# 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 *quar-tus*.

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 $\rightarrow$ SOPC Builder. This will probably ask you to start creating an SOPC builder system (if not, select File $\rightarrow$ 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.

New Project Wizard	
Directory, Name, Top-Level Entity [page 1 of 5]	
What is the working directory for this project?	
/home/sedwards/svn/classes/2013/4840/labs/lab3	
What is the name of this project?	
lab3	
What is the name of the top-level design entity for this project? This name is case sensi and must exactly match the entity name in the design file.	ive
lab3	
Use Existing Project Settings	
Help     < Back	ncel

Figure 1: Naming a new Quartus project

				Show in 'Avail	able devic	es' list	
Eamily: Cyclo	one II		\$	Package:	Any		\$
Devices: A	ces: All		\$	Pin <u>c</u> ount:	Any		\$
arget device			Speed grade:	Any ep2c35f672		\$	
Auto device	<ul> <li>Auto device selected by the Fitter</li> </ul>					Name filter:	
vailable device		LES	User	I/Os Memo	orv Bits	Embedde	d multiplier 9-bi
-	Core Voltage	<b>LEs</b> 33216	User 1	I/Os Memo 483840	ory Bits	Embedde	d multiplier 9-bi
- Name	Core Voltage				ory Bits		d multiplier 9-bi

Figure 2: Selecting the device in Quartus

S Create New System
System Name: nios_system
Target HDL: O Verilog © VHDL
OK Cancel

Figure 3: Naming a new system in SOPC Builder

OB Altera SOPC Builde	er.				
<u>E</u> ile <u>E</u> dit <u>M</u> odule <u>S</u> ystem <u>V</u> iew <u>T</u> o	ols <u>H</u> elp				
System Contents System Generation					
Component Library	Target	Clock Settings			
l X	Device Family: Cyclone II 💌	Name clk 0	Source External	MHz 50.0	Add
Project Wew Component Library RGBtogreyscaleconvertor		cin_u	LACOTHON	19970	Remove
be-Bridges     be-Configuration & Programming     be-DSP     be-Configuration & Programming     be-DSP     be-InterfaceOreproperture     be-Marcotex and Memory Controller     be-Marcotex and Memory Controller     be-Programming     be-Programming     be-Programming     be-Programming     be-Variation     be-Variation     be-Variation	Use [Con] Name	Descript	ion Clock	Base	
		Ш			•
New Edit 💠 Add				Fiters Fiter: Defa	-
	Egit Help	Prev Next	Generate		

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 $\rightarrow$ 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

MageCara' arameter ettings	s II Process			About Documentation
Core Nios II Core Nios II	aches and Memory Inter	faces 🔪 Advanced Featu	ires 🔪 MMU and MPU Settings 🔪	
elect a Nios II core:				
	⊙ Nios II/e	O Nios II/s	O Nios II/f	
Nios II Selector Guide Family: Cyclone II system: 50.0 MHz cpuid: 0	RISC 32-bit	RISC 32-bit Instruction Cache Branch Prediction Hardware Multiply Hardware Divide	PISC 32-bit Instruction Cache Branch Prediction Hardware Muthply Hardware Muthply Barrel Shifter Data Cache Dynamic Branch Prediction	
erformance at 50.0 MH		Up to 25 DMIPS	Up to 51 DMIPS	
ogic Usage lemory Usage	600-700 LEs Two M4Ks (or equiv.)	1200-1400 LEs Two M4Ks + cache	1400-1800 LEs Three M4Ks + cache	
lardware Multiply:	edded Multipliers	Hardware Divide		
teset Vector: Men		♥ Offset: 0x0     ♥ Offset: 0x20		
Include MMU Inly include the MMU will ast TLB Miss Exception		m that explicitly supports an MM	U Offset: 0x0	
Include MPU			re connected to the Nios I processor	

Figure 5: Adding an Nios II processor in SOPC Builder

Atera SOPC Builder     Bill Create new con     Nios Il Processo     Pridges and Adapter     Interface Protocols     Ceacy Components	Target Device Family: Cyclone II	click Setting:	Name	Source External	50.0 MH		Add Remove
Memories and Memo     Peripherals     Peripherals     PLL     User Logic     Video and Image Pro	Use Connec Module Name Popu Instruction_mas data_master Jtag_debug_mod	ster Avalor Avalor	Description Processor 1 Master 1 Master 1 Slave	Clock	Base IRQ 0 ≓ 0x00000800 0	End IRQ 3 x00000fff	
Add	Remove Ed		h. Move Lip		doress Map	Elter	

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.

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

```
end dp;
```

library ieee;

Figure 7: de2\_sram\_controller.vhd: VHDL source for the SRAM controller (inverters and a tristate buffer).

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 $\rightarrow$ 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

About Signals			1				
Name			Interfac	e	Signal Type	Width	Direction
chipselect	av	alon_slave_0			chipselect	1	input
write	av	alon_slave_0			write	1	input
read	av	alon_slave_0			read	1	input
address	av	avalon_slave_0			address	18	input
readdata	av	avalon_slave_0			readdata	16	output
writedata	av	alon_slave_0			writedata	16	input
byteenable		alon_slave_0			byteenable	2	input
SRAM_DQ		nduit_end			export	16	bidir
SRAM_ADDR	CO	nduit_end			export	18	output
SRAM_UB_N		nduit_end			export	1	output
					export	1	output
SRAM_LB_N	co	nduit_end			export		output
		nduit_end nduit_end			export	1	output
SRAM_WE_N SRAM_CE_N	co	nduit_end nduit_end			export export	1	output output
SRAM_WE_N SRAM_CE_N	co	nduit_end			export	1	output
SRAM_WE_N SRAM_CE_N	co	nduit_end nduit_end			export export	1	output output
SRAM_CE_II SRAM_CE_II SRAM_OE_II	со со со	nduit_end nduit_end nduit_end	Add Si		export export export	1	output output output
SRAM_WE_N SRAM_CE_N SRAM_OE_N	00000000000000000000000000000000000000	nduit_end nduit_end nduit_end			export export export	1	output output output
SRAM_UE_II SRAM_CE_II SRAM_OE_II SRAM_OE_II & Warning_avalor C Error: avalon_s	co co co co co	nduit_end nduit_end nduit_end	e an assoc	iated clock	export export export	1	output output output

Figure 8: Associating the signals with interfaces

8	Component	Editor	- de2_sran	n_contro	oller_hw.	tcl*			
Eile	<u>T</u> emplates								
Co	mponent Type H	DL Files	Parameters	Signals	Interfaces	•			
•	About Interfaces				±				
-									
E.	"avalon_slave_	0" (Avalor	Memory Mapp	ed Slave) –					î
	Nam	e: avalon	slave 0			Documer	atation		
			Memory Mappe	d Slave	•	Docume	itation		
	Тур		memory mappe	u slave					
	Associated Cloc	k: none			-				
	Associated Rese	t: none			-				
	Assignment	s:	Edit						
	* Block Diagram	1 🛛 🖓	Assignmer	its For: a	avalon_s	lave_0			
		-		Key				Value	
			ddedsw.confi				0		
	avalon		ddedsw.confi				1		0000000
	chipseled		ddedsw.confi ddedsw.confi				0		-
	write	embe	dueusw.com	guration.is	FinableD	evice	U		
	read	-							
	address[	1.							
	readdata							OK Cancel	
	writedata								
•	byteenab	ien.ui			I	- 41			) i i
			Add Inte	rface	Romovo k	atorfooon M	/ith No Signals		
			Add Int	anace	Remove i	iter faces vi	auti no Signais		
······									
	Warning: avalon_sla Error: avalon_slave								
<b>•</b>	Error. avalon_stave	_U. Interna	ce must have a	n associate	U CIOCK.				
			Help	E	rev	Next 🕨	Einish		

Figure 9: Editing the assignments

0 I.T. V.I.D.	Files Parameters	Signals Interfaces			
Component Type HDL	Files Parameters	Signals Interfaces			
About Signals					
Name		Interface	Signal Type	Width	Direction
lk	clock	into raco	cik	1	input
eset n	clock		reset n	1	input
ead	avalon_slave_0		read	1	input
rite	avalon_slave_0		write	1	input
hipselect	avalon_slave_0		chipselect	1	input
ddress	avalon_slave_0		address	5	input
eaddata	avalon_slave_0		readdata	16	output
vritedata	avalon_slave_0		writedata	16	input
eds	conduit_end		export	16	output
		Add Signal Remove	e Signal		
-		Add Signal Remov	e Signal		
Info: No errors or warn		Add Signal Remov			

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  $Project \rightarrow Set$  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;
                  std_logic;
  read
             : in
                  std_logic;
  write
             : in
  chipselect : in
                  std_logic;
            : in std_logic_vector(4 downto 0);
  address
  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

#### begin

ram\_address <= unsigned(address(3 downto 0));</pre>

```
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));</pre>
           elsif write = '1' then
             RAM(to_integer(ram_address)) <= writedata;</pre>
           end if:
        else
           if write = '1' then
             counter_delay <= unsigned(writedata);</pre>
             counter <= unsigned(writedata) & x"0000";</pre>
           end if;
        end if:
      else
        leds <= RAM(to_integer(display_address));</pre>
        if counter = x"00000000" then
           counter <= counter_delay & x"0000";</pre>
           display_address <= display_address + 1;</pre>
        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 4 the hold time for each address.

stem Contents System Generation	1	Clock Settings				
mponent Library	Target	-				
X	Device Family: Cyclone II	Name clik Ø	External	urce 50.0	MHz	Ac
Ethernet  High Speed  High Speed  Filder Interlaken  PCI						Ren
► SDI ► SDI II	Use Connec Name	Descriptio	n Clock	Base	End	IRC
SOI II Example Design     SOI II Example Design     SOI II Testbench     Serial     e Avalon-ST JTAG	epu_0 instruction_ma data_master itag debug me	Avalon Memory Mapped	Master [ck]	IF	NO 0	IR0 31-
Avalon-ST Serial     UTAG UART	e sram	de2_sram_controller		0x0010_0000	0x000f ffff	
<ul> <li>9 SPI (3 Wire Serial</li> </ul>	🖌 😑 leds	de2_led_flasher	[clock]	-	_	
<ul> <li>e UART (RS-232 S</li> </ul>	avalon_slave_			<pre>@ 0x0010_1000</pre>	0x0010_103f	
O USB2.0 HS Devic Legacy Components Memories and Memory Contro		JTAG UART Avaion Memory Mapped	[clk] Slave clk_0	# 0x0010_1040	0x0010_1047	<u> </u>
	4		11			
ew Edt 🗣 Add	🗶 Remove 🔯 Edt		Address Map	iters Filter: Default		
Varning: sram.avalon_slave_0: Intert						
/arning: cpu_0.data_master/leds.av	walon_slave_0 leds.avalon_slave_0	does not have byteenables. Narrov	v (less than 32-bit) writes fro	m cpu_0.data_master wil re	esult in spurious writes	to leds.avalo

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 $\rightarrow$ 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 $\rightarrow$ New $\rightarrow$ 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';</pre>
      else
        reset_n <= '0';</pre>
        counter <= counter + 1;</pre>
      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,
    SRAM_CE_N_from_the_sram
    SRAM_DQ_to_and_from_the_sram => SRAM_DQ,
    SRAM_LB_N_from_the_sram
                                  => SRAM_LB_N,
                                  => SRAM_OE_N,
    SRAM_OE_N_from_the_sram
    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

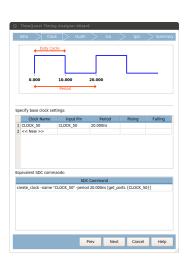


Figure 14: Imposing a clock timing constraint

Nios II Application and BSP from Template
Nios II Software Examples
Create a new application and board support package based on a software example template
Target hardware information SOPC Information File name: //home/sedwards/svn/classes/2013/4840/labs/ CPU name:
Application project
Project name: lab3_software
Use default location     Project location:     //home/sedwards/svn/classes/2013/4840/labs/lab3/soft     remplates     Template description     Blank Project     Bank Proje
Sack Next > Cancel Finish

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

```
#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));</pre>
    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++) {</pre>
    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 $\rightarrow$ Run As $\rightarrow$ 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,  $\omega_c$  is the carrier frequency (the fundamental tone we hear),  $\omega_m$  is the modulating frequency, and I is the modulation depth. The timbre of the sound is largely determined by the ratio  $\omega_c/\omega_m$ , 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 I<sup>2</sup>C 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 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 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.