CSEE W4840 Embedded System Design Lab 3

Stephen A. Edwards

Due March 28, 2013. Work together in project groups on this lab Submit one solution per group

Abstract

Use Quartus and SOPC builder to create one of two mixed hardware/software designs: an FM sound synthesizer or a bouncing video ball.

1 Introduction

This lab is about combining your own hardware and software components. You have a choice of implementing one of two "canned" designs that we started for you: an FM sound synthesizer that generates pleasing-sounding notes under keyboard control or a bouncing video ball in which software controls the trajectory of a circle on the screen displayed by custom video hardware.

First, follow the instructions in Section 2 to gain some practice building a simple system using SOPC Builder. Then, choose one of the two projects described in Sections 3 and 4.

2 Building a Nios II System with SOPC Builder

SOPC Builder is an Altera-supplied program for quickly assembling Nios II-based processor systems. It effectively writes VHDL for you.

The tutorial below explains how to make a simple "bouncing ball" LED display using SOPC Builder. Go though this tutorial first to see how the tools work, then start working on one of the three designs.

2.1 Quartus, part 1

Create a new directory (e.g., "lab3"), *cd* into it, and start *quartus*.

Select File→New Project Wizard.

In the new project wizard dialog, select the directory (e.g., "lab3") you just created. Name the project something like "lab3." The two names do not have to match, but only use letters, digits, and underscores in the project name. See Figure 1.

Don't add any files to the project yet.

For for the device, select the "Cyclone II" family and the "EP2C35F672C6" chip. See Figure 2.

Click "Finish" to create the project.

2.2 SOPC Builder

Inside Quartus, select Tools→SOPC Builder. This will probably ask you to start creating an SOPC builder system (if not, select File→New System). Name it differently than the project, e.g., "nios_system," and select VHDL as the language. See Figure 3.

You should now be at the SOPC Builder main window (Figure 4). Make sure the Device Family is set to Cyclone II and that there is a single external 50 MHz clock listed.

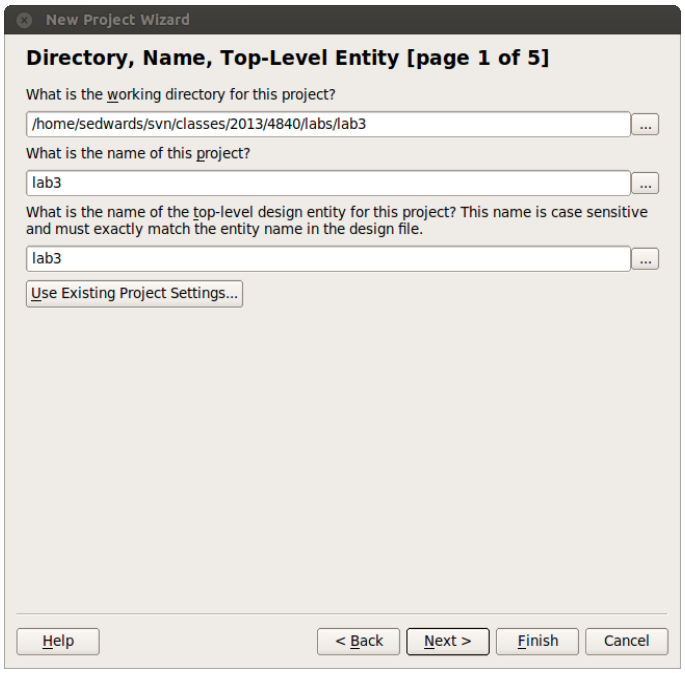

Figure 1: Naming a new Quartus project

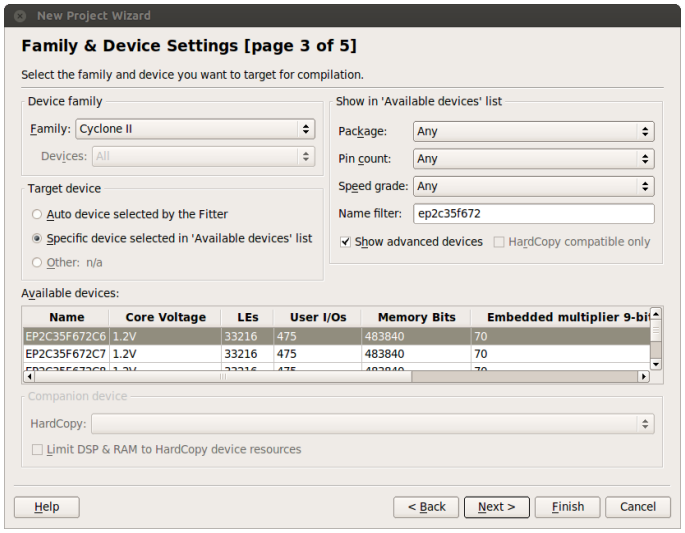

Figure 2: Selecting the device in Quartus

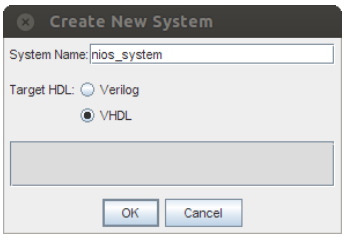

Figure 3: Naming a new system in SOPC Builder

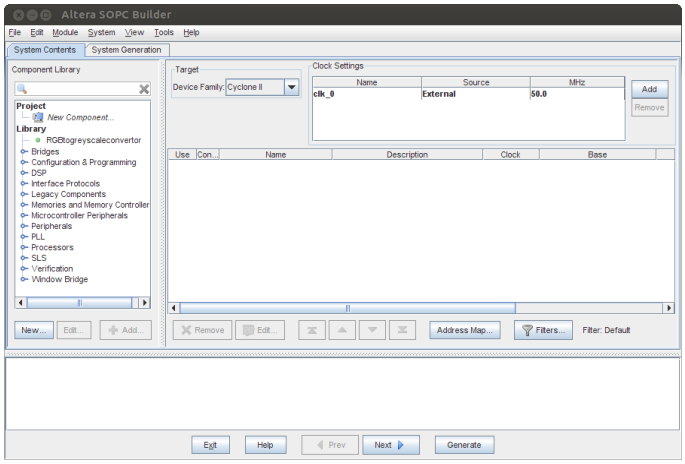

Figure 4: The SOPC Builder main window. Available components are listed on the left.

Add the processor by opening Processors and double-clicking "Nios II Processor." This should bring up the Nios II dialog in Figure 5. Select the Nios II/e, the smallest of the three and click "Finish." You don't need to adjust the other parameters.

At this point (Figure 6), you have a single processor with a JTAG debug module connected to it. By itself, this is useless because it has no memory.

We will use the off-chip 512K SRAM by creating a new component (peripheral) that does the nearly-trivial translation from the protocol spoken by the Avalon bus (i.e., that is connected to the Nios II) to that for the SRAM.

First, you need a VHDL file for the component called de2_sram_controller.vhd. Its contents are shown in Figure 7.

This does almost nothing: it connects and inverts the various Avalon signals (named avs_s1_...) for the SRAM chip and controls the tri-state output drivers by indicating the SRAM_DQ bus should only be driven when the Avalon *write* signal is asserted.

Create a new SOPC Builder component by selecting File→New Component... Under HDL Files, select this .vhd file. A dialog will come up showing the file is being parsed and give you a bunch of warnings about signals having type "export," which is fine. Make sure the Top Level Module is set to "de2_sram_controller."

Go to the "Signals" tab and change the interface for each SRAM signal (e.g., SRAM_DQ, SRAM_ADDR) from "avalon_slave_0" to "conduit_end." Select "New Conduit..." when you change the first signal. This will create "conduit_end," which you can select for the remaining signals. For each signal changed, set the signal type to "export." This tells

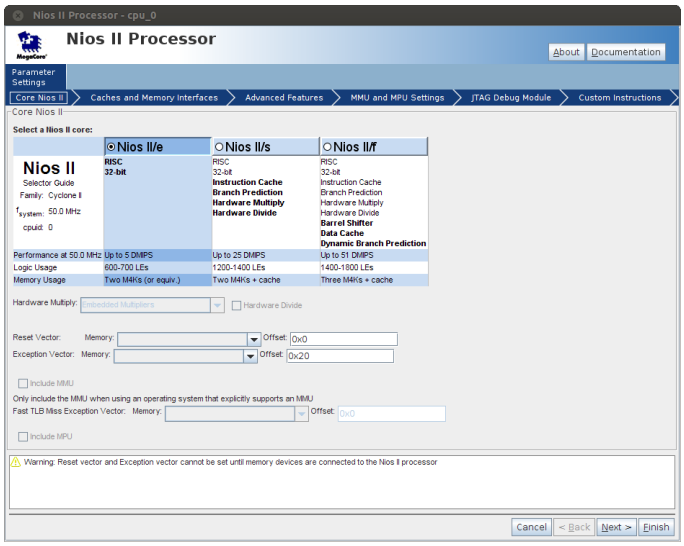

Figure 5: Adding an Nios II processor in SOPC Builder

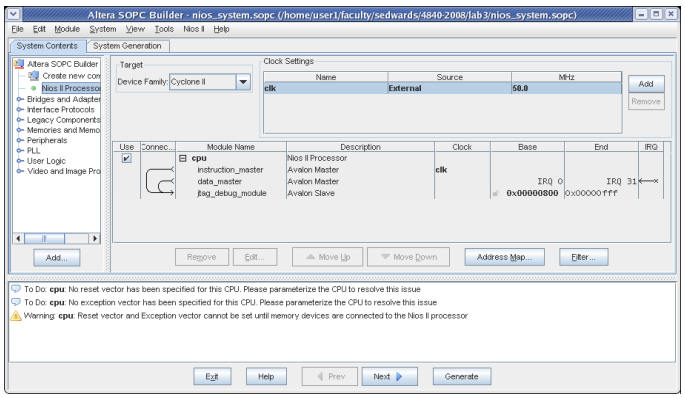

Figure 6: The system with only the Nios II processor

SOPC to defer this signal's behavior to the next level of your design hierarchy. That is, to create an endpoint for this signal so that it can be used in your custom VHDL. The list should look like Figure 8.

Next, go to the "Interfaces" tab and click on "Remove Interfaces With No Signals."

Also in the Interfaces tab, under the avalon_slave_0 interface, click the "Assignments: Edit..." button. Change the value of isMemoryDevice from 0 to 1. Figure 9 shows this. Click on "Finish" and save your new component. This creates the file de2_sram_controller_hw.tcl, which contains information about the component and its assignments.

Return to the main SOPC builder window, select the new "de2_sram_controller" component in the left pane, and click on "Add..." and then "Finish." Right-click on the module name (it defaults to "de2_sram_controller_inst") and rename it to "sram." Make sure both the instruction_master and data_master connections from cpu_0 lead to the avalon_slave_0 connection on the sram component; this allows the CPU to store both programs and data in the SRAM. Clicking in the "Connec..." panel will let you adjust these connections if necessary.

Congratulations: your processor system now has some memory and could actually run programs.

```
library ieee;
use ieee.std_logic_1164.all;

entity de2_sram_controller is

  port (
    signal chipselect : in std_logic;
    signal write, read : in std_logic;
    signal address  :  in std_logic_vector(17 downto 0);
    signal readdata : out std_logic_vector(15 downto 0);
    signal writedata : in std_logic_vector(15 downto 0);
    signal byteenable : in std_logic_vector(1 downto 0);

    signal SRAM_DQ : inout std_logic_vector(15 downto 0);
    signal SRAM_ADDR : out std_logic_vector(17 downto 0);
    signal SRAM_UB_N, SRAM_LB_N : out std_logic;
    signal SRAM_WE_N, SRAM_CE_N : out std_logic;
    signal SRAM_OE_N            : out std_logic
    );

end de2_sram_controller;

architecture dp of de2_sram_controller is
begin

  SRAM_DQ <= writedata when write = '1'
             else (others => 'Z');
  readdata <= SRAM_DQ;
  SRAM_ADDR <= address;
  SRAM_UB_N <= not byteenable(1);
  SRAM_LB_N <= not byteenable(0);
  SRAM_WE_N <= not write;
  SRAM_CE_N <= not chipselect;
  SRAM_OE_N <= not read;

end dp;
```

Figure 7: de2_sram_controller.vhd: VHDL source for the SRAM controller (inverters and a tristate buffer).

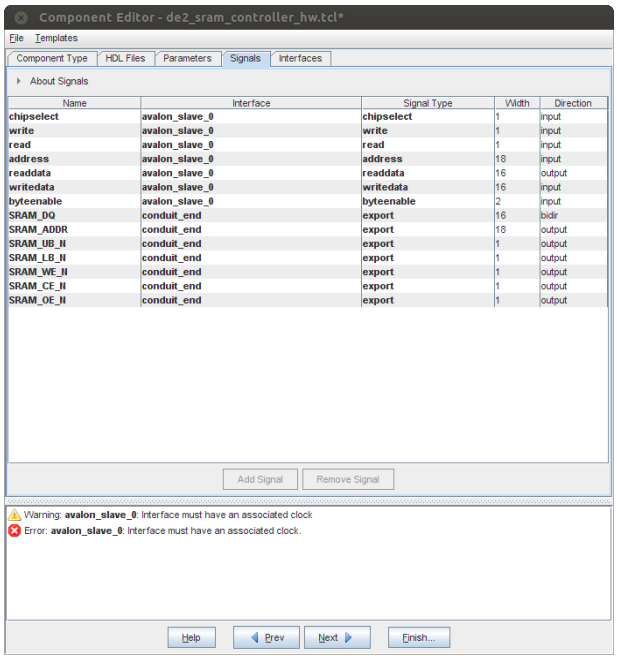

Figure 8: Associating the signals with interfaces

If you later change the VHDL code for your component (e.g., during the development process), you must re-edit the component by right-clicking the component on the left menu and selecting "Edit."

Double-click on the cpu component and choose the "sram" memory for both the reset vector and the exception vector. This should turn off some warnings. If you can't select sram as the memory, you probably forgot to change the "isMemoryDevice" setting in the avalon_slave_0 interface for the SRAM controller.

Using the same procedure, create a new component called "de2_led_flasher." The VHDL for this is shown in Figure 11. Again, remember to change the interface of the "leds" signal to "conduit_end" and its signal type to "export." Connect the "clk" and "reset_n" signals the "clock" interface and set their types to "clk" and "reset_n" respectively. The signals tab should look like Figure 10.

Add an instance of your new "led_flasher" component to the system and rename it to "leds."

For debugging output, add a Interface Protocols/Serial/JTAG UART component from the library. Just click "Finish" to accept the default parameters.

Run System→Assign Base Addresses to locate each component in memory. The completed system configuration is shown in Figure 12.

Finally, click on the "System Generation" tab, make sure

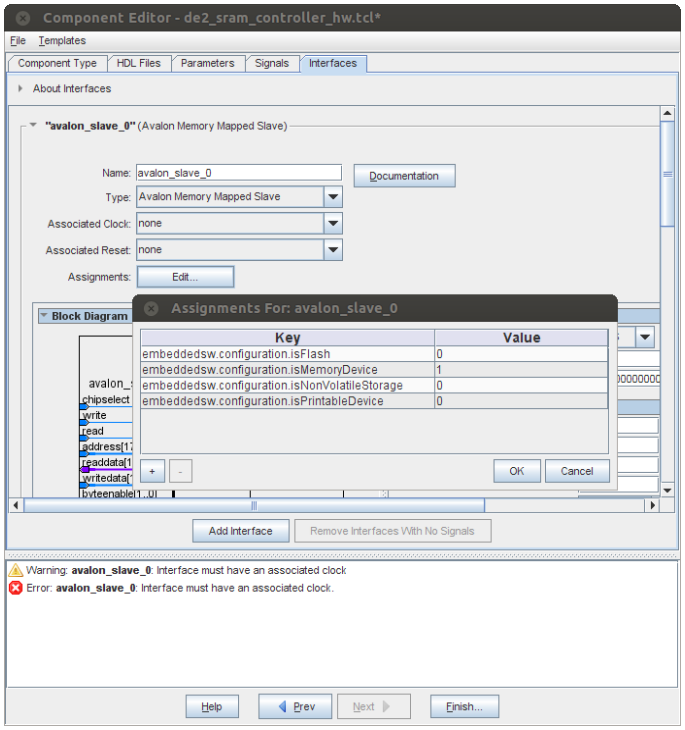

Figure 9: Editing the assignments

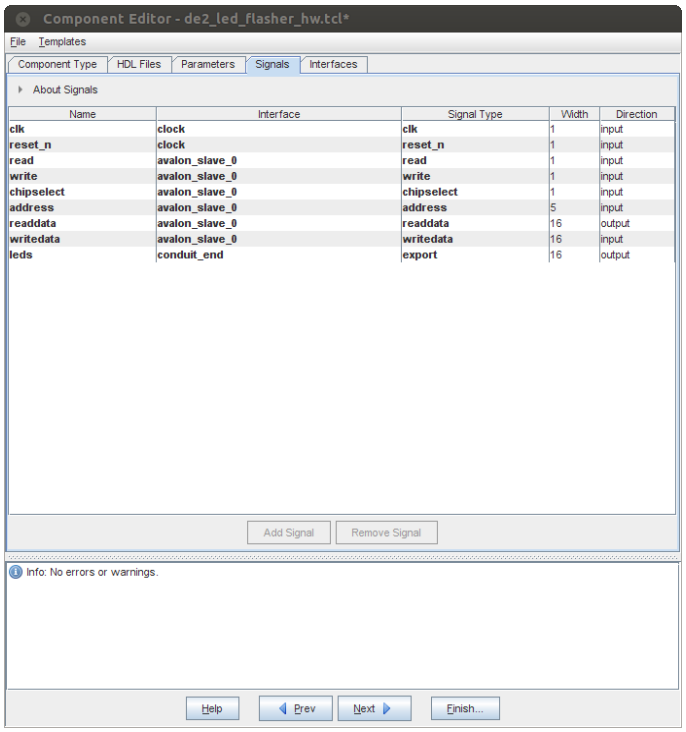

Figure 10: Signals for the LED flasher

"Simulation. Create simulator project files" is disabled (simulation with the DE2 does not work well without models for the various off-chip peripherals) and click "Generate." You may save your SOPC system as "nios_system.sopc" (an XML file). Running "Generate" should fill your project directory with many .vhd files.

When system generation completes (this takes a while), click on Exit and return to the Quartus II GUI.

2.3 Quartus, part 2

Once SOPC Builder has generated the system, we need to import it into a Quartus II project.

First, you need to create a top-level VHDL file that instantiates the Nios II system that was just generated and whatever hardware you want to connect to it. In this case, we only need to wire the Nios II to the external clock and connect the SRAM and LEDs to their pins.

The nios_system entity was generated by the SOPC Builder and is defined in nios_system.vhd (along with a lot of other things). As usual, its component definition is essentially just the ports on the entity, which were named by SOPC Builder.

Figure 13 shows the top-level VHDL file. Put this in the project directory and add it to the Quartus project. Also add the "nios_system.vhd" file. Finally, make sure the nios_system.qip file is also part of project; it contains other files generated by SOPC builder. Make sure you put "lab3.vhd" *below* the others (it won't find the nios_system entity otherwise).

By default, the name of the top-level entity is the name of the project2. Open lab3.vhd in Quartus and use Project→Set as Top-Level Entity to change this.

Match the pin names to locations by selecting Assignments→Import Assignments and choosing the DE2.qsf

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity de2_led_flasher is

  port (
    clk        : in  std_logic;
    reset_n    : in  std_logic;
    read       : in  std_logic;
    write      : in  std_logic;
    chipselect : in  std_logic;
    address    : in  std_logic_vector(4 downto 0);
    readdata   : out std_logic_vector(15 downto 0);
    writedata  : in  std_logic_vector(15 downto 0);

    leds       : out std_logic_vector(15 downto 0)
    );

end de2_led_flasher;

architecture rtl of de2_led_flasher is

  type ram_type is array(15 downto 0) of
            std_logic_vector(15 downto 0);
  signal RAM : ram_type;
  signal ram_address, display_address : unsigned(3 downto 0);
  signal counter_delay : unsigned(15 downto 0);
  signal counter : unsigned(31 downto 0);

begin
  ram_address <= unsigned(address(3 downto 0));

  process (clk)
  begin
    if rising_edge(clk) then
      if reset_n = '0' then
        readdata <= (others => '0');
        display_address <= (others => '0');
        counter <= (others => '0');
        counter_delay <= (others => '1');
      else
        if chipselect = '1' then
          if address(4) = '0' then
            if read = '1' then
              readdata <= RAM(to_integer(ram_address));
            elsif write = '1' then
              RAM(to_integer(ram_address)) <= writedata;
            end if;
          else
            if write = '1' then
              counter_delay <= unsigned(writedata);
              counter <= unsigned(writedata) & x"0000";
            end if;
          end if;
        else
          leds <= RAM(to_integer(display_address));
          if counter = x"00000000" then
            counter <= counter_delay & x"0000";
            display_address <= display_address + 1;
          else
            counter <= counter - 1;
          end if;
        end if;
      end if;
    end if;
  end process;

end rtl;
```

Figure 11: de2_led_flasher.vhd: VHDL source for the LED flash controller. This memory-maps a 16×16 RAM into 16 half-words and a single "delay" register into another 16. When the RAM is not being written, a counter steps through the contents of the RAM, displaying it on the LEDs. The delay register sets the hold time for each address.

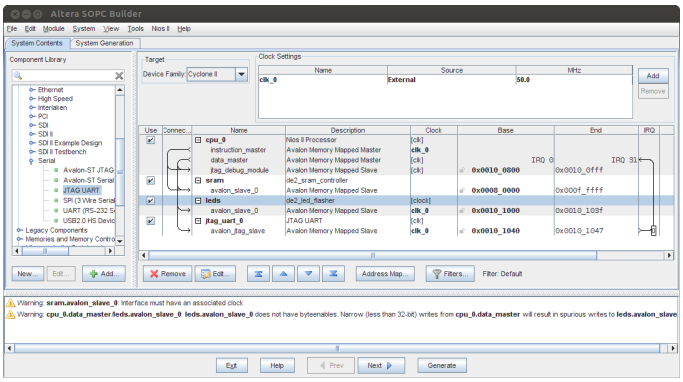

Figure 12: The final configuration of the LED flasher system

file, which is included in lab3.tar.gz.

Impose a global timing constraint by choosing Assignments→Time Quest Timing Analyzer Wizard. Create a clock named "CLOCK_50" on input pin "CLOCK_50" and set its period to 20 ns (50 MHz). See Figure 14. You don't have to set anything else.

Compile the project and download it to the board. Congratulations! You just built a computer.

2.4 Nios II IDE

Next, create a new software project for your new computer system. Since each system is different (e.g., different memory layout, different peripherals), the software is tied to the system.

Run nios2-ide and switch the workspace to your project directory.

Select File→New→Nios Application and BSP from template.

For the SOPC Information File name, select the nios_system.sopcinfo file that SOPC builder generated as part of your system. This describes the processor, memory map, etc. of your system. When you select this file, it should set the CPU name to "cpu_0." Incidentally, "BSP" stands for Board Support Package.

Name the new (software) project something like lab3_software (this is arbitrary—it creates a directory with this name in your project directory).

Finally, select the "Hello World" template and click Finish. Figure 15 illustrates this.

To compile, download, and run the software, program the FPGA using the Quartus II downloader then instruct the Nios II IDE to download and run the program. Once you have programmed the FPGA, you can download and run new software as many times as you like.

Program the FPGA from the Nios II IDE by selecting Nios II→Quartus II Programmer. Click the Hardware Setup button, select USB-Blaster, and close the window. Click "Auto Detect" to make sure the EP2C35 FPGA is detected. Next, select output_files/lab3.sof for the file to download. Select Program/Configure, then click "Start." The progress meter should quickly indicate "Successful" and the LEDs on the board should change. You may quit the Quartus II programmer at this point.

Note that if the Nios II IDE is running and communicating with the board, FPGA programming will fail.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lab3 is
  port (
    signal CLOCK_50 : in std_logic; -- 50 MHz
    signal LEDR : out std_logic_vector(17 downto 0); -- LEDs

    SRAM_DQ : inout std_logic_vector(15 downto 0);
    SRAM_ADDR : out std_logic_vector(17 downto 0);
    SRAM_UB_N,              -- High-byte Data Mask
    SRAM_LB_N,              -- Low-byte Data Mask
    SRAM_WE_N,              -- Write Enable
    SRAM_CE_N,              -- Chip Enable
    SRAM_OE_N : out std_logic  -- Output Enable
    );
end lab3;

architecture rtl of lab3 is
  signal counter : unsigned(15 downto 0);
  signal reset_n : std_logic;
begin

  LEDR(17) <= '1';
  LEDR(16) <= '1';

  process (CLOCK_50)
  begin
    if rising_edge(CLOCK_50) then
      if counter = x"ffff" then
        reset_n <= '1';
      else
        reset_n <= '0';
        counter <= counter + 1;
      end if;
    end if;
  end process;

  nios : entity work.nios_system port map (
    clk_0                       => CLOCK_50,
    reset_n                     => reset_n,
    leds_from_the_leds       => LEDR(15 downto 0),
    SRAM_ADDR_from_the_sram     => SRAM_ADDR,
    SRAM_CE_N_from_the_sram     => SRAM_CE_N,
    SRAM_DQ_to_and_from_the_sram => SRAM_DQ,
    SRAM_LB_N_from_the_sram     => SRAM_LB_N,
    SRAM_OE_N_from_the_sram     => SRAM_OE_N,
    SRAM_UB_N_from_the_sram     => SRAM_UB_N,
    SRAM_WE_N_from_the_sram     => SRAM_WE_N
    );

end rtl;
```

Figure 13: lab3.vhd: The top-level entity

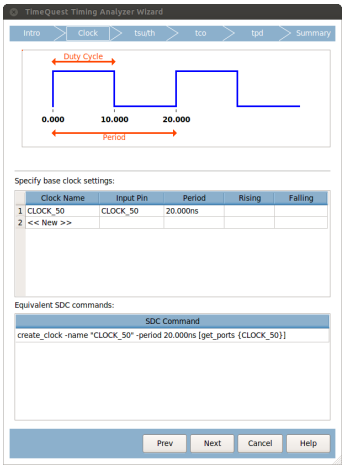

Figure 14: Imposing a clock timing constraint

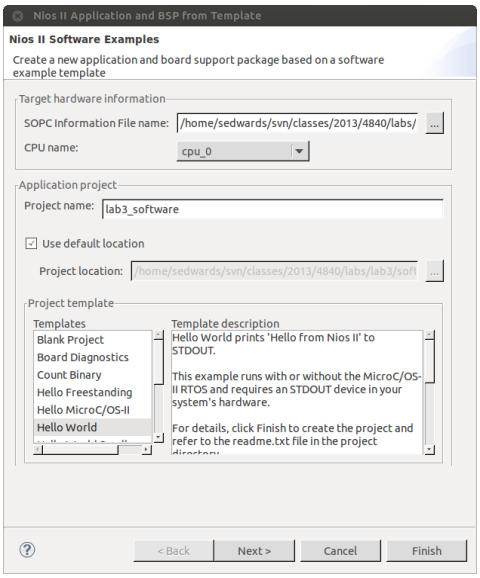

Figure 15: Creating a new application and BSP from a template

```c
#include <io.h>
#include <system.h>
#include <stdio.h>

#define IOWR_LED_DATA(base, offset, data) \
  IOWR_16DIRECT(base, (offset) * 2, data)
#define IORD_LED_DATA(base, offset) \
  IORD_16DIRECT(base, (offset) * 2)
#define IOWR_LED_SPEED(base, data) \
  IOWR_16DIRECT(base + 32, 0, data)

int main()
{
  int i;
  printf("Welcome to Lab 3\n");

  IOWR_LED_SPEED(LEDS_BASE, 0x0040);

  for (i = 0 ; i < 8 ; i++) {
    IOWR_LED_DATA(LEDS_BASE, i, 3 << (i * 2));
    printf("writing %x\n", i);
  }

  for (i = 8 ; i < 16 ; i++) {
    IOWR_LED_DATA(LEDS_BASE, i, 3 << (32 - (i * 2)));
    printf("writing %x\n", i);
  }

  for (i = 0 ; i < 16 ; i++) {
    printf("reading %x = %x\n", i,
           IORD_LED_DATA(LEDS_BASE, i));
  }

  printf("Goodbye\n");

  return 0;
}
```

Figure 16: A hello_world.c file that imitates KITT from Knight Rider (yes, I lived through the 80s). It sets the cycling speed, fills the LED_flasher peripheral with a pattern, then reads it back to verify it works as memory.

Run the program from the Nios II IDE by selecting the lab3_software project, then Run→Run As→Nios II Hardware. The first time, this should compile a number of files, download the program to the Nios II processor system on the FPGA, and finally run the program. The stock "Hello World" code prints "Hello from Nios II!" on the Nios II console.

Now, replace "hello_world.c" in the software/lab3 directory (i.e., the name of the software project you specified) with the code in Figure 16, which exercises the LED flasher peripheral we added earlier.

If you change the hardware and regenerate the SOPC project, you need to update the BSP. Right-click the lab3_bsp project in the Nios II IDE and select Nios II→Generate BSP. Then recompile in the IDE and run again. Run→Run Configurations... can help you get around mismatched system ID and timestamp issues.

3 An FM Sound Synthesizer

This project is a stripped-down version of Ron Weiss, Gabriel Glaser, and Scott Arfin's *Terrormouse* project from 4840 in spring 2004. Feel free to use it as reference and adapt what VHDL you can, but make sure you understand what you are

using.

In 1973, John Chowing introduced the idea of FM synthesis and the world has not sounded the same since. His basic insight is that FM waveforms are easy to produce and are "natural sounding." The basic FM equation is

$$x(t) = \sin\left(\omega_c t + I\sin(\omega_m t)\right)$$

where $x(t)$ is the amplitude at time t, ω_c is the carrier frequency (the fundamental tone we hear), ω_m is the modulating frequency, and I is the modulation depth. The timbre of the sound is largely determined by the ratio ω_c/ω_m, which is generally set to an integer ratio (e.g., $\omega_c = 3\omega_m$).

The fundamental frequency of musical notes follow an exponential scale. The A above middle C is 440 Hz, and going up an octave doubles the frequency.

Western music is built on a scale of twelve semitones, each in equal ratio. Thus, the frequencies of a standard scale are of the form

$$f = 440 \cdot 2^{p/12}$$

where f is the frequency in Hertz, $p = 0$ is the A above middle C, $p = 1$ is A♯, $p = 2$ is B, $p = 3$ is C, $p = 12$ is the A the octave above, $p = -12$ is the A the octave below, etc.

3.1 Starting Points

In the lab3.tar.gz file, we have supplied some helpful files you should use as a starting point. The most interesting is de2_wm8731_audio.vhd, which implements an interface to the Wolfson WM8371 audio codec on the DE2 board. This operates either in a test mode that generates a sinewave (a pure tone), or as a parallel-to-serial converter.

We included two Verilog files that configure the WM8371: de2_i2c_controller.v and de2_i2c_av_config.v. You should be able to just instantiate them without modification. They send initialization commands through the two-wire I^2C bus.

lab3_audio.vhd is a simple top-level module that instantiates the audio controller in test mode and the two I^2C bus components. You can build a new Quartus project with this as a starting point and should hear a tone on line out.

Finally, we have included a PS/2 keyboard controller.

3.2 The PS/2 Controller

The file de2_ps2.vhd is the core of an Avalon peripheral that can read data coming from a PS/2 keyboard. This is simpler than the one you used in lab 2 (e.g., it cannot send data to the keyboard), but will suffice. Use SOPC Builder to create a new component around it and connect the two PS/2 lines (clock and data) to the appropriate pins.

This peripheral presents a simple two-word interface: reading the first byte of the first word returns 1 if a byte is available and zero otherwise. Reading the first byte of the second word returns the byte received from the keyboard.

Thus, if DE2_PS2_BASE is the base address of the PS/2 controller peripheral, you can wait for the next data byte using

```
unsigned char code;

while (!IORD_8DIRECT(DE2_PS2_BASE, 0)) ; /* Poll the status */
code = IORD_8DIRECT(DE2_PS2_BASE, 4);
/* Get received byte */
```

3.3 What To Do

You have two things to design: an Avalon peripheral that can generate an FM waveform under software control that you feed to the supplied WM8371 audio controller, and a C program that translates key events from the PS/2 keyboard into commands for your FM oscillator. Basically, make the PS/2 keyboard behave like a dumb piano keyboard.

Using the LED flasher example peripheral, build an Avalon peripheral that presents registers that control the oscillation frequency, the modulation depth, and a simple volume control (on/off) that lets you turn off the oscillator when no key is pressed.

Use a sinewave lookup table to generate the waveform. Step through it at different rates to generate the different tones.

First, develop the oscillator functionality first using ModelSim to test that your waveform is as you expect. Then, integrate it with the supplied audio codec controller and make a VHDL-only design that actually generates sound. Finally, add an Avalon interface to your oscillator, use SOPC Builder to integrate a Nios II, the supplied PS/2 keyboard controller, and your new component, and develop the software.

4 A Bouncing Video Ball

After you implement this project, you will feel a much stronger connection with Nolan Bushnell, the inventor of the first commercially-successful videogame, Pong. Of course, you won't find it quite as lucrative.

You have two things to design: an Avalon component that displays a small white circle on the screen under software control, and a C program that controls the position of this circle.

Use the code in de2_vga_raster.vhd as a starting point for your Avalon component. It is a simple VGA controller that displays a large white rectangle against a blue background. It currently does not have a bus interface. You need to add one and change its behavior so that it displays a small circle. The lab3_vga.vhd file holds a simple top-level for this component that can be used to build a skeleton project.

First, adapt the video generator to display a circle instead of a rectangle. Make sure you add signals that control where on the screen the circle appears. While developing this, you can just set these to constants; later software will supply them.

Your other challenge is building an Avalon peripheral. Use the LED flasher from the tutorial as a basis for building a peripheral. First, get an Avalon peripheral working by building the registers you plan to use in the end for your video controller and connect them to some LEDs to verify you can communicate from the software to the hardware.

Once you have a working peripheral, integrate your modified video controller with it.

Finally, write a simple C program that bounces the ball around the screen.