Final Project

16.575 - FPGA Logic Design Techniques

Michael Nickelson

December 4, 2014

Contents

1 Summary **1**

2 Top-Level Module **3**

3 VGA Controller **5**

4 RAM **8**

5 RAM Initialization **10**

6 Update Logic **11**

7 Testbench **14**

8 Figures **15**

Summary

This project implements Conway's Game of Life in VHDL. It is targeted for use on the Altera Cyclone IV as present in the DE-0 Nano development board.

The implementation includes a top-level Game of Life module which implements a block of VRAM, a VGA Controller, and an update element. Within the VRAM there is a block RAM pulled from the Altera superfunction library. Inside the VGA Controller are two parameterized counters to produce hSync and vSync, and a clock divider to provide a 25MHz enable signal to the counters. The block diagram of this setup can be seen in Figure 8.1.

The VGA controller used is the same as what was developed for HW5 with the only change being delay of hSyncL_O, vSyncL_O, vidEnable_O, and newFrame_O to work with the RAM block.

The memory block itself is imported from Altera's mega function library so the source is not included. The parameters used for generation are a 32bit by 2048 line, unregistered, two port RAM.

The memory is initialized using an mif file. In this instance the stored data is a single blinker. I have chosen this shape as it is easy to verify using waveforms and it demonstrates all of the possible cellular activities (survival, death, and birth).

The update logic is implemented as a state machine (Figure 8.2) which waits in an idle state until the current configuration has been displayed a set number of times. This value is set up as a generic which is passed at instantiation and can allow up to 1s between updates. There is room for improvement in the implementation of the state machine, particularly the transition from updating back to the idle state. However, given the time constraints of this project the focus was on functionality.

To calculate the next step, golUpdate.vhd uses four 256-bit registers. One each to represent the current states of the last, current, and next lines to be calculated and the next state of the current line. Once these registers are loaded the next state of each bit in the current line is calculated based on the state of its 8 neighbors. This value is saved back to the updatedLine register. This register is then loaded back into RAM in the addresses of the original line. Once the last line is updated the system returns to the idle state.

The testbench generates a clock and sends reset to the system then allows the system to run with no further input.

There is a small amount of I/O so pin selection is not terribly complicated. The clock input is determined by the FPGA itself and is located at pin R8. For reset one of the available pushbuttons is used. It is located at pin J15. The outputs hSync, vSync, and pixel are chosen from pins available

at the GPIO header. Pins A3, C3, and D3 have been chosen. These 3 are all available on bank 8.

The intent was to select pins compatible with the Papilio VGA Wing available from Sparkfun Electronics (https://www.sparkfun.com/products/11569) however due to a mechanical incompatibility and the fact that I do not have a monitor to experiment with, this project has been tested in simulation only.

Figures 8.3 through 8.6 show a complete cycle of the blinker that is initialized in memory. It alternates between a 3x1 line segment and a 1x3 line segment and is situated in the top left corner of the screen for convenience.

The total chip utilization is shown below:

Family Cyclone IV E
Device EP4CE22F17C6
Total logic elements 1,762 / 22,320 (8 %)
Total combinational functions 1,537 / 22,320 (7 %)
Dedicated logic registers 1,123 / 22,320 (5 %)
Total registers 1123
Total pins 5 / 154 (3 %)
Total memory bits 65,536 / 608,256 (11 %)

At 85° C the maximum clock speed is 112MHz so timing is not a concern for this project.

Top-Level Module

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

-- This is the top-level module for Conway's Game of Life. This level primarily
-- instantiates and ties together other modules.
-- The only logical element in this piece is an and gate which ties together
-- the pixel_O signal from VRAM and vidEnable_O from the VGA Controller.

entity gol is
  port(
    clk_I    : in  std_logic;
    reset_I  : in  std_logic;
    hSyncL_O : out std_logic;
    vSyncL_O : out std_logic;
    pixel_O  : out std_logic);
end entity;

architecture rtl of gol is
  signal i_newFrame     : std_logic;
  signal i_dispAddr     : std_logic_vector(15 DOWNTO 0);
  signal i_updateAddr   : std_logic_vector(10 DOWNTO 0);
  signal i_videoEnable  : std_logic;
  signal i_pixel        : std_logic;
  signal i_updatedData  : std_logic_vector(31 DOWNTO 0);
  signal i_dataToUpdate : std_logic_vector(31 DOWNTO 0);
  signal i_updateEnable : std_logic;

  component vgaController
    port(
      clk_I        : in  std_logic;
      reset_I      : in  std_logic;
      hSyncL_O     : out std_logic;
      vSyncL_O     : out std_logic;
      vidEnable_O  : out std_logic;
      newFrame_O   : out std_logic;
      memAddress_O : out std_logic_vector(15 DOWNTO 0));
  end component;

  component myVRAM
    PORT(
      clk_I            : IN  STD_LOGIC := '1';
      reset_I          : IN  STD_LOGIC;
      updateAddress_I  : IN  STD_LOGIC_VECTOR(10 DOWNTO 0);
      displayAddress_I : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
      updateData_I     : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
      writeEnable_I    : IN  STD_LOGIC := '0';
      updateData_O     : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
      pixel_O          : OUT STD_LOGIC
    );
  end component;

  component golUpdate
    generic(
      staticFrames : std_logic_vector(4 DOWNTO 0));
    port(
      clk_I         : in  std_logic;
      reset_I       : in  std_logic;
      newFrame_I    : in  std_logic;
      oldData_I     : in  std_logic_vector(31 DOWNTO 0);
      writeEnable_O : out std_logic;
      newData_O     : out std_logic_vector(31 DOWNTO 0);
```

```
      address_O     : out std_logic_vector(10 DOWNTO 0));
  end component;

begin
  vga : vgaController
    port map(
      clk_I        => clk_I,
      reset_I      => reset_I,
      hSyncL_O     => hSyncL_O,
      vSyncL_O     => vSyncL_O,
      vidEnable_O  => i_videoEnable,
      newFrame_O   => i_newFrame,
      memAddress_O => i_dispAddr);

  ram : myVRAM
    port map(
      clk_I            => clk_I,
      reset_I          => reset_I,
      updateAddress_I  => i_updateAddr,
      displayAddress_I => i_dispAddr,
      updateData_I     => i_updatedData,
      writeEnable_I    => i_updateEnable,
      updateData_O     => i_dataToUpdate,
      pixel_O          => i_pixel);

  updater : golUpdate
    generic map(
      staticFrames => std_logic_vector(to_unsigned(2, 5)))
    port map(
      clk_I         => clk_I,
      reset_I       => reset_I,
      newFrame_I    => i_newFrame,
      oldData_I     => i_dataToUpdate,
      writeEnable_O => i_updateEnable,
      newData_O     => i_updatedData,
      address_O     => i_updateAddr);

pixel_O <= i_pixel AND i_videoEnable;
end architecture rtl;
```

VGA Controller

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

-- This project provides a VGA controller based on a single counter instantiated
-- twice to provide hSync and vSync signals.

entity vgaController is
  port(
    clk_I        : in  std_logic;
    reset_I      : in  std_logic;
    hSyncL_O     : out std_logic;
    vSyncL_O     : out std_logic;
    vidEnable_O  : out std_logic;
    newFrame_O   : out std_logic;
    memAddress_O : out std_logic_vector(15 DOWNTO 0));
end entity;

architecture rtl of vgaController is
  signal i_vgaClock    : std_logic;
  signal i_hVideoOn    : std_logic;
  signal i_vVideoOn    : std_logic;
  signal i_hCarry      : std_logic;
  signal i_vCarry      : std_logic;
  signal i_hPulseL     : std_logic;
  signal i_vPulseL     : std_logic;
  signal i_vSyncEnable : std_logic;
  signal i_hCount      : std_logic_vector(9 DOWNTO 0);
  signal i_vCount      : std_logic_vector(9 DOWNTO 0);

  signal  i_delayHSyncL : std_logic_vector(1 DOWNTO 0);
  signal i_delayVSyncL : std_logic_vector(1 DOWNTO 0);
  signal i_delayVidEnable : std_logic_vector(1 DOWNTO 0);
  signal i_delayNewFrame : std_logic_vector(1 DOWNTO 0);

  component clockDivider
    port(
      clk50_I   : in  std_logic;
      reset_I : in  std_logic;
      clk25_O : out std_logic);
  end component;

  component counter
    generic(
      maxCount     : std_logic_vector(9 DOWNTO 0);
      pulseWidth   : std_logic_vector(9 DOWNTO 0);
      backPorch    : std_logic_vector(9 DOWNTO 0);
      blankSection : std_logic_vector(9 DOWNTO 0);
      displayOn    : std_logic_vector(9 DOWNTO 0);
      frontPorch   : std_logic_vector(9 DOWNTO 0));
    port(
      clk_I     : in  std_logic;
      reset_I   : in  std_logic;
      enable_I  : in  std_logic;
      videoOn_O : out std_logic;
      carry_O   : out std_logic;
      pulseL_O  : out std_logic;
      count_O   : out std_logic_vector(9 DOWNTO 0));
  end component;

begin
  -- Enable for the vSync counter is a combination of the 25MHz clock and the
```

```
  -- begining of the hSync pulse.
  i_vSyncEnable <= i_vgaClock AND i_hCarry;

  memAddress_O <= i_vCount(8 DOWNTO 1) & i_hCount(8 DOWNTO 1);

-- The signals below are delayed so they will match output from the myVRAM
-- module which needs to access RAM.
  hSyncL_O <= i_delayHSyncL(1);
  vSyncL_O <= i_delayVSyncL(1);
-- The actual video signal should only be on when both sync counters are in the
-- active period.
  vidEnable_O <= i_delayVidEnable(1);
-- Alert the update module when a new frame is being displayed.
  newFrame_O <= i_delayNewFrame(1);

  -- Instantiate clockDivider to provide a 25MHz clock enable signal to both
  -- counters.
  vgaClock : clockDivider
    port map(
      clk50_I   => clk_I,
      reset_I => reset_I,
      clk25_O => i_vgaClock);

  -- Instantiate hSync and vSync counters with appropriate generics.
  hCounter : counter
    generic map(
      maxCount     => std_logic_vector(to_unsigned(800, 10)),
      pulseWidth   => std_logic_vector(to_unsigned(96, 10)),
      backPorch    => std_logic_vector(to_unsigned(48, 10)),
      blankSection => std_logic_vector(to_unsigned(80, 10)),
      displayOn    => std_logic_vector(to_unsigned(480, 10)),
      frontPorch   => std_logic_vector(to_unsigned(16, 10)))
    port map(
      clk_I     => clk_I,
      reset_I   => reset_I,
      enable_I  => i_vgaClock,
      videoOn_O => i_hVideoOn,
      carry_O   => i_hCarry,
      pulseL_O  => i_hPulseL,
      count_O   => i_hCount);

  vCounter : counter
    generic map(
      maxCount     => std_logic_vector(to_unsigned(525, 10)),
      pulseWidth   => std_logic_vector(to_unsigned(2, 10)),
      backPorch    => std_logic_vector(to_unsigned(33, 10)),
      blankSection => std_logic_vector(to_unsigned(0, 10)),
      displayOn    => std_logic_vector(to_unsigned(480, 10)),
      frontPorch   => std_logic_vector(to_unsigned(10, 10)))
    port map(
      clk_I     => clk_I,
      reset_I   => reset_I,
      enable_I  => i_vSyncEnable,
      videoOn_O => i_vVideoOn,
      carry_O   => i_vCarry,
      pulseL_O  => i_vPulseL,
      count_O   => i_vCount);

-- The delay is implemented as an inline shift register.
process(clk_I,reset_I)
begin
   if(reset_I='1') then
      i_delayHSyncL <= (others=>'0');
      i_delayVSyncL<= (others=>'0');
      i_delayVidEnable <= (others=>'0');
      i_delayNewFrame <= (others=>'0');
   elsif(rising_edge(clk_I)) then
      --bit shift register
      i_delayHSyncL(1) <= i_delayHSyncL(0);
      i_delayVSyncL(1) <= i_delayVSyncL(0);
      i_delayVidEnable(1) <= i_delayVidEnable(0);
      i_delayNewFrame(1) <= i_delayNewFrame(0);

      i_delayHSyncL(0) <= i_hPulseL;
      i_delayVSyncL(0) <= i_vPulseL;
      i_delayVidEnable(0) <= i_hVideoOn AND i_vVideoOn;
```

```
      i_delayNewFrame(0) <= i_vCarry AND i_vSyncEnable;
   end if;
end process;
end architecture rtl;
```

RAM

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

-- This module instantiates a block of RAM from the Altera superfunction
-- library. It then feeds individual pixels back to the top-level module as
-- needed based on the hSync and vSync counter values.

entity myVRAM is
  PORT(
    clk_I            : IN  STD_LOGIC := '1';
    reset_I          : IN  STD_LOGIC;
    updateAddress_I  : IN  STD_LOGIC_VECTOR(10 DOWNTO 0);
    displayAddress_I : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
    updateData_I     : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
    writeEnable_I    : IN  STD_LOGIC := '0';
    updateData_O     : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    pixel_O          : OUT STD_LOGIC);
end entity;

architecture rtl of myVRAM is
  type PixelDelay_t is array (0 to 1) of std_logic_vector(4 downto 0);
  signal i_displayLine : std_logic_vector(31 DOWNTO 0);
  signal i_myAddress : std_logic_vector(10 DOWNTO 0);
  signal i_pixelDelay : PixelDelay_t;

  component golMemory
    port(
      address_a : IN  STD_LOGIC_VECTOR(10 DOWNTO 0);
      address_b : IN  STD_LOGIC_VECTOR(10 DOWNTO 0);
      clock     : IN  STD_LOGIC := '1';
      data_a    : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
      data_b    : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
      wren_a    : IN  STD_LOGIC := '0';
      wren_b    : IN  STD_LOGIC := '0';
      q_a       : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
      q_b       : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
  end component;

begin

  myRam : golMemory
    port map(
      address_a => updateAddress_I,
      address_b => displayAddress_I(15 DOWNTO 5),
      clock     => clk_I,
      data_a    => updateData_I,
      data_b    => x"00000000",
      wren_a    => writeEnable_I,
      wren_b    => '0',
      q_a       => updateData_O,
      q_b       => i_displayLine
    );

   i_myAddress <= displayAddress_I(15 DOWNTO 5);

-- displayAddress_I is delayed before being fed to pixel_O to allow a cycle for
-- RAM access.
   pixel_O <= i_displayLine(to_integer(unsigned(i_pixelDelay(1))));

--Again, the delay is implemented as a simple inline shift register.
```

```
process(clk_I,reset_I)
begin
   if(reset_I='1') then
      i_pixelDelay <= (others=>(others=>'0'));
   elsif(rising_edge(clk_I)) then
      --bit shift register
      i_pixelDelay(1) <= i_pixelDelay(0);

      i_pixelDelay(0) <= displayAddress_I(4 DOWNTO 0);

   end if;
end process;

end architecture rtl;
```

RAM Initialization

```
-- Copyright (C) 1991-2014 Altera Corporation. All rights reserved.
-- Your use of Altera Corporation's design tools, logic functions
-- and other software and tools, and its AMPP partner logic
-- functions, and any output files from any of the foregoing
-- (including device programming or simulation files), and any
-- associated documentation or information are expressly subject
-- to the terms and conditions of the Altera Program License
-- Subscription Agreement, the Altera Quartus II License Agreement,
-- the Altera MegaCore Function License Agreement, or other
-- applicable license agreement, including, without limitation,
-- that your use is for the sole purpose of programming logic
-- devices manufactured by Altera and sold by Altera or its
-- authorized distributors.  Please refer to the applicable
-- agreement for further details.

-- Quartus II generated Memory Initialization File (.mif)

WIDTH=32;
DEPTH=2048;

ADDRESS_RADIX=UNS;
DATA_RADIX=UNS;

CONTENT BEGIN
        [0..7]  :   0;
        8       :   7;
        [9..2047]   :   0;
END;
```

Update Logic

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

-- This module waits until a specified number of frames have been displayed
-- then updates the playing field by loading one display line at a time then
-- scanning across this line and calculating the next state for bit of memory.

entity golUpdate is
  generic(
-- StaticFrames is the number of times to display the board before updating.
    staticFrames : std_logic_vector(4 DOWNTO 0) := (OTHERS => '1'));
  port(
    clk_I         : in  std_logic;
    reset_I       : in  std_logic;
    newFrame_I    : in  std_logic;
    oldData_I     : in  std_logic_vector(31 DOWNTO 0);
    writeEnable_O : out std_logic;
    newData_O     : out std_logic_vector(31 DOWNTO 0);
    address_O     : out std_logic_vector(10 DOWNTO 0));
end entity;

architecture rtl of golUpdate is
  TYPE State_t IS (idle_st, firstLoad_st, switch_st, load_st, calc_st, update_st, updateToLoad_st,
      switchIdle_st);

  signal i_curState  : State_t;
  signal i_nextState : State_t;

  signal i_loadedData : std_logic_vector(10 DOWNTO 0);

  signal i_frameCount    : std_logic_vector(4 DOWNTO 0);
  signal i_lastLine      : std_logic_vector(255 DOWNTO 0);
  signal i_thisLine      : std_logic_vector(255 DOWNTO 0);
  signal i_nextLine      : std_logic_vector(255 DOWNTO 0);
  signal i_updatedLine   : std_logic_vector(255 DOWNTO 0);
  signal i_baseAddress   : std_logic_vector(10 DOWNTO 0);
  signal i_nextAddress   : std_logic_vector(10 DOWNTO 0);
  signal i_updatedPixels : integer range 0 to 255;
  signal i_neighbors     : std_logic_vector(3 DOWNTO 0);

  signal i_NW : std_logic_vector(3 DOWNTO 0);
  signal i_N  : std_logic_vector(3 DOWNTO 0);
  signal i_NE : std_logic_vector(3 DOWNTO 0);
  signal i_W  : std_logic_vector(3 DOWNTO 0);
  signal i_E  : std_logic_vector(3 DOWNTO 0);
  signal i_SW : std_logic_vector(3 DOWNTO 0);
  signal i_S  : std_logic_vector(3 DOWNTO 0);
  signal i_SE : std_logic_vector(3 DOWNTO 0);

begin
-- i_nextAddress is used when loading the display line after the one currently
-- being calculated.
  i_nextAddress <= i_baseAddress + 8;

-- Sum of a given pixels neighbors in order to calculate next state.
  i_NW <= ("000" & i_lastLine(i_updatedPixels - 1)) when (i_updatedPixels > 0) else "0000";
  i_N  <= ("000" & i_lastLine(i_updatedPixels));
  i_NE <= ("000" & i_lastLine(i_updatedPixels + 1)) when (i_updatedPixels < 239) else "0000";
  i_W <= ("000" & i_thisLine(i_updatedPixels - 1)) when (i_updatedPixels > 0) else "0000";
  i_E <= ("000" & i_thisLine(i_updatedPixels + 1)) when (i_updatedPixels < 239) else "0000";
  i_SW <= ("000" & i_nextLine(i_updatedPixels - 1)) when (i_updatedPixels > 0) else "0000";
  i_S  <= ("000" & i_nextLine(i_updatedPixels));
```

```
  i_SE <= ("000" & i_nextLine(i_updatedPixels + 1)) when (i_updatedPixels < 239) else "0000";
  i_neighbors <= i_NW + i_N + i_NE + i_W + i_E + i_SW + i_S + i_SE;

  process(clk_I, reset_I)
  begin
    if reset_I = '1' then
      i_curState      <= idle_st;
      i_nextState     <= idle_st;
      writeEnable_O   <= '0';
      address_O       <= (OTHERS => '0');
      newData_O       <= (OTHERS => '0');
      i_frameCount    <= (OTHERS => '0');
      i_baseAddress   <= (OTHERS => '0');
      i_lastLine      <= (OTHERS => '0');
      i_thisLine      <= (OTHERS => '0');
      i_nextLine      <= (OTHERS => '0');
      i_loadedData    <= (OTHERS => '0');
      i_updatedLine   <= (OTHERS => '0');
      i_updatedPixels <= 0;

    elsif rising_edge(clk_I) then
      case i_curState is
-- Idle state counts frames then switches to firstLoad_st.
        when idle_st =>
          address_O    <= i_baseAddress;
          if newFrame_I = '1' then
            i_frameCount <= (i_frameCount + '1');
          end if;
          if i_frameCount >= staticFrames then
            i_nextState     <= firstLoad_st;
            i_thisLine      <= (OTHERS => '0');
            i_frameCount    <= (OTHERS => '0');
            i_loadedData    <= (OTHERS => '0');
            i_updatedPixels <= 0;
          end if;
-- First state after idle. Uses i_baseAddress to access RAM.
        when firstLoad_st =>
          if i_loadedData <= 9 then
            address_O    <= i_baseAddress + i_loadedData;
            i_nextLine   <= oldData_I & i_nextLine((i_nextLine'length-1) DOWNTO (oldData_I'length));
            i_loadedData <= i_loadedData + '1';
          else
            i_nextState <= switch_st;
          end if;
-- Shift last/this/next line registers as needed.
        when switch_st =>
          i_loadedData <= (OTHERS => '0');
          writeEnable_O <= '0';
          address_O <= i_baseAddress + 8;
-- It feels like this shouldn't be necessary. The state machine implementation
-- is probably screwy and could be improved with time.
          if i_nextState /= load_st then
            i_thisLine   <= i_nextLine;
            i_lastLine   <= i_thisLine;
          end if;
          i_nextState  <= load_st;
-- Basically the same as firstLoad_st but uses i_nextAddress to access RAM.
        when load_st =>
          if i_loadedData <= 9 then
            address_O    <= i_nextAddress + i_loadedData;
            i_nextLine   <= oldData_I & i_nextLine((i_nextLine'length-1) DOWNTO (oldData_I'length));
            i_loadedData <= i_loadedData + '1';
          else
            i_updatedPixels <= 0;
            i_nextState     <= calc_st;
          end if;
-- Calculate the next state of each pixel based on number of neighbors and
-- store in a temporary register.
        when calc_st =>
            i_loadedData <= (OTHERS => '0');
            i_updatedPixels <= i_updatedPixels + 1;
            if (i_neighbors = 2) then
              i_updatedLine(i_updatedPixels) <= i_thisLine(i_updatedPixels);
            elsif (i_neighbors = 3) then
              i_updatedLine(i_updatedPixels) <= '1';
            else
```

```
              i_updatedLine(i_updatedPixels) <= '0';
            end if;
          if (i_updatedPixels >= 238) then
            i_nextState <= update_st;
          end if;
-- Load new data back into RAM then either continue loading data or go to idle.
        when update_st =>
          if i_baseAddress > 1912 then
            i_nextState <= switchIdle_st;
          elsif i_loadedData >= 9 then
            i_nextState <= updateToLoad_st;
            writeEnable_0 <= '0';
          else
            writeEnable_0 <= '1';
            newData_0     <= i_updatedLine(31 DOWNTO 0);
            i_updatedLine   <= x"00000000" & i_updatedLine(255 DOWNTO 32);
            address_0     <= i_baseAddress + i_loadedData;
            i_loadedData <= i_loadedData + '1';
          end if;
-- Increment baseAddress for loading and then move to either switch_st or
-- idle_st depending on where in the load sequence we are.
        when updateToLoad_st =>
          if i_baseAddress > 1912 then
            i_nextState <= switchIdle_st;
          else
            i_nextState <= switch_st;
          end if;
          if (i_nextState /= switch_st) and (i_nextState /= switchIdle_st) then
            i_baseAddress <= i_baseAddress + 8;
          end if;
-- Reset memory access address and disable write before returning to idle.
        when switchIdle_st =>
            i_nextState <= idle_st;
            i_baseAddress <= (OTHERS => '0');
            writeEnable_0 <= '0';

      end case;
      i_curState <= i_nextState;
    end if;
  end process;
end rtl;
```

Testbench

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY t_gol IS
END ENTITY t_gol;

ARCHITECTURE TestBench OF t_gol IS
  SIGNAL clk_I   : STD_LOGIC := '0';
  SIGNAL reset_I : STD_LOGIC;
  SIGNAL hSyncL_O : STD_LOGIC;
  SIGNAL vSyncL_O : STD_LOGIC;
  signal pixel_O : std_logic;

  constant halfPeriod : time := 10 ns;

begin
  s0 : entity WORK.gol(rtl)
  port map(
    clk_I => clk_I,
    reset_I=> reset_I,
    hSyncL_O=>hSyncL_O,
    vSyncL_O=>vSyncL_O,
    pixel_O=>pixel_O);

  clock : process is
  begin
    clk_I <= '0';
    wait for (halfPeriod);
    clk_I <= '1';
    wait for (halfPeriod);
  end process clock;

  process is
  begin
    reset_I <= '0';
    WAIT FOR (halfPeriod);
    reset_I <= '1';
    WAIT FOR (halfPeriod * 4);
    reset_I <= '0';
    WAIT;
  end process;
END ARCHITECTURE TestBench;
```

Figures

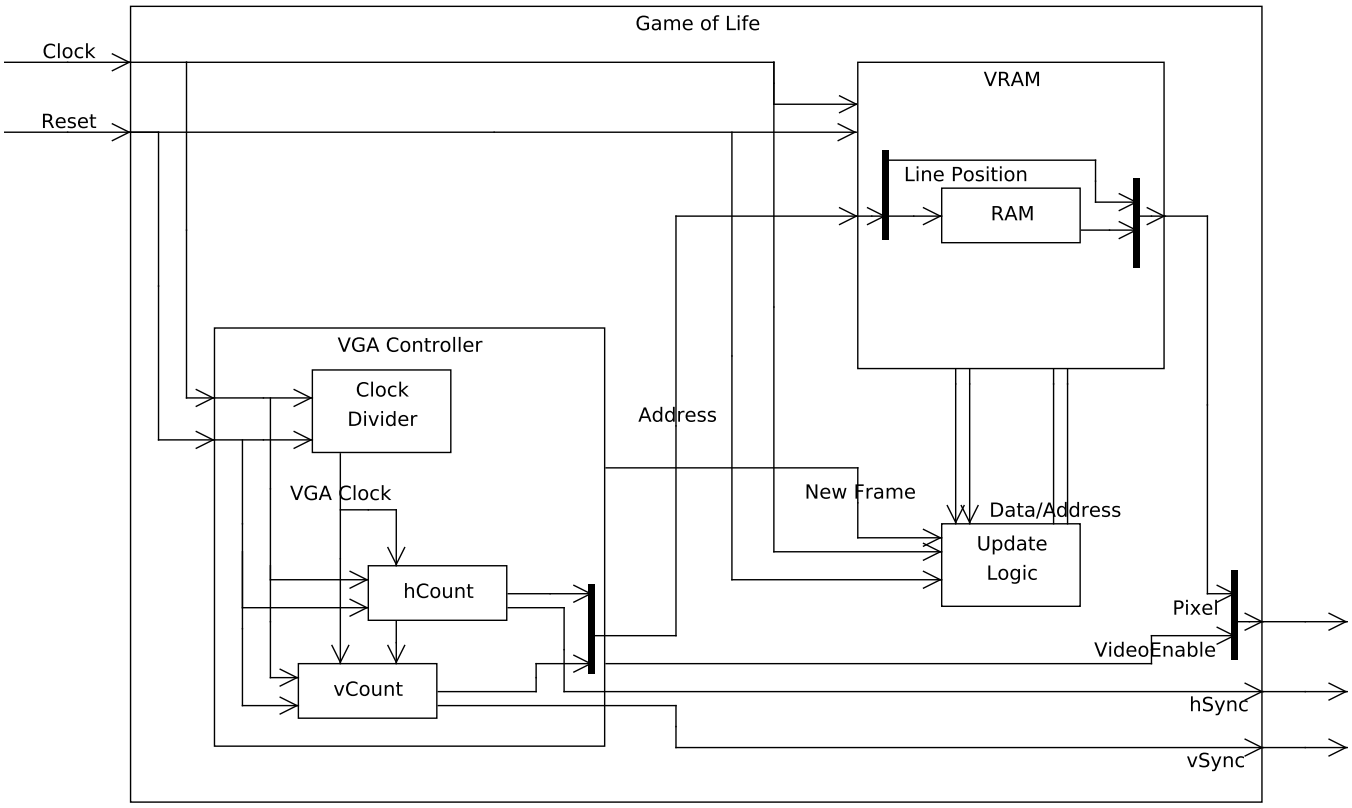

Figure 8.1: Block diagram of Game of Life project.

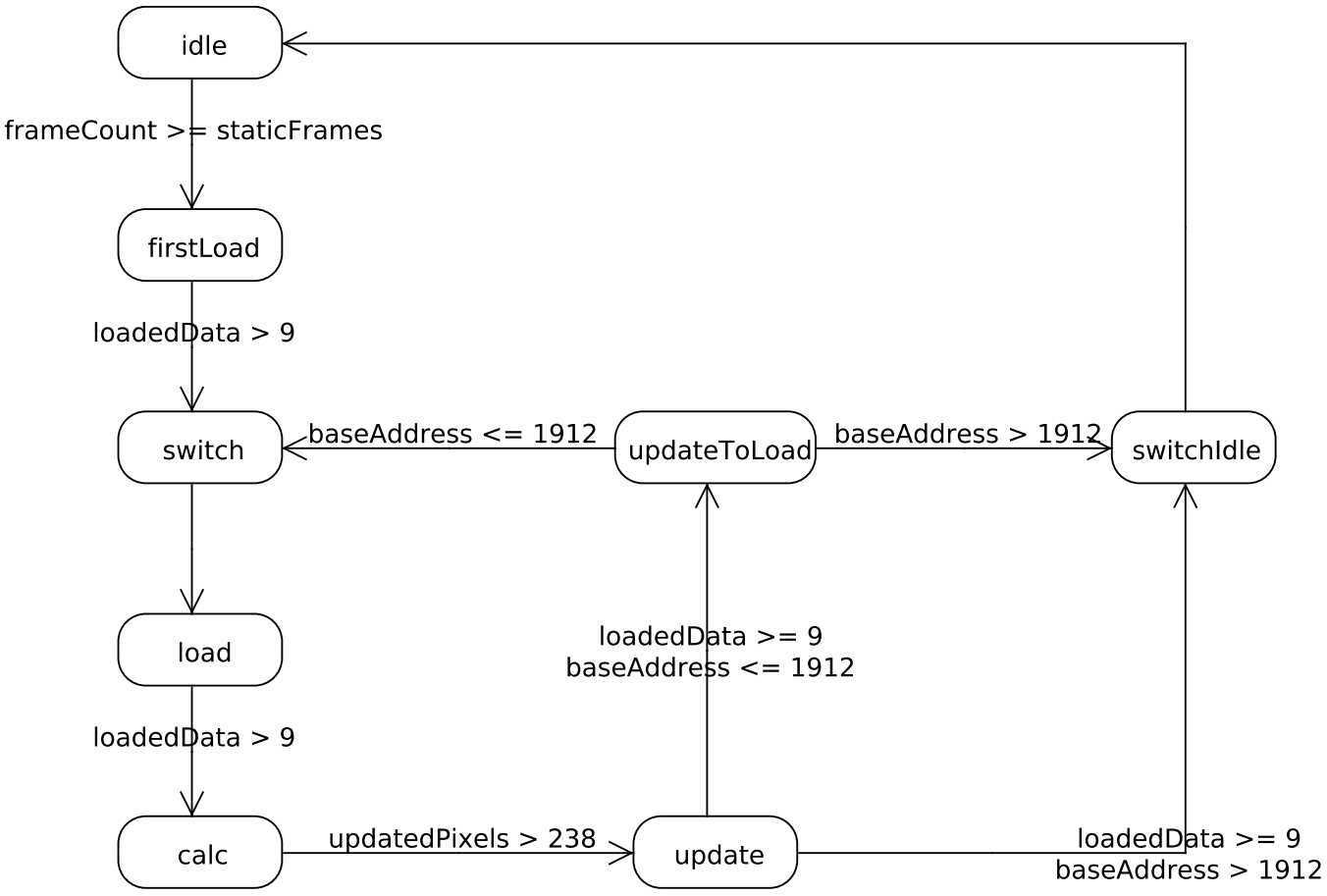

Figure 8.2: State machine used to update the play field.

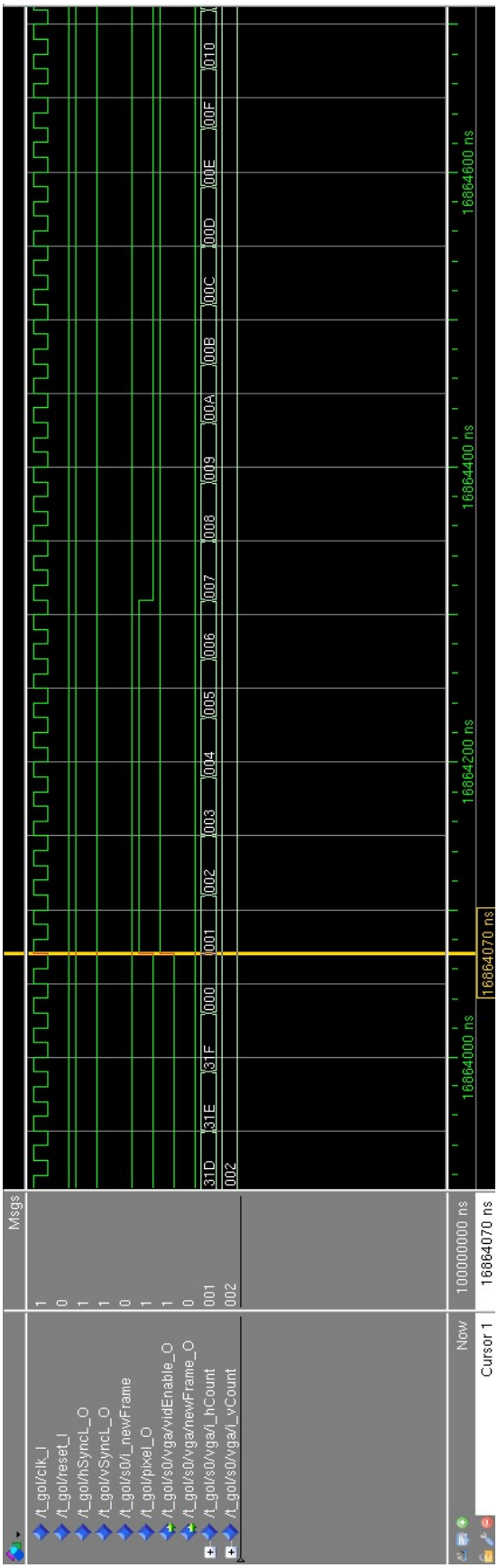

Figure 8.3: Memory is initialized as a 3x1 blinker in row 1, columns 0-2. This figure shows the initial memory block being displayed. HSync is offset by one for RAM access, VSync is at 2 because each pixel is shown twice.

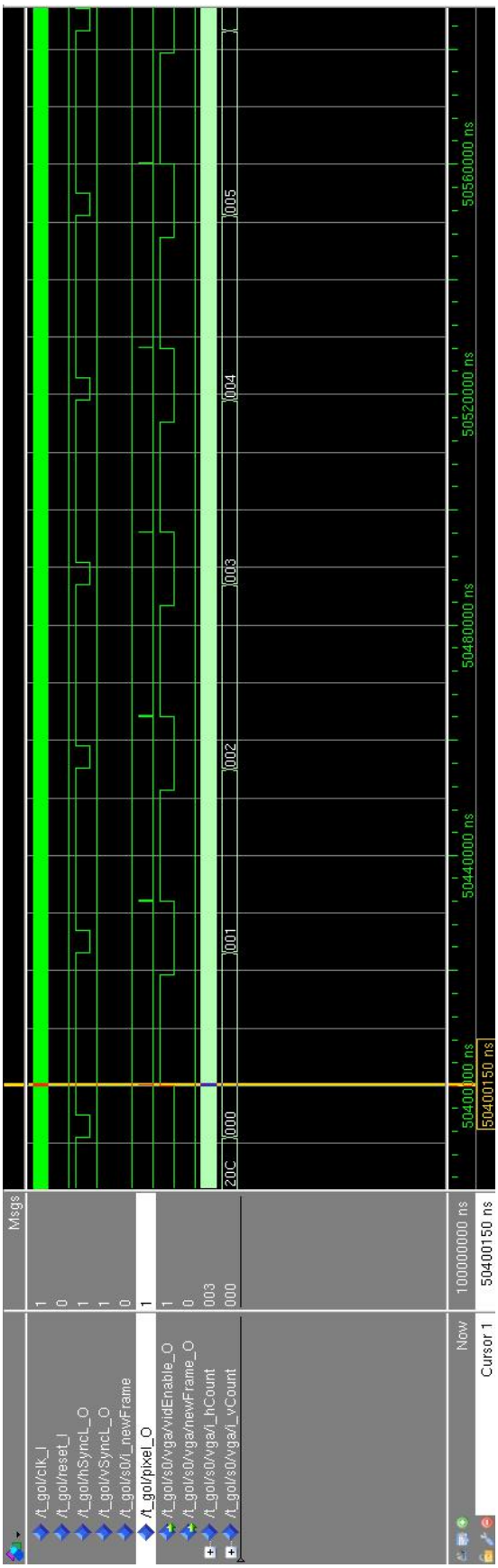

Figure 8.4: This is a zoomed out view of the next figure. It shows the flipped blinker 3 blocks tall (6 pixels) and 1 block wide.

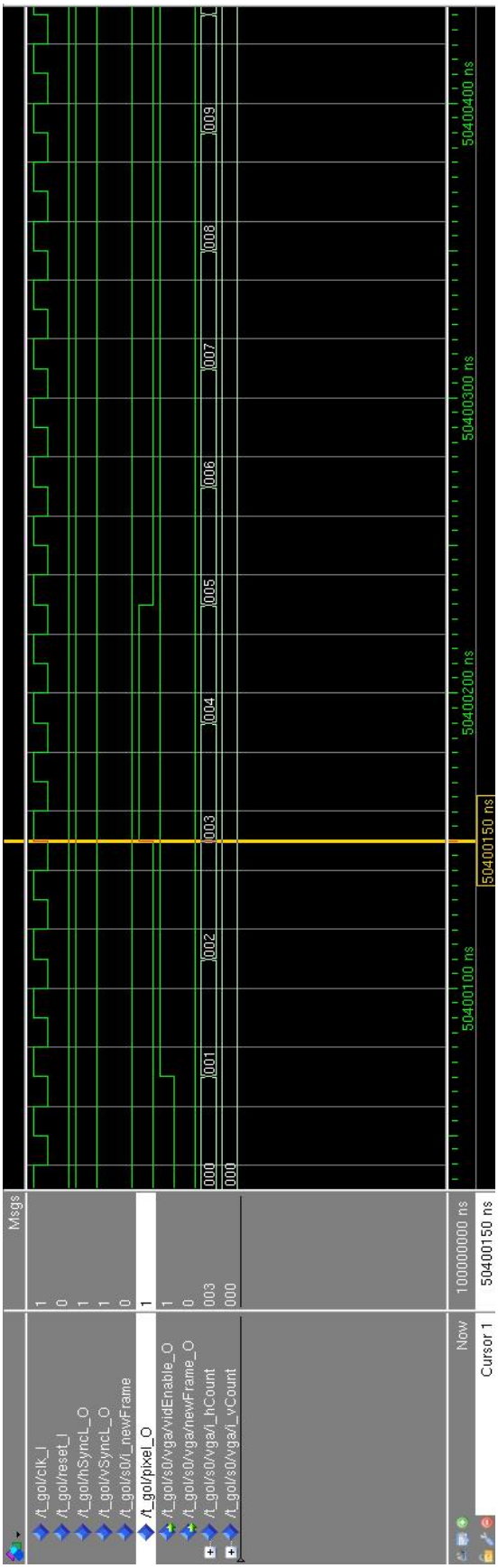

Figure 8.5: Zoomed in view of Figure 8.4 showing that the pixel is on for 2 hCounts.

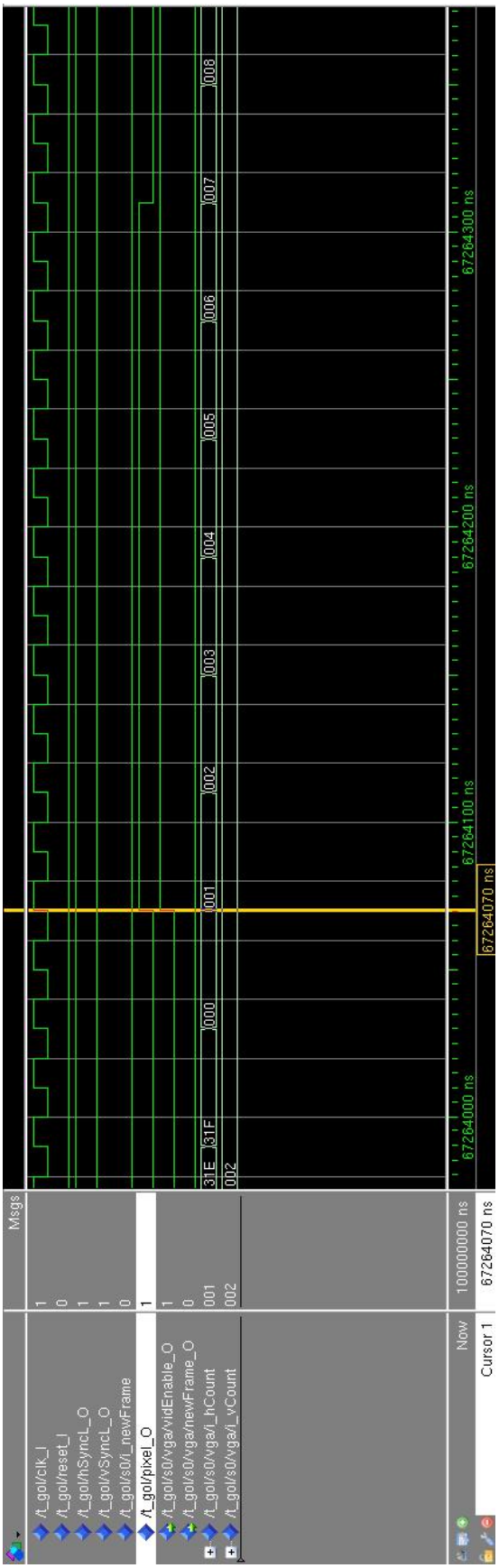

Figure 8.6: The blinker has completed its cycle and displays the same configuration as Figure 8.3.