 00111100 ; % \*\*\*\* %**

**107 : 00000000 ; % %**

**110 : 00111100 ; % \*\*\*\* %**

**111 : 01100110 ; % \*\* \*\* %**

**112 : 01100110 ; % \*\* \*\* %**

**113 : 00111110 ; % \*\*\*\*\* %**

**114 : 00000110 ; % \*\* %**

**115 : 01100110 ; % \*\* \*\* %**

**116 : 00111100 ; % \*\*\*\* %**

**117 : 00000000 ; % %**

**END;**