XiNES

A Nintendo Entertainment System simulator coded in pure VHDL

William Blinn, Dave Coulthart, Jay Fernandez, Neel Goyal, Jeff Lin

Table of Contents

Chapter 1 – *Introduction (Project Proposal)*

XiNES is a Nintendo Entertainment System simulator coded in pure VHDL and ported to the XSB-300E board, which utilizes a Xilinx Spartan FPGA. The NES itself consists of three main parts: a customized 6502 CPU, a Picture Processing Unit (PPU), and a memory hierarchy that includes the actual game ROM. Our initial goal was to implement all of these to get a single commercial game to run at full speed on the board. Due to time constraints and difficulties in implementing the complex PPU we decided to simplify the PPU. Our PPU now draws a test ROM consisting of a single frame.

The main bulk of the project was spent implementing the system's PPU. We used all resources available to us, including online documentation, open source emulators, and even patented schematics (all which is cited and credited). The 6502 was obtained by using a free, open-source VHDL implementation of the 6502, called Free-6502. Our goal was to connect our PPU and this 6502 implementation in some way such that the test ROM would display the frame.

Our main objective in this project was graphics implementation. We did not have time to implement sound or to connect a controller.

Chapter 2 – *Project Design*

6502 Processor

The 6502 processor is the main CPU of the NES. The VHDL component declaration of the 6502 is:

```
component core_6502
  port (clk   :in std_logic;
    reset     :in std_logic;
    irq       :in std_logic;
    nmi       :in std_logic;
    addr      :out std_logic_vector (15 downto 0);
    din       :in  std_logic_vector (7 downto 0);
    dout      :out std_logic_vector (7 downto 0);
    dout_oe   :out std_logic;
    wr        :out std_logic;
    rd        :out std_logic;
    sync      :out std_logic
  );
end component;

Signal descriptions
clk: The main system clock.  All synchronous signals
     are clocked off the rising edge of clk.
reset: An active high reset signal, asynchronous to
     clk.
irq: An active high, level triggered, asynchronous,
     interrupt input.
nmi: A rising edge triggered non-maskable interrupt
     input.
addr: The address bus output.
din: Data bus input
dout: Data bus output
dout_oe: Data bus output enable, used to control
     external tri-state buffers.  Active high.
wr: An active high write signal
rd: An active high read signal.
sync: High during the first byte of an instruction
     fetch.
```

The 6502 processor contains 64K of memory. There are four banks of 2K for RAM, 12K for registers, 4K for expansion modules, 8K for

WRAM (which is used for games that allow saving), and two banks of 16K for Program ROM.

Registers $2006 and $2007 are used for reading from and writing data to the VRAM. The address in VRAM to be read from or written to is specified in $2006 and the data to be read or written is specified in $2007. When reading from register $2007, the first read is invalid and needs to be discarded.

Multi-Memory Controllers

Multi-Memory Controllers (MMCs) are used in cartridges for addressing extra memory. The 6502 processor's memory limit is 64K, of which 32K is used for the Program ROM. The PPU's VRAM memory limit is 16K. If either the 6502 or the PPU's memory limit is exceeded, an MMC is needed to address the extra memory.

Of note, even though the 6502 supports 64K memory, there is only 32K available for Program ROM, so ROMs larger than 32K will require the use of an MMC.

The Program ROM memory region on the CPU is divided into two banks of 16K each. If a ROM is smaller than 16K, it will load into the upper bank of memory. Larger ROMs will load into the lower 16K bank as well.

The ROM

We initially decided that the ROM image that we wanted to use was the game Mario Brothers. It was chosen for its simplicity. The ROM is less

than 16K in size, which means that it does not require the use of an MMC. The game also has no scrolling involved so there will be a less complex PPU. We then had our problem with sprites and frames so we decided to use a test ROM that is one frame and displays “R G B” in big letters in its respective colors.

Memory Hierarchy

There are two main memory components of the NES – the 64 KB main memory interfacing with the 6502 CPU and the 16 KB Video RAM (VRAM) used by the Picture Processing Unit (PPU). Because of these high memory requirements, the two memories will be stored in SRAM. The 256-byte Sprite RAM, which is not a part of either the CPU or PPU address space, is the remaining piece of the memory hierarchy of the NES.

CPU Memory

The NES’s CPU memory is divided for different uses as follows:

Starting Address	Size (bytes)	Use
0x0000	2K	RAM
0x0800	2K	RAM (mirrored from 0x0000)
0x1000	2K	RAM (mirrored from 0x0000)
0x1800	2K	RAM (mirrored from 0x0000)
0x2000	12K	Registers
0x5000	4K	Expansion Modules
0x6000	8K	Writeable RAM (WRAM)
0x8000	16K	Program ROM (PRG-ROM) (Lower)
0xC000	16K	PRG-ROM (Upper)

While we will provide the entire CPU memory address space (to avoid the need for complicated address translation), memory associated

with certain advanced functionality will remain unused. In particular, the WRAM used by games for saving state and the expansion module memory will be unused. The PRG-ROM is used to hold the actual game code. Because our simplified design does not include a Multi-Memory Controller only the Upper PRG-ROM will be used to hold games up to 16 KB in size.

The registers are used primarily for communicating with the PPU, outputting sound, and managing the joystick. The PPU-associated registers are explained further in the PPU section of the document, while the sound registers and the joystick registers will be ignored because we did not implement them.

Both the CPU and the PPU have to access VRAM and there could be a collision if they are both trying to access it. Therefore we implemented a MUX that decides whether to let the CPU or the PPU into VRAM. Basically, the VRAM gets to access VRAM whenever it wants because drawing to the screen is the priority. If both are trying to access VRAM then the PPU is allowed to first.

PPU Memory

The division of the PPU VRAM is as follows:

Starting Address	Size (bytes)	Use
0x0000	4K	Pattern Table #0
0x1000	4K	Pattern Table #1
0x2000	960	Name Table #0
0x23C0	64	Attribute Table #0
0x2400	960	Name Table #1

0x27C0	64	Attribute Table #1
0x2800	960	Name Table #2 (based on mirroring)
0x2BC0	64	Attribute Table #2 (based on mirroring)
0x2C00	960	Name Table #3 (based on mirroring)
0x2FC0	64	Attribute Table #3 (based on mirroring)
0x3000	3840	EMPTY
0x3F00	16	Image Palette
0x3F10	16	Sprite Palette
0x3F20	224	EMPTY

The name tables are used to store indices for obtaining the actual color information stored in the matching pattern table. The address for the color information is calculated as: (IndexValue * 16) + PatternTableBaseAddress. Only two bits of the color information for a pixel (out of the four used for each pixel) are found in the pattern table. The upper two bits of color for each pixel are obtained from the attribute table. Each byte in the attribute table holds the upper two bits for sixteen 8x8 tiles (the same upper two bits are used for each set of four tiles).

Sprite RAM

The NES supports up to 64 concurrent sprites. The Sprite RAM is used to hold the attributes of these sprites. Each entry consists of: x and y coordinates (of upper left corner), sprite tile index number (for obtaining the actual sprite pattern from the pattern table in PPU memory), horizontal/vertical flip, priority (above/behind background), and the upper two bits of color (color selection is explained in the PPU section).

Sprite RAM was implemented as part of the ColorGen RAM module but is left unused by our system.

Picture Processing Unit

The Picture Processing Unit (PPU) is the graphical hardware behind the NES. The PPU can be thought of as a block with input and output pins.

Component declaration of the PPU is:

```
ENTITY nes_ppu_still IS
    PORT (
       b : OUT std_logic_vector(7 DOWNTO 0);
       g : OUT std_logic_vector(7 DOWNTO 0);
       r : OUT std_logic_vector(7 DOWNTO 0);
       v_addr : OUT std_logic_vector(13 DOWNTO 0);
       v_data : OUT std_logic_vector(7 DOWNTO 0);
       v_read : OUT std_logic;
       v_write : OUT std_logic;
       ppu_go : OUT std_logic;
       addr : IN std_logic_vector(15 DOWNTO 0);
       clock : IN std_logic;
       cpu_data : IN std_logic_vector( 7 DOWNTO 0);
       cpu_r : IN std_logic;
       cpu_w : IN std_logic;
       rst : IN std_logic;
       v_in : IN std_logic_vector(7 DOWNTO 0);
    );
END nes_ppu_still;
```

The PPU is the only component that has access to VRAM and Sprite RAM, meaning the CPU must access the PPU in order to either write or read from these memory spaces. Fortunately, this can be done by writing to various 8-bit registers, acting as I/O ports, that the CPU can see. Here is a list of them and the hexadecimal address the CPU sees them as:

```
$2000:    PPU Control Register which determines where in
VRAM and Sprite RAM data is being written to or read from
and the size of the sprites.
$2001:    PPU Control Register which determines various
properties regarding the image being displayed, such as the
background color and clipping information.
$2002:    PPU Status register which changes to indicate
whether the screen needs to be refreshed, a sprite needs to
be displayed, or too many sprites are on a line at a time.
$2003:    This register holds the address of Sprite RAM to
read or write to.
$2004:    Holds the data being written to or read from
Sprite RAM specified by the address in  $2003.
$2005:    Register which handles information regarding
screen scrolling.  Since we are trying to simulate a very
simple game, we will probably not use this.
$2006:    This is a double write register that determines
the location in VRAM to be written to or read from.  Since
VRAM is addressed via 14-bits, the first write writes the
upper byte of  the address, and the lower byte is written
second.
$2007:    Similar to $2004, this holds the data being
written to or read from VRAM.
```

In addition to being the mediator between the CPU and VRAM and Sprite RAM, the PPU generates the graphics outputted to the display. The NES displays graphics as tiles, each 8 pixels by 8 pixels in dimension. Sprites are either 8x8 or 8x16 pixels. Each pixel in a tile is generated by 4-bits taken from VRAM (or Sprite RAM if the tile pertains to a sprite) which are then converted to RGB via a color lookup table.

Two bytes from the pattern tables in VRAM and a byte from the attribute table are needed to generate this code. To draw the 5th pixel in a line on a tile, the fifth bit in the first pattern table byte is appended to the fifth bit of the second pattern table byte. Two bits from the attribute table are appended to the front of these two bits based on the location of

the tile. This makes up the 4-bit code, which also illustrates the NES's ability to only display 16 colors on the screen at a time.

The attribute byte should be explained a bit more in detail. Essentially, this byte holds information for 16 tiles, arranged in a 4x4 manner. The NTSC NES has a resolution of 256x240, meaning 32x30 tiles. This would imply that 8 attribute bytes are needed in order to draw the whole image. Assuming the 8-bit registers are bit numbered 7 down to 0, bits 1 and 0 represent the upper two bits of the color code of the upper left 4 tiles in the 4x4 tile arrangement. Bits 3 and 2 handle the upper right 4 tiles, and bits 5 and 4 handle the lower left.

It is important to note that the PPU is not driven by instructions and acts based on its registers. It reacts whenever a VBLANK occurs, which is stored in register $2002, and begins to redraw the image on the screen line by line. (Our simple implementation does not implement VBLANK as we only draw one frame over and over again.)

The 4 bits of still picture data is read from VRAM memory. VRAM is addressed with 14 bits. (The address where data should be read from is stored in the picture address register $2006) For the still data the VRAM data returns 2 bits for a character pattern and 2 bits for a color all for a single pixel. These 4 bits are fed into a lookup table called the color generator.

The color generator holds 32 6-bit codes. The top 16 codes of the RAM make up the sprite palette and the bottom 16 codes make up the

background palette. Our color generator has only 16 6-bit codes because we did not implement the sprite palette RAM. The 4-bit value serves as an address that looks up the appropriate color code in the Color generator RAM. When the correct code is found, the 6-bit value is outputted to the decoder.

The decoder's job is to generate a byte each for the three colors red, green and blue. The 6-bit code that is inputted to the decoder is made up a 4-bit code that specifies one of 16 different phases (hue) that the color is. The other 2 bits specify 1 of 4 levels. Based on these values and through a number of calculations R, G, and B values are generated and outputted to the line doubler.

The Line Doubler

The goal of the line doubler is to enlarge the image onscreen so that it is easier to see. To accomplish this, we will copy every pixel so that for every one pixel we had before we will have four new ones. Each pixel will be copied once to the position immediately to the right, then the same line will be drawn twice to given the effect of enlarging. For example, two lines that looked like this:

```
Xi@
NES
```

will be doubled to yield this:

```
XXii@@
XXii@@
NNEESS
```

NNEESS

The technique used will be very similar to the one in lab 5 for Embedded Systems Design. The interface presented to the PPU will be one that emphasizes simplicity: the input will be the bits corresponding to the pixel that needs to be displayed, a pixel clock and a line clock.

The port map to the line doubler looks as follows:

```
  port (
    doubler_clk  : in std_logic;
    doubler_data : in std_logic_vector(7 downto 0);
    doubler_reset : in std_logic;
    double_r    : out std_logic_vector(9 downto 0);
    double_g    : out std_logic_vector(9 downto 0);
    double_b    : out std_logic_vector(9 downto 0)
);
```

The PPU will send the line doubler at half the speed that the line doubler operates. The line doubler will use the extra clock cycles to display the double the pixels on the screen.

The mode of operation for the line doubler is as follows: when the PPU is feeding the line doubler line N, the line doubler will be outputting line N-1. Line N will be saved in a BRAM, to be accessed and outputted when the PPU feeds the doubler the next line. When outputting to the screen, the line doubler reads from the BRAM and sends the output signal to be displayed on screen.

In the time the PPU feeds one line to the line doubler, the line doubler will have output two lines to the screen. This is possible because the line doubler's clock is twice that of the PPU.

Another job of the line doubler is to center the image on the screen. To accomplish this, the line doubler will write into the BRAM starting at the beginning of the line, but will only read from the BRAM starting at an offset so that the image appears centered.

NTSC, the mode of video output for the NES, outputs at 30 Hz, while VGA operates at 60 Hz. Therefore, when NTSC has drawn one line, VGA has already drawn one line, moved its strobe back to the beginning of the line and drawn another. This helps us out because we can accept input at NTSC speeds and output them at VGA speeds without a hitch.

The end result of the line doubler will be a signal that is a centered 512x480 image instead of the native resolution of the NES, which is 256x240.

Figure 1. Block Diagram of Basic NES Design

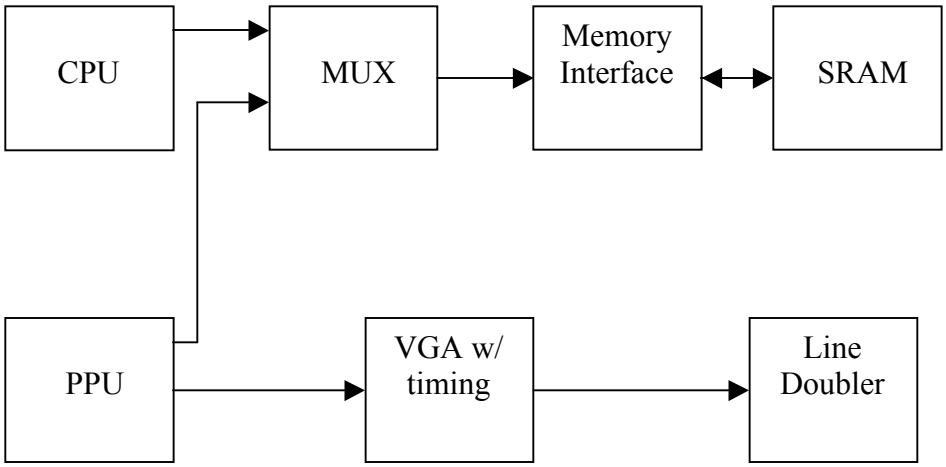
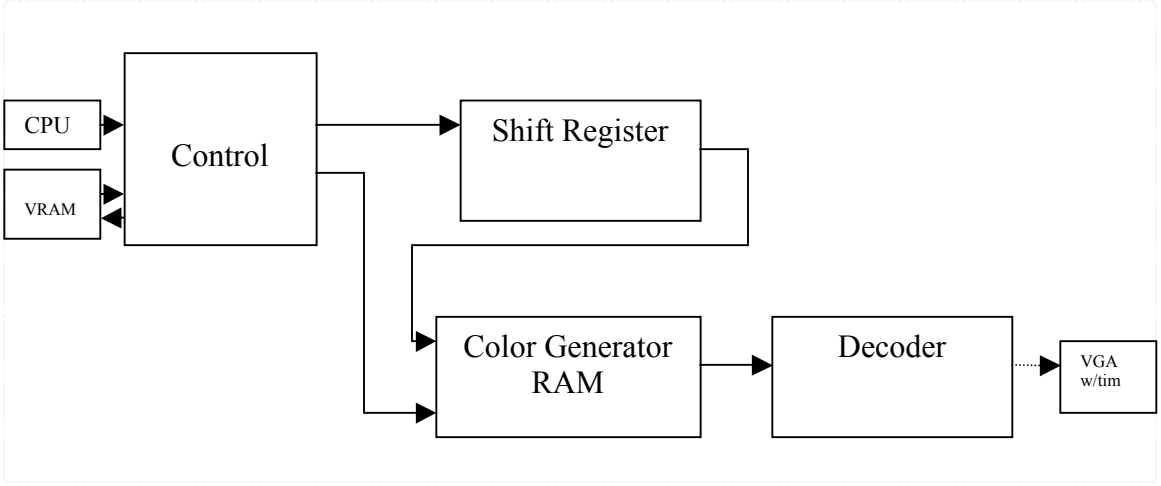


Figure 2. Block Diagram of XiNES PPU



Chapter 3 - *What Worked, What Didn't Work*

We encountered many problems as we began developing the XiNES. The first problem we came across was how to covert the NTSC phase (hue) and luminance values to RGB values. We found a BASIC implementation of this conversion online and translated it to VHDL. The Cadence simulation worked well however when we tried to put the program on the FPGA board we ran into problems. There were complex calculations involving floating-point numbers and trigonometric functions such as sine and cosine. We quickly learned that these calculations are tremendously expensive so the board did not support floating-point types.

Since the due date was a day away, we did not have time to think of other implementations. We solved the problem by using a Cadence simulation to determine which 16-image palette values were written to the Color Generator. We then manually worked out the calculations for these 16 values. When a 6-bit code from the image palette is chosen, the RGB values stored in the decoder are output to the line doubler.

Sprites did not work as well. The incredible complexity of the motion picture overwhelmed us after months of trying to understand how it worked and implement it. We coded up many components of the sprite section of the PPU, but at the end we decided to focus our attention on

getting a single frame of a background to display. We got rid of the sprite components and changed our design to only show backgrounds.

Another issue we ran into was that the resolution for the NES is 256x240. Also, we needed a monitor that supported raster scan and NTSC. Our original solution was to buy an old monitor much like the monitor that the Apple IIGS used. We discarded this idea when it proved expensive and we decided to write a line doubler in VHDL as explained in the project design. Basically, we copy every pixel so that for every one pixel we had before we will have four new ones. Each pixel will be copied once to the position immediately to the right, then the same line will be drawn twice to given the effect of enlarging. This solved the resolution problem.

In the end, the biggest setback to the project was trying to get the SRAM loaded with the appropriate program ROM. Using the `bin2hex` program we attempted to convert the binary character and program ROMs into the appropriate hex format for the Xilinx `xsload` utility with the appropriate memory offsets for interaction with the 6502 CPU. Unfortunately, it seems that `xsload` ignored the memory-offset information because the CPU never saw the correct data. We tried to work around this by using the `Xio_In` functions to load SRAM, but this either resulted in integer overflow, SRAM overflow, or LISP errors during the compilation phase.

Chapter 4 – *Who Did What*

Because of our high ambition of a fully working PPU, we initially wrote a lot of code based on the PPU patent schematics. Although the XiNES unit supports writing a single frame and does not support sprites, Neel and Jay spent a lot of time designing and trying to implement a full PPU. So although the PPU is pretty simple right now, there was much work on trying to get the full PPU to work.

William Blinn – Worked on SRAM interface, line doubler, and attempted controller interface.

Dave Coulthart – Documentation of XiNES. SRAM interface and controller.

Jay Fernandez – Primarily worked on PPU.

Neel Goyal – Worked on PPU, connecting everything, and simulation.

Jeff Lin – Documentation and testing. SRAM memory addressing.

Chapter 5 – *Lessons Learned & Advice*

Individual Lessons Learned

Billy: Overall, I could have put significantly more effort into the project. I had a good start by getting the line doubler out of the way relatively early, but there were a few weeks during the semester where I didn't work very hard on the project. I wasn't aggressive enough in finding out what needed to be done, figuring out how to do it and asking for help when necessary. I believe if I had started on the SRAM interface a few weeks earlier and asked Cristian or Professor Edwards for help, it would have been completed without much of a hitch.

We had a problem distributing work in our group, with Jay and Neel doing all of the PPU (which was easily the most complicated module). If I had taken the time to learn how to use Cadence and helped Jay and Neel out with coding and testing, we probably would have had more parts of the NES working.

Hoping to meet in the lab a couple days before the project is due and glue everything together at the last second doesn't work. Better time budgeting, communication and distribution of duties would have made for a much better project.

Jay: In order to finish development in time there needs to be specific intermediate deadlines for various modules. Although we had some

deadlines written up, they weren't taken as seriously as they should have been. The work built up at the end of the semester and caused for unpleasant last minute coding. The problem is that there are unexpected delays that are difficult to predict so it is very difficult to forecast development time. The lesson I learned was to allow adequate time for each module and to take these deadlines seriously.

Another important lesson I learned is that communication is crucial but it takes time. Teams should be kept small because it is very hard to keep five people up to date with what is going on. With such large groups, team positions should have been given out. We needed a leader who can get people together, give out responsibilities and make sure that those tasks are completed at the intermediate deadlines. Interpersonal and motivational skills for the leader are even more important in school projects than in the real world. The leader does not have as much power to enforce that things are done compared to a supervisor in the real world. If you do not perform well at a company there is the threat of losing your job. However, if you do not do your work in school projects someone else will have to do that work for you.

Jeff: Working on this project has shown that the less well-defined a project is at the assigned level, the more effort must be made by each person on the team to make sure he knows what needs to be accomplished. Unfortunately, I approached the project like my previous

projects. That is, I had a general understanding of the area, and using the project guidelines, I was able to quickly figure out what to do. In this broader, group-defined project, when each of us began choosing individual tasks to complete, I did not know enough about each part to make a choice that would fit my capabilities. As a result, I ended up working rather aimlessly, picking up the loose ends that other members of the group were not covering and not really feeling as though I had made a significant contribution to the project as a whole. If I had approached the project with more knowledge of what exactly needed to be done, I think I might have been able to do more to help the group finish the project.

Dave: The most important lessons I learned from this project are that defined roles for each team member are a *requirement* for working with such a large group, each team member (or pair of team members) should have a specific set of tasks to complete, and communication within a group along with personal motivation are critical. A group leader must be chosen who will set internal deadlines, assign appropriate tasks, and ensure that everyone is focused on their work. For a project consisting of so many components (CPU, PPU, ROM, line doubler), a single member of the group should be in charge of ensuring all members are integrating the different pieces and communicating their interfacing needs. A group is always composed of members with different skill levels and areas of

expertise; tasks must be assigned based on these considerations. For each task, the person assigned to it should provide a brief proposal of the iterations that will be taken to complete the task along with deadlines for each development cycle. The entire group must then meet regularly to discuss progress and the next steps. I personally felt out of the loop on a number of the key components being developed by other group members and there was not enough effort on either side to keep the team in sync. While individuals must commit time beyond group meetings to complete all of the work, I believe it should be a course requirement that teams meet at least twice a week, either by sign-in during class lab time, or by requiring a meeting log. I personally don't believe I contributed as much as I could have to the group, feeling as those the large tasks such as the PPU were being handled by other members, and mandatory meetings leading to more group communication would have better kept me on track.

Advice to Future Groups

The biggest piece of advice we can give to students next year is that while ambition often results in innovative results, it's important to keep the promises reasonable. Often, features promised are not worth the cost to implement, not necessary, or simply impossible to implement in the given timetable. We jumped into the PPU promising to implement everything and we quickly realized that this was way beyond what we could do in one semester. We learned that given a certain amount of

time, you have to quickly determine what the group is capable of doing in that timeframe. For example, we wasted valuable time researching and trying to implement sprites but near the end of the year we hit a dead-end and did not understand how to complete it.

Chapter 6 – *Source Code*

Line Doubler

```
-- linedoubler.vhd
library IEEE;
use IEEE.std_logic_1164.All;
use IEEE.std_logic_unsigned.All;

entity line_doubler is
  port (
    doubler_clk  : in std_logic;
    doubler_data : in std_logic_vector(7 downto 0);
    doubler_reset : in std_logic;
    double_r    : out std_logic_vector(9 downto 0);
    double_g    : out std_logic_vector(9 downto 0);
    double_b    : out std_logic_vector(9 downto 0));
end line_doubler;

architecture Behavioral of line_doubler is

  component RAMB4_S8_S8

  port (DIA    : in STD_LOGIC_VECTOR (7 downto 0);
        DIB    : in STD_LOGIC_VECTOR (7 downto 0);
        ENA    : in STD_logic;
        ENB    : in STD_logic;
        WEA    : in STD_logic;
        WEB    : in STD_logic;
        RSTA   : in STD_logic;
        RSTB   : in STD_logic;
        CLKA   : in STD_logic;
        CLKB   : in STD_logic;
        ADDRA  : in STD_LOGIC_VECTOR (8 downto 0);
        ADDRB  : in STD_LOGIC_VECTOR (8 downto 0);
        DOA    : out STD_LOGIC_VECTOR (7 downto 0);
        DOB    : out STD_LOGIC_VECTOR (7 downto 0));

end component;


  constant NES_WIDTH : integer := 256;
  constant NES_HEIGHT : integer := 240;
  constant DOUBLED_WIDTH : integer := 512;
  constant DOUBLED_HEIGHT : integer := 480;
```

```
  constant CENTER_OFFSET : integer := 64;
  constant HALF_CENTER_OFFSET : integer := 32;

  constant H_ACTIVE      : integer := 640;
  constant H_FRONT_PORCH : integer := 16;
  constant H_BACK_PORCH  : integer := 48;
  --may need to modify h_total because of the difference
between
  --the nes clock and vga's clock
  constant H_TOTAL       : integer := 800;

  constant V_ACTIVE      : integer := 480;
  constant V_FRONT_PORCH : integer := 11;
  constant V_BACK_PORCH  : integer := 31;
  constant V_TOTAL       : integer := 524;

  signal d_bram_out : std_logic_vector(7 downto 0);

  --read from the input byte
  signal hor_read_in_bounds : std_logic := '1';
  signal ver_read_in_bounds : std_logic := '1';

  --write to the rgb signals
  signal hor_write_in_bounds : std_logic := '0';
  signal ver_write_in_bounds : std_logic := '0';

  signal ram_write_enable : std_logic;
  signal ram_read_enable : std_logic;

  signal address_a : std_logic_vector(8 downto 0);
  signal address_b : std_logic_vector(8 downto 0);

  signal pixel_count : std_logic_vector(10 downto 0);
  signal line_count : std_logic_vector(9 downto 0);

  signal r_temp : std_logic_vector(9 downto 0);
  signal g_temp : std_logic_vector(9 downto 0);
  signal b_temp : std_logic_vector(9 downto 0);

begin

    -- Pixel counter

  process ( doubler_clk, doubler_reset )
  begin
    if doubler_reset = '1' then
      pixel_count <= "00000000000";
```

```
    elsif doubler_clk'event and doubler_clk = '1' then
      if pixel_count = (H_TOTAL - 1) then
        pixel_count <= "00000000000";
      else
        pixel_count <= pixel_count + 1;
      end if;
    end if;
  end process;

  -- Line counter

  process ( doubler_clk, doubler_reset )
  begin
    if doubler_reset = '1' then
      line_count <= "0000000000";
    elsif doubler_clk'event and doubler_clk = '1' then
      if ((line_count = V_TOTAL - 1) and (pixel_count =
H_TOTAL - 1)) then
        line_count <= "0000000000";
      elsif pixel_count = (H_TOTAL - 1) then
        line_count <= line_count + 1;
      end if;
    end if;
  end process;

  -- create a signal to determine whether we want to write
to our output signals
  -- if we're in bounds vertically

  --we don't need to take into account the porches, so i
deleted the vtotal-front-back
  --bill
  process(doubler_clk)
  begin
    if doubler_clk'event and doubler_clk = '1' then
      if ((line_count = (DOUBLED_HEIGHT - 1)) and
(pixel_count = (H_TOTAL - 1))) then
        ver_write_in_bounds <= '0';
      elsif ((line_count = (V_TOTAL - 1)) and (pixel_count
= (H_TOTAL - 1))) then
        ver_write_in_bounds <= '1';
      end if;
    end if;
  end process;

  -- create a signal to determine whether we want to read
from our input signal
```

```
  -- if we're in bounds vertically

  --we don't need to take into account the porches, so i
deleted the vtotal-front-back
  --bill
  process(doubler_clk)
  begin
    if doubler_clk'event and doubler_clk = '1' then
      if ((line_count = (DOUBLED_HEIGHT - 1)) and
(pixel_count = (H_TOTAL - 1))) then
        ver_read_in_bounds <= '0';
      elsif ((line_count = (V_TOTAL - 1)) and (pixel_count
= (H_TOTAL - 1))) then
        ver_read_in_bounds <= '1';
      end if;
    end if;
  end process;

    -- create a signal to determine whether we want to
write to our output signals
  -- if we're in bounds horizontally
  process(doubler_clk)
  begin
    if doubler_clk'event and doubler_clk = '1' then
      if pixel_count = (DOUBLED_WIDTH + CENTER_OFFSET - 1)
then
        hor_write_in_bounds <= '0';
      elsif pixel_count = (CENTER_OFFSET - 1) then
        hor_write_in_bounds <= '1';
      end if;
    end if;
  end process;


  -- create a signal to determine whether we want to read
from our input signal
  -- if we're in bounds horizontally
  process(doubler_clk)
  begin
    if doubler_clk'event and doubler_clk = '1' then
      if pixel_count = (DOUBLED_WIDTH - 1) then
        hor_read_in_bounds <= '0';
      elsif pixel_count = (H_TOTAL - 1) then
        hor_read_in_bounds <= '1';
      end if;
    end if;
  end process;
```

```
  --create ram signals
  process(doubler_clk)
  begin
    if doubler_clk'event and doubler_clk = '1' then
      ram_write_enable <= hor_read_in_bounds and
ver_read_in_bounds and pixel_count(0);
      address_a <= line_count(1) & pixel_count(8 downto 1);
      address_b <= not line_count(1) & pixel_count(8 downto
1) - HALF_CENTER_OFFSET + 1;
    end if;
  end process;


 --BRAM
 line_bram : RAMB4_S8_S8 port map (
         DIA    => doubler_data ,
         ENA    => '1',
         WEA    => ram_write_enable,
         RSTA   => '0',
         CLKA   => doubler_clk,
         ADDRA  => address_a,
         DOA    => open,

         DIB    => X"00",
         ENB    => '1',
         WEB    => '0',
         RSTB   => '0',
         CLKB   => doubler_clk,
         ADDRB  => address_b,
         DOB    => d_bram_out
 );

  --assign the output signals
   double_r <= d_bram_out(7 downto 5) & "0000000"
               when ((hor_write_in_bounds = '1') and
(ver_write_in_bounds = '1'))
               else "1111111111";-- when
(hor_write_in_bounds = '0') else "0000000000";
   double_g <= d_bram_out(4 downto 2) & "0000000"
               when ((hor_write_in_bounds = '1') and
(ver_write_in_bounds = '1'))
               else "1111111111";-- when
(hor_write_in_bounds = '0') else "0000000000";
   double_b <= d_bram_out(1 downto 0) & "00000000"
               when ((hor_write_in_bounds = '1') and
(ver_write_in_bounds = '1'))
```

```
               else "1111111111";-- when
(hor_write_in_bounds = '0') else "0000000000";


end Behavioral;
```

```
-- vga.vhd
------------------------------------------------------------
--------------------
--
-- VGA video generator
--
-- Uses the vga_timing module to generate hsync etc.
-- Massages the RAM address and requests cycles from the
memory controller
-- to generate video using one byte per pixel
--
-- Cristian Soviani, Dennis Lim, and Stephen A. Edwards
--
------------------------------------------------------------
--------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity vga is
  port (
     clk            : in std_logic;
     pix_clk        : in std_logic;
     rst            : in std_logic;
     video_data     : in std_logic_vector(15 downto 0);
     video_addr     : out std_logic_vector(19 downto 0);
     video_req      : out std_logic;
    VIDOUT_CLK     : out std_logic;
    VIDOUT_RCR     : out std_logic_vector(9 downto 0);
    VIDOUT_GY      : out std_logic_vector(9 downto 0);
    VIDOUT_BCB     : out std_logic_vector(9 downto 0);
    VIDOUT_BLANK_N : out std_logic;
    VIDOUT_HSYNC_N : out std_logic;
    VIDOUT_VSYNC_N : out std_logic);
end vga;

architecture Behavioral of vga is

  -- Fast low-voltage TTL-level I/O pad with 12 mA drive
```

```
  component OBUF_F_12
    port (
      O : out STD_ULOGIC;
      I : in STD_ULOGIC);
  end component;

  -- Basic edge-sensitive flip-flop

  component FD
    port (
      C : in std_logic;
      D : in std_logic;
      Q : out std_logic);
  end component;

  -- Force instances of FD into pads for speed

  attribute iob : string;
  attribute iob of FD : component is "true";

  component vga_timing
    port (
      h_sync_delay         : out std_logic;
      v_sync_delay         : out std_logic;
      blank                : out std_logic;
      vga_ram_read_address : out std_logic_vector (19
downto 0);
      pixel_clock          : in std_logic;
      reset                : in std_logic);
  end component;

   component line_doubler
     port (
       doubler_clk  : in std_logic;
       doubler_data : in std_logic_vector(7 downto 0);
       doubler_reset : in std_logic;
       double_r    : out std_logic_vector(9 downto 0);
       double_g    : out std_logic_vector(9 downto 0);
       double_b    : out std_logic_vector(9 downto 0));
   end component;

  signal r                     : std_logic_vector (9
downto 0);
  signal g                     : std_logic_vector (9
downto 0);
  signal b                     : std_logic_vector (9
downto 0);
```

```
  signal blank                  : std_logic;
  signal hsync                  : std_logic;
  signal vsync                  : std_logic;
  signal vga_ram_read_address   : std_logic_vector(19
downto 0);
  signal vreq                   : std_logic;
  signal vreq_1                 : std_logic;
  signal load_video_word        : std_logic;
  signal vga_shreg              : std_logic_vector(15
downto 0);
  signal d_data                 : std_logic_vector(7 downto
0);

  --old signals
--   signal video_data            : std_logic_vector(15
downto 0);
--   signal clk                   : std_logic;
--   signal rst                   : std_logic;
--   signal pix_clk               : std_logic;

begin

--   clk <= OPB_Clk;
--   pix_clk <= pixel_clock;
--   rst <= OPB_Rst;
--   video_data <= "1110000000011100";

  st : vga_timing port map (
    pixel_clock => pix_clk,
    reset => rst,
    h_sync_delay => hsync,
    v_sync_delay => vsync,
    blank => blank,
    vga_ram_read_address => vga_ram_read_address);

   doubler : line_doubler port map (
     doubler_clk   => pix_clk,
     doubler_data  => d_data,
     doubler_reset => rst,
     double_r     => r,
     double_g     => g,
     double_b     => b);

  -- Video request is true when the RAM address is even

  -- FIXME: This should be disabled during blanking to
reduce memory traffic
```

```
  vreq <= not vga_ram_read_address(0);

  -- Generate load_video_word by delaying vreq two cycles

  process (pix_clk)
  begin
    if pix_clk'event and pix_clk='1' then
      vreq_1 <= vreq;
      load_video_word <= vreq_1;
    end if;
  end process;

  -- Generate video_req (to the RAM controller) by delaying
vreq by
  -- a cycle synchronized with the pixel clock

   process (clk)
   begin
     if clk'event and clk='1' then
       video_req <= pix_clk and vreq;
     end if;
   end process;

  -- The video address is the upper 19 bits from the VGA
timing generator
  -- because we are using two pixels per word and the RAM
address counts words

  video_addr <= '0' & vga_ram_read_address(19 downto 1);

  -- The video shift register: either load it from RAM or
shift it up a byte

  process (pix_clk)
  begin
    if pix_clk'event and pix_clk='1' then
      if load_video_word = '1' then
        vga_shreg <= video_data;
      else
        -- Shift the low byte of read video data into the
high byte
        vga_shreg <= vga_shreg(7 downto 0) & "00000000";
      end if;
    end if;
  end process;
```

```
  -- Copy the upper byte of the video word to the color
signals
  -- Note that we use three bits for red and green and two
for blue.

--    r(9 downto 7) <= vga_shreg (15 downto 13);
--    r(6 downto 0) <= "0000000";
--    g(9 downto 7) <= vga_shreg (12 downto 10);
--    g(6 downto 0) <= "0000000";
--    b(9 downto 8) <= vga_shreg (9 downto 8);
--    b(7 downto 0) <= "00000000";
--     d_data <= vga_shreg(15) or
--               vga_shreg(14) or
--               vga_shreg(13) or
--               vga_shreg(12) or
--               vga_shreg(11) or
--               vga_shreg(10) or
--               vga_shreg(9) or
--               vga_shreg(8);
  d_data <= vga_shreg(15 downto 8);

  -- Video clock I/O pad to the DAC

  vidclk : OBUF_F_12 port map (
    O => VIDOUT_clk,
    I => pix_clk);

  -- Control signals: hsync, vsync, and blank

  hsync_ff : FD port map (
    C => pix_clk,
    D => not hsync,
    Q => VIDOUT_HSYNC_N );

  vsync_ff : FD port map (
    C => pix_clk,
    D => not vsync,
    Q => VIDOUT_VSYNC_N );

  blank_ff :  FD port map (
    C => pix_clk,
    D => not blank,
    Q => VIDOUT_BLANK_N );

  -- Three digital color signals

  rgb_ff : for i in 0 to 9 generate
```

```
    r_ff : FD port map (
      C => pix_clk,
      D => r(i),
      Q => VIDOUT_RCR(i) );

    g_ff : FD port map (
      C => pix_clk,
      D => g(i),
      Q => VIDOUT_GY(i) );

    b_ff : FD port map (
      C => pix_clk,
      D => b(i),
      Q => VIDOUT_BCB(i) );

  end generate;

end Behavioral;
```

Picture Processing Unit

```
--Create Entity:
--Library=NES,Cell=nes_ppu_still,View=entity
--Time:Sat May  8 18:10:11 2004
--By:neel

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
ENTITY nes_ppu_still IS
    PORT(
        b : OUT std_logic_vector(7 DOWNTO 0);
        g : OUT std_logic_vector(7 DOWNTO 0);
        r : OUT std_logic_vector(7 DOWNTO 0);
        v_addr : OUT std_logic_vector(13 DOWNTO 0);
        v_data : OUT std_logic_vector(7 DOWNTO 0);
        v_read : OUT std_logic;
        v_write : OUT std_logic;
        ppu_go : out std_logic;
        addr : IN std_logic_vector(15 DOWNTO 0);
        clock : IN std_logic;
        cpu_data : IN std_logic_vector(7 DOWNTO 0);
        cpu_r : IN std_logic;
        cpu_w : IN std_logic;
        rst : IN std_logic;
```

```
        v_in : IN std_logic_vector(7 DOWNTO 0)
    );
END nes_ppu_still;

--Netlist:
--Library=NES,Cell=nes_ppu_still,View=schematic
--Time:Wed May  5 16:49:38 2004
--By:neel

ARCHITECTURE schematic OF nes_ppu_still IS
    COMPONENT decoder
        PORT(
            din : IN std_logic_vector(5 DOWNTO 0);
            rout : OUT std_logic_vector(7 DOWNTO 0);
            bout : OUT std_logic_vector(7 DOWNTO 0);
            gout : OUT std_logic_vector(7 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT colorgen
        PORT(
            clk : IN std_logic;
            we : IN std_logic;
            addr_vram : IN std_logic_vector(3 DOWNTO 0);
            addr_mux : IN std_logic_vector(3 DOWNTO 0);
            di : IN std_logic_vector(7 DOWNTO 0);
            do : OUT std_logic_vector(5 DOWNTO 0)
        );
    END COMPONENT;

--      COMPONENT vram
--          PORT(
--              clk : IN std_logic;
--              addr : IN std_logic_vector(13 DOWNTO 0);
--              din : IN std_logic_vector(7 DOWNTO 0);
--              dout : OUT std_logic_vector(7 DOWNTO 0);
--              wr : IN std_logic;
--              rd : IN std_logic
--          );
--      END COMPONENT;

    COMPONENT shift_reg
        PORT(
            output : OUT std_logic_vector(1 DOWNTO 0);
            data1 : IN std_logic_vector(7 DOWNTO 0);
            data2 : IN std_logic_vector(7 DOWNTO 0);
            load1 : IN std_logic;
```

```
        load2 : IN std_logic;
        clk : IN std_logic
    );
END COMPONENT;

COMPONENT control
    PORT(
        cpu_in : IN std_logic_vector(7 DOWNTO 0);
        addr_in : IN std_logic_vector(15 DOWNTO 0);
        cpu_read : IN std_logic;
        cpu_write : IN std_logic;
        ppu_clock : IN std_logic;
        reset : IN std_logic;
        vram_addr : OUT std_logic_vector(13 DOWNTO 0);
        vram_data : OUT std_logic_vector(7 DOWNTO 0);
        vram_write : OUT std_logic;
        loadsr1 : OUT std_logic;
        loadsr2 : OUT std_logic;
        sr1_data : OUT std_logic_vector(7 DOWNTO 0);
        sr2_data : OUT std_logic_vector(7 DOWNTO 0);
        vram_in : IN std_logic_vector(7 DOWNTO 0);
        attrib_out : OUT std_logic_vector(1 DOWNTO 0);
        write_color : OUT std_logic;
        ppu_going : out std_logic;
        color_data : OUT std_logic_vector(7 DOWNTO 0);
        color_addr : OUT std_logic_vector(3 DOWNTO 0);
        vram_read : OUT std_logic
    );
END COMPONENT;

SIGNAL net33 : std_logic;
SIGNAL bits : std_logic_vector(3 DOWNTO 0);
SIGNAL net35 : std_logic_vector(0 TO 3);
SIGNAL net50 : std_logic;
SIGNAL net36 : std_logic_vector(0 TO 7);
SIGNAL net34 : std_logic_vector(0 TO 7);
SIGNAL net49 : std_logic;
SIGNAL net48 : std_logic_vector(0 TO 7);
SIGNAL net43 : std_logic;
SIGNAL net20 : std_logic_vector(0 TO 5);
SIGNAL net46 : std_logic_vector(0 TO 13);
SIGNAL net47 : std_logic_vector(0 TO 7);
SIGNAL net44 : std_logic;
SIGNAL net45 : std_logic_vector(0 TO 7);
ALIAS clock_wire : std_ulogic IS clock;
```

```
BEGIN


    \I5\ : decoder
        PORT MAP(
            din(5 DOWNTO 0) => net20(0 TO 5),
            rout(7 DOWNTO 0) => r(7 DOWNTO 0),
            bout(7 DOWNTO 0) => b(7 DOWNTO 0),
            gout(7 DOWNTO 0) => g(7 DOWNTO 0)
        );

    \I4\ : colorgen
        PORT MAP(
            clk => clock_wire,
            we => net33,
            addr_vram(3 DOWNTO 0) => net35(0 TO 3),
            addr_mux(3 DOWNTO 0) => bits(3 DOWNTO 0),
            di(7 DOWNTO 0) => net34(0 TO 7),
            do(5 DOWNTO 0) => net20(0 TO 5)
        );

--     \I2\ : vram
--         PORT MAP(
--             clk => clock_wire,
--             addr(13 DOWNTO 0) => net46(0 TO 13),
--             din(7 DOWNTO 0) => net45(0 TO 7),
--             dout(7 DOWNTO 0) => net36(0 TO 7),
--             wr => net43,
--             rd => net44
--         );

    \I1\ : shift_reg
        PORT MAP(
            output(1 DOWNTO 0) => bits(1 DOWNTO 0),
            data1(7 DOWNTO 0) => net48(0 TO 7),
            data2(7 DOWNTO 0) => net47(0 TO 7),
            load1 => net50,
            load2 => net49,
            clk => clock_wire
        );

    \I0\ : control
        PORT MAP(
            cpu_in(7 DOWNTO 0) => cpu_data(7 DOWNTO 0),
            addr_in(15 DOWNTO 0) => addr(15 DOWNTO 0),
            cpu_read => cpu_r,
            cpu_write => cpu_w,
```

```
            ppu_clock => clock_wire,
            reset => rst,
            vram_addr(13 DOWNTO 0) => v_addr(13 downto 0),
-- net46(0 TO 13),
            vram_data(7 DOWNTO 0) => v_data(7 downto 0), --
net45(0 TO 7),
            vram_write => v_write, --net43,
            loadsr1 => net50,
            loadsr2 => net49,
            sr1_data(7 DOWNTO 0) => net48(0 TO 7),
            sr2_data(7 DOWNTO 0) => net47(0 TO 7),
            vram_in(7 DOWNTO 0) => v_in(7 downto 0), --
net36(0 TO 7),
            attrib_out(1 DOWNTO 0) => bits(3 DOWNTO 2),
            write_color => net33,
            ppu_going => ppu_go,
            color_data(7 DOWNTO 0) => net34(0 TO 7),
            color_addr(3 DOWNTO 0) => net35(0 TO 3),
            vram_read => v_read --net44
        );

END schematic;
```

```
-- control.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity control is

  port (
    cpu_in     : in  std_logic_vector(7 downto 0);
    addr_in    : in  std_logic_vector(15 downto 0);
    cpu_read   : in  std_logic;
    cpu_write  : in  std_logic;
    ppu_clock  : in  std_logic;
    reset      : in  std_logic;
    vram_addr  : out std_logic_vector(13 downto 0);
    vram_data  : out std_logic_vector(7 downto 0);
    vram_write : out std_logic;
    loadsr1 : out std_logic;
    loadsr2 : out std_logic;
    sr1_data : out std_logic_vector(7 downto 0);
    sr2_data : out std_logic_vector(7 downto 0);
    vram_in : in std_logic_vector(7 downto 0);
```

```
    attrib_out : out std_logic_vector(1 downto 0);
    write_color : out std_logic;
    ppu_going : out std_logic;
    color_data : out std_logic_vector(7 downto 0);
    color_addr : out std_logic_vector(3 downto 0);
    vram_read  : out std_logic);

end control;
architecture behavior of control is


signal p2002, p2001, p2000, p2007 : std_logic_vector(7
downto 0);
signal p2006top, p2006bot : std_logic_vector(7 downto 0);
--signal loaded2006 : std_logic;
signal nametable, attrib, pattern1, pattern2 :
std_logic_vector(13 downto 0);
signal namebyte, attribyte, pbyte1, pbyte2 :
std_logic_vector(7 downto 0);
--signal go_do : std_logic;
signal current_state, next_state : std_logic_vector(3
downto 0);
--gnal hcount : std_logic_vector(7 downto 0);
--signal vcount : std_logic_vector(6 downto 0);
signal pixel_count, line_count : std_logic_vector(7 downto
0);
--signal reset_pixels : std_logic;
signal writex, writey : std_ulogic;
signal shit, shit1, shit2, loaded2006, go_do, reset_pixels
: std_ulogic;
signal clock_tick : std_ulogic;

constant IDLE : std_logic_vector(3 downto 0) := "0000";
constant FETCH : std_logic_vector(3 downto 0) := "0001";
constant FETCH_A : std_logic_vector(3 downto 0) := "0010";
constant FETCH_P1 : std_logic_vector(3 downto 0) := "0011";
constant FETCH_P2 : std_logic_vector(3 downto 0) := "0100";
constant LOADPATS : std_logic_vector(3 downto 0) := "0101";
constant GO : std_logic_vector(3 downto 0) := "0110";
constant GET_ATTR : std_logic_vector(3 downto 0) := "0111";
constant GET_P1 : std_logic_vector(3 downto 0) := "1000";
constant GET_P2 : std_logic_vector(3 downto 0) := "1001";
constant LOAD : std_logic_vector(3 downto 0) := "1010";


begin  -- behavior
```

```
  process(ppu_clock, reset)
    begin
      if reset = '1' then
        shit <= '1';
        current_state <= IDLE;
        clock_tick <= '0';
      elsif ppu_clock = '1' and ppu_clock'event then
        clock_tick <= not(clock_tick);
        if addr_in = "0010000000000010" and cpu_read = '1'
then
          p2002 <= "00000000";
          shit <= '0';
          current_state <= IDLE;
        else
          current_state <= next_state;
          shit <= '0';
        end if;
      end if;
    end process;

  process(writex, writey)
    begin
      if writex = '0' and writey = '0' then
        attrib_out <= attribyte(1 downto 0);
      elsif writex = '1' and writey = '0' then
        attrib_out <= attribyte(3 downto 2);
      elsif writex = '0' and writey = '1' then
        attrib_out <= attribyte(5 downto 4);
      elsif writex = '1' and writey = '1' then
        attrib_out <= attribyte(7 downto 6);
      end if;
    end process;

  process(ppu_clock, reset_pixels)
    begin
      if reset_pixels = '1' then
        pixel_count <= "00000000";
      elsif ppu_clock = '1' and ppu_clock'event then
        if pixel_count = "11111111" then
          pixel_count <= "00000000";
        else
          pixel_count <= pixel_count + 1;
        end if;
      end if;
    end process;

  process(pixel_count, reset_pixels)
```

```
    begin
      if reset_pixels = '1' then
        line_count <= "00000000" ;
      elsif pixel_count = "11111111" and line_count = 239
then
        line_count <= "00000000" ;
      elsif pixel_count = "11111111" then
        line_count <= line_count + 1;
      end if;
    end process;

  process(pixel_count, line_count, reset, reset_pixels)
    begin
      if reset = '1' or reset_pixels = '1' then
        writex <= '0';
        writey <= '0';
      else
        if pixel_count(3 downto 0) = "1111" then
          writex <= not(writex);
        end if;
        if line_count(3 downto 0) = "1111" then
          writey <= not(writey);
        end if;
      end if;
    end process;

--  process(cpu_in, addr_in, cpu_read, cpu_write,
current_state, reset, clock_tick)

    vram_write <= cpu_write when addr_in =
"0010000000000111" else
                  '0';

    vram_read <= '0' when cpu_write = '1' and addr_in =
"0010000000000111" else
                 '1';

    vram_data <= cpu_in when addr_in = "0010000000000111"
else
                 "00000000";


    ppu_going <= '0' when current_state = IDLE else '1';

    process(clock_tick, current_state)
    begin
```

```
      reset_pixels <= '0';
      loadsr1 <= '0';
      loadsr2 <= '0';
--      vram_read <= '0';

      case current_state is
        when IDLE =>
          if shit = '1' then
            loaded2006 <= '0';
            go_do <= '0';
            reset_pixels <= '1';
            next_state <= IDLE;
          end if;
          -- here it handles all vram writes, etc.
          vram_addr <= p2006top(5 downto 0) & p2006bot(7
downto 0);


          -- for sram mux
--           if addr_in(3 downto 0) = "0111" then
--             write2007 <= '1';
--           else
--             write2007 <= '0';
--           end if;

--            shit2 <= '1';
            if addr_in(15 downto 12) = "0010" then
              if cpu_write = '1' then
                shit1 <= '1';
                if addr_in(3 downto 0) = "0000" then
                  p2000 <= cpu_in;
                  next_state <= IDLE;
                elsif addr_in(3 downto 0) = "0001" then
                  p2001 <= cpu_in;
                  if go_do = '0' then
                    go_do <= '1';
                    next_state <= IDLE;
                  else
                    go_do <= '0';
                    next_state <= FETCH;
                    --tilecount <= "0000000000";
                    -- will need to fetch first nametable
byte
                    nametable <= "10" & p2000(1 downto 0) &
"0000000000";
                    vram_addr <= "10" & p2000(1 downto 0) &
"0000000000";
```

```
                  end if;
                elsif addr_in(3 downto 0) = "0110" then
                  if loaded2006 = '0' then
                    p2006top <= cpu_in;
                    loaded2006 <= '1';
                    next_state <= IDLE;
                  else
                    p2006bot <= cpu_in;
                    loaded2006 <= '0';
                    next_state <= IDLE;
                  end if;
                elsif addr_in(3 downto 0) = "0111" then
                  p2007 <= cpu_in;
                  --vram_data <= cpu_in;
                  --vram_write <= '1';
                  next_state <= IDLE;
                  if p2000(2) = '1' then
                    p2006bot <= p2006bot + 32;
                  else
                    p2006bot <= p2006bot + 1;
                  end if;
                end if;
                if p2006top(5 downto 0) & p2006bot(7 downto
4) = "1111110000" then
                  -- writing to pallete
                  color_addr <= p2006bot(3 downto 0);
                  write_color <= '1';
                  color_data <= cpu_in; --p2007;
                else
                  write_color <= '0';
                end if;
              end if;
            end if;




        when FETCH =>
--          if ppu_clock = '1' and ppu_clock'event then
          -- commenting for latching testing purposes
            --vram_addr <= nametable;
--            vram_read <= '1';
            namebyte <= vram_in;
            if p2000(2) = '1' then
              nametable <= nametable + 32;
            else
              nametable <= nametable + 1;
```

```
            end if;

            attrib <= nametable + 960;
            vram_addr <= nametable + 960;
            if attrib = "10001111010000" then
              attrib <= "10001111011000";
              vram_addr <= "10001111011000";
            end if;

            reset_pixels <= '1';
            -- location of first attribute byte
            next_state <= FETCH_A;
--          end if;

        when FETCH_A =>
--          if ppu_clock = '1' and ppu_clock'event then
            --vram_addr <= attrib;
--            vram_read <= '1';
            attribyte <= vram_in;
           -- now find location of pattern table byte 1
            if p2000(4) = '1' then
              pattern1 <= "01" & namebyte(7 downto 0) & '0'
& line_count(2 downto 0);
              vram_addr <= "01" & namebyte(7 downto 0) &
'0' & line_count(2 downto 0);
            else
              pattern1 <= "00" & namebyte(7 downto 0) & '0'
& line_count(2 downto 0);
              vram_addr <= "00" & namebyte(7 downto 0) &
'0' & line_count(2 downto 0);
            end if;
            next_state <= FETCH_P1;
--          end if;

        when FETCH_P1 =>
--          if ppu_clock = '1' and ppu_clock'event then
--            vram_addr <= pattern1;
--            vram_read <= '1';
            pbyte1 <= vram_in;
            -- now find location of pattern table byte 2
            if p2000(4) = '1' then
              pattern2 <= "01" & namebyte(7 downto 0) & '1'
& line_count(2 downto 0);
              vram_addr <= "01" & namebyte(7 downto 0) &
'1' & line_count(2 downto 0);
            else
```

```
              pattern2 <= "00" & namebyte(7 downto 0) & '1'
& line_count(2 downto 0);
              vram_addr <=  "00" & namebyte(7 downto 0) &
'1' & line_count(2 downto 0);
            end if;
            next_state <= FETCH_P2;
--          end if;

        when FETCH_P2 =>
--          if ppu_clock = '1' and ppu_clock'event then
--            vram_addr <= pattern2;
--            vram_read <= '1';
            pbyte2 <= vram_in;
            -- now both shift registers are loaded with the
first two bytes to
            -- make the first 8 pixels
            reset_pixels <= '1';
            next_state <= LOADPATS;
--          end if;

        when LOADPATS =>
--          if ppu_clock = '1' and ppu_clock'event then
            sr1_data <= pbyte1;
            sr2_data <= pbyte2;
            loadsr1 <= '1';
            loadsr2 <= '1';
            reset_pixels <= '1';
            vram_addr <= nametable;
            next_state <= GO;
--          end if;

        when GO =>
          -- this is the main function loop
          -- will need to grab a new nametable byte before
8 pixels are drawn
          --      will need to update address after every
fetch and at end of line
          --      if pixel_count = 255 then need to
subtract 32 or 32*32 from
          --      nametable address based on p2000
          -- will need to grab a new attribute byte if
necessary for next tile
          --      attribute table address will be updated
in LOAD state
          -- will need to grab two pattern table bytes and
load them when
          --      pixel_count(2 downto 0) = "111"
```

```
          -- already incremented nametable byte address in
FETCH state
--          if ppu_clock = '1' and ppu_clock'event then
--            vram_addr <= nametable;
--            vram_read <= '1';
          vram_addr <= attrib;
          if attrib = "10001111010000" then
              attrib <= "10001111011000";
              vram_addr <= "10001111011000";
          end if;
          namebyte <= vram_in;
          next_state <= GET_ATTR;
  --              end if;

        when GET_ATTR =>
--          if ppu_clock = '1' and ppu_clock'event then
--          vram_addr <= attrib;
--            vram_read <= '1';
            attribyte <= vram_in;
            -- update pattern 1 address
            if p2000(4) = '1' then
              pattern1 <= "01" & namebyte(7 downto 0) & '0'
& line_count(2 downto 0);
              vram_addr <=  "01" & namebyte(7 downto 0) &
'0' & line_count(2 downto 0);
            else
              pattern1 <= "00" & namebyte(7 downto 0) & '0'
& line_count(2 downto 0);
              vram_addr <=  "00" & namebyte(7 downto 0) &
'0' & line_count(2 downto 0);
            end if;
            next_state <= GET_P1;
--          end if;

        when GET_P1 =>
--          if ppu_clock = '1' and ppu_clock'event then
--            vram_addr <= pattern1;
--            vram_read <= '1';
            pbyte1 <= vram_in;
            if p2000(4) = '1' then
              pattern2 <= "01" & namebyte(7 downto 0) & '1'
& line_count(2 downto 0);
              vram_addr <=  "01" & namebyte(7 downto 0) &
'1' & line_count(2 downto 0);
            else
              pattern2 <= "00" & namebyte(7 downto 0) & '1'
& line_count(2 downto 0);
```

```
              vram_addr <= "00" & namebyte(7 downto 0) &
'1' & line_count(2 downto 0);
            end if;
            next_state <= GET_P2;
--          end if;


        when GET_P2 =>
--          if ppu_clock = '1' and ppu_clock'event then
--            vram_addr <= pattern2;
--            vram_read <= '1';
            pbyte2 <= vram_in;
            next_state <= LOAD;
--          end if;

        when LOAD =>
--         if ppu_clock = '1' and ppu_clock'event then
            if pixel_count(2 downto 0) = "111" then
              loadsr1 <= '1';
              sr1_data <= pbyte1;
              loadsr2 <= '1';
              sr2_data <= pbyte2;
              -- name table address update
              -- update by 1 or 32 if tile is done
              -- subtract by 32 or 32*32 if at end of line,
but not row of tiles
              -- update by 1 or 32 if row is done
              if pixel_count = 255 then
                if line_count(2 downto 0) = "111" then  --
done with a row of tiles
                  if p2000(2) = '1' then
                    nametable <= nametable + 32;
                  else
                    nametable <= nametable + 1;
                  end if;
                else
                  if p2000(2) = '1' then
                    nametable <= nametable - 992;  --
offsets may be wrong
                  else
                    nametable <= nametable - 31;
                  end if;
                end if;
              else
                -- pixel_count(2 downto 0) = "111" but not
at end of line
                if p2000(2) = '1' then
```

```
                nametable <= nametable + 32;
                vram_addr <= nametable + 32;
              else
                nametable <= nametable + 1;
                vram_addr <= nametable + 1;
              end if;
            end if;                    -- end if 255 for
nametable

            -- update attribute address
            if pixel_count = 255 then
              if line_count(4 downto 0) = "11111" then
                attrib <= attrib + 1;
              else
                attrib <= attrib - 7;
              end if;
            else
              if pixel_count(4 downto 0) = "11111" then
                attrib <= attrib + 1;
              end if;
            end if;

            -- check if at end of screen
            if pixel_count = 255 and line_count = 240
then
              nametable <= "01" & p2000(1 downto 0) &
"0000000000";
              vram_addr <=  "01" & p2000(1 downto 0) &
"0000000000";
              attrib <= "01" & p2000(1 downto 0) &
"1111000000";  -- 960 offset
              reset_pixels <= '1';
              next_state <= FETCH;
            else
              next_state <= GO;
            end if;
--          next_state <= GO;
          else
            next_state <= LOAD;
          end if;
--        end if;


        when others => null;
      end case;
    end process;
```

```
end behavior;
```

```
-- meminterface.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity mem_interface is

  port (
    data_to_mem : in std_logic_vector(15 downto 0);
    write_to_mem : in std_logic;
    read_from_mem : in std_logic;
    chip_enable : out std_logic;
    mem_bus : inout std_logic_vector(15 downto 0);
    data_from_mem : out std_logic_vector(15 downto 0);
    upper_en : out std_logic;
    lower_en : out std_logic;
    output_en : out std_logic;
    write_en : out std_logic
  );

end mem_interface;

architecture behavior of mem_interface is

begin  -- behavior

  -- chip enable is active low
  chip_enable <= '0' when write_to_mem = '1' or
read_from_mem = '1'
                 else '1';

  upper_en <= '0';                      -- enable upper
byte
  lower_en <= '0';                      -- enable lower
byte

  data_from_mem <= mem_bus when read_from_mem = '1' else
                   "XXXXXXXXXXXXXXXX";

  write_en <= not(write_to_mem);        -- write_to_mem is
active high
  output_en <= not(read_from_mem);      -- read_from_mem is
active high
```

```
  mem_bus <= data_to_mem when write_to_mem = '1' else
             "ZZZZZZZZZZZZZZZZ";


end behavior;
```

```
-- decoder.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
--use ieee.math_real.all;
use ieee.std_logic_arith.all;



entity decoder is
  port (
    din         : in  std_logic_vector(5 downto 0);
    rout        : out std_logic_vector(7 downto 0);
    bout        : out std_logic_vector(7 downto 0);
    gout        : out std_logic_vector(7 downto 0)

    );
end decoder;

architecture behavior of decoder is


  signal r : std_logic_vector(7 downto 0);
  signal g : std_logic_vector(7 downto 0);
  signal b : std_logic_vector(7 downto 0);



begin  -- imp
--changed
  rout <= "11111111" when din = "100000" or din = "000000"
or din = "001101" else
--  rout <= "11111111" when din = "100000" or din =
"001101" else
          "10111011" when din = "000110" else
          "11100100" when din = "010110" else
          "11111111" when din = "100110" or din = "001101"
else
          "00000000" when din = "001010" else
```

```
          "00000111" when din = "011010" else
          "01001011" when din = "101010" else
          "11111111" when din = "001101" else
          "01000110" when din = "000010" else
          "01101111" when din = "010010" else
          "10110110" when din = "100010" else
--          "00000000" when din = "000000" else
          "00000000";

  gout <= "11111111" when din = "100000" or din = "000000"
or din = "001101" else
--  gout <= "11111111" when din = "100000" or din =
"001101" else
          "00011100" when din = "000110" else
          "01000100" when din = "010110" else
          "10001100" when din = "100110" else
          "11111111" when din = "001101" else
          "10001110" when din = "001010" else
          "10110111" when din = "011010" else
          "11111111" when din = "101010" or din = "001101"
else
          "00110100" when din = "000010" else
          "01011101" when din = "100010" else
          "10100101" when din = "100010" else
--          "00000000" when din = "000000" else
          "00000000";

  bout <= "11111111" when din = "100000" or din = "000000"
or din = "001101" else
 -- bout <= "11111111" when din = "100000" or din =
"001101" else
          "00001110" when din = "000110" else
          "00110111" when din = "010110" else
          "01111110" when din = "100110" else
          "11111111" when din = "001101" else
          "00000110" when din = "001010" else
          "00101111" when din = "011010" else
          "01110111" when din = "101010" else
          "11111111" when din = "001101" else
          "11001010" when din = "000010" else
          "11110011" when din = "010010" else
          "11111111" when din = "100010" else
   --       "11111111" when din = "000000" else
          "00000000" ;


--   process(Clk)
```

```
--   begin

--     if ((din = "100000") or (din = "000000") or (din =
"001101"))  -- 1, 2,3, 4
--       r <= "11111111";
--       g <= "11111111";
--       b <= "11111111";
--     elsif (din = "000110") then  -- 5
--       r <= "10111011";
--       g <= "00011100";
--       b <= "00001110";
--     elsif (din = "010110") then  -- 6
--       r <= "11100100";
--       g <= "01000100";
--       b <= "00110111";
--     elsif (din = "100110") then  --7
--       r <= "11111111";
--       g <= "10001100";
--       b <= "01111110";
--     elsif (din = "001101") then  -- 8
--       r <= "11111111";
--       g <= "11111111";
--       b <= "11111111";
--     elsif (din = "001010") then  -- 9
--       r <= "00000000";
--       g <= "10001110";
--       b <= "00000110";
--     elsif (din = "011010") then  -- 10
--       r <= "00000111";
--       g <= "10110111";
--       b <= "00101111";
--     elsif (din = "101010") then  -- 11
--       r <= "01001011";
--       g <= "11111111";
--       b <= "01110111";
--     elsif (din = "001101") then  -- 12
--       r <= "11111111";
--       g <= "11111111";
--       b <= "11111111";
--     elsif (din = "000010") then  -- 13
--       r <= "01000110";
--       g <= "00110100";
--       b <= "11001010";
--     elsif (din = "010010") then  -- 14
--       r <= "01101111";
--       g <= "01011101";
--       b <= "11110011";
```

```
--     elsif (din = "100010")   -- 15
--       r <= "10110110";
--       g <= "10100101";
--       b <= "11111111";
--   else
--                         -- unknown
--       r <= "00000000";
--       g <= "00000000";
--       b <= "00000000";

--   end if;
-- end process;

-- rout <= r;
-- gout <= g;
-- bout <= b;


end behavior;
```

```
-- colorgen.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity colorgen is
  port (
    Clk         : in  std_logic;
    WE          : in  std_logic;
 --   EN          : in  std_logic;
    addr_vram   : in  std_logic_vector(3 downto 0);
    addr_mux    : in  std_logic_vector(3 downto 0);
    di          : in std_logic_vector(7 downto 0);
    do          : out std_logic_vector(5 downto 0)
    );
end colorgen;
architecture behavior of colorgen is

   type ram_type is array(15 downto 0) of
std_logic_vector(5 downto 0);
--   signal RAM : ram_type;
constant RAM : ram_type :=
  ("000000", "100000", "000000", "000000",
   "001101", "000110", "010110", "100110",
   "001101", "001010", "011010", "101010",
   "001101", "000010", "010010", "100010");
```

```
begin
  process(we, addr_mux, addr_vram, di)
  begin
--    if Clk'event and Clk = '1' then
--      if en = '1' then
--         if we = '1' then
--           RAM(conv_integer(addr_vram)) <= di;
--           do <= di(5 downto 0);
--         else
          do <= RAM(conv_integer(addr_mux))(5 downto 0);
--        end if;
--      end if;
--    end if;
  end process;

end behavior;
```

```
--Create Entity:
--Library=NES,Cell=shift_reg,View=entity
--Time:Wed May  5 15:18:16 2004
--By:neel

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY shift_reg IS
    PORT(
        output : OUT std_logic_vector(1 DOWNTO 0);
        data1 : IN std_logic_vector(7 DOWNTO 0);
        data2 : IN std_logic_vector(7 DOWNTO 0);
        load1 : IN std_logic;
        load2 : IN std_logic;
        clk : IN std_logic
    );
END shift_reg;
architecture behavior of shift_reg is

signal shift1, shift2 : std_logic_vector(7 downto 0);
signal count8 : std_logic_vector(2 downto 0);

begin  -- imp

    process(clk, load1, load2)
      begin
        if clk = '1' and clk'event then
          if (load1 = '1' or load2 = '1') then
```

```
            count8 <= "000";
            if load1 = '1' then
              shift1 <= data1;
            end if;
            if load2 = '1' then
              shift2 <= data2;
            end if;
          end if;
        for i in 6 to 0 loop
          shift1(i + 1) <= shift1(i);
          shift2(i + 1) <= shift2(i);
        end loop;  -- i

        end if;
      end process;

      output <= shift2(7) & shift1(7);

end behavior;
```

```
-- sram_mux.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity sram_mux is

  port (
    ppu_active  : in std_logic;
    v_read  :  in std_logic;
    v_write :  in std_logic;
    cpu_read : in std_logic;
    cpu_write : in std_logic;
    v_data : in std_logic_vector(7 downto 0);
    cpu_data : in std_logic_vector(7 downto 0);
    --writing_2007 :in std_logic;
    sram_addr_cpu  : in std_logic_vector(15 downto 0);
    sram_addr_ppu  : in std_logic_vector(13 downto 0);
    sram_data_out : out std_logic_vector(15 downto 0);
    sram_read : out std_logic;
    sram_write : out std_logic;
    sram_addr : out std_logic_vector(17 downto 0));
end sram_mux;

architecture behavior of sram_mux is
```

```
--signal sram_addr_temp : std_logic_vector(17 downto 0);

begin  -- behavior


--         if ppu_active = '1' or sram_addr_cpu =
"0010000000000111" then
--           sram_addr_temp = "0000" & sram_addr_ppu;
--         else
--           sram_addr_temp = "1000" & sram_addr_cpu;
--         end if;
--       end if;
--     end process;

--     sram_addr <= sram_addr_temp;
--     sram_data_out <= "00000000" & v_data;

  sram_addr <= "0000" & sram_addr_ppu when ppu_active = '1'
or sram_addr_cpu = "0010000000000111" else
               "10" & sram_addr_cpu;

  sram_read <= v_read when ppu_active = '1' or
sram_addr_cpu = "0010000000000111" else
              cpu_read;

  sram_write <= v_write when ppu_active = '1' or
sram_addr_cpu = "0010000000000111" else
               cpu_write;

  sram_data_out <= "00000000" & v_data when ppu_active =
'1' or sram_addr_cpu = "0010000000000111" else
                  "00000000" & cpu_data;

end behavior;
```

References

"Free-6502 Interface."

http://www.free-ip.com/6502/interface.htm

"NES Development"

http://nesdev.parodius.com

"NES Palette Generator"

http://nesdev.parodius.com/kevin_palette.txt

"Nintendo Entertainment System Documentation v. 0.40."

http://db.gamefaqs.com/console/nes/file/nes_tech.txt

Ueda et al. "TV Game System Having Reduced Memory Needs." United States Patent #4,824,106. April 25, 1989.