CSEE W4840 Embedded System Design Lab 3

Stephen A. Edwards

Due February 15, 2007 (Incomplete Draft!)

Abstract

Use Quartus and SOPC builder to create one of three mixed hardware/software designs: an FM sound synthesizer, a bouncing video ball, or an image convolver that uses a 3×3 kernel.

1 Introduction

This lab is about combining your own hardware and software components. You have a choice of implementing one of three "canned" designs that we started for you: an FM sound synthesizer that generates pleasing-sounding notes under keyboard control, a bouncing video ball in which software controls the trajectory of a square on the screen displayed by custom video hardware, or a image convolver that uses FPGA hardware to accelerate a 3×3 convolution kernel such as a blur or edge finder.

First, follow the instructions in Section 2 to gain some practice building a simple system using SOPC Builder. Then, choose one of the three projects described in Sections 3, 4, and 5.

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
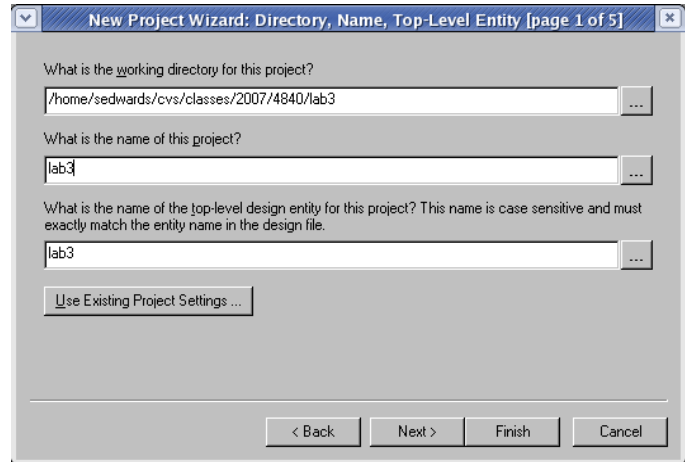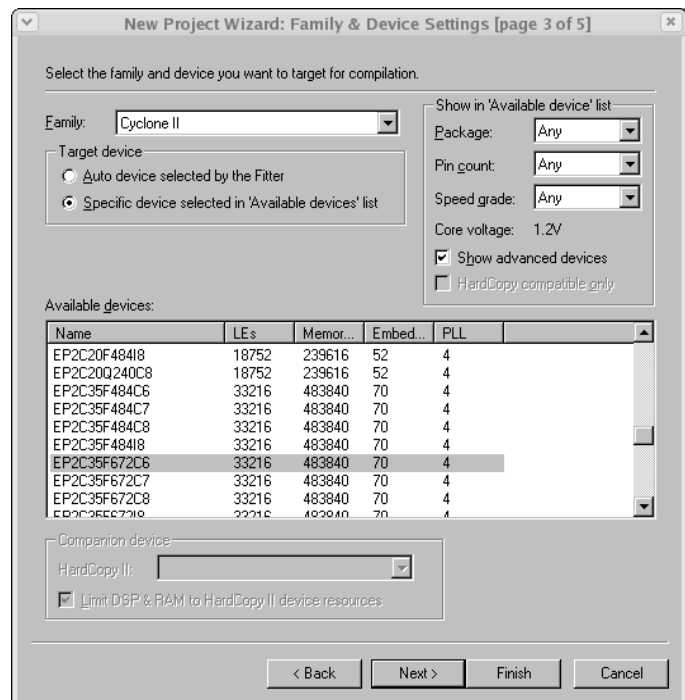

Figure 1: Naming a new Quartus project

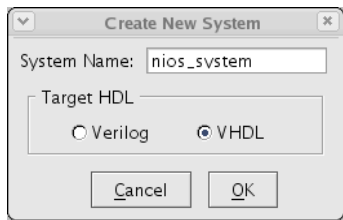Figure 2: Selecting the device in Quartus
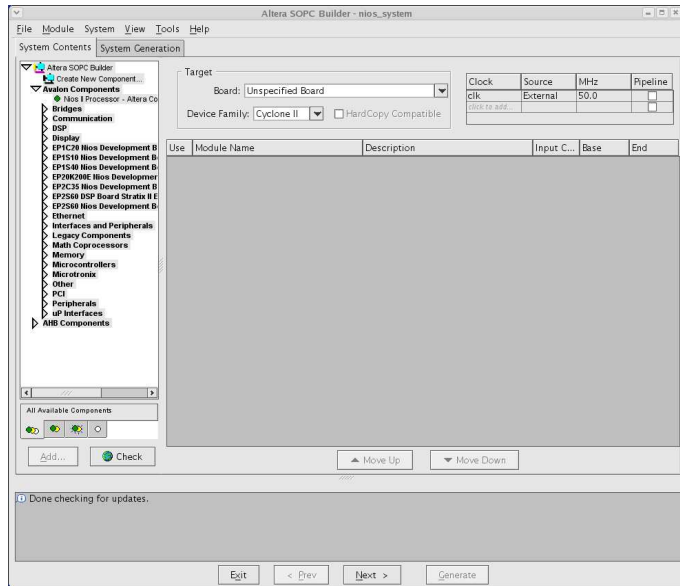
Figure 3: Naming a new system in SOPC Builder

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

Figure 5: Adding an Nios II processor in SOPC Builder
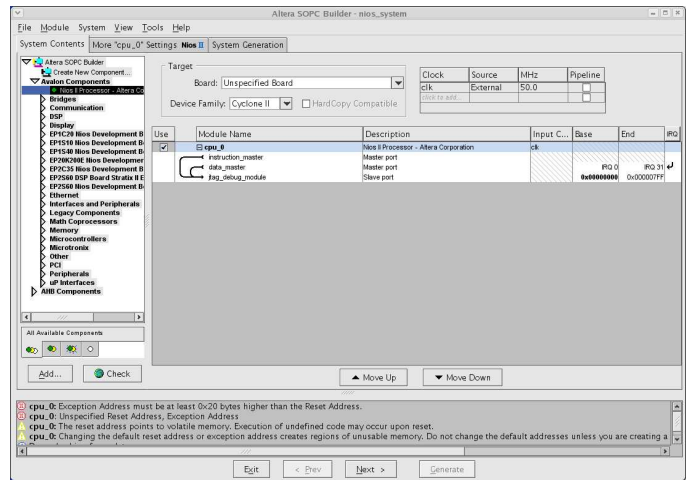
Figure 6: The system with only the Nios II processor

that there is a single external 50 MHz clock listed. "Unspecified Board" is correct.

Add the processor by opening Avalon Components and double-clicking "Nios II Processor—Altera Corporation." 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. Once the filename stops flashing green (it is being checked), the dialog should indicate "de2_sram_controller" is the top-level module (Figure 8).
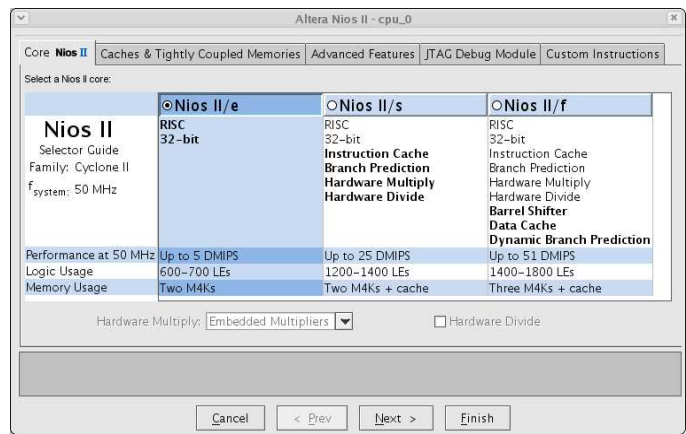
Under the "Signals" tab, make sure all the "SRAM" signals are connected to the "global_signals" interface (otherwise, SOPC builder assumes you have two bus connections on the component). See Figure 9.

Under the "Interfaces" tab, click "Remove Interfaces with No Signals." This should leave just the "s1" interface. Turn off "Can receive stderr/stdout" for the "s1" interface (since it is not a communication port—it is a memory). Note that at this point you can set the speed and other parameters of the bus interface for the component. In this case the defaults—one cycle each for read and write—are what we want.

"Slave addressing" is an important choice. The "Memory" setting indicates that the bus will be dynamically resized to accommodate the data width of the peripheral—exactly what we want for the SRAM component. The "Register" setting disables this: the bus always appears as 32 bits wide and the peripheral is expected to align its data on 32 bit boundaries.

Finally, under the "Component Wizard" tab, set the Component Group to "User Logic" and click "Finish."

You should now have de2_sram_controller under Avalon Components/User Logic on the list on the left.

Double-click it to add it to the system and right-click on its module name to rename it "sram." Congratulations: your processor system now has some memory and could actually run

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity de2_sram_controller is

port (
  signal avs_s1_clk,
         avs_s1_chipselect,
         avs_s1_write,
         avs_s1_read : in std_logic;
  signal avs_s1_address : in std_logic_vector(17 downto 0);
  signal avs_s1_readdata : out std_logic_vector(15 downto 0);
  signal avs_s1_writedata : in std_logic_vector(15 downto 0);
  signal avs_s1_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,
         SRAM_WE_N,
         SRAM_CE_N,
         SRAM_OE_N : out std_logic
);

end de2_sram_controller;

architecture datapath of de2_sram_controller is

begin

  SRAM_DQ <= avs_s1_writedata when avs_s1_write = '1' else
             (others => 'Z');
  avs_s1_readdata <= SRAM_DQ;
  SRAM_ADDR <= avs_s1_address;
  SRAM_UB_N <= not avs_s1_byteenable(1);
  SRAM_LB_N <= not avs_s1_byteenable(0);
  SRAM_WE_N <= not avs_s1_write;
  SRAM_CE_N <= not avs_s1_chipselect;
  SRAM_OE_N <= not avs_s1_read;

end datapath;
```

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

programs.

Note that 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." After this, the system will instruct you to remove and re-add every instance of the component in your system.

Using the same procedure, create a new component called "led_flasher." The VHDL for this is shown in Figure 11. Again, remember to change the interface of the "leds" signal to global_signals and remove interfaces with no signals.

Add an instance of your new "led_flasher" component to the system and rename it to "leds" (instead of led_flasher_0).

For debugging output, add Avalon Components/Communication/JTAG UART. Just click "Finish" to accept the default parameters.

Run System→Auto-Assign Base Addresses to locate each component in memory. The completed system configuration is shown in Figure 13.

Click on the "More "cpu_0" Settings" tab. Make sure the Reset Address and Exception Address functions are mapped to the "sram" memory module, not the "leds" module, which is not memory.

Finally, click on the "System Generation" tab, disable "Simulation. Create simulator project files," (simulation with the DE2 does not work well without models for the various off-chip pe-
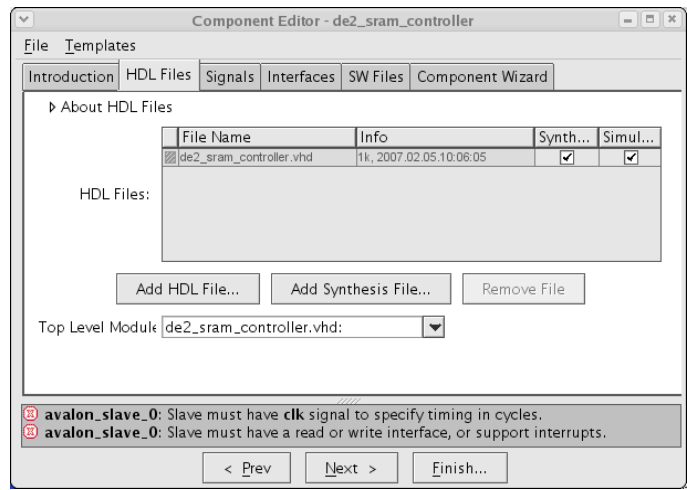

Figure 8: Selecting the top-level VHDL file for a new SOPC component
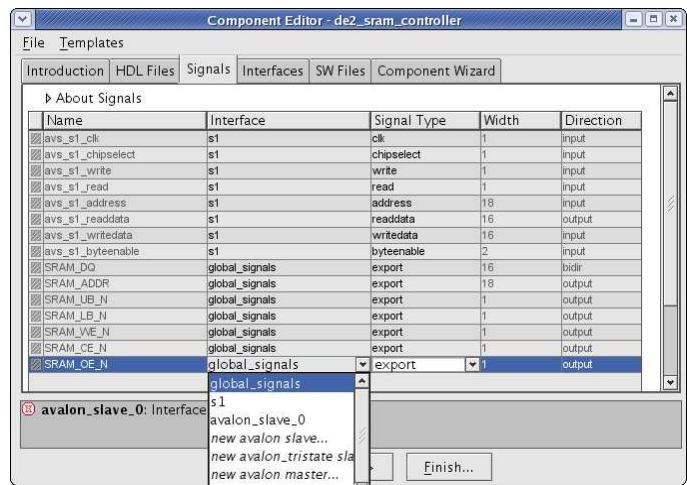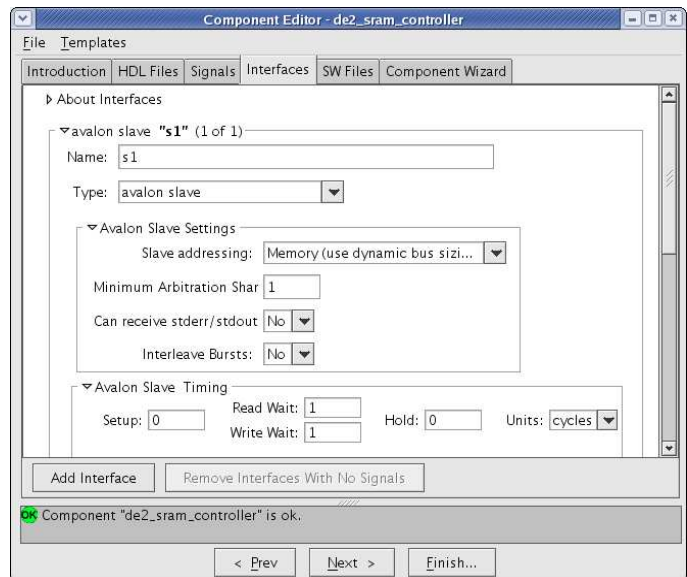

Figure 9: Setting the interfaces for the signals

Figure 10: Configuring the Avalon bus interfaces for the component

3

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity led_flasher is

  port (
    signal avs_s1_clk,
           avs_s1_reset_n,
           avs_s1_read,
           avs_s1_write,
           avs_s1_chipselect : in std_logic;
    signal avs_s1_address
         : in std_logic_vector(4 downto 0);
    signal avs_s1_readdata
         : out std_logic_vector(15 downto 0);
    signal avs_s1_writedata
         : in std_logic_vector(15 downto 0);

    signal leds : out std_logic_vector(15 downto 0)
  );

end led_flasher;

architecture rtl of led_flasher is

signal clk, reset_n : std_logic;

type ram_type is array(15 downto 0)
          of std_logic_vector(15 downto 0);

signal RAM : ram_type;
signal ram_address, display_address
  : std_logic_vector(3 downto 0);
signal counter_delay : std_logic_vector(15 downto 0);
signal counter : std_logic_vector(31 downto 0);

begin

  clk <= avs_s1_clk;
  reset_n <= avs_s1_reset_n;
  ram_address <= avs_s1_address(3 downto 0);

  process (clk)
  begin
    if clk'event and clk = '1' then
      if reset_n = '0' then
        avs_s1_readdata <= (others => '0');
        display_address <= (others => '0');
        counter <= (others => '0');
        counter_delay <= (others => '1');
      else
        if avs_s1_chipselect = '1' then
          if avs_s1_address(4) = '0' then
            if avs_s1_read = '1' then
              avs_s1_readdata <=
                  RAM(conv_integer(ram_address));
            elsif avs_s1_write = '1' then
              RAM(conv_integer(ram_address)) <=
                      avs_s1_writedata;
            end if;
          else
            if avs_s1_write = '1' then
              counter_delay <= avs_s1_writedata;
            end if;
          end if;
        else
          leds <= RAM(conv_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: led_flasher.vhd: VHDL source for the LED flash controller. This memory-maps a 16×16 RAM into 16 halfwords 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.
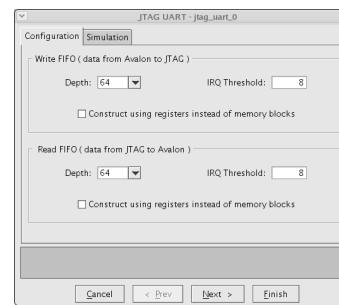

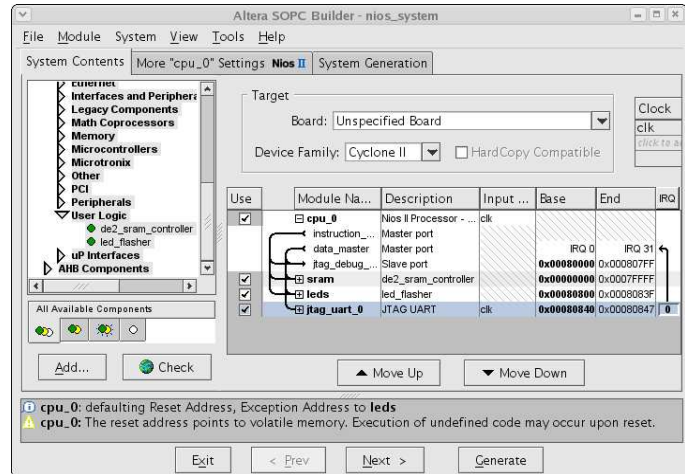
Figure 12: Adding a JTAG UART

Figure 13: The final configuration of the system

ripherals) and click "Generate." This 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 14 shows the top-level VHDL file. Put this in the project directory.

Add all the generated VHDL files to the Quartus project. Select Project→Add/Remove Files in Project and select all the .vhd files in the project directory. This is most easily done by clicking "Add All" and then removing the non-VHDL files. The "altera_europa_support.vhd" file is also not necessary.

By default, the name of the top-level entity is the name of the project2. You can use Project→Set as Top-Level Entity to change this.

4

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity lab3 is

  port (
    signal CLOCK_50 : in std_logic;
    signal LEDR : out std_logic_vector(17 downto 0);

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

end lab3;

architecture rtl of lab3 is

  component nios_system is
  port (
    signal clk                  : in std_logic;
    signal reset_n              : in std_logic;
    signal leds_from_the_leds
              : out std_logic_vector(15 downto 0);
    signal SRAM_ADDR_from_the_sram
              : out std_logic_vector (17 downto 0);
    signal SRAM_CE_N_from_the_sram : out std_logic;
    signal SRAM_DQ_to_and_from_the_sram
              : inout std_logic_vector (15 downto 0);
    signal SRAM_LB_N_from_the_sram : out std_logic;
    signal SRAM_OE_N_from_the_sram : out std_logic;
    signal SRAM_UB_N_from_the_sram : out std_logic;
    signal SRAM_WE_N_from_the_sram : out std_logic
  );
  end component;

  signal counter : std_logic_vector(15 downto 0);
  signal reset_n : std_logic;

begin

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

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

  nios : nios_system port map (
    clk                         => 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;
```
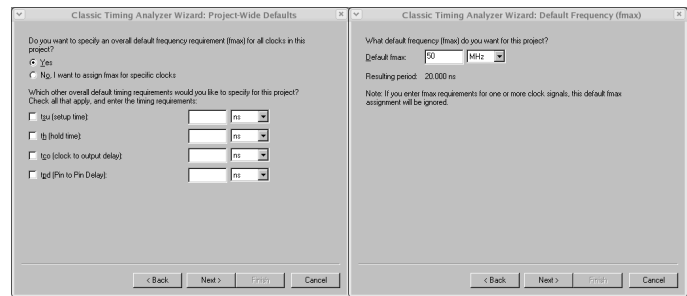
Figure 14: lab3.vhd: The top-level entity

Figure 15: Imposing a global timing constraint

Match the pin names to locations by selecting Assignments→Import Assignments and choosing the DE2_pin_assignments.csv file.

Impose a global timing constraint by choosing Assignments→Classic Timing Analyzer Wizard.

Select an overall default frequency requirement, then set Default fmax to 50 MHz (Figure 15). Leave the defaults alone on the next window, then click Finish.

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 II C/C++ Application.

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

Select the "nios_system.ptf" file in your project directory as the SOPC Builder System. This should set the CPU to "cpu_0."

Finally, select the "Hello World" template and click Finish.

At this point, you can build and run the project on your board, but it does not do much. Instead, replace "hello_world.c" in the lab3_software 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.

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

```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("Hello Michael\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.

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$\sharp$, $p = 2$ 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 I²C bus.

lab3_audio.vhd is a simple top-level module that instantiates the audio controller in test mode and the two I²C 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.

Important: for this peripheral, set the "Slave addressing" mode to "Register." This affects whether the peripheral will appear as two words (register mode) or two bytes (memory mode).

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

```c
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 Model-Sim 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 rectangle on the screen under software control, and a C program that controls the position of this rectangle.

Use the code in video_display.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 rectangle.

Your main 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.

5   An Image Filter